Tarantool

Выпуск 2.2.1

янв. 28, 2021

Оглавление

1 Создаем свою первую базу данных на Tarantool **2**
- 1.1 Использование Docker-образа . . . 2
- 1.2 Использование менеджера пакетов . . . 5

2 Подключаемся к базе из разных языков программирования **10**
- 2.1 Подключение из Python . . . 10
- 2.2 Подключение из PHP . . . 13
- 2.3 Подключение из Go . . . 16

3 Создаем свое первое приложение на Tarantool Cartridge **21**

4 Руководство пользователя **27**
- 4.1 Предисловие . . . 27
- 4.2 Функциональность СУБД . . . 28
- 4.3 Tarantool Cartridge . . . 73
- 4.4 Сервер приложений . . . 247
- 4.5 Администрирование . . . 287
- 4.6 Репликация . . . 344
- 4.7 Коннекторы . . . 375
- 4.8 Вопросы и ответы . . . 386

5 Справочники **389**
- 5.1 SQL reference . . . 389
- 5.2 Справочник по встроенным модулям . . . 439
- 5.3 Справочник по сторонним библиотекам . . . 729
- 5.4 Справочник по настройке . . . 810
- 5.5 Справочник по C API . . . 836
- 5.6 Детали реализации . . . 864
- 5.7 Interactive console . . . 879
- 5.8 Утилита *tarantoolctl* . . . 880
- 5.9 Рекомендации по Lua-синтаксису . . . 882

6 Практические задания **885**
- 6.1 Практические задания на Lua . . . 885
- 6.2 Практическое задание на C . . . 898
- 6.3 SQL tutorial . . . 905
- 6.4 Улучшаем работу MySQL с помощью Tarantool . . . 916

6.5 Практические задания по *libslave* 921

7 Примечания к версиям **925**
7.1 Version 2.x 925
7.2 Версия 1.10 932
7.3 Версия 1.9 939
7.4 Version 1.8 940
7.5 Версия 1.7 941
7.6 Версия 1.6 953

8 Руководство разработчика **959**
8.1 Содействие в разработке 959
8.2 Рекомендации 966

Lua Module Index **1011**

В этой главе объясняются основы работы с Tarantool как с СУБД, а также приводятся способы подключения к базе на Tarantool из других языков программирования.

ГЛАВА 1

Создаем свою первую базу данных на Tarantool

Первым делом давайте установим Tarantool, запустим его и создадим простую базу данных.

Вы можете установить Tarantool и работать с ним либо локально, либо в Docker – как вам удобнее.

1.1 Использование Docker-образа

Для практики и тестирования мы рекомендуем использовать официальные образы Tarantool'а для Docker. Официальный образ содержит определенную версию Tarantool'а и все популярные внешние модули для Tarantool'а. Все необходимое уже установлено и настроено на платформе Linux. Данные образы - это самый простой способ установить и запустить Tarantool.

Примечание: Если вы никогда раньше не работали с Docker, рекомендуем сперва прочитать эту обучающую статью.

1.1.1 Запуск контейнера

Если Docker не установлен на вашей машине, следуйте официальным инструкциям по установке для вашей ОС.

Для использования полнофункционального экземпляра Tarantool'а запустите контейнер с минимальными настройками:

```
$ docker run \
  --name mytarantool \
  -d -p 3301:3301 \
  -v /data/dir/on/host:/var/lib/tarantool \
  tarantool/tarantool:2
```

Эта команда запускает новый контейнер с именем „mytarantool“. Docker запускает его из официального образа „tarantool/tarantool:2“ с предустановленным Tarantool'ом `2.2` и всеми внешними модулями.

Tarantool будет принимать входящие подключения по адресу `localhost:3301`. Можно сразу начать его использовать как key-value хранилище.

Tarantool *сохраняет данные* внутри контейнера. Чтобы ваше тестовые данные остались доступны после остановки контейнера, эта команда также монтирует директорию `/data/dir/on/host` (здесь необходимо указать абсолютный путь до существующей локальной директории), расположенную на машине, в директорию `/var/lib/tarantool` (Tarantool традиционно использует эту директорию в контейнере для сохранения данных), расположенную в контейнере. Таким образом все изменения в смонтированной директории, внесенные на стороне контейнера, также отражаются в расположенной на пользовательском диске директории.

Модуль Tarantool'а для работы с базой данных уже *настроен* и запущен в контейнере. Ручная настройка не требуется, если только вы не используете Tarantool как *сервер приложений* и не запускаете его вместе с приложением.

Примечание: Если ваш контейнер рушится вскоре после запуска, перейдите на эту страницу, чтобы найти возможное решение.

1.1.2 Подключение к экземпляру Tarantool'а

Для подключения к запущенному в контейнере экземпляру Tarantool'а, выполните эту команду:

```
$ docker exec -i -t mytarantool console
```

Эта команда:

- Требует от Tarantool'а открыть порт с интерактивной консолью для входящих подключений.
- Подключается через стандартный Unix-сокет к Tarantool-серверу, запущенному внутри контейнера, из-под пользователя `admin`.

Tarantool показывает приглашение командной строки:

```
tarantool.sock>
```

Теперь вы можете вводить запросы в командной строке.

Примечание: На боевых серверах интерактивный режим Tarantool'а предназначен только для системных администраторов. Мы же используем его в большинстве примеров в данном руководстве, потому что интерактивный режим хорошо подходит для обучения.

1.1.3 Создание базы данных

Подключившись к консоли, давайте создадим простую тестовую базу данных.

Сначала создайте первый *спейс* (с именем `tester`):

```
tarantool.sock> s = box.schema.space.create('tester')
```

Форматируйте созданный спейс, указав имена и типы полей:

```
tarantool.sock> s:format({
              > {name = 'id', type = 'unsigned'},
              > {name = 'band_name', type = 'string'},
              > {name = 'year', type = 'unsigned'}
              > })
```

Создайте первый индекс (с именем `primary`):

```
tarantool.sock> s:create_index('primary', {
              > type = 'hash',
              > parts = {'id'}
              > })
```

Это первичный индекс по полю `id` в каждом кортеже.

Вставьте в созданный спейс три *кортежа* (наш термин для записей):

```
tarantool.sock> s:insert{1, 'Roxette', 1986}
tarantool.sock> s:insert{2, 'Scorpions', 2015}
tarantool.sock> s:insert{3, 'Ace of Base', 1993}
```

Для выборки кортежей по первичному индексу `primary` выполните команду:

```
tarantool.sock> s:select{3}
```

Теперь вывод в окне терминала выглядит следующим образом:

```
tarantool.sock> s = box.schema.space.create('tester')
---
...
tarantool.sock> s:format({
              > {name = 'id', type = 'unsigned'},
              > {name = 'band_name', type = 'string'},
              > {name = 'year', type = 'unsigned'}
              > })
---
...
tarantool.sock> s:create_index('primary', {
              > type = 'hash',
              > parts = {'id'}
              > })
---
- unique: true
  parts:
  - type: unsigned
    is_nullable: false
    fieldno: 1
  id: 0
  space_id: 512
  name: primary
  type: HASH
...
tarantool.sock> s:insert{1, 'Roxette', 1986}
---
- [1, 'Roxette', 1986]
...
tarantool.sock> s:insert{2, 'Scorpions', 2015}
```

(continues on next page)

(продолжение с предыдущей страницы)

```
---
- [2, 'Scorpions', 2015]
...
tarantool.sock> s:insert{3, 'Ace of Base', 1993}
---
- [3, 'Ace of Base', 1993]
...
tarantool.sock> s:select{3}
---
- - [3, 'Ace of Base', 1993]
...
```

Для добавления вторичного индекса по полю `band_name` используйте эту команду:

```
tarantool.sock> s:create_index('secondary', {
              > type = 'hash',
              > parts = {'band_name'}
              > })
```

Для выборки кортежей по вторичному индексу `secondary` выполните команду:

```
tarantool.sock> s.index.secondary:select{'Scorpions'}
---
- - [2, 'Scorpions', 2015]
...
```

Чтобы удалить индекс, выполните:

```
tarantool> s.index.secondary:drop()
---
...
```

1.1.4 Остановка контейнера

После завершения тестирования для корректной остановки контейнера выполните эту команду:

```
$ docker stop mytarantool
```

Это был временный контейнер, поэтому после остановки содержимое его диска/памяти обнулилось. Но так как вы монтировали локальную директорию в контейнер, все данные Tarantool'а сохранились на диске вашей машины. Если вы запустите новый контейнер и смонтируете в него ту же директорию с данными, Tarantool восстановит все данные с диска и продолжит с ними работать.

1.2 Использование менеджера пакетов

Для промышленной разработки мы рекомендуем устанавливать Tarantool с помощью официального менеджера пакетов. Можно выбрать одну из трех версий: LTS, stable или beta. Автоматическая система сборки создает, тестирует и публикует пакеты после каждого коммита в соответствующую ветку репозитория Tarantool'а на GitHub.

Чтобы скачать и установить подходящий пакет, откройте командную строку и введите инструкции, которые даны для вашей операционной системы на странице для скачивания.

1.2.1 Запуск экземпляра Tarantool'а

Чтобы начать работу с Tarantool, выполните эту команду:

```
$ tarantool
$ # при этом создается новый экземпляр Tarantool
```

Tarantool запускается в интерактивном режиме и показывает приглашение командной строки:

```
tarantool>
```

Теперь вы можете вводить запросы в командной строке.

Примечание: На боевых серверах интерактивный режим Tarantool'а предназначен только для системных администраторов. Мы же используем его в большинстве примеров в данном руководстве, потому что интерактивный режим хорошо подходит для обучения.

1.2.2 Создание базы данных

Далее объясняется, как создать простую тестовую базу данных после установки Tarantool'а.

1. Чтобы Tarantool хранил данные в определенном месте, создайте предназначенную специально для тестов директорию:

   ```
   $ mkdir ~/tarantool_sandbox
   $ cd ~/tarantool_sandbox
   ```

   Ее можно удалить после окончания тестирования.

2. Проверьте доступность порта, используемого по умолчанию для прослушивания на экземпляре базы данных.

   В зависимости от версии, Tarantool может во время установки запустить экземпляр `example.lua`, который настроен на прослушивание по порту 3301 по умолчанию. В файле `example.lua` показана базовая конфигурация; его можно найти в директории `/etc/tarantool/instances.enabled` или `/etc/tarantool/instances.available`.

   Тем не менее, мы предлагаем провести установку самостоятельно с целью обучения.

   Убедитесь, что свободен порт, используемый по умолчанию:

   1. Чтобы проверить статус работы демонстрационного экземпляра, выполните команду:

      ```
      $ lsof -i :3301
      COMMAND    PID USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
      tarantool 6851 root   12u  IPv4  40827      0t0  TCP *:3301 (LISTEN)
      ```

   2. Если он запущен, отключите соответствующий процесс. В данном примере:

      ```
      $ kill 6851
      ```

3. Чтобы запустить модуль Tarantool'а для работы с базой данных и сделать так, чтобы запущенный экземпляр принимал TCP-запросы на порту 3301, выполните эту команду:

   ```
   tarantool> box.cfg{listen = 3301}
   ```

4. Создайте первый *спейс* (с именем `tester`):

```
tarantool> s = box.schema.space.create('tester')
```

5. Форматируйте созданный спейс, указав имена и типы полей:

```
tarantool> s:format({
         > {name = 'id', type = 'unsigned'},
         > {name = 'band_name', type = 'string'},
         > {name = 'year', type = 'unsigned'}
         > })
```

6. Создайте первый индекс (с именем `primary`):

```
tarantool> s:create_index('primary', {
         > type = 'hash',
         > parts = {'id'}
         > })
```

   Это первичный индекс по полю `id` в каждом кортеже.

7. Вставьте в созданный спейс три *кортежа* (наш термин для записей):

```
tarantool> s:insert{1, 'Roxette', 1986}
tarantool> s:insert{2, 'Scorpions', 2015}
tarantool> s:insert{3, 'Ace of Base', 1993}
```

8. Для выборки кортежей по первичному индексу `primary` выполните команду:

```
tarantool> s:select{3}
```

   Теперь вывод в окне терминала выглядит следующим образом:

```
tarantool> s = box.schema.space.create('tester')
---
...
tarantool> s:format({
         > {name = 'id', type = 'unsigned'},
         > {name = 'band_name', type = 'string'},
         > {name = 'year', type = 'unsigned'}
         > })
---
...
tarantool> s:create_index('primary', {
         > type = 'hash',
         > parts = {'id'}
         > })
---
- unique: true
  parts:
  - type: unsigned
    is_nullable: false
    fieldno: 1
  id: 0
  space_id: 512
  name: primary
  type: HASH
...
tarantool> s:insert{1, 'Roxette', 1986}
```

(continues on next page)

(продолжение с предыдущей страницы)

```
---
- [1, 'Roxette', 1986]
...
tarantool> s:insert{2, 'Scorpions', 2015}
---
- [2, 'Scorpions', 2015]
...
tarantool> s:insert{3, 'Ace of Base', 1993}
---
- [3, 'Ace of Base', 1993]
...
tarantool> s:select{3}
---
- - [3, 'Ace of Base', 1993]
...
```

9. Для добавления вторичного индекса по полю `band_name` используйте эту команду:

```
tarantool> s:create_index('secondary', {
         > type = 'hash',
         > parts = {'band_name'}
         > })
```

10. Для выборки кортежей по вторичному индексу `secondary` выполните команду:

```
tarantool> s.index.secondary:select{'Scorpions'}
---
- - [2, 'Scorpions', 2015]
...
```

11. Теперь, чтобы подготовиться к примеру в следующем разделе, попробуйте следующее:

```
tarantool> box.schema.user.grant('guest', 'read,write,execute', 'universe')
```

1.2.3 Установка удаленного подключения

В запросе `box.cfg{listen = 3301}`, который мы отправили ранее, параметр `listen` может принимать в качестве значения *URI* (унифицированный идентификатор ресурса) любой формы. В нашем случае это просто локальный порт 3301. Вы можете отправлять запросы на указанный URI, используя:

(1) `telnet`,

(2) *коннектор*,

(3) другой экземпляр Tarantool'а (с помощью модуля *console*), либо

(4) утилиту *tarantoolctl*.

Давайте попробуем вариант с `tarantoolctl`.

Переключитесь на другой терминал. Например, в Linux-системе для этого нужно запустить еще один экземпляр Bash. В новом терминале можно сменить текущую рабочую директорию на любую другую, необязательно использовать `~/tarantool_sandbox`.

Запустите утилиту `tarantoolctl`:

```
$ tarantoolctl connect '3301'
```

Данная команда означает «использовать утилиту `tarantoolctl` для подключения к Tarantool-серверу, который слушает по адресу `localhost:3301`».

Введите следующий запрос:

```
localhost:3301> box.space.tester:select{2}
```

Это означает «послать запрос тому Tarantool-серверу и вывести результат на экран». Результатом в данном случае будет один из кортежей, что вы вставляли ранее. В окне терминала теперь должно отображаться примерно следующее:

```
$ tarantoolctl connect 3301
/usr/local/bin/tarantoolctl: connected to localhost:3301
localhost:3301> box.space.tester:select{2}
---
- - [2, 'Scorpions', 2015]
...
```

Вы можете посылать запросы `box.space...:insert{}` и `box.space...:select{}` неограниченное количество раз на любом из двух запущенных экземпляров Tarantool'а.

Закончив тестирование, выполните следующие шаги:

- Для удаления спейса: `s:drop()`
- Для остановки `tarantoolctl`: ctrl+C или ctrl+D
- Для остановки Tarantool'а (альтернативный вариант): стандартная Lua-функция os.exit()
- Для остановки Tarantool'а (из другого терминала): `sudo pkill -f tarantool`
- Для удаления директории-песочницы: `rm -r ~/tarantool_sandbox`

ГЛАВА 2

Подключаемся к базе из разных языков программирования

Итак, мы создали базу данных в Tarantool. Теперь давайте посмотрим, как к ней можно подключиться из Python, PHP и Go.

2.1 Подключение из Python

2.1.1 Подготовка

Перед тем как идти дальше, выполним следующие действия:

1. Установим библиотеку `tarantool`. Рекомендуется использовать `python3` и `pip3`.
2. *Запустим* Tarantool (локально или в Docker) и обязательно создадим базу данных с тестовыми данными, как показано в *предыдущем разделе*:

```
box.cfg{listen = 3301}
s = box.schema.space.create('tester')
s:format({
         {name = 'id', type = 'unsigned'},
         {name = 'band_name', type = 'string'},
         {name = 'year', type = 'unsigned'}
         })
s:create_index('primary', {
         type = 'hash',
         parts = {'id'}
         })
s:create_index('secondary', {
         type = 'hash',
         parts = {'band_name'}
         })
s:insert{1, 'Roxette', 1986}
s:insert{2, 'Scorpions', 2015}
s:insert{3, 'Ace of Base', 1993}
```

Важно: Не закрывайте окно терминала с запущенным Tarantool – оно пригодится нам позднее.

3. Чтобы иметь возможность подключаться к Tarantool в качестве администратора, сменим пароль пользователя `admin`:

```
box.schema.user.passwd('pass')
```

2.1.2 Подключение к Tarantool

Для подключения к серверу достаточно выполнить следующее:

```
>>> import tarantool
>>> connection = tarantool.connect("localhost", 3301)
```

Также при необходимости можно указать имя пользователя и пароль:

```
>>> tarantool.connect("localhost", 3301, user=username, password=password)
```

По умолчанию используется пользователь `guest`.

2.1.3 Работа с данными

Спейс – это контейнер для **кортежей**. Чтобы обратиться к спейсу как к именованному объекту, воспользуемся функцией `connection.space`:

```
>>> tester = connection.space('tester')
```

Вставка данных

Для вставки нового кортежа в спейс воспользуемся функцией `insert`:

```
>>> tester.insert((4, 'ABBA', 1972))
[4, 'ABBA', 1972]
```

Получение данных

Сначала выберем кортеж по первичному ключу (в нашем примере первичный индекс —это индекс `primary`, построенный по полю `id` в каждом кортеже). Воспользуемся функцией `select`:

```
>>> tester.select(4)
[4, 'ABBA', 1972]
```

Теперь поищем кортежи по вторичному ключу. Для этого нужно указать номер *или* имя вторичного индекса.

Сначала сделаем запрос по номеру индекса:

```
>>> tester.select('Scorpions', index=1)
[2, 'Scorpions', 2015]
```

(Мы указываем `index=1`, потому что индексы в Tarantool нумеруются с нуля, а в данном случае мы обращаемся к индексу, который создавали вторым.)

Теперь сделаем аналогичный запрос по имени индекса и получим тот же результат:

```
>>> tester.select('Scorpions', index='secondary')
[2, 'Scorpions', 2015]
```

А чтобы выбрать все кортежи из спейса, вызовем `select` без аргументов:

```
>>> tester.select()
```

Обновление данных

Обновим значение поля с помощью `update`:

```
>>> tester.update(4, [('=', 1, 'New group'), ('+', 2, 2)])
```

Здесь мы обновляем значение поля `1` и увеличиваем значение поля `2` для кортежа с `id` = 4. Если кортежа с таким `id` нет, то Tarantool вернет ошибку.

Теперь с помощью функции `replace` мы полностью заменим кортеж с совпадающим первичным ключом. Если кортежа с указанным первичным ключом не существует, то эта операция ни к чему не приведет.

```
>>> tester.replace((4, 'New band', 2015))
```

Также мы можем обновлять данные с помощью функции `upsert`, которая работает аналогично `update`, но создает новый кортеж, если старый не был найден.

```
>>> tester.upsert((4, 'Another band', 2000), [('+', 2, 5)])
```

Здесь мы увеличиваем на 5 значение поля `2` в кортеже с `id` = 4 – или же вставляем кортеж `(4, "Another band", 2000)`, если такого нет.

Удаление данных

Чтобы удалить кортеж, нужно использовать `delete(primary_key)`:

```
>>> tester.delete(4)
[4, 'New group', 2012]
```

Для удаления всех кортежей в спейсе (или всего спейса целиком) нужно воспользоваться функцией `call`. Мы поговорим о ней подробнее в *следующем разделе*.

Чтобы удалить все кортежи в спейсе, нужно вызвать функцию `space:truncate`:

```
>>> connection.call('box.space.tester:truncate', ())
```

Чтобы удалить весь спейс, нужно вызвать функцию `space:drop`. Для выполнения следующей команды необходимо подключиться из-под пользователя `admin`:

```
>>> connection.call('box.space.tester:drop', ())
```

2.1.4 Исполнение хранимых процедур

Перейдем в терминал с запущенным Tarantool.

Примечание: О том, как установить удаленное подключение к Tarantool, можно прочитать здесь:

- *как подключиться к Tarantool, запущенному локально*
- *как подключиться к Tarantool, запущенному в Docker-контейнере*

Напишем простую функцию на Lua:

```
function sum(a, b)
    return a + b
end
```

Итак, теперь у нас есть функция, описанная в Tarantool. Чтобы вызвать ее из `python`, нам нужна функция `call`:

```
>>> connection.call('sum', (3, 2))
5
```

Также мы можем передать на выполнение любой Lua-код. Для этого воспользуемся функцией `eval`:

```
>>> connection.eval('return 4 + 5')
9
```

2.2 Подключение из PHP

2.2.1 Подготовка

Перед тем как идти дальше, выполним следующие действия:

1. Установим библиотеку `tarantool/client`.
2. *Запустим* Tarantool (локально или в Docker) и обязательно создадим базу данных с тестовыми данными, как показано в *предыдущем разделе*:

```
box.cfg{listen = 3301}
s = box.schema.space.create('tester')
s:format({
         {name = 'id', type = 'unsigned'},
         {name = 'band_name', type = 'string'},
         {name = 'year', type = 'unsigned'}
         })
s:create_index('primary', {
         type = 'hash',
         parts = {'id'}
         })
s:create_index('secondary', {
         type = 'hash',
         parts = {'band_name'}
         })
s:insert{1, 'Roxette', 1986}
```

(continues on next page)

(продолжение с предыдущей страницы)

```
s:insert{2, 'Scorpions', 2015}
s:insert{3, 'Ace of Base', 1993}
```

Важно: Не закрывайте окно терминала с запущенным Tarantool – оно пригодится нам позднее.

3. Чтобы иметь возможность подключаться к Tarantool в качестве администратора, сменим пароль пользователя `admin`:

```
box.schema.user.passwd('pass')
```

2.2.2 Подключение к Tarantool

Для настройки подключения к серверу достаточно выполнить следующее:

```
use Tarantool\Client\Client;

require __DIR__.'/vendor/autoload.php';
$client = Client::fromDefaults();
```

Само подключение будет установлено при первом запросе. Также при необходимости можно указать имя пользователя и пароль:

```
$client = Client::fromOptions([
    'uri' => 'tcp://127.0.0.1:3301',
    'username' => '<username>',
    'password' => '<password>'
]);
```

По умолчанию используется пользователь `guest`.

2.2.3 Работа с данными

Спейс – это контейнер для **кортежей**. Чтобы обратиться к спейсу как к именованному объекту, воспользуемся функцией `getSpace`:

```
$tester = $client->getSpace('tester');
```

Вставка данных

Для вставки нового кортежа в спейс воспользуемся функцией `insert`:

```
$result = $tester->insert([4, 'ABBA', 1972]);
```

Получение данных

Сначала выберем кортеж по первичному ключу (в нашем примере первичный индекс —это индекс `primary`, построенный по полю `id` в каждом кортеже). Воспользуемся функцией `select`:

```
use Tarantool\Client\Schema\Criteria;

$result = $tester->select(Criteria::key([4]));
printf(json_encode($result));
```

```
[[4, 'ABBA', 1972]]
```

Теперь поищем кортежи по вторичному ключу. Для этого нужно указать номер *или* имя вторичного индекса.

Сначала сделаем запрос по номеру индекса:

```
$result = $tester->select(Criteria::index(1)->andKey(['Scorpions']));
printf(json_encode($result));
```

```
[2, 'Scorpions', 2015]
```

(Мы указываем `index(1)`, потому что индексы в Tarantool нумеруются с нуля, а в данном случае мы обращаемся к индексу, который создавали вторым.)

Теперь сделаем аналогичный запрос по имени индекса и получим тот же результат:

```
$result = $tester->select(Criteria::index('secondary')->andKey(['Scorpions']));
printf(json_encode($result));
```

```
[2, 'Scorpions', 2015]
```

А чтобы выбрать все кортежи из спейса, вызовем `select`:

```
$result = $tester->select(Criteria::allIterator());
```

Обновление данных

Обновим значение поля с помощью `update`:

```
use Tarantool\Client\Schema\Operations;

$result = $tester->update([4], Operations::set(1, 'New group')->andAdd(2, 2));
```

Здесь мы обновляем значение поля `1` и увеличиваем значение поля `2` для кортежа с `id` = 4. Если кортежа с таким `id` нет, то Tarantool вернет ошибку.

Теперь с помощью функции `replace` мы полностью заменим кортеж с совпадающим первичным ключом. Если кортежа с указанным первичным ключом не существует, то эта операция ни к чему не приведет.

```
$result = $tester->replace([4, 'New band', 2015]);
```

Также мы можем обновлять данные с помощью функции `upsert`, которая работает аналогично `update`, но создает новый кортеж, если старый не был найден.

```
use Tarantool\Client\Schema\Operations;

$tester->upsert([4, 'Another band', 2000], Operations::add(2, 5));
```

Здесь мы увеличиваем на 5 значение поля 2 в кортеже с `id` = 4 – или же вставляем кортеж `(4, "Another band", 2000)`, если такого нет.

Удаление данных

Чтобы удалить кортеж, нужно использовать `delete(primary_key)`:

```
$result = $tester->delete([4]);
```

Для удаления всех кортежей в спейсе (или всего спейса целиком) нужно воспользоваться функцией `call`. Мы поговорим о ней подробнее в *следующем разделе*.

Чтобы удалить все кортежи в спейсе, нужно вызвать функцию `space:truncate`:

```
$result = $client->call('box.space.tester:truncate');
```

Чтобы удалить весь спейс, нужно вызвать функцию `space:drop`. Для выполнения следующей команды необходимо подключиться из-под пользователя `admin`:

```
$result = $client->call('box.space.tester:drop');
```

2.2.4 Исполнение хранимых процедур

Перейдем в терминал с запущенным Tarantool.

Примечание: О том, как установить удаленное подключение к Tarantool, можно прочитать здесь:

- *как подключиться к Tarantool, запущенному локально*
- *как подключиться к Tarantool, запущенному в Docker-контейнере*

Напишем простую функцию на Lua:

```
function sum(a, b)
    return a + b
end
```

Итак, теперь у нас есть функция, описанная в Tarantool. Чтобы вызвать ее из `php`, нам нужна функция `call`:

```
$result = $client->call('sum', 3, 2);
```

Также мы можем передать на выполнение любой Lua-код. Для этого воспользуемся функцией `eval`:

```
$result = $client->evaluate('return 4 + 5');
```

2.3 Подключение из Go

2.3.1 Подготовка

Перед тем как идти дальше, выполним следующие действия:

1. Установим библиотеку `go-tarantool`.
2. *Запустим* Tarantool (локально или в Docker) и обязательно создадим базу данных с тестовыми данными, как показано в *предыдущем разделе*:

```
box.cfg{listen = 3301}
s = box.schema.space.create('tester')
s:format({
         {name = 'id', type = 'unsigned'},
         {name = 'band_name', type = 'string'},
         {name = 'year', type = 'unsigned'}
         })
s:create_index('primary', {
         type = 'hash',
         parts = {'id'}
         })
s:create_index('secondary', {
         type = 'hash',
         parts = {'band_name'}
         })
s:insert{1, 'Roxette', 1986}
s:insert{2, 'Scorpions', 2015}
s:insert{3, 'Ace of Base', 1993}
```

Важно: Не закрывайте окно терминала с запущенным Tarantool – оно пригодится нам позднее.

3. Чтобы иметь возможность подключаться к Tarantool в качестве администратора, сменим пароль пользователя `admin`:

```
box.schema.user.passwd('pass')
```

2.3.2 Подключение к Tarantool

Простая программа, выполняющая подключение к серверу, будет выглядеть так:

```
package main

import (
    "fmt"

    "github.com/tarantool/go-tarantool"
)

func main() {

    conn, err := tarantool.Connect("127.0.0.1:3301", tarantool.Opts{
            User: "admin",
            Pass: "pass",
    })

    if err != nil {
            log.Fatalf("Connection refused")
    }

    defer conn.Close()
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    // Ваш код общения с базой

}
```

По умолчанию используется пользователь `guest`.

2.3.3 Работа с данными

Вставка данных

Для вставки нового кортежа в спейс воспользуемся функцией `Insert`:

```
resp, err = conn.Insert("tester", []interface{}{4, "ABBA", 1972})
```

В этом примере в спейс `tester` вставляется кортеж `(4, "ABBA", 1972)`.

Код ответа и данные можно получить из структуры tarantool.Response:

```
code := resp.Code
data := resp.Data
```

Получение данных

Чтобы выбрать кортеж из спейса, воспользуемся функцией Select:

```
resp, err = conn.Select("tester", "primary", 0, 1, tarantool.IterEq, []interface{}{4})
```

В этом примере мы ищем кортеж по первичному ключу с `offset` = 0 и `limit` = 1 в спейсе `tester` (первичный индекс в нашем примере – это индекс `primary`, построенный по полю `id` в каждом кортеже).

Теперь поищем по вторичному ключу:

```
resp, err = conn.Select("tester", "secondary", 0, 1, tarantool.IterEq, []interface{}{"ABBA"})
```

Наконец, было бы интересно сделать полную выборку даных из спейса. Но в рамках языка Go эта задача не решается в одну строчку. *Пример* такой программы вы можете посмотреть в отдельном разделе документации.

Более сложные примеры выборок можно увидеть тут: https://github.com/tarantool/go-tarantool#usage

Обновление данных

Обновим значение поля с помощью `Update`:

```
resp, err = conn.Update("tester", "primary", []interface{}{4}, []interface{}{[]interface{}{"+", 2,␣
↪3}})
```

Здесь мы увеличиваем на 3 значение поля 2 для кортежа с `id` = 4. Если кортежа с таким `id` нет, то Tarantool вернет ошибку.

Теперь с помощью функции `Replace` мы полностью заменим кортеж с совпадающим первичным ключом. Если кортежа с указанным первичным ключом не существует, то эта операция ни к чему не приведет.

```
resp, err = conn.Replace("tester", []interface{}{4, "New band", 2011})
```

Также мы можем обновлять данные с помощью функции `Upsert`, которая работает аналогично `Update`, но создает новый кортеж, если старый не был найден.

```
resp, err = conn.Upsert("tester", []interface{}{4, "Another band", 2000}, []interface{}{[]interface
↪{}{"+", 2, 5}})
```

Здесь мы увеличиваем на 5 значение третьего поля в кортеже с `id` = 4 – или же вставляем кортеж `(4, "Another band", 2000)`, если такого нет.

Удаление данных

Чтобы удалить кортеж, воспользуемся функцией `connection.Delete`:

```
resp, err = conn.Delete("tester", "primary", []interface{}{4})
```

Для удаления всех кортежей в спейсе (или всего спейса целиком), нужно воспользоваться функцией `Call`. Мы поговорим о ней подробнее в *следующем разделе*.

Чтобы удалить все кортежи в спейсе, нужно вызвать функцию `space:truncate`:

```
resp, err = conn.Call("box.space.tester:truncate", []interface{}{})
```

Чтобы удалить весь спейс, нужно вызвать функцию `space:drop`. Для выполнения следующей команды необходимо подключиться из-под пользователя `admin`:

```
resp, err = conn.Call("box.space.tester:drop", []interface{}{})
```

2.3.4 Исполнение хранимых процедур

Перейдем в терминал с запущенным Tarantool.

Примечание: О том, как установить удаленное подключение к Tarantool, можно прочитать здесь:

- *как подключиться к Tarantool, запущенному локально*
- *как подключиться к Tarantool, запущенному в Docker-контейнере*

Напишем простую функцию на Lua:

```
function sum(a, b)
    return a + b
end
```

Итак, теперь у нас есть функция, описанная в Tarantool. Чтобы вызвать ее из `go`, нам нужна функция `Call`:

```
resp, err = conn.Call("sum", []interface{}{2, 3})
```

Также мы можем передать на выполнение любой Lua-код. Для этого воспользуемся функцией `Eval`:

```
resp, err = connection.Eval("return 4 + 5", []interface{}{})
```

ГЛАВА 3

Создаем свое первое приложение на Tarantool Cartridge

Здесь мы показываем, как сделать простое кластерное приложение.

Первым делом настройте среду разработки.

Затем создайте приложение с именем `myapp`. Выполните:

```
$ cartridge create --name myapp
```

Эта команда создает новое Tarantool Cartridge-приложение в директории `./myapp`. Там теперь содержатся созданные по шаблону файлы и директории.

Войдите внутрь этой директории и запустите ваше приложение:

```
$ cd ./myapp
$ cartridge build
$ cartridge start
```

Эта команда собирает приложение локально, стартует 5 экземпляров Tarantool и запускает приложение в том виде, как оно было создано – без какой-либо интересной бизнес-логики.

Откуда взялись 5 экземпляров? Загляните внутрь файла `instances.yml`.Там задается *конфигурация* всех экземпляров, которые вы можете настроить внутри вашего кластера. По умолчанию, там задана конфигурация 5 экземпляров.

```
myapp.router:
  workdir: ./tmp/db_dev/3301
  advertise_uri: localhost:3301
  http_port: 8081

myapp.s1-master:
  workdir: ./tmp/db_dev/3302
  advertise_uri: localhost:3302
  http_port: 8082

myapp.s1-replica:
```

(continues on next page)

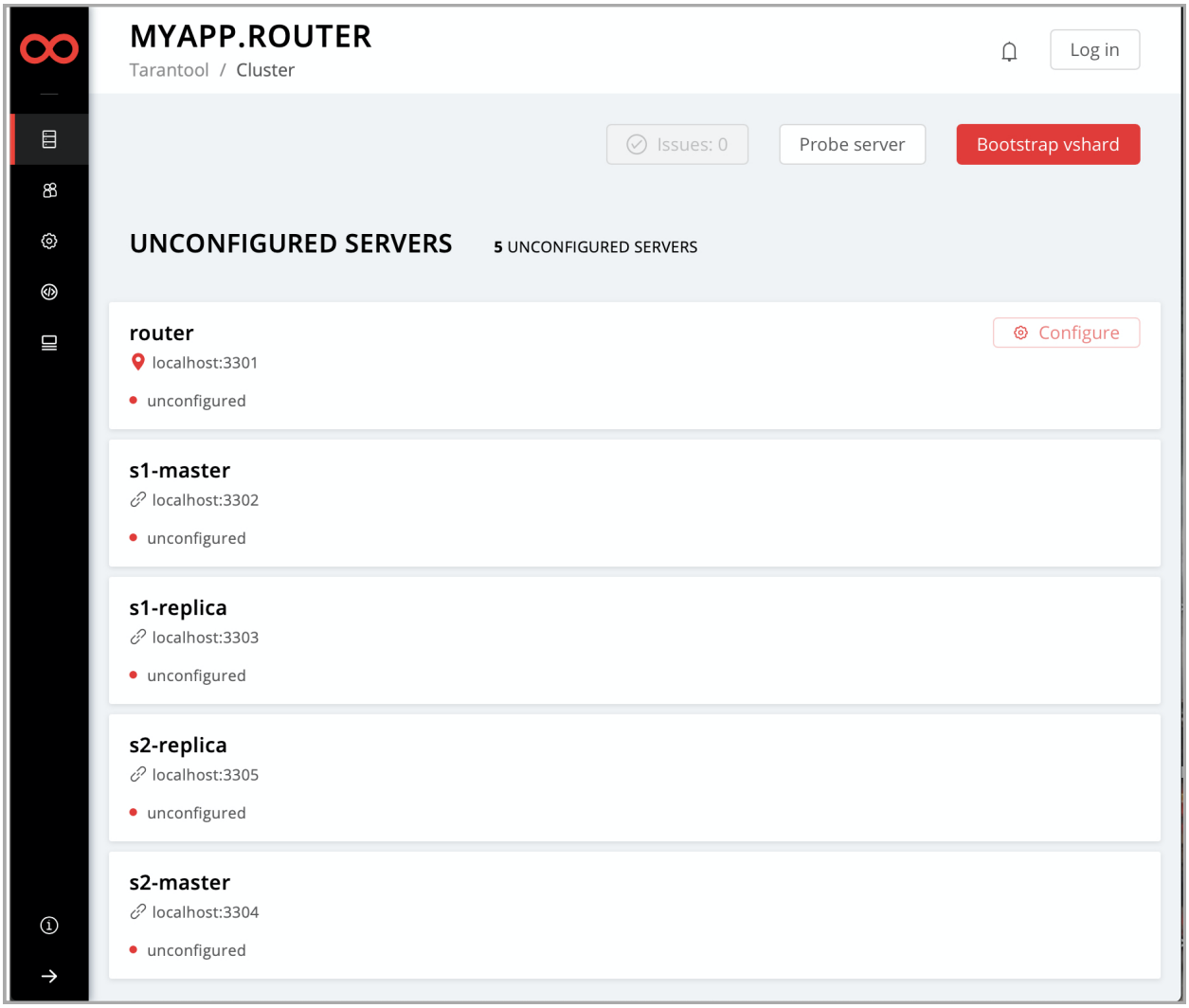
(продолжение с предыдущей страницы)

```
  workdir: ./tmp/db_dev/3303
  advertise_uri: localhost:3303
  http_port: 8083

myapp.s2-master:
  workdir: ./tmp/db_dev/3304
  advertise_uri: localhost:3304
  http_port: 8084

myapp.s2-replica:
  workdir: ./tmp/db_dev/3305
  advertise_uri: localhost:3305
  http_port: 8085
```

Вы можете увидеть все эти экземпляры в веб-интерфейсе для управления кластером по адресу http://localhost:8081 (порт 8081 – это HTTP-порт первого экземпляра из файла `instances.yml`).



Теперь временно остановите кластер с помощью `Ctrl + C`.

Пора заняться написанием бизнес-логики для вашего приложения. Чтобы не слишком усложнять наш пример, возьмем канонический «Hello world!»».

Переименуйте шаблонный файл `app/roles/custom.lua` в `hello-world.lua`.

```
$ mv app/roles/custom.lua app/roles/hello-world.lua
```

Это будет ваша *роль*. Роль в Tarantool Cartridge – это Lua-модуль, в котором реализованы специфичные для экземпляра Tarantool функции и логика. Далее мы покажем, как добавлять в роль свой код, собирать ее, назначать и тестировать.

У вашей роли уже есть некоторый код внутри функции `init()`.

```
local function init(opts) -- luacheck: no unused args
    -- if opts.is_master then
    -- end

    local httpd = cartridge.service_get('httpd')
    httpd:route({method = 'GET', path = '/hello'}, function()
        return {body = 'Hello world!'}
    end)

    return true
end
```

Этот код экспортирует конечную точку `/hello` для выполнения HTTP-запросов. Например, для первого экземпляра из файла `instances.yml` это будет http://localhost:8081/hello . Если вы зайдете по этому адресу в браузере после того, как роль будет назначена (чуть позже мы покажем, как это делается), то увидите на странице слова «Hello world!».

Добавим сюда еще немного кода.

```
local function init(opts) -- luacheck: no unused args
    -- if opts.is_master then
    -- end

    local httpd = cartridge.service_get('httpd')
    httpd:route({method = 'GET', path = '/hello'}, function()
        return {body = 'Hello world!'}
    end)

    local log = require('log')
    log.info('Hello world!')

    return true
end
```

Здесь мы пишем «Hello, world!» в консоль в момент назначения роли, что даст вам возможность отследить данное событие. Пока ничего сложного.

Далее изменим значение параметра `role_name` в «return»-блоке файла `hello-world.lua`. Этот текст будет показан в качестве имени вашей роли в веб-интерфейсе для управления кластером.

```
return {
    role_name = 'Hello world!',
    init = init,
    stop = stop,
    validate_config = validate_config,
    apply_config = apply_config,
}
```

Последнее, что осталось сделать – это добавить ваше роль в список доступных ролей кластера, в файл `init.lua`.

```
local ok, err = cartridge.cfg({
    workdir = 'tmp/db',
    roles = {
        'cartridge.roles.vshard-storage',
        'cartridge.roles.vshard-router',
        'app.roles.hello-world'
    },
    cluster_cookie = 'myapp-cluster-cookie',
})
```

Теперь кластер будет знать про вашу роль.

Почему мы указали `app.roles.hello-world`? По умолчанию, имя роли в данном файле должно включать в себя полный путь от корня приложения (`./myapp`) до файла роли (`app/roles/hello-world.lua`).

Отлично! Роль готова. Теперь заново соберите и запустите ваше приложение:

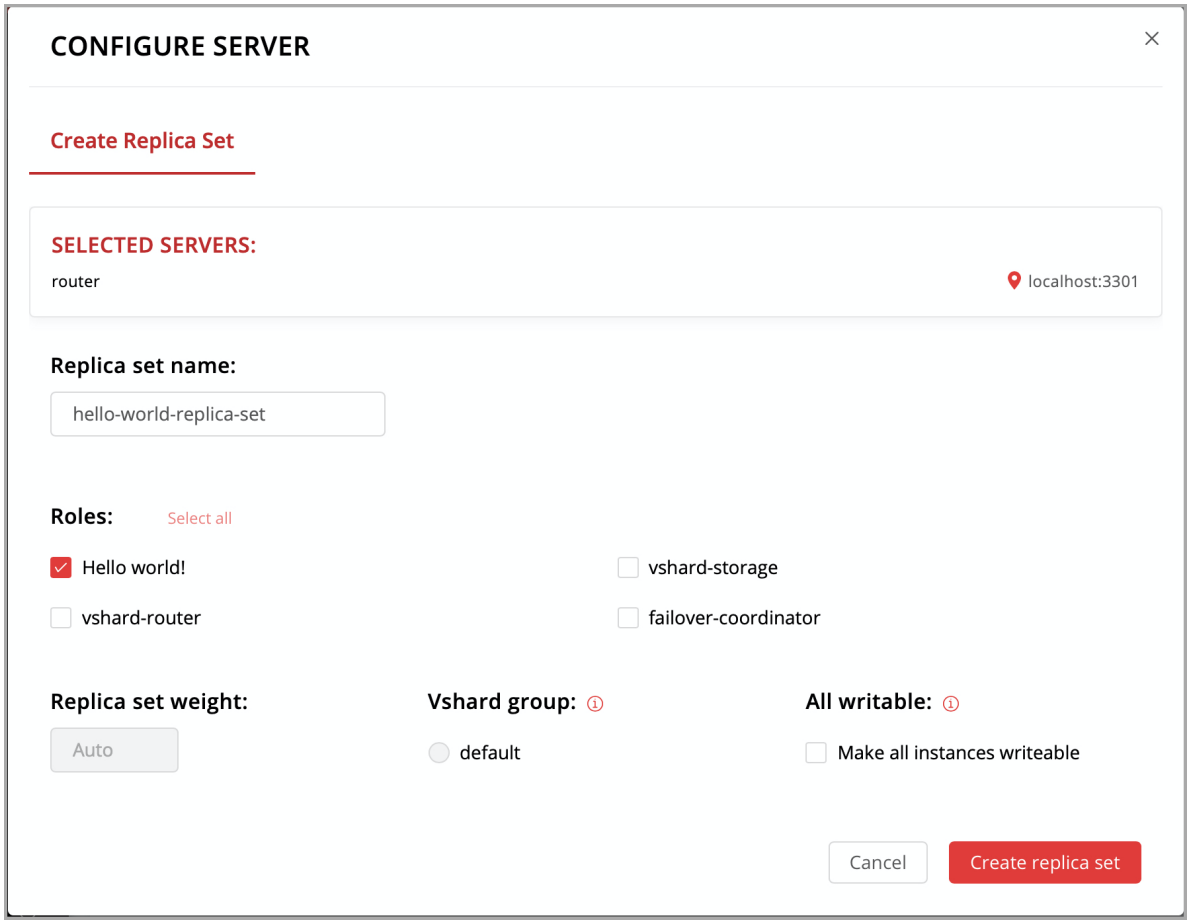
```
$ cartridge build
$ cartridge start
```

Все экземпляры запущены, но они пока ничего не делают, а ждут, что им назначат роли.

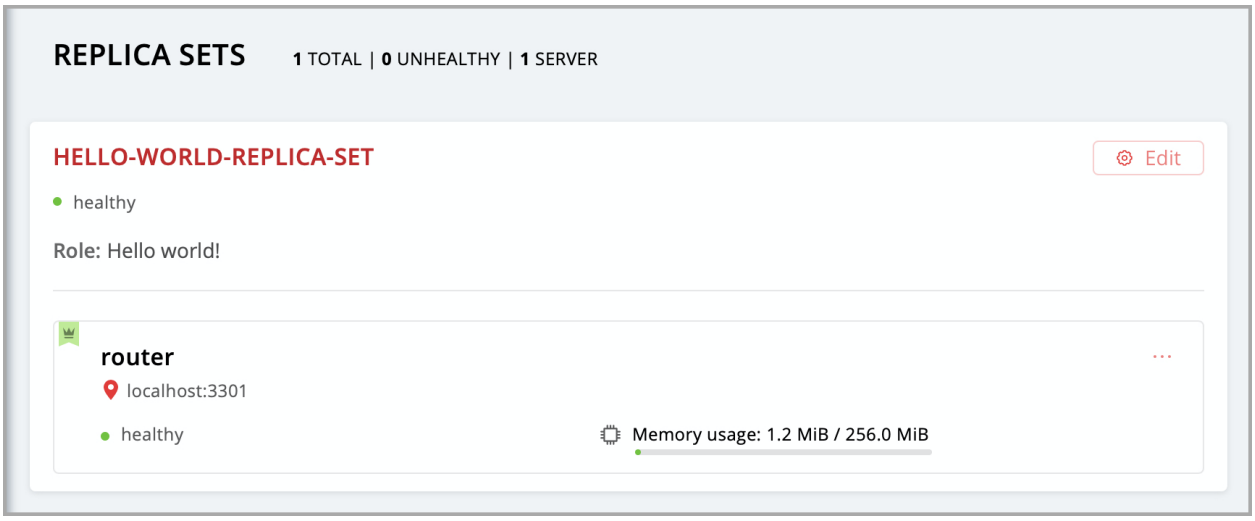
Экземпляры (реплики) в кластере Tarantool Cartridge должны быть собраны в *наборы реплик*. Роли назначаются каждому набору, и любой экземпляр в наборе реплик видит все роли, которые назначены этому набору.

Давайте создадим набор реплик, в котором будет всего один экземпляр, и назначим этому набору вашу роль.

1. Откройте веб-интерфейс для управления кластером по адресу http://localhost:8081.
2. Нажмите кнопку **Configure**.
3. Установите флажок напротив роли `Hello world!`, чтобы назначить ее. Заметьте, что имя роли здесь совпадает с тем текстом, который вы задали в параметре `role_name` в файле `hello-world.lua`.
4. (По желанию) Задайте имя набора реплик, например «hello-world-replica-set».



5. Нажмите кнопку **Create replica set**. Информация о вашем наборе реплик появится в веб-интерфейсе.



Итак, ваша роль назначена. В консоли вы можете увидеть такое сообщение:

```
router | 2020-04-17 18:10:17.254 [96390] main/123/remote_control/127.0.0.1:64591 I> set 'replication' configuration option to ["admin@localhost:3301"]
router | 2020-04-17 18:10:17.254 [96390] main/123/remote_control/127.0.0.1:64591 I> Instance state changed: ConnectingFullmesh -> BoxConfigured
router | 2020-04-17 18:10:17.254 [96390] main/123/remote_control/127.0.0.1:64591 I> Instance state changed: BoxConfigured -> ConfiguringRoles
router | 2020-04-17 18:10:17.255 [96390] main/123/remote_control/127.0.0.1:64591 I> Failover disabled
router | 2020-04-17 18:10:17.255 [96390] main/123/remote_control/127.0.0.1:64591 I> Hello world!
router | 2020-04-17 18:10:17.255 [96390] main/123/remote_control/127.0.0.1:64591 I> Roles configuration finished
router | 2020-04-17 18:10:17.255 [96390] main/123/remote_control/127.0.0.1:64591 I> Instance state changed: ConfiguringRoles -> RolesConfigured
router | 2020-04-17 18:10:17.255 [96390] main/3694/http/127.0.0.1:61751 twophase.lua:340 W> Committed config at localhost:3301
router | 2020-04-17 18:10:17.255 [96390] main/3694/http/127.0.0.1:61751 twophase.lua:377 W> Clusterwide config updated successfully
router | 2020-04-17 18:10:17.257 [96390] main/3925/applier/admin@localhost:3301 I> remote master 8d946f46-6e1d-45fb-97e3-69e0a85510ac at [::1]:3301 ru
nning Tarantool 2.3.1
```

А если вы сейчас откроете в браузере страницу `http://localhost:8081/hello` , то увидите ответ вашей роли на HTTP GET-запрос.

Все работает! Что же дальше?

- Загляните в *это руководство*, чтобы настроить оставшиеся наборы реплик и опробовать разные кластерные возможности.
- Посмотрите эти примеры приложений и реализуйте более сложную логику для вашей роли.
- Упакуйте ваше приложение для дальнейшего распространения. Вы можете выбрать любой из поддерживаемых видов пакетов: DEB, RPM, архив TGZ или Docker-образ.

ГЛАВА 4

Руководство пользователя

4.1 Предисловие

Добро пожаловать в мир Tarantool! Сейчас вы читаете «Руководство пользователя». Мы советуем начинать именно с него, а затем переходить к *«Справочникам»*, если вам понадобятся более подробные сведения.

4.1.1 Как пользоваться документацией

Для начала можно установить и запустить Tarantool, используя *Docker-контейнер*, *менеджер пакетов* или онлайн-сервер Tarantool'а http://try.tarantool.org. В любом случае для пробы можно сделать вводные упражнения из *главы 2 «Руководство для начинающих»*. Если хотите получить практический опыт, переходите к *Практическим заданиям* после работы с главой 2.

В *главе 3 «Функциональность СУБД»* рассказано о возможностях Tarantool'а как NoSQL СУБД, а в *главе 4 «Сервер приложений»* – о возможностях Tarantool'а как сервера приложений Lua.

Глава 5 «Администрирование серверной части» и *Глава 6 «Репликация»* предназначены в первую очередь для системных администраторов.

Глава 7 «Коннекторы» актуальна только для тех пользователей, которые хотят устанавливать соединение с Tarantool'ом с помощью программ на других языках программирования (например C, Perl или Python) – для прочих пользователей эта глава неактуальна.

Глава 8 «Вопросы и ответы» содержит ответы на некоторые часто задаваемые вопросы о Tarantool'е.

Опытным же пользователям будут полезны *«Справочники»*, *«Руководство участника проекта»* и комментарии в исходном коде.

4.1.2 Как связаться с сообществом разработчиков Tarantool'а

Оставить сообщение о найденных дефектах или сделать запрос на новые функции можно тут: http://github.com/tarantool/tarantool/issues

Пообщаться напрямую с командой разработки Tarantool'а можно в telegram или на форумах (англоязычном или русскоязычном).

4.1.3 Условные обозначения, используемые в руководстве

В квадратные скобки [и] включается синтаксис необязательных элементов.

Две точки подряд .. означают, что предыдущие токены могут повторяться.

Вертикальная черта | означает, что предыдущий и последующий токены представляют собой взаимоисключающие альтернативы.

4.2 Функциональность СУБД

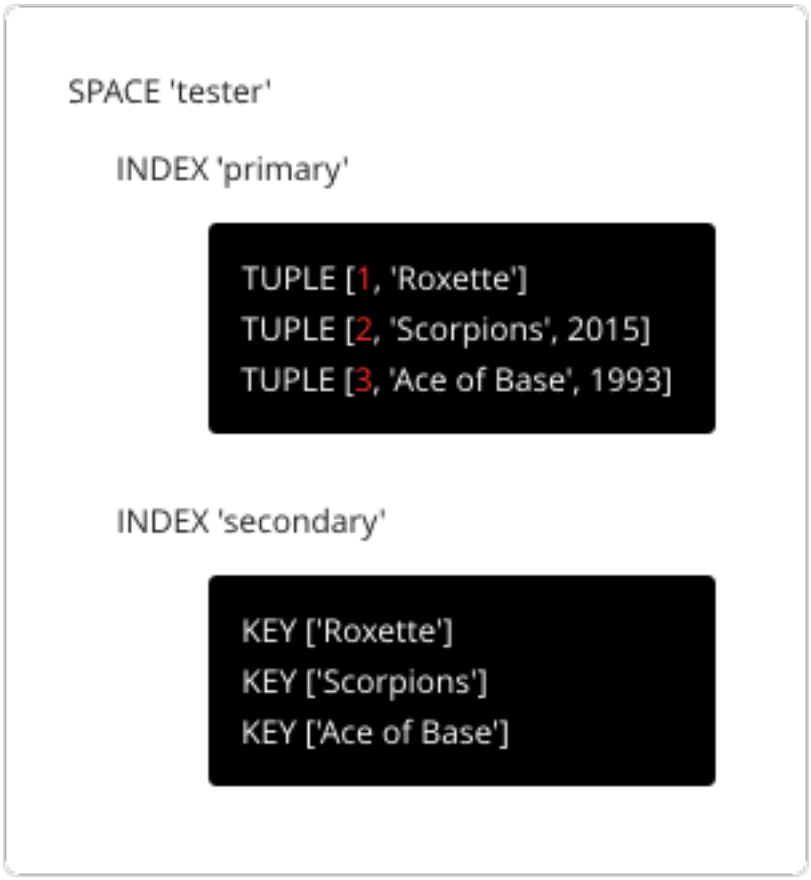
В данной главе мы рассмотрим основные понятия при работе с Tarantool'ом в качестве системы управления базой данных.

Эта глава состоит из следующих разделов:

4.2.1 Модель данных

В этом разделе описывается то, как в Tarantool'е организовано хранение данных и какие операции с данными он поддерживает.

Если вы пробовали создать базу данных, как предлагается в упражнениях в *«Руководстве для начинающих»*, то ваша тестовая база данных выглядит следующим образом:



Спейсы

Спейс – с именем „tester“ в нашем примере – это контейнер.

Когда Tarantool используется для хранения данных, всегда существует хотя бы один спейс. У каждого спейса есть уникальное **имя**, указанное пользователем. Кроме того, пользователь может указать уникальный **числовой идентификатор**, но обычно Tarantool назначает его автоматически. Наконец, в спейсе всегда есть **движок**: *memtx* (по умолчанию) – in-memory движок, быстрый, но ограниченный в размере, или *vinyl* – дисковый движок для огромного количества данных.

Спейс – это контейнер для *кортежей*. Для работы ему необходим первичный индекс. Также возможно использование вторичных индексов.

Кортежи

Кортеж играет такую же роль, как “строка” или “запись”, а компоненты кортежа (которые мы называем “полями”) играют такую же роль, что и “столбец” или “поле записи”, не считая того, что:

- поля могут представлять собой композитные структуры, такие как таблицы типа массива или ассоциативного массива, а также
- полям не нужны имена.

В любом кортеже может быть любое количество полей, и это могут быть поля разных *типов*. Идентификатором поля является его номер, начиная с 1 (в Lua и других языках с индексацией с 1) или с 0 (в PHP или C/C++). Например, `1` или `0` могут использоваться в некоторых контекстах для обозначения первого поля кортежа.

Количество кортежей в спейсе не ограничено.

Кортежи в Tarantool'е хранятся в виде массивов MsgPack.

Когда Tarantool выводит значение в кортеже в консоль, используется формат *YAML*, например: `[3, 'Ace of Base', 1993]`.

Индексы

Индекс – это совокупность значений ключей и указателей.

Как и для спейсов, индексам следует указать **имена**, а Tarantool определит уникальный **числовой идентификатор** («ID индекса»).

У индекса всегда есть определенный **тип**. Тип индекса по умолчанию – „TREE“. Все движки Tarantool'а предоставляют TREE-индексы, которые могут индексировать уникальные и неуникальные значения, поддерживают поиск по компонентам ключа, сравнение ключей и упорядоченные результаты. Кроме того, движок memtx поддерживает следующие индексы: HASH, RTREE и BITSET.

Индекс может быть **многокомпонентным**, то есть можно объявить, что ключ индекса состоит из двух или более полей в кортеже в любом порядке. Например, для обычного TREE-индекса максимальное количество частей равно 255.

Индекс может быть **уникальным**, то есть можно объявить, что недопустимо дважды задавать одно значение ключа.

Первый индекс, определенный для спейса, называется **первичный индекс**. Он должен быть уникальным. Все остальные индексы называются **вторичными индексами**, они могут строиться по неуникальным значениям.

Индекс может содержать идентификаторы полей кортежа и их предполагаемые **типы** (см. допустимые *типы индексированных полей*).

Примечание: Рекомендуется проектировать модель данных так, чтобы первичные ключи были первыми полями в котреже, чтобы их было быстрее сравнивать.

В нашем примере для начала определяем первичный индекс (под названием „primary“) по полю №1 каждого кортежа:

```
tarantool> i = s:create_index('primary', {type = 'hash', parts = {{field = 1, type = 'unsigned'}}}
```

Смысл в том, что поле №1 должно существовать и содержать целое число без знака для всех кортежей в спейсе „tester“. Тип индекса – „hash“, поэтому значения в поле №1 должны быть уникальными, поскольку ключи в HASH-индексах уникальны.

После этого мы определим вторичный индекс (под названием „secondary“) по полю №2 каждого кортежа:

```
tarantool> i = s:create_index('secondary', {type = 'tree', parts = {field = 2, type = 'string'}})
```

Смысл в том, что поле №2 должно существовать и содержать строку для всех кортежей в спейсе „tester“. Тип индекса – „tree“, поэтому значения в поле №2 не должны быть уникальными, поскольку ключи в TREE-индексах могут не быть уникальными.

Примечание: Определения спейса и определения индексов хранятся в системных спейсах Tarantool'а *_space* и *_index* соответственно (для получения подробной информации см. справочник по вложенному модулю *box.space*).

Можно добавлять, опускать или изменять определения во время исполнения кода с некоторыми ограничениями. Более подробно о синтаксисе см. в справочнике по модулю *box*.

Подробнее об операциях с индексом читайте *здесь*.

Типы данных

Tarantool представляет собой базу данных и сервер приложений одновременно. Следовательно, разработчик часто работает с двумя наборами типов: типы языка программирования (например, Lua) и типы формата хранилища Tarantool (MsgPack).

Lua в сравнении с MsgPack

Скалярный / составной	MsgPack-тип	Lua-тип	Пример значения
скалярный	nil	«nil» (нулевое значение)	msgpack.NULL
скалярный	boolean (логический)	«boolean» (логическое значение)	true
скалярный	string (строка)	«string» (строка)	„A B C“
скалярный	integer (целое число)	«number» (число)	12345
скалярный	double (числа с двойной точностью)	«number» (число)	1,2345
скалярный	bin	«cdata»	[!!binary 3t7e]
составной	map (ассоциативный массив)	«table» (таблица со строковыми ключами)	{„a“: 5, „b“: 6}
составной	array (массив)	«table» (таблица с целочисленными ключами)	[1, 2, 3, 4, 5]
составной	array (массив)	tuple («cdata») (кортеж)	[12345, „A B C“]

В языке Lua тип *nil* (нулевой) может иметь только одно значение, также называемое *nil* (отображаемое как **null** в командной строке Tarantool'а, поскольку значения выводятся в формате YAML). Нулевое значение можно сравнивать со значениями любых типов с помощью операторов == (равен) или ~= (не равен), но никакие другие операции для нулевых значений не доступны. Нулевые значения также нельзя использовать в Lua-таблицах; вместо нулевого значения в таком случае можно указать *msgpack.NULL*

Тип *boolean* (логический) может иметь только значения `true` или `false`.

Тип **string** (строка) представляет собой последовательность байтов переменной длины, обычно представленную буквенно-цифровые символы в одинарных кавычках. Как в Lua, так и в MsgPack строки рассматриваются как бинарные данные без попыток определить набор символов строки или выполнить преобразование строки – кроме случаев, когда есть опциональное *сравнение символов*. Таким образом, обычно сортировка и сравнение строк выполняются побайтово, не применяя дополнительных правил сравнения символов. (Пример: числа упорядочены по их положению на числовой прямой, поэтому 2345 больше, чем 500; а строки упорядочены по кодировке первого байта, затем кодировке второго байта и так далее, таким образом, „2345“ меньше, чем „500“.)

В языке Lua тип **number** (число) – это число с плавающей запятой двойной точности, но в Tarantool'е можно использовать как целые числа, так и числа с плавающей запятой. Tarantool по возможности сохраняет числа языка Lua в виде чисел с плавающей запятой, если числовое значение содержит десятичную запятую или если оно очень велико (более 100 триллионов = 1e14). В противном случае,

Tarantool сохраняет такое значение в виде целого числа. Чтобы даже очень большие величины гарантированно обрабатывались как целые числа, используйте функцию *tonumber64*, либо приписывайте в конце суффикс LL (Long Long) или ULL (Unsigned Long Long). Вот примеры записи чисел в обычном представлении, экспоненциальном, с суффиксом ULL и с использованием функции `tonumber64`: `-55`, `-2.7e+20`, `100000000000000ULL`, `tonumber64('18446744073709551615')`.

A **bin** (binary) value is not directly supported by Lua but there is a Tarantool type `VARBINARY` which is encoded as MessagePack binary. For an (advanced) example showing how to insert VARBINARY into a database, see the Cookbook Recipe for *ffi_varbinary_insert*.

В Lua **tables** (таблицы) со строковыми ключами хранятся как ассоциативные массивы в MsgPack; Lua-таблицы с целочисленными ключами, начиная с 1, хранятся как массивы в MsgPack. Нулевые значения нельзя использовать в Lua-таблицах; вместо нулевого значения в таком случае можно указать *msgpack.NULL*

Тип **tuple** (кортеж) представляет собой легкую ссылку на массив MsgPack, который хранится в базе данных. Это особый тип (cdata), чтобы избежать конвертации в Lua-таблицу при выборке данных. Некоторые функции могут возвращать таблицы с множеством кортежей. Примеры с кортежами см. в *box.tuple*.

Примечание: Tarantool использует формат MsgPack для хранения в базе данных переменной длины. Поэтому, например, для наименьшего числа требуется только один байт, но для наибольшего числа требуется девять байтов.

Примеры запроса вставки с разными типами данных:

```
tarantool> box.space.K:insert{1,nil,true,'A B C',12345,1.2345}
---
- [1, null, true, 'A B C', 12345, 1.2345]
...
tarantool> box.space.K:insert{2,{['a']=5,['b']=6}}
---
- [2, {'a': 5, 'b': 6}]
...
tarantool> box.space.K:insert{3,{1,2,3,4,5}}
---
- [3, [1, 2, 3, 4, 5]]
...
```

Типы индексированных полей

Индексы ограничивают значения, которые может содержать MsgPack в Tarantool'е. Вот почему, например, тип „unsigned“ (без знака) представляет собой отдельный **тип индексированного поля** в сравнении с типом данных 'integer' (целое число) в MsgPack: оба содержат значения с целыми числами, но индекс „unsigned“ содержит только *неотрицательные* целые числовые значения, а индекс 'integer' содержит *все* целые числовые значения.

Вот как типы индексированных полей в Tarantool'е соответствуют типам данных MsgPack.

Тип индексированного поля	Тип данных MsgPack (и возможные значения)	Тип индекса	Примеры
unsigned (без знака – может также называться 'uint' или 'num', но 'num' объявлен устаревшим)	**integer** (целое число в диапазоне от 0 до 18 446 744 073 709 551 615, т.е. около 18 квинтиллионов)	TREE, BITSET или HASH	123456
integer (целое число – может также называться 'int')	`integer` (целое число в диапазоне от -9 223 372 036 854 775 808 до 18 446 744 073 709 551 615)	TREE или HASH	-2^63
number	`integer` (целое число в диапазоне от -9 223 372 036 854 775 808 до 18 446 744 073 709 551 615) **double** (число с плавающей запятой с одинарной точностью или с двойной точностью)	TREE или HASH	1,234 -44 1,447e+44
string (строка – может также называться 'str')	**string** (строка – любая последовательность октетов до максимальной длины)	TREE, BITSET или HASH	'A B C' '\65 \66 \67'
varbinary	**bin** (любая последовательность октетов до максимальной длины)	TREE или HASH	'\65 \66 \67'
boolean	**bool** (логический – true или false)	TREE или HASH	true
array	**array** (массив – список чисел, который представляет собой точки в геометрической фигуре)	RTREE	{10, 11} {3, 5, 9, 10}
scalar	**null** **bool** (логический – true или false) `integer` (целое число в диапазоне от -9 223 372 036 854 775 808 до 18 446 744 073 709 551 615) **double** (число с плавающей запятой с одинарной точностью или с двойной точностью) **string** (строковое значение, т.е. любая последовательность октетов) **varbinary** (любая последовательность октетов) Примечание: в сочетании различных типов порядок будет следующим: null, затем булевские значения, затем числовые, затем строковые, затем varbinary.	TREE или HASH	msgpack.NULL true -1 1,234 '' 'py'

Сортировка

По умолчанию, когда Tarantool сравнивает строки, он использует то, что мы называем **«бинарной» сортировкой**. Единственный фактор, который учитывается, это числовое значение каждого байта в строке. Таким образом, если строка кодируется по ASCII или UTF-8, то `'A' < 'B' < 'a'`, поскольку в кодировке „A“ (что раньше называлось «значение ASCII») соответствует 65, „B“ – 66, а „a“ – 98.

Бинарная сортировка подходит лучше всего для быстрого детерминированного простого обслуживания и поиска с помощью индексов Tarantool'a.

Однако если необходимо распределение, как в телефонных справочниках и словарях, то вам нужна **опциональная сортировка** Tarantool'a – `unicode` и `unicode_ci` – которые обеспечивают `'a' < 'A' < 'B'` и `'a' = 'A' < 'B'` соответственно.

Опциональная сортировка unicode и unicode_ci использует распределение в соответствии с Таблицей сортировки символов Юникода по умолчанию (DUCET) и правилами, указанными в Техническом стандарте Юникода №10 – Алгоритм сортировки по Юникоду (Unicode® Technical Standard #10 Unicode Collation Algorithm (UTS #10 UCA)). Единственное отличие между двумя сортировками – вес:

- сортировка `unicode` принимает во внимание уровни веса L1, L2 и L3 (уровень = „tertiary“, третичный),
- сортировка `unicode_ci` принимает во внимание только вес L1 (уровень = „primary“, первичный), поэтому, например, „a“ = „A“ = „á“ = „Á“.

Для примера возьмем некоторые русские слова:

```
'ЕЛЕ'
'елейный'
'ёлка'
'еловый'
'елозить'
'Ёлочка'
'ёлочный'
'ЕЛь'
'ель'
```

. . . и покажем разницу в упорядочении и выборке по индексу:

- с сортировкой по `unicode`:

```
tarantool> box.space.T:create_index('I', {parts = {{field = 1, type = 'str', collation=
↪'unicode'}}})
...
tarantool> box.space.T.index.I:select()
---
- - ['ЕЛЕ']
  - ['елейный']
  - ['ёлка']
  - ['еловый']
  - ['елозить']
  - ['Ёлочка']
  - ['ёлочный']
  - ['ель']
  - ['ЕЛь']
...
tarantool> box.space.T.index.I:select{'ЁлКа'}
---
- []
...
```

- с сортировкой по `unicode_ci`:

```
tarantool> box.space.T:create_index('I', {parts = {{field = 1, type ='str', collation=
↪'unicode_ci'}}})
...
```

(continues on next page)

(продолжение с предыдущей страницы)

```
tarantool> box.space.S.index.I:select()
---
- - ['ЕЛЕ']
  - ['елейный']
  - ['ёлка']
  - ['еловый']
  - ['елозить']
  - ['Ёлочка']
  - ['ёлочный']
  - ['ЕЛь']
...
tarantool> box.space.S.index.I:select{'ЁлКа'}
---
- - ['ёлка']
...
```

In all, collation involves much more than these simple examples of upper case / lower case and accented / unaccented equivalence in alphabets. We also consider variations of the same character, non-alphabetic writing systems, and special rules that apply for combinations of characters.

For English: use «unicode» and «unicode_ci». For Russian: use «unicode» and «unicode_ci» (although a few Russians might prefer the Kyrgyz collation which says Cyrillic letters „Е“ and „Ё“ are the same with level-1 weights). For Dutch, German (dictionary), French, Indonesian, Irish, Italian, Lingala, Malay, Portuguese, Southern Soho, Xhosa, or Zulu: «unicode» and «unicode_ci» will do.

The tailored optional collations: For other languages, Tarantool supplies tailored collations for every modern language that has more than a million native speakers, and for specialized situations such as the difference between dictionary order and telephone book order. To see a complete list say `box.space._collation:select()`. The tailored collation names have the form unicode_[language code]_[strength] where language code is a standard 2-character or 3-character language abbreviation, and strength is s1 for «primary strength» (level-1 weights), s2 for «secondary», s3 for «tertiary». Tarantool uses the same language codes as the ones in the «list of tailorable locales» on man pages of Ubuntu and Fedora. Charts explaining the precise differences from DUCET order are in the Common Language Data Repository.

Последовательности

Последовательность – это генератор упорядоченных значений целых чисел.

Как и для спейсов и индексов, для последовательностей следует указать **имена**, а Tarantool определит уникальный **числовой идентификатор** («ID последовательности»).

Кроме того, можно указать несколько параметров при создании новой последовательности. Параметры определяют, какое значение будет генерироваться при использовании последовательности.

Параметры для `box.schema.sequence.create()`

Имя параметра	Тип и значение	Значение по умолчанию	Примеры
start (начало)	Целое число. Значение генерируется, когда последовательность используется впервые	1	start=0
min (мин)	Целое число. Ниже указанного значения не могут генерироваться	1	min=-1000
max (макс)	Целое число. Выше указанного значения не могут генерироваться	9 223 372 036 854 775 807	max=0
cycle (цикл)	Логическое значение. Если значения не могут быть сгенерированы, начинать ли заново	false (ложь)	cycle=true
cache (кэш)	Целое число. Количество значений для хранения в кэше	0	cache=0
step (шаг)	Целое число. Что добавить к предыдущему сгенерированному значению, когда генерируется новое значение	1	step=-1
if_not_exists (если отсутствует)	Логическое значение. Если выставлено в true (истина) и существует последовательность с таким именем, то игнорировать другие опции и использовать текущие значения	false (ложь)	if_not_exists=true

Существующую последовательность можно изменять, опускать, сбрасывать, заставить сгенерировать новое значение или ассоциировать с индексом.

Для первоначального примера сгенерируем последовательность под названием „S“.

```
tarantool> box.schema.sequence.create('S',{min=5, start=5})
---
- step: 1
  id: 5
  min: 5
  cache: 0
  uid: 1
  max: 9223372036854775807
  cycle: false
  name: S
  start: 5
...
```

В результате видим, что в новой последовательность есть все значения по умолчанию, за исключением указанных `min` и `start`.

Затем получаем следующее значение с помощью функции `next()`.

```
tarantool> box.sequence.S:next()
---
- 5
...
```

Результат точно такой же, как и начальное значение. Если мы снова вызовем `next()`, то получим 6 (потому что предыдущее значение плюс значение шага составит 6) и так далее.

Затем создадим новую таблицу и скажем, что ее первичный ключ можно получить из последовательности.

```
tarantool> s=box.schema.space.create('T');s:create_index('I',{sequence='S'})
---
...
```

Затем вставим кортеж, не указывая значение первичного ключа.

```
tarantool> box.space.T:insert{nil,'other stuff'}
---
- [6, 'other stuff']
...
```

В результате имеем новый кортеж со значением 6 в первом поле. Такой способ организации данных, когда система автоматически генерирует значения для первичного ключа, иногда называется «автоинкрементным» (т.е. с автоматическим увеличением) или «по идентификатору».

Для получения подробной информации о синтаксисе и методах реализации см. справочник по *box.schema.sequence*.

Персистентность

В Tarantool'е обновления базы данных записываются в так называемые *файлы журнала упреждающей записи (WAL-файлы)*. Это обеспечивает персистентность данных. При отключении электроэнергии или случайном завершении работы экземпляра Tarantool'а данные в оперативной памяти теряются. В такой ситуации WAL-файлы используются для восстановления данных так: Tarantool прочитывает WAL-файлы и повторно выполняет запросы (это называется «процессом восстановления»). Можно изменить временные настройки метода записи WAL-файлов или отключить его с помощью *wal_mode*.

Tarantool также сохраняет ряд файлов со статическими снимками данных (*snapshots*). Файл со снимком – это дисковая копия всех данных в базе на какой-то момент. Вместо того, чтобы зачитывать все WAL-файлы, появившиеся с момента создания базы, Tarantool в процессе восстановления может загрузить самый свежий снимок и затем зачитать только те WAL-файлы, которые были сделаны с момента сохранения снимка. После создания новых файлов, старые WAL-файлы могут быть удалены в целях экономии места на диске.

Чтобы принудительно создать файл со снимком, можно использовать запрос *box.snapshot()* в Tarantool'е. Чтобы включить автоматическое создание файлов со снимком, можно использовать *демон создания контрольных точек* Tarantool'а. Демон создания контрольных точек определяет интервалы для принудительного создания контрольных точек. Он обеспечивает синхронизацию и сохранение на диск образов движков базы данных (как memtx, так и vinyl), а также автоматически удаляет старые WAL-файлы.

Файлы со снимками можно создавать, даже если WAL-файлы отсутствуют.

Примечание: Движок memtx регулярно создает контрольные точки с интервалом, указанным в настройках *демона создания контрольных точек*.

Движок vinyl постоянно сохраняет состояние в контрольной точке в фоновом режиме.

Для получения более подробной информации о методе записи WAL-файлов и процессе восстановления см. раздел *Внутренняя реализация*.

Операции

Операции с данными

Tarantool поддерживает следующие основные операции с данными:

- пять операций по изменению данных (INSERT, UPDATE, UPSERT, DELETE, REPLACE) и
- одну операцию по выборке данных (SELECT).

Все они реализованы в виде функций во вложенном модуле *box.space*.

Примеры:

- *INSERT*: добавить новый кортеж к спейсу „tester“.

  Первое поле, field[1], будет 999 (тип MsgPack – *integer*, целое число).

  Второе поле, field[2], будет „Taranto“ (тип MsgPack – *string*, строка).

```
tarantool> box.space.tester:insert{999, 'Taranto'}
```

- *UPDATE*: обновить кортеж, изменяя поле field[2].

  Оператор «{999}» со значением, которое используется для поиска поля, соответствующего ключу в первичном индексе, является обязательным, поскольку в запросе `update()` должен быть оператор, который указывает уникальный ключ, в данном случае – field[1].

  Оператор «{{„=“, 2, „Tarantino“}}» указывает, что назначение нового значения относится к field[2].

```
tarantool> box.space.tester:update({999}, {{'=', 2, 'Tarantino'}})
```

- *UPSERT*: обновить или вставить кортеж, снова изменяя поле field[2].

  Синтаксис `upsert()` похож на синтаксис `update()`. Однако логика выполнения двух запросов отличается. UPSERT означает UPDATE или INSERT, в зависимости от состояния базы данных. Кроме того, выполнение UPSERT откладывается до коммита транзакции, поэтому в отличие от“update()“, `upsert()` не возвращает данные.

```
tarantool> box.space.tester:upsert({999, 'Taranted'}, {{'=', 2, 'Tarantism'}})
```

- *REPLACE*: заменить кортеж, добавляя новое поле.

  Это действие также можно выполнить с помощью запроса `update()`, но обычно запрос `update()` более сложен.

```
tarantool> box.space.tester:replace{999, 'Tarantella', 'Tarantula'}
```

- *SELECT*: провести выборку кортежа.

  Оператор «{999}» все еще обязателен, хотя в нем не должен упоминаться первичный ключ.

```
tarantool> box.space.tester:select{999}
```

- *DELETE*: удалить кортеж.

  В этом примере мы определяем поле, соответствующее ключу в первичном индексе.

```
tarantool> box.space.tester:delete{999}
```

Подводя итоги по примерам:

- Функции `insert` и `replace` принимают кортеж (где первичный ключ – это часть кортежа).
- Функция `upsert` принимает кортеж (где первичный ключ – это часть кортежа), а также операции по обновлению.
- Функция `delete` принимает полный ключ любого уникального индекса (первичный или вторичный).
- Функция `update` принимает полный ключ любого уникального индекса (первичный или вторичный), а также операции к выполнению.
- Функция `select` принимает любой ключ: первичный/вторичный, уникальный/неуникальный, полный/часть.

Для получения более *подробной информации по использованию операций с данными* см. справочник по `box.space`.

Примечание: Помимо Lua можно использовать *коннекторы к Perl, PHP, Python или другому языку программирования*. Клиент-серверный протокол открыт и задокументирован. См. *БНФ с комментариями*.

Операции с индексами

Операции с индексами производятся автоматически. Если запрос по манипулированию данными меняет данные в кортеже, то меняются и ключи в индексе для данного кортежа.

Простая операция по созданию индекса, которую мы рассматривали ранее, выглядит следующим образом:

`box.space.`*`space-name`*`:create_index('`*`index-name`*`')`

По умолчанию, при этом создается TREE-индекс по первому полю для всех кортежей (обычно его называют «Field#1»). Предполагается, что индексируемое поле является числовым.

Вот простой SELECT-запрос, который мы рассматривали ранее:

`box.space.`*`space-name`*`:select(`*`value`*`)`

Такой запрос ищет отдельный кортеж по первичному индексу. Поскольку первичный индекс всегда уникален, то данный запрос вернет не более одного кортежа. Можно также вызвать `select()` без аргументов, чтобы вернуть все кортежи.

Продолжим работу со спейсом „tester“, созданным в упражнениях из *«Руководства для начинающих»*, но сначала его немного модифицируем:

```
tarantool> box.space.tester:format({
         > {name = 'id', type = 'unsigned'},
         > {name = 'band_name', type = 'string'},
         > {name = 'year', type = 'unsigned'},
         > {name = 'rate', type = 'unsigned', is_nullable=true}})
---
...
```

Добавим рейтинг „rate“ кортежам #1 и #2:

```
tarantool> box.space.tester:update(1, {{'=', 4, 5}})
---
- [1, 'Roxette', 1986, 5]
```

(continues on next page)

(продолжение с предыдущей страницы)

```
...
tarantool> box.space.tester:update(2, {{'=', 4, 4}})
---
- [2, 'Scorpions', 2015, 4]
...
```

И создадим еще один кортеж:

```
tarantool> box.space.tester:insert({4, 'Roxette', 2016, 3})
---
- [4, 'Roxette', 2016, 3]
...
```

Существующие вариации SELECT:

1. Помимо условия равенства, при поиске могут использоваться и другие условия сравнения.

```
tarantool> box.space.tester:select(1, {iterator = 'GT'})
---
- - [2, 'Scorpions', 2015, 4]
  - [3, 'Ace of Base', 1993]
  - [4, 'Roxette', 2016, 3]
...
```

Можно использовать следующие *операторы сравнения*: LT (меньше), LE (меньше или равно), EQ (равно, результаты отсортированы в порядке возрастания по ключу), REQ (равно, результаты отсортированы в порядке убывания по ключу), GE (больше или равно), GT (больше). Сравнения имеют смысл только для индексов типа „TREE“.

Этот вариант поиска может вернуть более одного кортежа. В таком случае кортежи будут отсортированы в порядке убывания по ключу (если использовался оператор LT, LE или REQ), либо в порядке возрастания (во всех остальных случаях).

2. Поиск может производиться по вторичному индексу.

При поиске по первичному индексу имя индекса можно не указывать. При поиске же по вторичному индексу имя индекса указывать необходимо.

```
tarantool> box.space.tester:create_index('secondary', {parts = {{field=3, type='unsigned'}}})
---
- unique: true
  parts:
  - type: unsigned
    is_nullable: false
    fieldno: 3
  id: 2
  space_id: 512
  type: TREE
  name: secondary
...
tarantool> box.space.tester.index.secondary:select({1993})
---
- - [3, 'Ace of Base', 1993]
...
```

3. Поиск можно осуществить по некоторым частям ключа, используя его префикс. Обратите внимание, что частичный поиск по ключу доступен только в TREE индексах.

```
-- Создаем индекс, состоящий из трех частей
tarantool> box.space.tester:create_index('tertiary', {parts = {{field = 2, type = 'string'},
↪{field=3, type='unsigned'}, {field=4, type='unsigned'}}})
---
- unique: true
  parts:
  - type: string
    is_nullable: false
    fieldno: 2
  - type: unsigned
    is_nullable: false
    fieldno: 3
  - type: unsigned
    is_nullable: true
    fieldno: 4
  id: 6
  space_id: 513
  type: TREE
  name: tertiary
...
-- Выполняем частичный поиск
tarantool> box.space.tester.index.tertiary:select({'Scorpions', 2015})
---
- - [2, 'Scorpions', 2015, 4]
...
```

4. Поиск может производиться по всем полям через запись в виде таблицы:

```
tarantool> box.space.tester.index.tertiary:select({'Roxette', 2016, 3})
---
- - [4, 'Roxette', 2016, 3]
...
```

либо же по одному полю (в этом случае используется таблица или скалярное значение):

```
tarantool> box.space.tester.index.tertiary:select({'Roxette'})
---
- - [1, 'Roxette', 1986, 5]
  - [4, 'Roxette', 2016, 3]
...
```

Работа с BITSET и RTREE

Примеры BITSET:

```
tarantool> box.schema.space.create('bitset_example')
tarantool> box.space.bitset_example:create_index('primary')
tarantool> box.space.bitset_example:create_index('bitset',{unique=false,type='BITSET', parts={2,
↪'unsigned'}})
tarantool> box.space.bitset_example:insert{1,1}
tarantool> box.space.bitset_example:insert{2,4}
tarantool> box.space.bitset_example:insert{3,7}
tarantool> box.space.bitset_example:insert{4,3}
tarantool> box.space.bitset_example.index.bitset:select(2, {iterator='BITS_ANY_SET'})
```

Мы получим следующий результат:

```
---
- - [3, 7]
  - [4, 3]
...
```

поскольку (7 AND 2) не равно 0 и (3 AND 2) не равно 0.

Примеры RTREE:

```
tarantool> box.schema.space.create('rtree_example')
tarantool> box.space.rtree_example:create_index('primary')
tarantool> box.space.rtree_example:create_index('rtree',{unique=false,type='RTREE', parts={2,'ARRAY
↪'}})
tarantool> box.space.rtree_example:insert{1, {3, 5, 9, 10}}
tarantool> box.space.rtree_example:insert{2, {10, 11}}
tarantool> box.space.rtree_example.index.rtree:select({4, 7, 5, 9}, {iterator = 'GT'})
```

Мы получим следующий результат:

```
---
- - [1, [3, 5, 9, 10]]
...
```

поскольку прямоугольник с углами в координатах `4,7,5,9` лежит целиком внутри прямоугольника с углами в координатах `3,5,9,10`.

Кроме того, есть *операции с итераторами с индексом*. Их можно использовать только с кодом на языках Lua и C/C++. Итераторы с индексом предназначены для обхода индексов по одному ключу за раз, поскольку используют особенности каждого типа индекса, например оценка логических выражений при обходе BITSET-индексов или обход TREE-индексов в порядке по убыванию.

Полный список операций над индексами, таких как *alter()* (изменение индекса) и *drop()* (удаление индекса), приводится во вложенном модуле *box.index*.

Факторы сложности

Во вложенных модулях *box.space* и *box.index* содержится информация о том, как факторы сложности могут повлиять на использование каждой функции.

Фактор сложности	Эффект
Размер индекса	Количество ключей в индексе равно количеству кортежей в наборе данных. В случае с TREE-индексом: с ростом количества ключей увеличивается время поиска, хотя зависимость здесь, конечно же, не линейная. В случае с HASH-индексом: с ростом количества ключей увеличивается объем оперативной памяти, но количество низкоуровневых шагов остается примерно тем же.
Тип индекса	Как правило, поиск по HASH-индексу работает быстрее, чем по TREE-индексу, если в спейсе более одного кортежа.
Количество обращений к индексам	Обычно для выборки значений одного кортежа используется только один индекс. Но при обновлении значений в кортеже требуется N обращений, если в спейсе N индексов. Примечание по движку базы данных: Vinyl отклоняет такой доступ, если обновление не затрагивает поля вторичного индекса. Таким образом, этот фактор сложности влияет только на memtx, поскольку он всегда создает копию всего кортежа при каждом обновлении.
Количество обращений к кортежам	Некоторые запросы, например SELECT, могут возвращать несколько кортежей. Как правило, это наименее важный фактор из всех.
Настройки WAL	Важным параметром для записи в WAL является *wal_mode*. Если запись в WAL отключена или задана запись с задержкой, но этот фактор не так важен. Если же запись в WAL производится при каждом запросе на изменение данных, то при каждом таком запросе приходится ждать, пока отработает обращение к более медленному диску, и данный фактор становится важнее всех остальных.

4.2.2 Транзакции

Транзакции в Tarantool'е происходят в **файберах** в одном **потоке**. Вот почему Tarantool дает гарантию атомарности выполнения. На этом следует сделать акцент.

Потоки, файберы и передача управления

Как Tarantool выполняет основные операции? Для примера возьмем такой запрос:

```
tarantool> box.space.tester:update({3}, {{'=', 2, 'size'}, {'=', 3, 0}})
```

Это эквивалентно следующему SQL-выражению (оно работает с таблицей, где первичные ключи в `field[1]`):

```
UPDATE tester SET "field[2]" = 'size', "field[3]" = 0 WHERE "field[1]" = 3
```

Assuming this query is received by Tarantool via network, it will be processed with three operating system **threads**:

1. The **network thread** on the server side receives the query, parses the statement, checks if it's correct, and then transforms it into a special structure–a message containing an executable statement and its options.
2. Сетевой поток отправляет это сообщение в **поток обработки транзакций** с помощью шины передачи сообщений без блокировок. Lua-программы выполняются непосредственно в потоке обработки транзакций и не требуют разбора и подготовки.

   The instance's transaction processor thread uses the primary-key index on field[1] to find the location of the tuple. It determines that the tuple can be updated (not much can go wrong when you're merely

changing an unindexed field value).

3. The transaction processor thread sends a message to the *write-ahead logging (WAL) thread* to commit the transaction. When done, the WAL thread replies with a COMMIT or ROLLBACK result to the transaction processor which gives it back to the network thread, and the network thread returns the result to the client.

Обратите внимание, что в Tarantool'е есть только один поток обработки транзакций. Некоторые уже привыкли к мысли, что потоков для обработки данных в базе данных может быть много (например, поток №1 читает данные из строки №x, в то время как поток №2 записывает данные в столбец №y). В случае с Tarantool'ом такого не происходит. Доступ к базе есть только у потока обработки транзакций, и на каждый экземпляр Tarantool'а есть только один такой поток.

Как и любой другой поток Tarantool'а, поток обработки транзакций может управлять множеством *файберов*. Файбер – это набор команд, среди которых могут быть и сигналы «**передачи управления**». Поток обработки транзакций выполняет все команды, пока не увидит такой сигнал, и тогда он переключается на выполнение команд из другого файбера. Например, таким образом поток обработки транзакций сначала выполняет чтение данных из строки №x для файбера №1, а затем выполняет запись в строку №y для файбера №2.

Передача управления необходима, в противном случае, поток обработки транзакции заклинит на одном файбере. Есть два типа передачи управления:

- *неявная передача управления*: каждая операция по изменению данных или доступ к сети вызывают неявную передачу управления, а также каждое выражение, которое проходит через клиент Tarantool'а, вызывает неявную передачу управления.
- явная передача управления: в Lua-функции можно (и нужно) добавить выражения *«передачи управления»* для предотвращения захвата ЦП. Это называется **кооперативной многозадачностью**.

Кооперативная многозадачность

Кооперативная многозадачность означает, что если запущенный файбер намеренно не передаст управление, он не вытесняется каким-либо другим файбером. Но запущенный файбер намеренно передает управление, когда обнаруживает "точку передачи управления": коммит транзакции, вызов операционной системы или запрос явной *«передачи управления»*. Любой вызов системы, который может блокировать файбер, будет производиться асинхронно, а запущенный файбер, который должен ожидать системного вызова, будет вытеснен так, что другой готовый к работе файбер занимает его место и становится запущенным файбером.

This model makes all programmatic locks unnecessary: cooperative multitasking ensures that there will be no concurrency around a resource, no race conditions, and no memory consistency issues. The way to achieve this is quite simple: in critical sections, don't use yields, explicit or implicit, and no one can interfere into the code execution.

При небольших запросах, таких как простые UPDATE, INSERT, DELETE или SELECT, происходит справедливое планирование файберов: немного времени требуется на обработку запроса, планирование записи на диск и передачу управления на файбер, обслуживающий следующего клиента.

Однако функция может выполнять сложные расчеты или может быть написана так, что управление не передается в течение длительного времени. Это может привести к несправедливому планированию, когда отдельный клиент перекрывает работу остальной системы, или к явным задержкам в обработке запросов. Автору функции следует не допускать таких ситуаций.

Транзакции

В отсутствие транзакций любая функция, в которой есть точки передачи управления, может видеть изменения в состоянии базы данных, вызванные вытесняющими файберами. Составные транзакции предназначены для **изоляции**: каждая транзакция видит постоянное состояние базы данных и делает атомарные коммиты изменений. Во время *коммита* происходит передача управления, а все транзакционные изменения записываются в *журнал упреждающей записи* в отдельный пакет. Или, при необходимости, можно откатить изменения – *полностью* или на определенную *точку сохранения*.

In Tarantool, transaction isolation level is *serializable* with the clause «if no failure during writing to WAL». In case of such a failure that can happen, for example, if the disk space is over, the transaction isolation level becomes *read uncommitted*.

In *vynil*, to implement isolation Tarantool uses a simple optimistic scheduler: the first transaction to commit wins. If a concurrent active transaction has read a value modified by a committed transaction, it is aborted.

Кооперативный планировщик обеспечивает, что в отсутствие передачи управления составная транзакция не вытесняется, поэтому никогда не прерывается. Таким образом, понимание передачи управления необходимо для написания кода без прерываний.

Иногда при тестировании механизма транзакций в Tarantool можно заметить, что выдача после `box.begin()`, но перед любой операцией чтения/записи не приводит к прерыванию, как это должно происходить согласно описанию. Причина в том, что на самом деле `box.begin()` не запускает транзакцию: это просто метка, указывающая Tarantool запустить транзакцию после некоторого запроса к базе данных, который следует за этим.

In memtx, if an instruction that implies yields, explicit or implicit, is executed during a transaction, the transaction is fully rolled back. In vynil, we use more complex transactional manager that allows yields.

Примечание: На сегодняшний день нельзя смешивать движки базы данных в транзакции.

Правила неявной передачи управления

Единственные запросы явной передачи данных в Tarantool'е отправляют *fiber.sleep()* и *fiber.yield()*, но многие другие запросы «неявно» подразумевают передачу управления, поскольку цель Tarantool'а – избежать блокировок.

Database requests imply yields if and only if there is disk I/O. For memtx, since all data is in memory, there is no disk I/O during a read request. For vinyl, since some data may not be in memory, there may be disk I/O for a read (to fetch data from disk) or for a write (because a stall may occur while waiting for memory to be free). For both memtx and vinyl, since data-change requests must be recorded in the WAL, there is normally a commit. A commit happens automatically after every request in default «autocommit» mode, or a commit happens at the end of a transaction in «transaction» mode, when a user deliberately commits by calling *box.commit()*. Therefore for both memtx and vinyl, because there can be disk I/O, some database operations may imply yields.

Многие функции в модулях *fio*, *net_box*, *console* и *socket* (запросы «ОС» и «сети») передают управление.

Поэтому выполнение отдельных команд, таких как `select()`, `insert()`, `update()` в консоли внутри транзакции, приведет к прерыванию транзакции. Это связано с тем, что после выполнения каждого фрагмента кода в консоли происходит неявная передача управления (yield).

Пример №1

- *Engine = memtx* The sequence `select() insert()` has one yield, at the end of insertion, caused by implicit commit; `select()` has nothing to write to the WAL and so does not yield.

- *Engine = vinyl* The sequence `select() insert()` has one to three yields, since `select()` may yield if the data is not in cache, `insert()` may yield waiting for available memory, and there is an implicit yield at commit.
- Последовательность `begin() insert() insert() commit()` передает управление только при коммите, если движок – memtx, и может передавать управление до 3 раз, если движок – vinyl.

Пример №2

Assume that in the memtx space 'tester' there are tuples in which the third field represents a positive dollar amount. Let's start a transaction, withdraw from tuple#1, deposit in tuple#2, and end the transaction, making its effects permanent.

```
tarantool> function txn_example(from, to, amount_of_money)
         >   box.begin()
         >   box.space.tester:update(from, {{'-', 3, amount_of_money}})
         >   box.space.tester:update(to,   {{'+', 3, amount_of_money}})
         >   box.commit()
         >   return "ok"
         > end
---
...
tarantool> txn_example({999}, {1000}, 1.00)
---
- "ok"
...
```

Если *wal_mode* = 'none', то при коммите управление не передается неявно, потому что не идет запись в WAL-файл.

If a task is interactive – sending requests to the server and receiving responses – then it involves network I/O, and therefore there is an implicit yield, even if the request that is sent to the server is not itself an implicit yield request. Therefore, the following sequence

```
conn.space.test:select{1}
conn.space.test:select{2}
conn.space.test:select{3}
```

causes yields three times sequentially when sending requests to the network and awaiting the results. On the server side, the same requests are executed in common order possibly mixing with other requests from the network and local fibers. Something similar happens when using clients that operate via telnet, via one of the connectors, or via the *MySQL and PostgreSQL rocks*, or via the interactive mode when *using Tarantool as a client*.

После того, как файбер передал управление, а затем вернул его, он незамедлительно вызывает *testcancel*.

4.2.3 Управление доступом

В основном администраторы занимаются вопросами настроек безопасности. Однако обычные пользователи должны хотя бы бегло прочитать этот раздел, чтобы понять, как Tarantool позволяет администраторам не допустить неавторизованный доступ к базе данных и некоторым функциям.

Вкратце:

- Существует метод, который с помощью паролей проверяет, что пользователи являются теми, за кого себя выдают ("аутентификация").
- Существует системный спейс *_user*, где хранятся имена пользователей и хеши паролей.

- Существуют функции, чтобы дать определенным пользователям право совершать определенные действия (“права”).
- Существует системный спейс *_priv*, где хранятся права. Когда пользователь пытается выполнить операцию, проводится проверка на наличие у него прав на выполнение такой операции (“управление доступом”).

Подробная информация приводится ниже.

Пользователи

Для любой локальной или удаленной программы, работающей с Tarantool’ом, есть **текущий пользователь**. Если удаленное соединение использует *бинарный порт*, то текущим пользователем, по умолчанию, будет „**guest**“ (гость). Если соединение использует *порт для административной консоли*, текущим пользователем будет „**admin**“ (администратор). При выполнении *скрипта инициализации на Lua*, текущим пользователем также будет ‘**admin**’.

Имя текущего пользователя можно узнать с помощью *box.session.user()*.

Текущего пользователя можно изменить:

- Для соединения по бинарному порту – с помощью *команды протокола AUTH*, которая поддерживается большинством клиентов;
- Для соединения по порту для административной консоли и при выполнении скрипта инициализации на Lua – с помощью *box.session.su*;
- Для соединения по бинарному порту, которое вызывает хранимую функцию с помощью команды CALL – если для функции включена настройка *SETUID*, Tarantool временно заменит текущего пользователя на создателя функции со всеми правами создателя во время выполнения функции.

Пароли

У каждого пользователя (за исключением гостя „guest“) может быть **пароль**. Паролем является любая буквенно-цифровая строка.

Пароли Tarantool’а хранятся в системном спейсе *_user* с криптографической хеш-функцией, так что если паролем является ‘x’, хранится хеш-пароль в виде длинной строки, например ‘lL3OvhkIPOKh+Vn9Avlkx69M/Ck=‘. Когда клиент подключается к экземпляру Tarantool’а, экземпляр отправляет случайное значение соль, которое клиент должен сложить вместе с хеш-паролем перед отправкой на экземпляр. Таким образом, изначальное значение ‘x’ никогда не хранится нигде, кроме как в голове самого пользователя, а хешированное значение никогда не передается по сети, кроме как в смешанном с солью виде.

Примечание: Для получения дополнительной информации об алгоритме хеширования паролей (например, для написания нового клиентского приложения), прочтите файл заголовка scramble.h.

Система не дает злоумышленнику определить пароли путем просмотра файлов журнала или слежения за активностью. Это та же система, несколько лет назад внедренная в MySQL, которой оказалось достаточно для объектов со средней степенью безопасности. Тем не менее, администраторы должны предупреждать пользователей, что никакая система не защищена полностью от постоянных длительных атак, поэтому пароли следует охранять и периодически изменять. Администраторы также должны рекомендовать пользователям выбирать длинные неочевидные пароли, но сами пользователи выбирают свои пароли и изменяют их.

Для управления паролями в Tarantool'е есть две функции: *box.schema.user.passwd()* для изменения пароля пользователя и *box.schema.user.password()* для получения хеша пароля пользователя.

Владельцы и права

В Tarantool'е одна база данных. Она может называться «box.schema» или «universe». База данных содержит объекты базы данных, включая спейсы, индексы, пользователей, роли, последовательности и функции.

Владелец объекта базы данных – это пользователь, который создал его. Владельцем самой базы данных и объектов, которые изначально были созданы (системные спейсы и пользователи по умолчанию) является „**admin**".

У владельцев автоматически есть **права** на то, что они создают. Владельцы могут поделиться этими правами с другими пользователями или ролями с помощью запросов *box.schema.user.grant*. Можно предоставить следующие права:

- „read" (чтение), например, разрешить выборку из спейса
- „write" (запись), например, разрешить обновление спейса
- „execute" (выполнение), например, разрешить вызов функции, или (реже) разрешить использование роли
- „create" (создание), например, разрешить выполнение *box.schema.space.create* (также необходим доступ к определенным системным спейсам)
- „alter" (изменение), например, разрешить выполнение *box.space.x.index.y:alter* (также необходим доступ к определенным системным спейсам
- „drop", e.g. allow *box.sequence.x:drop* (access to certain system spaces is also necessary)
- „usage" (использование), например, допустимо ли любое действие, несмотря на другие права (иногда удобно отменить право на использование, чтобы временно заблокировать пользователя, не удаляя ег
- „session" (сессия), например, может ли пользователь выполнить подключение „connect".

Чтобы **создавать** объекты, у пользователей должны быть права на создание „create" и хотя бы права на чтение „read" и запись „write" в системный спейс с похожим именем (например, на спейс *_space*, если пользователю необходимо создавать спейсы.

Чтобы **получать доступ** к объектам, у пользователей должны быть соответствующие права на объект (например, права на выполнение „execute" на функцию F, если пользователям необходимо выполнить функцию F). См. ниже некоторые *примеры предоставления определенных прав*, которые может выдать „admin" или создатель объекта.

To drop an object, users must be „admin" or have the „super" role. Some objects may also be dropped by their creators. As the owner of the entire database, „admin" can drop any object including other users.

Чтобы предоставить права пользователю, владелец объекта выполняет команду *grant()*. Чтобы отменить права пользователя, владелец объекта выполняет команду *revoke()*. В любом случае можно использовать до пяти параметров:

```
(user-name, privilege, object-type [, object-name [, options]])
```

- `user-name` – это пользователь (или роль), который получит или потеряет права;
- `privilege` – это тип прав: „read", „write", „execute", „create", „alter", „drop", „usage" или „session" (или список прав, разделенных запятыми);

- `object-type` is any of „space“, „index“, „sequence“, „function“, „user“, „role“, or „universe“;
- `object-name` is what the privilege is for (omitted if `object-type` is „universe“) (may be omitted or `nil` if the intent is to grant for all objects of the same type);
- `options` – это список параметров, приведенный в скобках, например, `{if_not_exists=true|false}` (как правило, не указывается, поскольку допускаются значения по умолчанию).

  Все изменения прав пользователя сразу же отражаются на текущих сессиях и на объектах, например, функциях.

Пример предоставления нескольких типов прав одновременно

В данном примере пользователь „admin“ выдает много типов прав на множество объектов пользователю „U“ в едином запросе.

```
box.schema.user.grant('U','read,write,execute,create,drop','universe')
```

Примеры предоставления прав на определенные действия

In these examples an administrator grants precisely the minimal privileges necessary for particular operations, to user „U“.

```
-- So that 'U' can create spaces:
  box.schema.user.grant('U','create','space')
  box.schema.user.grant('U','write', 'space', '_schema')
  box.schema.user.grant('U','write', 'space', '_space')
-- So that 'U' can  create indexes on space T
  box.schema.user.grant('U','create,read','space','T')
  box.schema.user.grant('U','read,write','space','_space_sequence')
  box.schema.user.grant('U','write', 'space', '_index')
-- So that 'U' can  alter indexes on space T (assuming 'U' did not create the index)
  box.schema.user.grant('U','alter','space','T')
  box.schema.user.grant('U','read','space','_space')
  box.schema.user.grant('U','read','space','_index')
  box.schema.user.grant('U','read','space','_space_sequence')
  box.schema.user.grant('U','write','space','_index')
-- So that 'U' can alter indexes on space T (assuming 'U' created the index)
  box.schema.user.grant('U','read','space','_space_sequence')
  box.schema.user.grant('U','read,write','space','_index')
-- So that 'U' can create users:
  box.schema.user.grant('U','create','user')
  box.schema.user.grant('U', 'read,write', 'space', '_user')
  box.schema.user.grant('U', 'write', 'space', '_priv')
-- So that 'U' can create roles:
  box.schema.user.grant('U','create','role')
  box.schema.user.grant('U', 'read,write', 'space', '_user')
  box.schema.user.grant('U', 'write', 'space', '_priv')
-- So that 'U' can create sequence generators:
  box.schema.user.grant('U','create','sequence')
  box.schema.user.grant('U', 'read,write', 'space', '_sequence')
-- So that 'U' can create functions:
  box.schema.user.grant('U','create','function')
  box.schema.user.grant('U','read,write','space','_func')
-- So that 'U' can create any object of any type
  box.schema.user.grant('guest','read,write,create','universe')
-- So that 'U' can grant access on objects that 'U' created
  box.schema.user.grant('U','write','space','_priv')
-- So that 'U' can select or get from a space named 'T'
```

(continues on next page)

(продолжение с предыдущей страницы)

```
  box.schema.user.grant('U','read','space','T')
-- So that 'U' can update or insert or delete or truncate a space named 'T'
  box.schema.user.grant('U','write','space','T')
-- So that 'U' can execute a function named 'F'
  box.schema.user.grant('U','execute','function','F')
-- So that 'U' can use the "S:next()" function with a sequence named S
  box.schema.user.grant('U','read,write','sequence','S')
-- So that 'U' can use the "S:set()" or "S:reset() function with a sequence named S
  box.schema.user.grant('U','write','sequence','S')
-- So that 'U' can drop a sequence (assuming 'U' did not create it)
  box.schema.user.grant('U','drop','sequence')
  box.schema.user.grant('U','write','space','_sequence_data')
  box.schema.user.grant('U','write','space','_sequence')
-- So that 'U' can drop a function (assuming 'U' did not create it)
  box.schema.user.grant('U','drop','function')
  box.schema.user.grant('U','write','space','_func')
-- So that 'U' can drop a space that has some associated objects
  box.schema.user.grant('U','create,drop','space')
  box.schema.user.grant('U','write','space','_schema')
  box.schema.user.grant('U','write','space','_space')
  box.schema.user.grant('U','write','space','_space_sequence')
  box.schema.user.grant('U','read','space','_trigger')
  box.schema.user.grant('U','read','space','_fk_constraint')
  box.schema.user.grant('U','read','space','_ck_constraint')
  box.schema.user.grant('U','read','space','_func_index')
-- So that 'U' can drop any space (ignore if the privilege exists already)
  box.schema.user.grant('U','drop','space',nil,{if_not_exists=true})
```

Пример создания пользователей и объектов и последующей выдачи прав

Здесь создадим Lua-функцию, которая будет выполняться от ID пользователя, который является ее создателем, даже если она вызывается другим пользователем.

Для начала создадим два спейса („u“ и „i“) и дадим полный доступ к ним пользователю без пароля („internal“). Затем определим функцию („read_and_modify“), и пользователь без пароля становится создателем функции. Наконец, дадим другому пользователю („public_user“) доступ на выполнение Lua-функций, созданных пользователем без пароля.

```
box.schema.space.create('u')
box.schema.space.create('i')
box.space.u:create_index('pk')
box.space.i:create_index('pk')

box.schema.user.create('internal')

box.schema.user.grant('internal', 'read,write', 'space', 'u')
box.schema.user.grant('internal', 'read,write', 'space', 'i')
box.schema.user.grant('internal', 'create', 'universe')
box.schema.user.grant('internal', 'read,write', 'space', '_func')

function read_and_modify(key)
  local u = box.space.u
  local i = box.space.i
  local fiber = require('fiber')
  local t = u:get{key}
  if t ~= nil then
    u:put{key, box.session.uid()}
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    i:put{key, fiber.time()}
  end
end

box.session.su('internal')
box.schema.func.create('read_and_modify', {setuid= true})
box.session.su('admin')
box.schema.user.create('public_user', {password = 'secret'})
box.schema.user.grant('public_user', 'execute', 'function', 'read_and_modify')
```

Роли

Роль представляет собой контейнер для прав, которые можно предоставить обычным пользователям. Вместо того, чтобы предоставлять или отменять индивидуальные права, можно поместить все права в роль, а затем назначить или отменить роль.

Информация о роли хранится в спейсе *_user*, но третье поле кортежа – поле типа – это 'роль', а не 'пользователь'.

В управлении доступом на основе ролей один из главных моментов – это то, что роли могут быть **вложенными**. Например, роли R1 можно предоставить право типа «роль R2», то есть пользователи с ролью R1 тогда получат все права роли R1 и роли R2. Другими словами, пользователь получает все права, которые предоставляются ролям пользователя напрямую и опосредованно.

Фактически есть два способа предоставить или отменить роль: `box.schema.user.grant-or-revoke(`*имя-пользователя-или-имя-роли*`,'execute', 'role',`*имя-роли*`...)` или `box.schema.user.grant-or-revoke(`*имя-пользователя-или-имя-роли*`,`*имя-роли*`...)`. Рекомендуется использовать второй способ.

Права типов „usage“ и „session“ нельзя предоставить для роли.

Пример

```
-- Этот пример сработает для пользователя со множеством прав, например, 'admin'
-- или для пользователя с заданной ролью 'super'
-- Создать спейс T с первичным индексом
box.schema.space.create('T')
box.space.T:create_index('primary', {})
-- Создать пользователя U1, чтобы затем можно было заменить текущего пользователя на U1
box.schema.user.create('U1')
-- Создать две роли, R1 и R2
box.schema.role.create('R1')
box.schema.role.create('R2')
-- Предоставить роль R2 для роли R1, а роль R1 пользователю U1 (порядок не имеет значения)
-- Есть два способа предоставить роль, здесь используется более короткий способ
box.schema.role.grant('R1', 'R2')
box.schema.user.grant('U1', 'R1')
-- Предоставить права на чтение/запись на спейс T для роли R2
-- (но не для роли R1 и не пользователю U1)
box.schema.role.grant('R2', 'read,write', 'space', 'T')
-- Изменить текущего пользователя на пользователя U1
box.session.su('U1')
-- Теперь вставка в спейс T сработает, потому что благодаря вложенным ролям,
-- у пользователя U1 есть права на запись в спейс T
box.space.T:insert{1}
```

Более подробную информацию см. в справочнике по встроенным модулям: *box.schema.user.grant()* и *box.schema.role.grant()*.

Сессии и безопасность

Сессия – это состояние подключения к Tarantool'у. Она содержит:

- идентификатор в виде целого числа, определяющий соединение,
- *текущий пользователь*, использующий соединение,
- текстовое описание подключенного узла и
- локальное состояние сессии, например, переменные и функции на Lua.

В Tarantool'е отдельная сессия может выполнять несколько транзакций одновременно. Каждая транзакция определяется по уникальному идентификатору в виде целого числа, который можно запросить в начале транзакции с помощью *box.session.sync()*.

Примечание: Чтобы отследить все подключения и отключения, можно использовать *триггеры соединений и аутентификации*.

4.2.4 Триггеры

Триггеры, которые также называют **обратными вызовами**, представляют собой функции, которые выполняет сервер при наступлении определенных событий.

В Tarantool'е есть шесть типов триггеров:

- *box.session.on_connect()* (при установлении соединения) или *box.session.on_disconnect()* (при разрыве соединения),
- *box.session.on_auth()*,
- *space_object:on_replace()* (после замены) или *space_object:before_replace()* (перед заменой),
- *box.on_commit()* (при коммите) или *box.on_rollback()* (при откате),
- *net.box.on_connect()* or *net.box.on_disconnect()*,
- *net.box.on_schema_reload()*,
- *box.ctl.on_schema_init()* or *box.ctl.on_shutdown()*,
- *swim_object:on_member_event()* (при получении события от swim-участников).

У всех триггеров есть следующие особенности:

- Триггеры связывают функцию с событием. Запрос «определить триггер» подразумевает передачу функции триггера в одну из функций обработки событий «on_event()»:
  - *box.session.on_connect()* (при установлении соединения) или *box.session.on_disconnect()* (при разрыве соединения),
  - *box.session.on_auth()*,
  - *net.box.on_connect()* or *net.box.on_disconnect()*,
  - *net.box.on_schema_reload()*,
  - *space_object:on_replace()* (после замены) или *space_object:before_replace()* (перед заменой),

  - *box.on_commit()* (при коммите) или *box.on_rollback()* (при откате),
  - *box.ctl.on_schema_init()* or *box.ctl.on_shutdown()*,
  - *swim_object:on_member_event()* (при получении события от swim-участников).

- Только *пользователь „admin“* определяет триггеры.
- Триггеры хранятся в памяти экземпляра Tarantool'a, а не в базе данных. Поэтому триггеры пропадают, когда экземпляр отключают. Чтобы сохранить их, поместите определения функции и настройки триггера в *скрипт инициализации* Tarantool'a.
- Триггеры не тратят много ресурсов. Если триггер не задан, то требуется минимум вычислений — разыменование и проверка указателя. Если триггер определен, то стоимость вызова равна стоимости вызова функции.
- There can be multiple triggers for one event. In this case, triggers are executed in the reverse order that they were defined in. (Exception: member triggers are executed in the order that they appear in the member list.)
- Триггеры должны работать в контексте события. Однако результат не определен, если функция содержит запросы, которые при нормальных условиях не могут быть выполнены непосредственно после события, а только после возврата из события. Например, если указать os.exit() или *box.rollback()* в триггерной функции, запросы не будут выполняться в контексте события.
- Триггеры можно заменять. Запрос на «замену триггера» подразумевает передачу новой триггерной функции и старой триггерной функции в одну из функций обработки событий «on_event()».
- Во всех функциях обработки событий «on_event()» есть параметры, которые представляют собой указатели функции, и все они возвращают указатели функции. Следует запомнить, что определение Lua-функции, например, «function f() x = x + 1 end» совпадает с «f = function () x = x + 1 end» – в обоих случаях `f` получит указатель функции. А «trigger = box.session.on_connect(f)» – это то же самое, что «trigger = box.session.on_connect(function () x = x + 1 end)» – в обоих случаях `trigger` получит переданный указатель функции.
- Если вызвать любую из «on_event()» функций без аргументов, то она вернет список соответствующих триггеров. Например, `box.session.on_connect()` вернет таблицу со всеми connect-trigger функциями.
- Triggers can be useful in solving problems with replication. See details in

Resolving replication conflicts

Пример

Здесь мы записываем события подключения и отключения в журнал на сервере Tarantool'a.

```
log = require('log')

function on_connect_impl()
  log.info("connected "..box.session.peer().."", sid "..box.session.id())
end

function on_disconnect_impl()
  log.info("disconnected, sid "..box.session.id())
end

function on_auth_impl(user)
  log.info("authenticated sid "..box.session.id().."" as "..user)
end"
```

(continues on next page)

(продолжение с предыдущей страницы)

```
function on_connect() pcall(on_connect_impl) end
function on_disconnect() pcall(on_disconnect_impl) end
function on_auth(user) pcall(on_auth_impl, user) end

box.session.on_connect(on_connect)
box.session.on_disconnect(on_disconnect)
box.session.on_auth(on_auth)
```

4.2.5 Ограничения

Количество частей в индексе

Для TREE-индексов или HASH-индексов максимальное количество – 255 частей (`box.schema.INDEX_PART_MAX`). Для *RTREE-индексов* максимальное количество – 1, но это поля типа ARRAY (массив) с размерностью до 20. Для BITSET-индексов максимальное количество – 1.

Количество индексов в спейсе

128 (`box.schema.INDEX_MAX`).

Количество полей в кортеже

Теоретически максимальное количество составляет 2 147 483 647 полей (`box.schema.FIELD_MAX`). Практически максимальное количество указано в поле *field_count* спейса или соответствует максимальной длине кортежа.

Количество байтов в кортеже

Максимальное количество байтов в кортеже примерно равно *memtx_max_tuple_size* или *vinyl_max_tuple_size* (с ресурсами метаданных около 20 байтов на кортеж, которые добавляются к полезным байтам). Значение `memtx_max_tuple_size` или `vinyl_max_tuple_size` по умолчанию составляет 1 048 576. Чтобы его увеличить, укажите большее значение при запуске экземпляра Tarantool'а. Например, `box.cfg{memtx_max_tuple_size=2*1048576}`.

Количество байтов в индекс-ключе

Если поле в кортеже может содержать миллион байтов, то индекс-ключ может содержать миллион байтов, поэтому максимальное количество определяется такими факторами, как *количество байтов в кортеже*, а не параметрами индекса.

Количество спейсов

Теоретически максимальное количество составляет 2 147 483 647 (`box.schema.SPACE_MAX`), но практически максимальное количество – около 65 000.

Количество соединений

Практически пределом является количество файловых дескрипторов, которые можно определить с операционной системой.

Размер спейса

Итоговый максимальный размер всех спейсов фактически определяется в *memtx_memory*, который в свою очередь ограничен общим размером свободной памяти.

Число операций обновления

Максимальное количество операций, возможное в рамках одного обновления (для одного тапла), составляет 4000 (`BOX_UPDATE_OP_CNT_MAX`).

Количество пользователей и ролей

32 (`BOX_USER_MAX`).

Длина имени индекса, имени спейса или имени пользователя

65000 (`box.schema.NAME_MAX`).

Количество реплик в наборе реплик

32 (`vclock.VCLOCK_MAX`).

4.2.6 Движки базы данных

Движок базы данных представляет собой набор очень низкоуровневых процессов, которые фактически хранят и получают значения в кортежах. Tarantool предлагает два движка базы данных на выбор:

- memtx (in-memory движок базы данных) используется по умолчанию, который был первым.
- vinyl (движок для хранения данных на диске) – это рабочий движок на основе пар ключ-значение, который особенно понравится пользователям, предпочитающим записывать данные напрямую на диск, чтобы сократить время восстановления и увеличить размер базы данных.

  С другой стороны, vinyl'у не хватает некоторых функций и параметров, доступных в memtx'е. В соответствующих случаях дается дополнительное описание в виде примечания, которое начинается со слов **Примечание про движок базы данных**.

Далее в разделе рассмотрим подробнее метод хранения данных с помощью движка базы данных vinyl.

Чтобы указать, что следует использовать именно vinyl, необходимо при создании спейса добавить оператор `engine = 'vinyl'`, например:

```
space = box.schema.space.create('name', {engine='vinyl'})
```

Различия между движками memtx и vinyl

Основным различием между движками memtx и vinyl является то, что memtx представляет собой «in-memory» движок, тогда как vinyl – это «дисковый» движок. Как правило, in-memory движок быстрее (каждый запрос обычно выполняется меньше, чем за 1 мс), и движок memtx по праву используется в Tarantool'е по умолчанию, но если база данных больше объема доступной памяти, а добавление дополнительной памяти не представляется возможным, рекомендуется использовать дисковый движок как vinyl.

Характеристика	memtx	vinyl
Поддерживаемый тип индекса	TREE, HASH, *RTREE* или BITSET	TREE
Временные спейсы	Поддерживается	Не поддерживается
функция *random()*	Поддерживается	Не поддерживается
функция *alter()*	Поддерживается	Поддерживается с версии 1.10.2 (первичный индекс изменять нельзя)
функция *len()*	Возвращает количество кортежей в спейсе	Возвращает максимальное примерное количество кортежей в спейсе
функция *count()*	Занимает одинаковые периоды времени	Занимает различное количество времени в зависимости от состояния БД
функция *delete()*	Возвращает удаленный кортеж, если есть таковой	Всегда возвращает nil
передача управления	Не передает управление на запросах выборки, если не происходит коммит транзакции в журнал упреждающей записи (WAL)	Передает управление на запросах выборки или аналогичных: get() или pairs()

Хранение данных с помощью vinyl

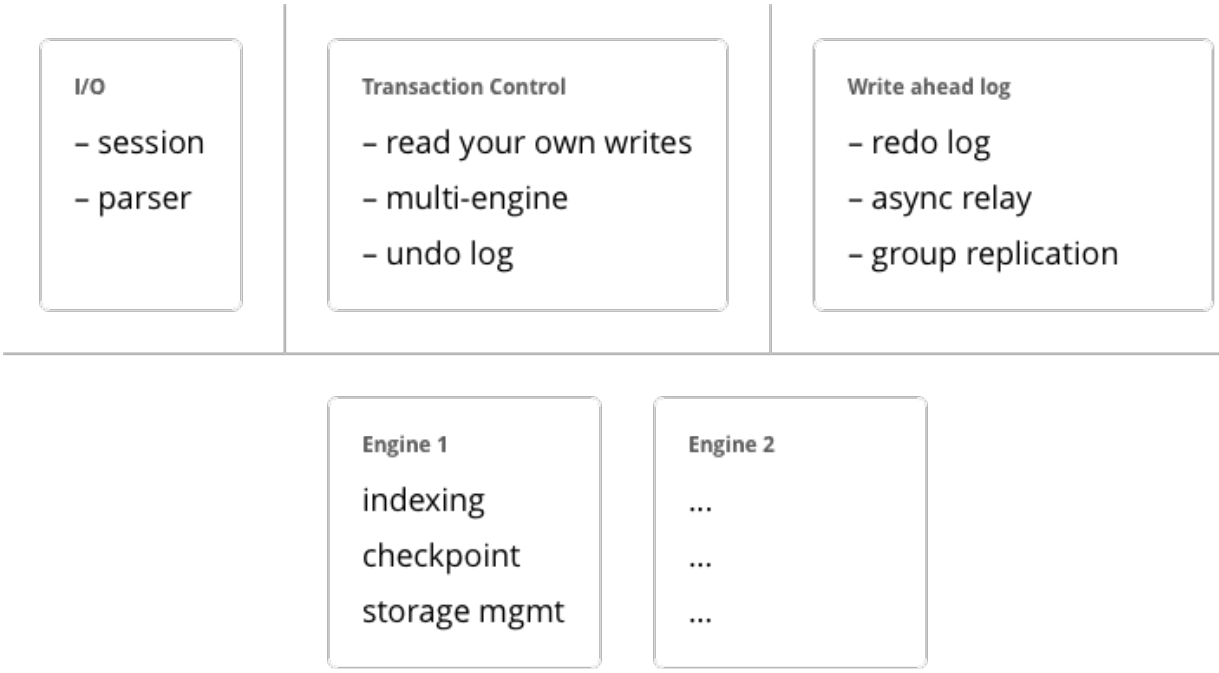
Tarantool – это транзакционная, персистентная СУБД, которая хранит 100% данных в оперативной памяти. Основными преимущества хранения данных оперативной памяти являются скорость и простота использования: нет необходимости в оптимизации, однако производительность остается стабильно высокой.

Несколько лет назад мы решили расширить продукт посредством реализации классической технологии хранения как в обычных СУБД: в оперативной памяти хранится лишь кэш данных, а основной объем данных находится на диске. Мы решили, что движок хранения можно будет выбирать независимо для каждой таблицы, как это реализовано в MySQL, но при этом с самого начала будет реализована поддержка транзакций.

Первый вопрос, на который нужен был ответ: создавать свой движок или использовать уже существующую библиотеку? Сообщество разработчиков открытого ПО предлагает готовые библиотеки на выбор. Активнее всего развивалась библиотека RocksDB, которая к настоящему времени стала одной из самых популярных. Есть также несколько менее известных библиотек: WiredTiger, ForestDB, NestDB, LMDB.

Тем не менее, изучив исходный код существующих библиотек и взвесив все «за» и «против», мы решили написать свой движок. Одна из причин – все существующие сторонние библиотеки предполагают, что запросы к данным могут поступать из множества потоков операционной системы, и поэтому содержат сложные примитивы синхронизации для управления одновременным доступом к данным. Если бы мы решили встраивать одну из них в Tarantool, то пользователи были бы вынуждены нести издержки многопоточных приложений, не получая ничего взамен. Дело в том, что в основе Tarantool'а лежит архитектура на основе акторов. Обработка транзакций в выделенном потоке позволяет обойтись без лишних блокировок, межпроцессного взаимодействия и других затрат ресурсов, которые забирают до 80% процессорного времени в многопоточных СУБД.

Процесс Tarantool'а состоит из заданного количества потоков



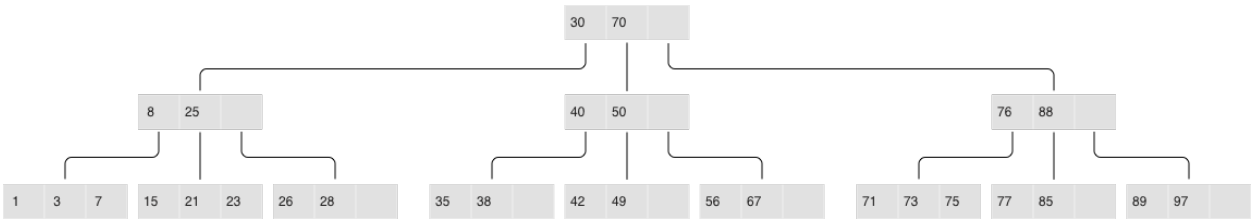
Если изначально проектировать движок с учетом кооперативной многозадачности, можно не только существенно ускорить работу, но и реализовать приемы оптимизации, слишком сложные для многопоточных движков. В общем, использование стороннего решения не привело бы к лучшему результату.

Алгоритм

Отказавшись от идеи внедрения существующих библиотек, необходимо было выбрать архитектуру для использования в качестве основы. Есть два альтернативных подхода к хранению данных на диске: старая модель с использованием B-деревьев и их разновидностей и новая – на основе журнально-структурированных деревьев со слиянием, или LSM-деревьев (Log Structured Merge Tree). MySQL, PostgreSQL и Oracle используют B-деревья, а Cassandra, MongoDB и CockroachDB уже используют LSM-деревья.

Считается, что B-деревья более эффективны для чтения, а LSM-деревья – для записи. Тем не менее, с распространением SSD-дисков, у которых в несколько раз выше производительность чтения по сравнению с производительностью записи, преимущества LSM-деревьев стали очевидны в большинстве сценариев.

Прежде чем разбираться с LSM-деревьями в Tarantool'е, посмотрим, как они работают. Для этого разберем устройство обычного B-дерева и связанные с ним проблемы. «B» в слове B-tree означает «Block», то есть это сбалансированное дерево, состоящее из блоков, которые содержат отсортированные списки пар ключ-значение. Вопросы наполнения дерева, балансировки, разбиения и слияния блоков выходят за рамки данной статьи, подробности вы сможете прочитать в Википедии. В итоге мы получаем отсортированный по возрастанию ключа контейнер, минимальный элемент которого хранится в крайнем левом узле, а максимальный – в крайнем правом. Посмотрим, как в B-дереве осуществляется поиск и вставка данных.



Классическое B-дерево

Если необходимо найти элемент или проверить его наличие, поиск начинается, как обычно, с вершины. Если ключ обнаружен в корневом блоке, поиск заканчивается; в противном случае, переходим в блок с наибольшим меньшим ключом, то есть в самый правый блок, в котором еще есть элементы меньше искомого (элементы на всех уровнях расположены по возрастанию). Если и там элемент не найден, снова переходим на уровень ниже. В конце концов окажемся в одном из листьев и, возможно, обнаружим искомый элемент. Блоки дерева хранятся на диске и читаются в оперативную память по одному, то есть в рамках одного поиска алгоритм считывает $logB(N)$ блоков, где N – это количество элементов в B-дереве. Запись в самом простом случае осуществляется аналогично: алгоритм находит блок, который содержит необходимый элемент, и обновляет (вставляет) его значение.

Чтобы наглядно представить себе эту структуру данных, возьмем B-дерево на 100 000 000 узлов и предположим, что размер блока равен 4096 байтов, а размер элемента – 100 байтов. Таким образом, в каждом блоке можно будет разместить до 40 элементов с учетом накладных расходов, а в B-дереве будет около 2 570 000 блоков, пять уровней, при этом первые четыре займут по 256 МБ, а последний – до 10 ГБ. Очевидно, что на любом современном компьютере все уровни, кроме последнего, успешно попадут в кэш файловой системы, и фактически любая операция чтения будет требовать не более одной операции ввода-вывода.

Ситуация выглядит существенно менее радужно при смене точки зрения. Предположим, что необходимо обновить один элемент дерева. Так как операции с B-деревьями работают через чтение и запись целых блоков, приходится прочитать 1 блок в память, изменить 100 байт из 4096, а затем записать обновленный блок на диск. Таким образом, нам пришлось записать в 40 раз больше, чем реальный объем измененных данных!

Принимая во внимание, что внутренний размер блока в SSD-дисках может быть 64 КБ и больше, и не любое изменение элемента меняет его целиком, объем «паразитной» нагрузки на диск может быть еще выше.

Феномен таких «паразитных» чтений в литературе и блогах, посвященных хранению на диске, называется read amplification (усложнение чтения), а феномен «паразитной» записи – write amplification (усложнение записи).

Коэффициент усложнения, то есть коэффициент умножения, вычисляется как отношение размера фактически прочитанных (или записанных) данных к реально необходимому (или измененному) размеру. В нашем примере с B-деревом коэффициент составит около 40 как для чтения, так и для записи.

Объем «паразитных» операций ввода-вывода при обновлении данных является одной из основных проблем, которую решают LSM-деревья. Рассмотрим, как это работает.

Ключевое отличие LSM-деревьев от классических B-деревьев заключается в том, что LSM-деревья не просто хранят данные (ключи и значения), а также операции с данными: вставки и удаления.

LSM-дерево:

- **Хранит операторы, а не значения:**

  - REPLACE
  - DELETE
  - UPSERT

- Для каждого оператора назначается LSN Обновление файлов происходит только путем присоединения новых записей, сборка мусора проводится после контрольной точки
- Журнал транзакций при любых изменениях в системе: vylog

Например, элемент для операции вставки, помимо ключа и значения, содержит дополнительный байт с кодом операции – обозначенный выше как REPLACE. Элемент для операции удаления содержит ключ элемента (хранить значение нет необходимости) и соответствующий код операции – DELETE. Также каждый элемент LSM-дерева содержит порядковый номер операции (log sequence number – LSN), то есть значение монотонно возрастающей последовательности, которое уникально идентифицирует каждую операцию. Таким образом, всё дерево упорядочено сначала по возрастанию ключа, а в пределах одного ключа – по убыванию LSN.

Key	lsn	Op code	Value
1	176	REPLACE	2018-05-07 15:00:01
1	53	INSERT	2017-12-31 23:59:01
2	174	REPLACE	2018-05-06 00:00:00
3	175	REPLACE	2018-05-07 09:04:19
3	9	REPLACE	2017-01-01 19:25:43
3	7	INSERT	2017-01-01 19:22:16
4	173	DELETE	
4	168	INSERT	2018-05-05 07:40:01

Один уровень LSM-дерева

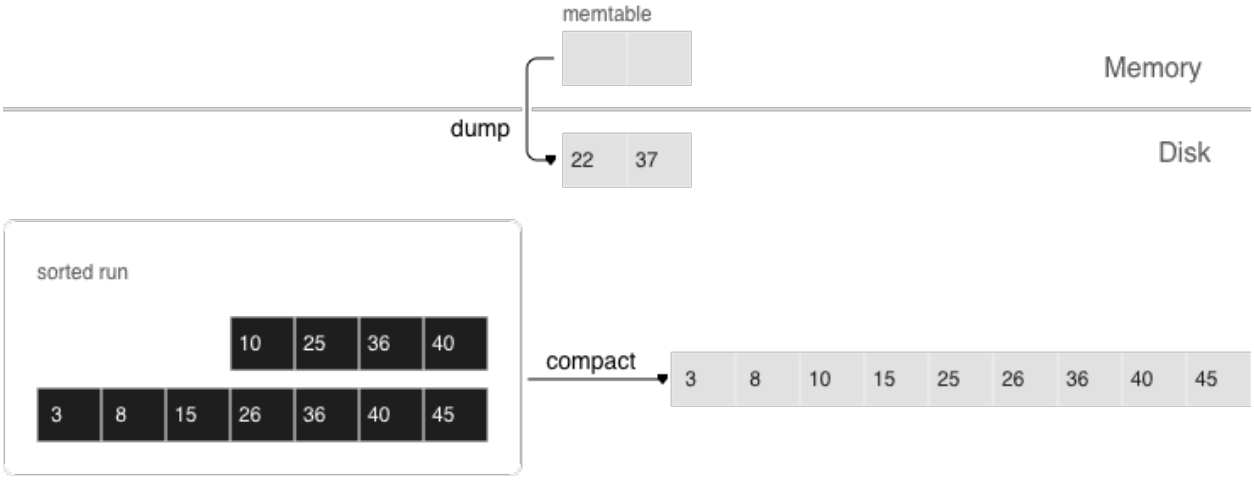
Наполнение LSM-дерева

В отличие от B-дерева, которое полностью хранится на диске и может частично кэшироваться в оперативной памяти, в LSM-дереве разделение между памятью и диском явно присутствует с самого

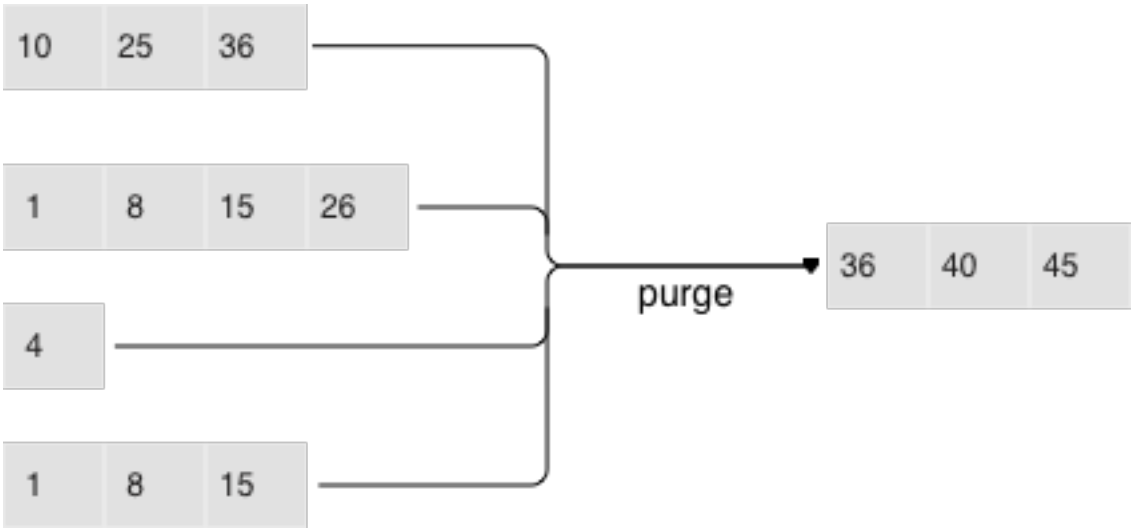
начала. При этом проблема сохранности данных, расположенных в энергозависимой памяти, выносится за рамки алгоритма хранения: ее можно решить разными способами, например, журналированием изменений.

Часть дерева, расположенную в оперативной памяти, называют L0 (level zero – уровень ноль). Объем оперативной памяти ограничен, поэтому для L0 отводится фиксированная область. В конфигурации Tarantool'a, например, размер L0 задается с помощью параметра `vinyl_memory`. В начале, когда LSM-дерево не содержит элементов, операции записываются в L0. Следует отметить, что элементы в дереве упорядочены по возрастанию ключа, а затем по убыванию LSN, так что в случае вставки нового значения по данному ключу легко обнаружить и удалить предыдущее значение. L0 может быть представлен любым контейнером, который сохраняет упорядоченность элементов. Например, для хранения L0 Tarantool использует B+*-дерево. Операции поиска и вставки – это стандартные операции структуры данных, используемой для представления L0, и мы их подробно рассматривать не будем.

Рано или поздно количество элементов в дереве превысит размер L0. Тогда L0 записывается в файл на диске (который называется забегом – «run») и освобождается под новые элементы. Эта операция называется «дамп» (dump).



Все дампы на диске образуют последовательность, упорядоченную по LSN: диапазоны LSN в файлах не пересекаются, а ближе к началу последовательности находятся файлы с более новыми операциями. Представим эти файлы в виде пирамиды, где новые файлы расположены вверху, а старые внизу. По мере появления новых файлов забегов, высота пирамиды растет. При этом более свежие файлы могут содержать операции удаления или замены для существующих ключей. Для удаления старых данных необходимо производиться сборку мусора (этот процесс иногда называется «слияние» – в английском языке «merge» или «compaction»), объединяя нескольких старых файлов в новый. Если при слиянии мы встречаем две версии одного и того же ключа, то достаточно оставить только более новую версию, а если после вставки ключа он был удален, то из результата можно исключить обе операции.



Ключевым фактором эффективности LSM-дерева является то, в какой момент и для каких файлов делается слияние. Представим, что LSM-дерево в качестве ключей хранит монотонную последовательность вида 1, 2, 3 . . . , и операций удаления нет. В этом случае слияние будет бесполезным – все элементы уже отсортированы, дерево не содержит мусор и можно однозначно определить, в каком файле находится каждый ключ. Напротив, если LSM-дерево содержит много операций удаления, слияние позволит освободить место на диске. Но даже если удалений нет, а диапазоны ключей в разных файлах сильно пересекаются, слияние может ускорить поиск, так как сократит число просматриваемых файлов. В этом случае имеет смысл выполнять слияние после каждого дампа. Однако следует отметить, что такое слияние приведет к перезаписи всех данных на диске, поэтому если чтений мало, то лучше делать слияния реже.

Для оптимальной конфигурации под любой из описанных выше сценариев в LSM-дереве все файлы организованы в пирамиду: чем новее операции с данными, тем выше они находятся в пирамиде. При этом в слиянии участвуют два или несколько соседних файлов в пирамиде; по возможности выбираются файлы примерно одинакового размера.

- Многоуровневое слияние может охватить любое количество уровней
- Уровень может содержать несколько файлов

Все соседние файлы примерно одинакового размера составляют уровень LSM-дерева на диске. Соотношение размеров файлов на различных уровнях определяет пропорции пирамиды, что позволяет оптимизировать дерево под интенсивные вставки, либо интенсивные чтения.

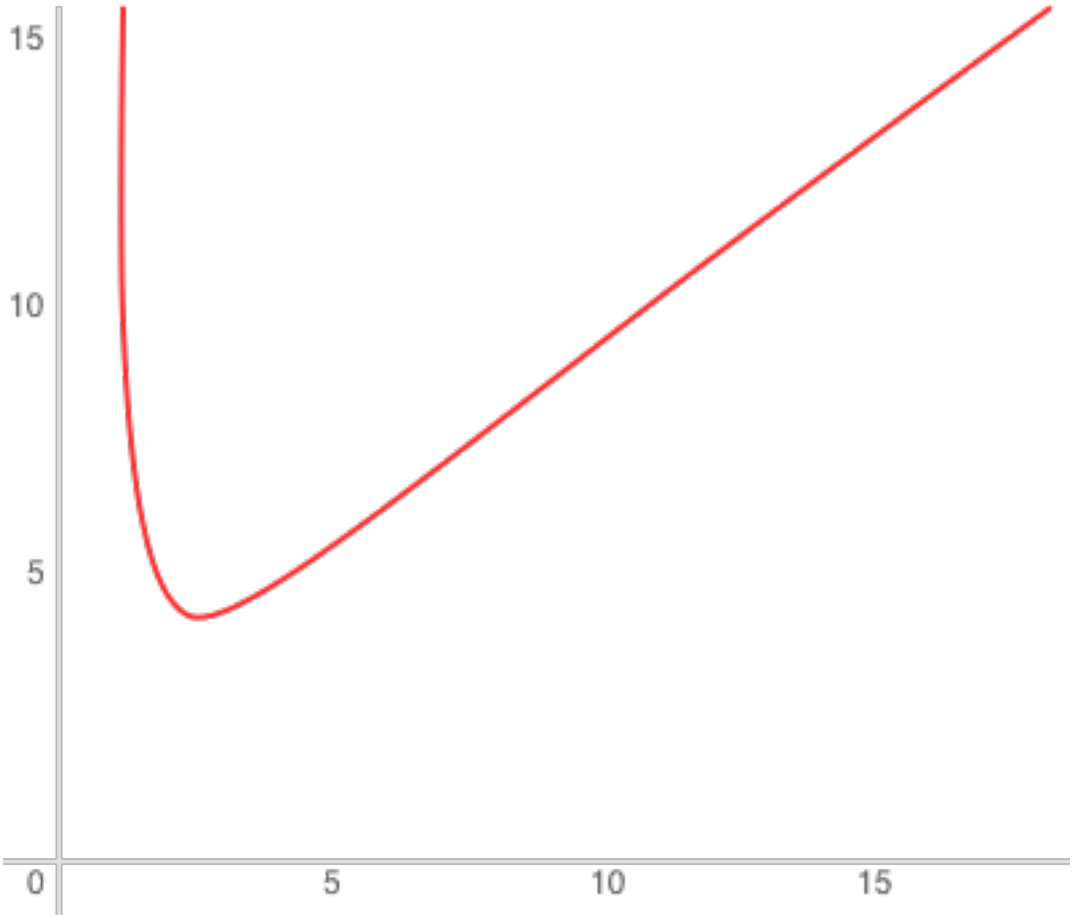
Предположим, что размер L0 составляет 100 МБ, а соотношение размеров файлов на каждом уровне (параметр `vinyl_run_size_ratio`) равно 5, и на каждом уровне может быть не более 2 файлов (пара-

метр `vinyl_run_count_per_level`). После первых трех дампов на диске появятся 3 файла по 100 МБ, эти файлы образуют уровень L1. Так как $3 > 2$, запустится слияние файлов в новый файл размером 300 МБ, а старые будут удалены. Спустя еще 2 дампа снова запустится слияние, на этот раз файлов в 100, 100 и 300 МБ, в результате файл размером 500 МБ переместится на уровень L2 (вспомним, что соотношение размеров уровней равно 5), а уровень L1 останется пустым. Пройдут еще 10 дампов, и получим 3 файла по 500 МБ на уровне L2, в результате чего будет создан один файл размером 1500 МБ. Спустя еще 10 дампов произойдет следующее: 2 раза произведем слияние 3 файлов по 100 МБ, а также 2 раза слияние файлов по 100, 100 и 300 МБ, что приведет к созданию двух файлов на уровне L2 по 500 МБ. Поскольку на уровне L2 уже есть три файла, запустится слияние двух файлов по 500 МБ и одного файла в 1500 МБ. Полученный в результате файл в 2500 МБ, в силу своего размера, переедет на уровень L3.

Процесс может продолжаться до бесконечности, а если в потоке операций с LSM-деревом будет много удалений, образовавшийся в результате слияния файл может переместиться не только вниз по пирамиде, но и вверх, так как окажется меньше исходных файлов, использовавшихся при слиянии. Иными словами, принадлежность файла к уровню достаточно отслеживать логически на основе размера файла и минимального и максимального LSN среди всех хранящихся в нем операций.

Управление формой LSM-дерева

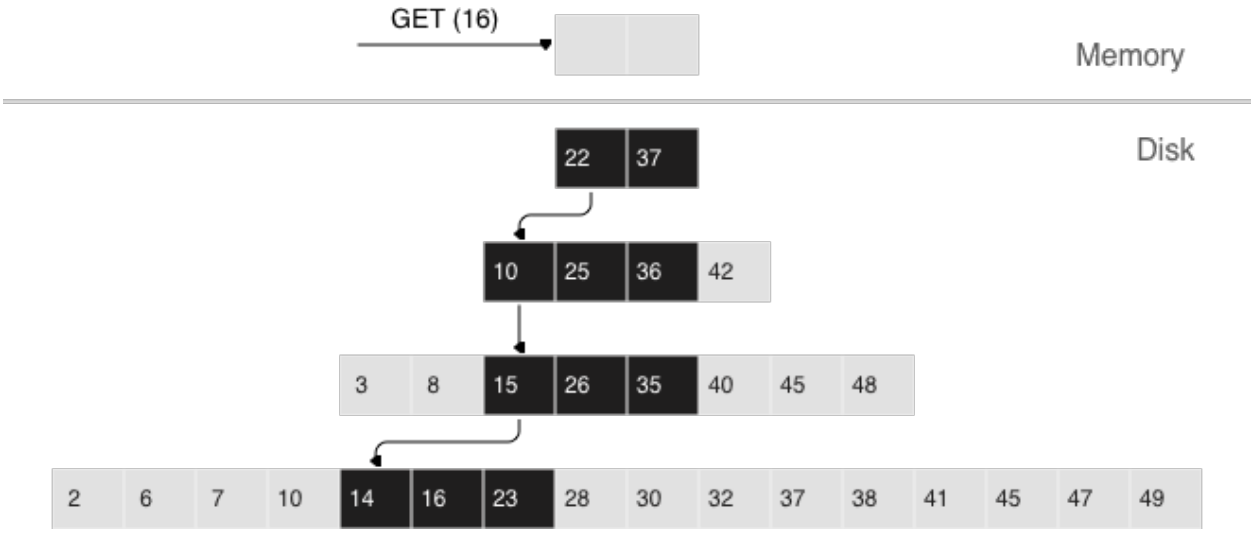
Если число файлов для поиска нужно уменьшить, то соотношение размеров файлов на разных уровнях можно увеличить, и, как следствие, уменьшается число уровней. Если, напротив, необходимо снизить затраты ресурсов, вызванные слиянием, то можно уменьшить соотношение размеров уровней: пирамида будет более высокой, а слияние хотя и выполняется чаще, но работает в среднем с файлами меньшего размера, за счет чего суммарно выполняет меньше работы. В целом, «паразитная запись» в LSM-дереве описывается формулой $log_x($
$fracNL0)x$ или x
$fracln(fracNC0)ln(x)$, где N – это общий размер всех элементов дерева, L0 – это размер уровня ноль, а x – это соотношение размеров уровней (параметр `level_size_ratio`). Если
$fracNC0 = 40$ (соотношение диск-память), график выглядит примерно вот так:



«Паразитное» чтение при этом пропорционально количеству уровней. Стоимость поиска на каждом уровне не превышает стоимости поиска в B-дереве. Возвращаясь к нашему примеру дерева в 100 000 000 элементов: при наличии 256 МБ оперативной памяти и стандартных значений параметров `vinyl_run_size_ratio` и `vinyl_run_count_per_level`, получим коэффициент «паразитной» записи равным примерно 13, коэффициент «паразитной» записи может доходить до 150. Разберемся, почему это происходит.

Поиск

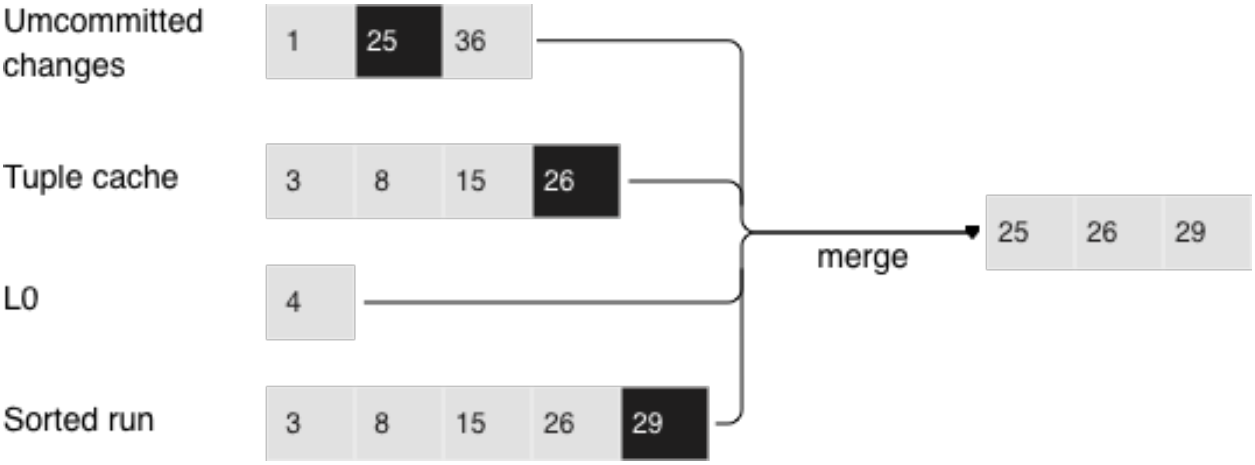
При поиске в LSM-дереве нам необходимо найти не сам элемент, а последнюю операцию с ним. Если это операция удаления, искомый элемент отсутствует в дереве. Если это операция вставки, то искомому элементу соответствует самое верхнее значение в LSM-пирамиде, и поиск можно остановить при первом совпадении ключа. В худшем случае значение в дереве изначально отсутствовало. Тогда поиск вынужден последовательно перебрать все уровни дерева, начиная с L0.



К сожалению, на практике этот худший случай довольно распространен. Например, при вставке в дерево необходимо убедиться в отсутствии дубликатов для первичного или уникального ключа. Поэтому для ускорения поиска несуществующих значений в LSM-деревьях применяется вероятностная структура данных, которая называется «фильтр Блума». О нем мы поговорим более детально в разделе, посвященном внутреннему устройству vinyl.

Поиск по диапазону

Если при поиске по одному ключу алгоритм завершается после первого совпадения, то для поиска всех значений в диапазоне (например, всех пользователей с фамилией «Иванов») необходимо просматривать все уровни дерева.



Поиск по диапазону [24,30)

Формирование искомого диапазона при этом происходит так же, как и при слиянии нескольких файлов: из всех источников алгоритм выбирает ключ с максимальным LSN, отбрасывает остальные операции по этому ключу, сдвигает позицию поиска на следующий ключ и повторяет процедуру.

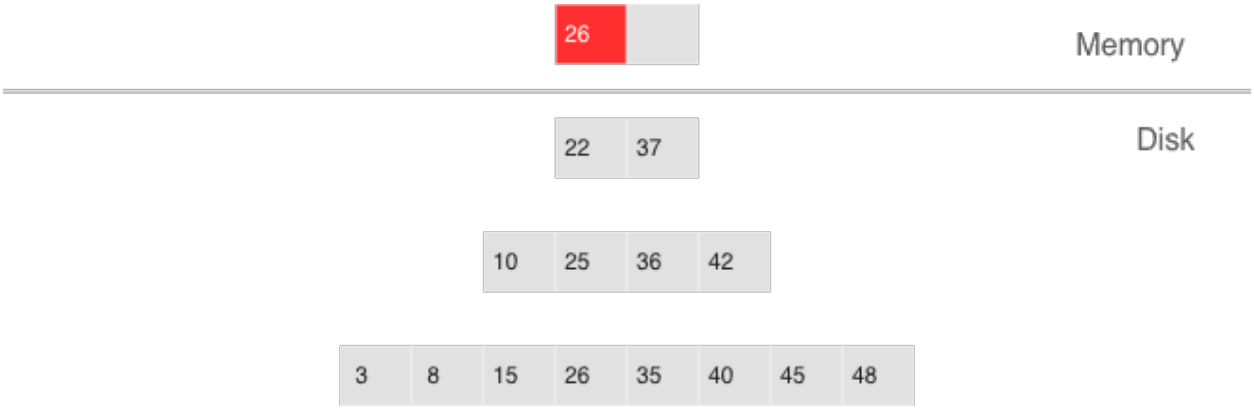
Удаление

Зачем вообще хранить операции удаления? И почему это не приводит к переполнению дерева, например, в сценарии for i=1,10000000 put(i) delete(i) end?

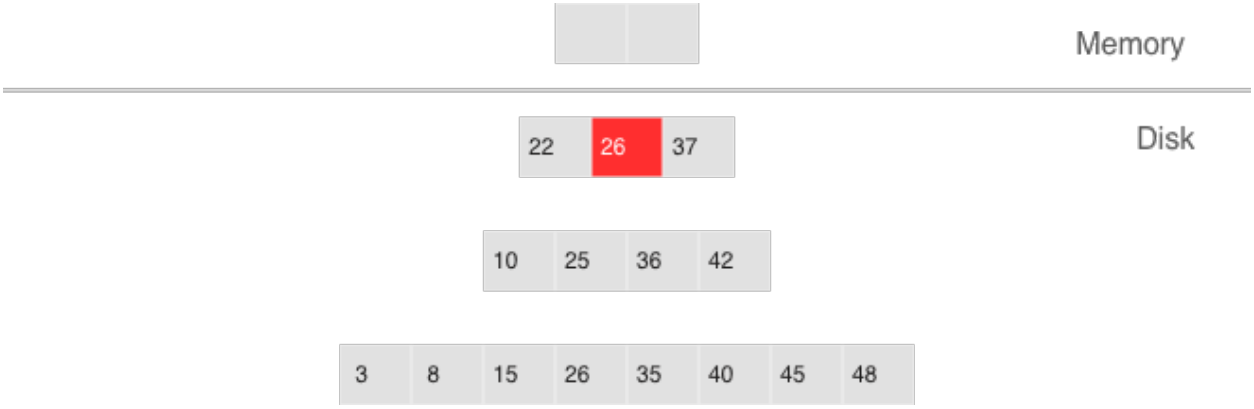
Роль операций удаления при поиске – сообщать об отсутствии искомого значения, а при слиянии – очищать дерево от «мусорных» записей с более старыми LSN.

Пока данные хранятся только в оперативной памяти, нет необходимости хранить операции удаления. Также нет необходимости сохранять операции удаления после слияния, если оно затрагивает в том числе самый нижний уровень дерева – на нем находятся данные самого старого дампа. Действительно, отсутствие значения на последнем уровне означает, что оно отсутствует в дереве.

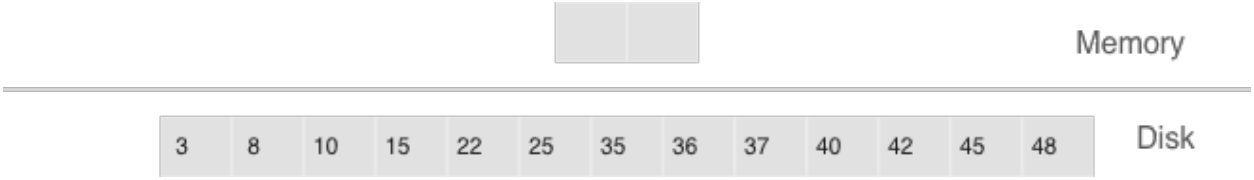
- Нельзя производить удаление из файлов, которые обновляются только путем присоединения новых записей
- Вместо этого на уровень L0 вносятся маркеры удаленных записей (tombstones)



Удаление, шаг 1: вставка удаленной записи в L0



Удаление, шаг 2: удаленная запись проходит через промежуточные уровни



Удаление, шаг 3: при значительном слиянии удаленная запись удаляется из дерева

Если мы знаем, что удаление следует сразу за вставкой уникального значения – а это частый случай при изменении значения во вторичном индексе – то операцию удаления можно отфильтровывать уже при слиянии промежуточных уровней. Эта оптимизация реализована в vinyl'е.

Преимущества LSM-дерева

Помимо снижения «паразитной» записи, подход с периодическими дампами уровня L0 и слиянием уровней L1-Lk имеет ряд преимуществ перед подходом к записи, используемым в B-деревьях:

- При дампах и слиянии создаются относительно большие файлы: стандартный размер L0 составляет 50-100 МБ, что в тысячи раз превышает размер блока B-дерева.
- Большой размер позволяет эффективно сжимать данные перед записью. В Tarantool'е сжатие происходит автоматически, что позволяет еще больше уменьшить «паразитную» запись.
- Издержки фрагментации отсутствуют, потому что в файле элементы следуют друг за другом без пустот/заполнений.
- Все операции создают новые файлы, а не заменяют старые данные. Это позволяет избавиться от столь ненавистных нам блокировок, при этом несколько операций могут идти параллельно, не приводя к конфликтам. Это также упрощает создание резервных копий и перенос данных на реплику.
- Хранение старых версий данных позволяет эффективно реализовать поддержку транзакций, используя подход управления параллельным доступом с помощью многоверсионности.

Недостатки LSM-дерева и их устранение

Одним из ключевых преимуществ B-дерева как структуры данных для поиска является предсказуемость: любая операция занимает не более чем $log_B(N)$. В классическом LSM-дереве скорость как чтения, так и записи могут может отличаться в лучшем и худшем случае в сотни и тысячи раз. Например, добавление всего лишь одного элемента в L0 может привести к его переполнению, что в свою очередь, может привести к переполнению L1, L2 и т.д. Процесс чтения может обнаружить исходный элемент в L0, а может задействовать все уровни. Чтение в пределах одного уровня также необходимо оптимизировать, чтобы добиться скорости, сравнимой с B-деревом. К счастью, многие недостатки можно скрасить или полностью устранить с помощью вспомогательных алгоритмов и структур данных. Систематизируем эти недостатки и опишем способы борьбы с ними, используемые в Tarantool'е.

Непредсказуемая скорость записи

Вставка данных в LSM-дерево почти всегда задействует исключительно L0. Как избежать простоя, если заполнена область оперативной памяти, отведенная под L0?

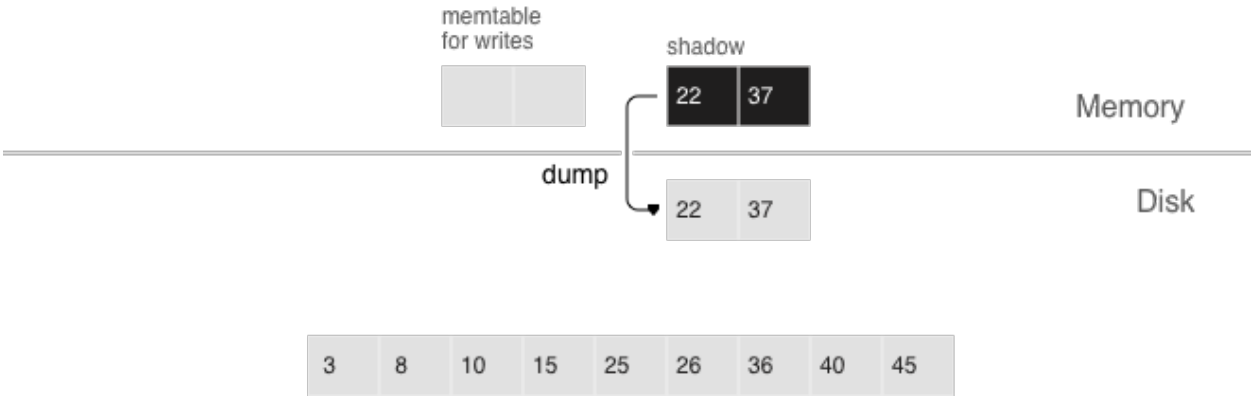
Освобождение L0 подразумевает две долгих операции: запись на диск и освобождение памяти. Чтобы избежать простоя во время записи L0 на диск, Tarantool использует упреждающую запись. Допустим, размер L0 составляет 256 МБ. Скорость записи на диск составляет 10 МБ/с. Тогда для записи L0 на диск понадобится 26 секунд. Скорость вставки данных составляет 10 000 запросов в секунду, а размер одного ключа – 100 байтов. На время записи необходимо зарезервировать около 26 МБ доступной оперативной памяти, сократив реальный полезный размер L0 до 230 МБ.

Все эти расчеты Tarantool делает автоматически, постоянно поддерживая скользящее среднее значение нагрузки на СУБД и гистограмму скорости работы диска. Это позволяет максимально эффективно использовать L0 и избежать истечения времени ожидания доступной памяти для операций записи.

При резком всплеске нагрузки ожидание все же возможно, поэтому также существует время ожидания операции вставки (параметр `vinyl_timeout`), значение которого по умолчанию составляет 60 секунд. Сама запись осуществляется в выделенных потоках, число которых (2 по умолчанию) задается в параметре `vinyl_write_threads`. Используемое по умолчанию значение 2 позволяет выполнять дамп параллельно со слиянием, что также необходимо для предсказуемой работы системы.

Слияния в Tarantool'е всегда выполняются независимо от дампов, в отдельном потоке выполнения. Это возможно благодаря природе LSM-дерева – после записи файлы в дереве никогда не меняются, а слияние лишь создает новый файл.

К задержкам также может приводить ротация L0 и освобождение памяти, записанной на диск: в процессе записи памятью L0 владеют два потока операционной системы – поток обработки транзакций и поток записи. Хотя в L0 во время ротации элементы не добавляются, он может участвовать в поиске. Чтобы избежать блокировок на чтение во время поиска, поток записи не освобождает записанную память, а оставляет эту задачу потоку обработки транзакций. Само освобождение после завершения дампа происходит мгновенно: для этого в L0 используется специализированный механизм распределения, позволяющий освободить всю память за одну операцию.

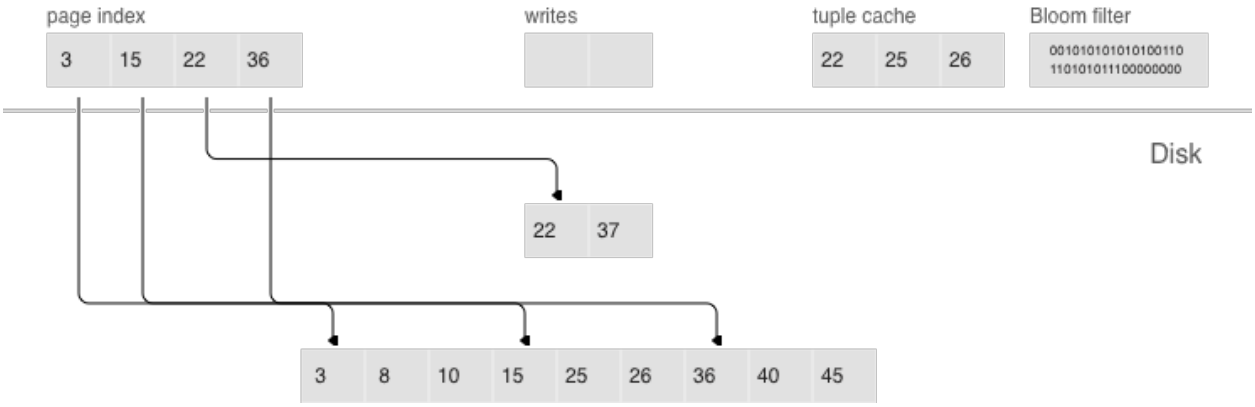


- упреждающий дамп
- загрузка

Дамп происходит из так называемого «теневого» L0, не блокируя новые вставки и чтения

Непредсказуемая скорость чтений

Чтение – самая сложная задача для оптимизации в LSM-деревьях. Главным фактором сложности является большое количество уровней: это не только значительно замедляет поиск, но и потенциально значительно увеличивает требования к оперативной памяти при почти любых попытках оптимизации. К счастью, природа LSM-деревьев, где файлы обновляются только путем присоединения новых записей, позволяет решать эти проблемы нестандартными для традиционных структур данных способами.



- постраничный индекс
- фильтры Блума
- кэш диапазона кортежей
- многоуровневое слияние

Сжатие и постраничный индекс

Сжатие данных в B-деревьях – это либо сложнейшая в реализации задача, либо больше средство маркетинга, чем действительно полезный инструмент. Сжатие в LSM-деревьях работает следующим образом:

При любом дампе или слиянии мы разбиваем все данные в одном файле на страницы. Размер страницы в байтах задается в параметре `vinyl_page_size`, который можно менять отдельно для каждого индекса. Страница не обязана занимать строго то количество байт, которое прописано `vinyl_page_size` – она может быть чуть больше или чуть меньше, в зависимости от хранящихся в ней данных. Благодаря этому страница никогда не содержит пустот.

Для сжатия используется потоковый алгоритм Facebook под названием «zstd». Первый ключ каждой страницы и смещение страницы в файле добавляются в так называемый постраничный индекс (page index) – отдельный файл, который позволяет быстро найти нужную страницу. После дампа или слияния постраничный индекс созданного файла также записывается на диск.

Все файлы типа *.index* кэшируются в оперативной памяти, что позволяет найти нужную страницу за одно чтение из файла *.run* (такое расширение имени файла используется в vinyl'е для файлов, полученных в результате дампа или слияния). Поскольку данные в странице отсортированы, после чтения и декомпрессии нужный ключ можно найти с помощью простого бинарного поиска. За чтение и декомпрессию отвечают отдельные потоки, их количество определяется в параметре `vinyl_read_threads`.

Tarantool использует единый формат файлов: например, формат данных в файле *.run* ничем не отличается от формата файла *.xlog* (файл журнала). Это упрощает резервное копирование и восстановление, а также работу внешних инструментов.

Фильтры Блума

Хотя постраничный индекс позволяет уменьшить количество страниц, просматриваемых при поиске в одном файле, он не отменяет необходимости искать на всех уровнях дерева. Есть важный частный случай, когда необходимо проверить отсутствие данных, и тогда просмотр всех уровней неизбежен: вставка в уникальный индекс. Если данные уже существуют, то вставка в уникальный индекс должна завершиться с ошибкой. Единственный способ вернуть ошибку до завершения транзакции в LSM-дереве –

произвести поиск перед вставкой. Такого рода чтения в СУБД образуют целый класс, называемый «скрытыми» или «паразитными» чтениями.

Другая операция, приводящая к скрытым чтениям, – обновление значения, по которому построен вторичный индекс. Вторичные ключи представляют собой обычные LSM-деревья, в которых данные хранятся в другом порядке. Чаще всего, чтобы не хранить все данные во всех индексах, значение, соответствующее данному ключу, целиком сохраняется только в первичном индексе (любой индекс, хранящий и ключ, и значение, называется покрывающим или кластерным), а во вторичном индексе сохраняются лишь поля, по которым построен вторичный индекс, и значения полей, участвующих в первичном индексе. Тогда при любом изменении значения, по которому построен вторичный ключ, приходится сначала удалять из вторичного индекса старый ключ, и только потом вставлять новый. Старое значение во время обновления неизвестно – именно его и нужно читать из первичного ключа с точки зрения внутреннего устройства.

Например:

```
update t1 set city='Moscow' where id=1
```

Чтобы уменьшить количество чтений с диска, особенно для несуществующих значений, практически все LSM-деревья используют вероятностные структуры данных. Tarantool не исключение. Классический фильтр Блума – это набор из нескольких (обычно 3-5) битовых массивов. При записи для каждого ключа вычисляется несколько хеш-функций, и в каждом массиве выставляется бит, соответствующий значению хеша. При хешировании могут возникнуть коллизии, поэтому некоторые биты могут быть проставлены дважды. Интерес представляют биты, которые оказались не проставлены после записи всех ключей. При поиске также вычисляются выбранные хеш-функции. Если хотя бы в одном из битовых массивов бит не стоит, то значение в файле отсутствует. Вероятность срабатывания фильтра Блума определяется теоремой Байеса: каждая хеш-функция представляет собой независимую случайную величину, благодаря чему вероятность того, что во всех битовых массивах одновременно произойдет коллизия, очень мала.

Ключевым преимуществом реализации фильтров Блума в Tarantool'е является простота настройки. Единственный параметр, который можно менять независимо для каждого индекса, называется `vinyl_bloom_fpr` (FPR в данном случае означает сокращение от «false positive ratio» – коэффициент ложноположительного срабатывания), который по умолчанию равен 0,05, или 5%. На основе этого параметра Tarantool автоматически строит фильтры Блума оптимального размера для поиска как по полному ключу, так и по компонентам ключа. Сами фильтры Блума хранятся вместе с постраничным индексом в файле *.index* и кэшируются в оперативной памяти.

Кэширование

Многие привыкли считать кэширование панацеей от всех проблем с производительностью: «В любой непонятной ситуации добавляй кэш». В vinyl'е мы смотрим на кэш скорее как на средство снижения общей нагрузки на диск, и, как следствие, получения более предсказуемого времени ответов на запросы, которые не попали в кэш. В vinyl'е реализован уникальный для транзакционных систем вид кэша под названием «кэш диапазона кортежей» (range tuple cache). В отличие от RocksDB, например, или MySQL, этот кэш хранит не страницы, а уже готовые диапазоны значений индекса, после их чтения с диска и слияния всех уровней. Это позволяет использовать кэш для запросов как по одному ключу, так и по диапазону ключей. Поскольку в кэше хранятся только горячие данные, а не, скажем, страницы (в странице может быть востребована лишь часть данных), оперативная память используется наиболее оптимально. Размер кэша задается в параметре `vinyl_cache`.

Управление сборкой мусора

Возможно, добравшись до этого места вы уже начали терять концентрацию и нуждаетесь в заслуженной дозе допамина. Самое время сделать перерыв, так как для того, чтобы разобраться с оставшейся частью, понадобятся серьезные усилия.

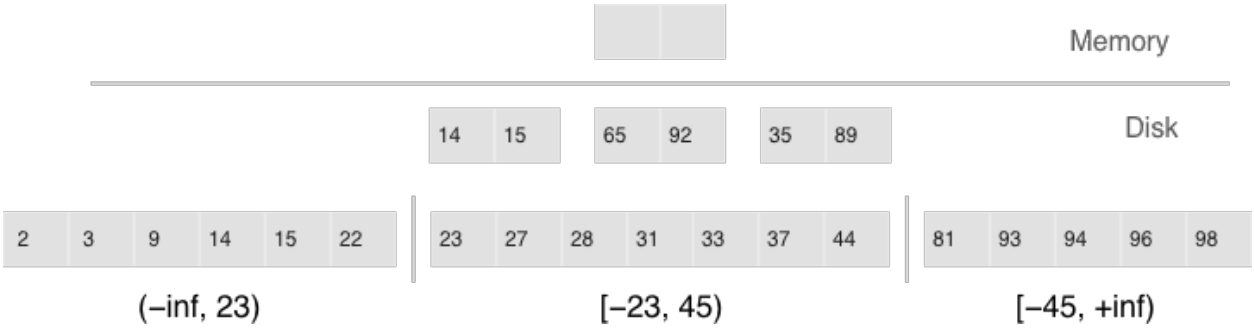
В vinyl'е устройство одного LSM-дерева – это лишь фрагмент мозаики. Vinyl создает и обслуживает несколько LSM-деревьев даже для одной таблицы (так называемого спейса) – по одному дереву на каждый индекс. Но даже один единственный индекс может состоять из десятков LSM-деревьев. Попробуем разобраться, зачем.

Рассмотрим наш стандартный пример: 100 000 000 записей по 100 байтов каждая. Через некоторое время на самом нижнем уровне LSM у нас может оказаться файл размером 10 ГБ. Во время слияния последнего уровня мы создадим временный файл, который также будет занимать около 10 ГБ. Данные на промежуточных уровнях тоже занимают место: по одному и тому же ключу дерево может хранить несколько операций. Суммарно для хранения 10 ГБ полезных данных нам может потребоваться до 30 ГБ свободного места: 10 ГБ на последний уровень, 10 ГБ на временный файл и 10 ГБ на всё остальное. А если данных не 1 ГБ, а 1 ТБ? Требовать, чтобы количество свободного места на диске всегда в несколько раз превышало объем полезных данных, экономически нецелесообразно, да и создание файла в 1ТБ может занимать десятки часов. При любой аварии или перезапуске системы операцию придется начинать заново.

Рассмотрим другую проблему. Представим, что первичный ключ дерева – это монотонная последовательность, например, временной ряд. В этом случае основные вставки будут приходиться на правую часть диапазона ключей. Нет смысла заново производить слияние лишь для того, чтобы дописать в конец и без того огромного файла еще несколько миллионов записей.

А если вставки происходят, в основном, в одну часть диапазона ключей, а чтения – из другой части? Как в этом случае оптимизировать форму дерева? Если оно будет слишком высоким, пострадают чтения, если слишком низким – запись.

Tarantool «факторизует» проблему, создавая не одно, а множество LSM-деревьев для каждого индекса. Примерный размер каждого поддерева можно задать в конфигурационном параметре `vinyl_range_size`. Такие поддеревья называется диапазонами («range»).



Факторизация больших LSM-деревьев с помощью диапазонов

- Диапазоны отражают статичную структуру упорядоченных файлов
- Срезы объединяют упорядоченный файл в диапазон

Изначально, пока в индексе мало элементов, он состоит из одного диапазона. По мере добавления элементов суммарный объем может превысить *максимальный размер диапазона*. В таком случае выполняется операция под названием «разделение» (split), которая делит дерево на две равные части. Разделение происходит по срединному элементу диапазона ключей, хранящихся в дереве. Например, если изначально дерево хранит полный диапазон -inf... +inf, то после разделения по срединному ключу X получим два поддерева: одно будет хранить все ключи от -inf до X, другое – от X до +inf. Таким

образом, при вставке или чтении мы однозначно знаем, к какому поддереву обращаться. Если в дереве были удаления и каждый из соседних диапазонов уменьшился, выполняется обратная операция под названием «объединение» (coalesce). Она объединяет два соседних дерева в одно.

Разделение и объединение не приводят к слиянию, созданию новых файлов и прочим тяжеловесным операциям. LSM-дерево – это лишь набор файлов. В vinyl'е мы реализовали специальный журнал метаданных, позволяющий легко отслеживать, какой файл принадлежит какому поддереву или поддеревьям. Журнал имеет расширение *.vylog*, по формату он совместим с файлом *.xlog*. Как и файл *.xlog*, происходит автоматическая ротация файла при каждой контрольной точке. Чтобы избежать повторного создания файлов при разделении и объединении, мы ввели промежуточную сущность – срез (slice). Это ссылка на файл с указанием диапазона значений ключа, которая хранится исключительно в журнале метаданных. Когда число ссылок на файл становится равным нулю, файл удаляется. А когда необходимо произвести разделение или объединение, Tarantool создает срезы для каждого нового дерева, старые срезы удаляет, и записывает эти операции в журнал метаданных. Буквально, журнал метаданных хранит записи вида <идентификатор дерева, идентификатор среза> или <идентификатор среза, идентификатор файла, мин, макс>.

Таким образом, непосредственно тяжелая работа по разбиению дерева на два поддерева, откладывается до слияния и выполняется автоматически. Огромным преимуществом подхода с разделением всего диапазона ключей на диапазоны является возможность независимо управлять размером L0, а также процессом создания дампов и слиянием для каждого поддерева. В результате эти процессы являются управляемыми и предсказуемыми. Наличие отдельного журнала метаданных также упрощает выполнение таких операций, как усечение и удаление – в vinyl'е они обрабатываются мгновенно, потому что работают исключительно с журналом метаданных, а удаление мусора выполняется в фоне.

Расширенные возможности vinyl'а

Upsert (обновление и вставка)

В предыдущих разделах упоминались лишь две операции, которые хранит LSM-дерево: удаление и замена. Давайте рассмотрим, как представлены все остальные. Вставку можно представить с помощью замены – необходимо лишь предварительно убедиться в отсутствии элемента указанным ключом. Для выполнения обновления необходимо предварительно считывать старое значение из дерева, так что и эту операцию проще записать в дерево как замену – это ускорит будущие чтения по этому ключу. Кроме того, обновление должно вернуть новое значение, так что скрытых чтений никак не избежать.

В B-деревьях скрытые чтения почти ничего не стоят: чтобы обновить блок, его в любом случае необходимо прочитать с диска. Для LSM-деревьев идея создания специальной операции обновления, которая не приводила бы к скрытым чтениям, выглядит очень заманчивой.

Такая операция должна содержать как значение по умолчанию, которое нужно вставить, если данных по ключу еще нет, так и список операций обновления, которые нужно выполнить, если значение существует.

На этапе выполнения транзакции Tarantool лишь сохраняет всю операцию в LSM-дереве, а «выполняет» ее уже только во время слияния.

Операция обновления и вставки:

```
space:upsert(tuple, {{operator, field, value}, ... })
```

- Обновление без чтения или вставка
- Отложенное выполнение
- Фоновое сжатие операций обновления и вставки предотвращает накапливание операций

К сожалению, если откладывать выполнение операции на этап слияния, возможностей для обработки ошибок не остается. Поэтому Tarantool стремится максимально проверять операции обновления и вставки upsert перед записью в дерево. Тем не менее, некоторые проверки можно выполнить лишь имея старые данные на руках. Например, если обновление прибавляет число к строке или удаляет несуществующее поле.

Операция с похожей семантикой присутствует во многих продуктах, в том числе в PostgreSQL и MongoDB. Но везде она представляет собой лишь синтаксический сахар, объединяющий обновление и вставку, не избавляя СУБД от необходимости выполнять скрытые чтения. Скорее всего, причиной этого является относительная новизна LSM-деревьев в качестве структур данных для хранения.

Хотя обновление и вставка upsert представляет собой очень важную оптимизацию, и ее реализация стоила нам долгой напряженной работы, следует признать, что ее применимость ограничена. Если в таблице есть вторичные ключи или триггеры, скрытых чтений не избежать. А если у вас есть сценарии, для которых не нужны вторичные ключи и обновление после завершения транзакции однозначно не приведет к ошибкам – эта операция для вас.

Небольшая история, связанная с этим оператором: vinyl только начинал «взрослеть», и мы впервые запустили операцию обновления и вставки upsert на рабочие серверы. Казалось бы, идеальные условия: огромный набор ключей, текущее время в качестве значения, операции обновления либо вставляют ключ, либо обновляют текущее время, редкие операции чтения. Нагрузочные тесты показали отличные результаты.

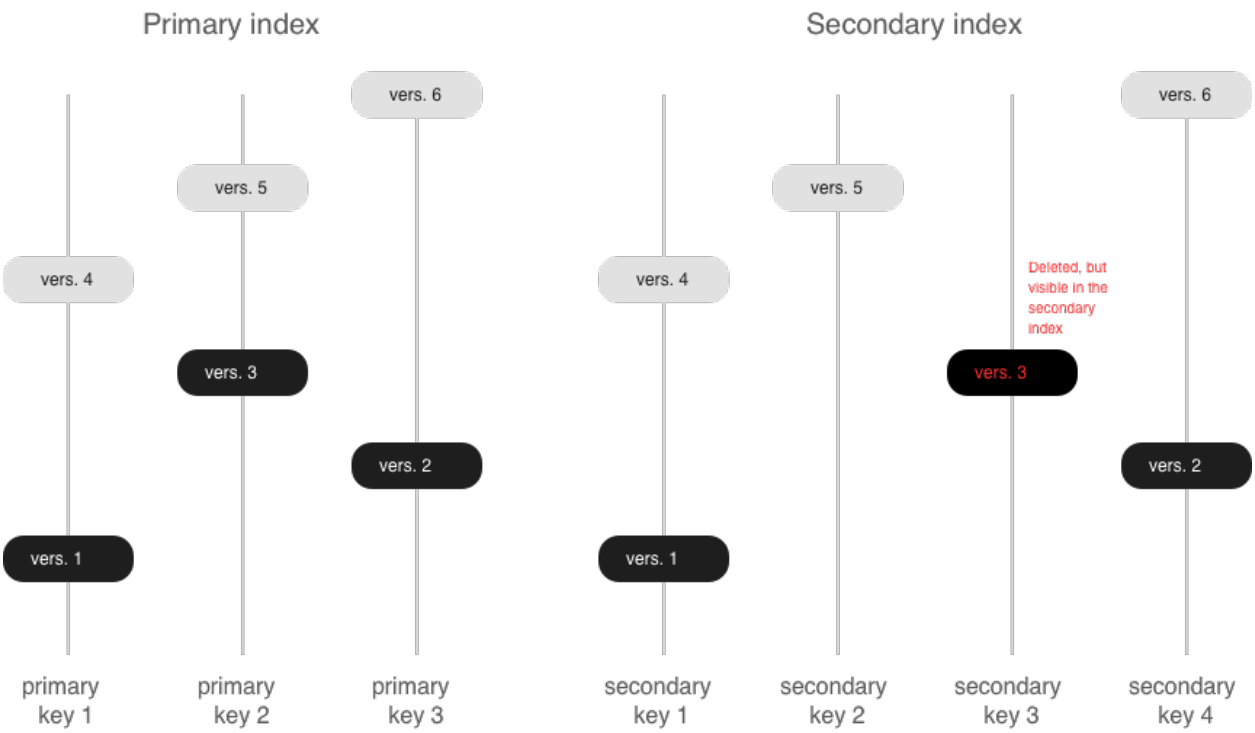
Тем не менее, после пары дней работы процесс Tarantool'а начал потреблять 100 % CPU, а производительность системы упала практически до нуля.

Начали подробно изучать проблему. Оказалось, что распределение запросов по ключам существенно отличалось от того, что мы видели в тестовом окружении. Оно было... очень неравномерное. Большая часть ключей обновлялась 1-2 раза за сутки, и база для них не была нагружена. Но были ключи гораздо более горячие – десятки тысяч обновлений в сутки. Tarantool прекрасно справлялся с этим потоком обновлений. А вот когда по ключу с десятком тысяч операций обновления и вставки upsert происходило чтение, всё шло под откос. Чтобы вернуть последнее значение, Tarantool'у приходилось каждый раз прочитать и «проиграть» историю из десятков тысяч команд обновления и вставки upsert. На стадии проекта мы надеялись, что это произойдет автоматически во время слияния уровней, но до слияния дело даже не доходило: памяти L0 было предостаточно, и дампы не создавались.

Решили мы проблему добавлением фонового процесса, осуществляющего упреждающие чтения для ключей, по которым накопилось больше нескольких десятков операций обновления и вставки upsert с последующей заменой на прочитанное значение.

Вторичные ключи

Не только для операции обновления остро стоит проблема оптимизации скрытых чтений. Даже операция замены при наличии вторичных ключей вынуждена читать старое значение: его нужно независимо удалить из вторичных индексов, а вставка нового элемента может этого не сделать, оставив в индексе мусор.



Если вторичные индексы не уникальны, то удаление из них «мусора» также можно перенести в фазу слияния, что мы и делаем в Tarantool'е. Природа LSM-дерева, в котором файлы обновляются путем присоединения новых записей, позволила нам реализовать в vinyl'е полноценные сериализуемые транзакции. Запросы только для чтения при этом используют старые версии данных и не блокируют запись. Сам менеджер транзакций пока довольно простой: в традиционной классификации он реализует класс MVTO (multiversion timestamp ordering – упорядочение временных меток на основе многоверсионности), при этом в конфликте побеждает та транзакция, что завершилась первой. Блокировок и свойственных им взаимоблокировок нет. Как ни странно, это скорее недостаток, чем преимущество: при параллельном выполнении можно повысить количество успешных транзакций, задерживая некоторые из них в нужный момент на блокировке. Развитие менеджера транзакций в наших ближайших планах. В текущей версии мы сфокусировались на том, чтобы сделать алгоритм корректным и предсказуемым на 100%. Например, наш менеджер транзакций – один из немногих в NoSQL-среде, поддерживающих так называемые «блокировки разрывов» (gap locks).

4.3 Tarantool Cartridge

Cluster management in Tarantool is powered by the Tarantool Cartridge framework.

Here we explain how you can benefit with Tarantool Cartridge, a framework for developing, deploying, and managing applications based on Tarantool.

This documentation contains the following sections:

4.3.1 Tarantool Cartridge

A framework for distributed applications development.

Содержание

- *Tarantool Cartridge*
  - *About Tarantool Cartridge*
  - *Getting started*
    - *Prerequisites*
    - *Create your first application*
    - *Next steps*
  - *Contributing*

About Tarantool Cartridge

Tarantool Cartridge allows you to easily develop Tarantool-based applications and run them on one or more Tarantool instances organized into a cluster.

This is the recommended alternative to the old-school practices of application development for Tarantool.

As a **software development kit (SDK)**, Tarantool Cartridge provides you with utilities and an application template to help:

- easily set up a development environment for your applications;
- plug the necessary Lua modules.

The resulting package can be installed and started on one or multiple servers as one or multiple instantiated services – independent or organized into a **cluster**.

A Tarantool cluster is a collection of Tarantool instances acting in concert. While a single Tarantool instance can leverage the performance of a single server and is vulnerable to failure, the cluster spans multiple servers, utilizes their cumulative CPU power, and is fault-tolerant.

To fully utilize the capabilities of a Tarantool cluster, you need to develop applications keeping in mind they are to run in a cluster environment.

As a **cluster management tool**, Tarantool Cartridge provides your cluster-aware applications with the following key benefits:

- horizontal scalability and load balancing via built-in automatic sharding;
- asynchronous replication;
- automatic failover;
- centralized cluster control via GUI or API;
- automatic configuration synchronization;
- instance functionality segregation.

A Tarantool Cartridge cluster can segregate functionality between instances via built-in and custom (user-defined) **cluster roles**. You can toggle instances on and off on the fly during cluster operation. This allows you to put different types of workloads (e.g., compute- and transaction-intensive ones) on different physical servers with dedicated hardware.

Tarantool Cartridge has an external utility called cartridge-cli which provides you with utilities and an application template to help:

- easily set up a development environment for your applications;

- plug the necessary Lua modules;
- pack the applications in an environment-independent way: together with module binaries and Tarantool executables.

Getting started

Prerequisites

To get a template application that uses Tarantool Cartridge and run it, you need to install several packages:

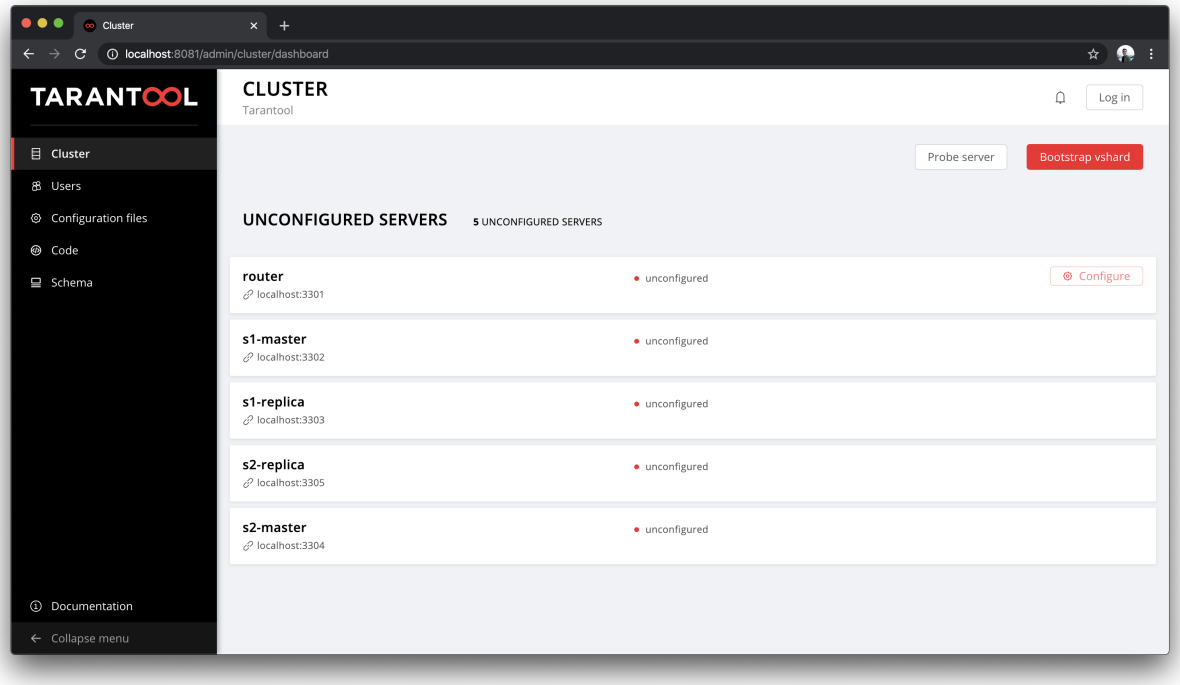
- `tarantool` and `tarantool-dev` (see these instructions);
- `cartridge-cli` (see these instructions)
- `git`, `gcc`, `cmake` and `make`.

Create your first application

Long story short, copy-paste this into the console:

```
cartridge create --name myapp
cd myapp
cartridge build
cartridge start
```

That's all! Now you can visit `http://localhost:8081` and see your application's Admin Web UI:



Next steps

See:

- A more detailed getting started guide
- More application examples
- Cartridge documentation
- Cartridge API reference

Contributing

The most essential contribution is your *feedback*, don't hesitate to open an issue. If you'd like to propose some changes in code, see the contribution guide.

4.3.2 Developer's guide

For a quick start, skip the details below and jump right away to the Cartridge getting started guide.

For a deep dive into what you can develop with Tarantool Cartridge, go on with the Cartridge developer's guide.

Introduction

Короче говоря, чтобы разработать и запустить приложение, вам необходимо выполнить следующие шаги:

1. *Install* Tarantool Cartridge and other components of the development environment.
2. *Create a project*.
3. Разработать приложение. Если это приложение с поддержкой кластеров, реализуйте его логику в виде отдельной (пользовательской) *кластерной роли*, чтобы инициализировать базу данных в кластерной среде.
4. *Развернуть* приложение на сервере или серверах. Это включает в себя настройку и *запуск* экземпляров.
5. Если это приложение с поддержкой кластеров, *развернуть кластер*.

В следующих разделах подробно описывается каждый из этих шагов.

Установка Tarantool Cartridge

1. Install `cartridge-cli`, a command-line tool for developing, deploying, and managing Tarantool applications.
2. Install `git`, a version control system.
3. Install `npm`, a package manager for `node.js`.
4. Install the `unzip` utility.

Creating a project

To set up your development environment, create a project using the Tarantool Cartridge project template. In any directory, say:

```
$ cartridge create --name <app_name> /path/to/
```

This will automatically set up a Git repository in a new `/path/to/<app_name>/` directory, tag it with *version* `0.1.0`, and put the necessary files into it.

В этом Git-репозитории можно разработать приложение (просто редактируя файлы из шаблона), подключить необходимые модули, а затем с легкостью упаковать все для развертывания на своих серверах.

The project template creates the `<app_name>/` directory with the following contents:

- файл `<имя_приложения>-scm-1.rockspec`, где можно указать зависимости приложения.
- скрипт `deps.sh`, который решает проблемы с зависимостями из файла `.rockspec`.
- файл `init.lua`, который является точкой входа в ваше приложение.
- файл `.git`, необходимый для Git-репозитория.
- файл `.gitignore`, чтобы не учитывать ненужные файлы.
- файл `env.lua`, который устанавливает общие пути для модулей, чтобы приложение можно было запустить из любой директории.
- файл `custom-role.lua`, который представляет собой объект-заполнитель для пользовательской *кластерной роли*.

The entry point file (`init.lua`), among other things, loads the `cartridge` module and calls its initialization function:

```
...
local cartridge = require('cartridge')
...
cartridge.cfg({
-- cartridge options example
  workdir = '/var/lib/tarantool/app',
  advertise_uri = 'localhost:3301',
  cluster_cookie = 'super-cluster-cookie',
  ...
}, {
-- box options example
  memtx_memory = 1000000000,
  ... })
 ...
```

Вызов `cartridge.cfg()` позволяет управлять экземпляром через административную консоль, но не вызывает `box.cfg()` для настройки экземпляров.

Предупреждение: Запрещается вызывать функцию `box.cfg()`.

Сам кластер сделает это за вас, когда придет время:

- загрузить текущий экземпляр, когда вы:
  - выполните `cartridge.bootstrap()` в административной консоли, или
  - нажмете **Create** (Создать) в веб-интерфейсе;

- присоединить экземпляр к существующему кластеру, когда вы:
  - выполните `cartridge.join_server({uri = ''uri_другого_экземпляра'})` в консоли, или
  - нажмете **Join** (Присоединить – к уже существующему набору реплик) или **Create** (Создать – для нового набора реплик) в веб-интерфейсе.

Обратите внимание, что вы можете указать cookie для кластера (параметр `cluster_cookie`), если необходимо запустить несколько кластеров в одной сети. Cookie может представлять собой любое строковое значение.

Now you can develop an application that will run on a single or multiple independent Tarantool instances (e.g. acting as a proxy to third-party databases) – or will run in a cluster.

If you plan to develop a cluster-aware application, first familiarize yourself with the notion of *cluster roles*.

Кластерные роли

Cluster roles are Lua modules that implement some specific functions and/or logic. In other words, a Tarantool Cartridge cluster segregates instance functionality in a role-based way.

Since all instances running cluster applications use the same source code and are aware of all the defined roles (and plugged modules), you can dynamically enable and disable multiple different roles without restarts, even during cluster operation.

Note that every instance in a replica set performs the same roles and you cannot enable/disable roles individually on some instances. In other words, configuration of enabled roles is set up *per replica set*. See a step-by-step configuration example in *this guide*.

Встроенные роли

В модуль `cartridge` входят две *встроенные* роли, которые реализуют автоматический шардинг:

- `vshard-router` обрабатывает *ресурсоемкие* вычисления в `vshard`: направляет запросы к узлам хранения данных.
- `vshard-storage` работает с *большим количеством транзакций* в `vshard`: хранит подмножество набора данных и управляет им.

Примечание: For more information on sharding, see the vshard module documentation.

With the built-in and *custom roles*, you can develop applications with separated compute and transaction handling – and enable relevant workload-specific roles on different instances running on physical servers with workload-dedicated hardware.

Пользовательские роли

You can implement custom roles for any purposes, for example:

- определять хранимые процедуры;
- implement extra features on top of `vshard`;
- полностью обойтись без `vshard`;
- внедрить одну или несколько дополнительных служб, таких как средство уведомления по электронной почте, репликатор и т.д.

To implement a custom cluster role, do the following:

1. Take the `app/roles/custom.lua` file in your project as a sample. Rename this file as you wish, e.g. `app/roles/custom-role.lua`, and implement the role's logic. For example:

```
-- Implement a custom role in app/roles/custom-role.lua
#!/usr/bin/env tarantool
local role_name = 'custom-role'

local function init()
...
end

local function stop()
...
end

return {
    role_name = role_name,
    init = init,
    stop = stop,
}
```

   Here the `role_name` value may differ from the module name passed to the `cartridge.cfg()` function. If the `role_name` variable is not specified, the module name is the default value.

Примечание: Имена ролей должны быть уникальными, поскольку невозможно зарегистрировать несколько ролей с одним именем.

---

2. Зарегистрируйте новую роль в кластере, изменив вызов `cartridge.cfg()` в файле входа в приложение `init.lua`:

```
-- Register a custom role in init.lua
...
local cartridge = require('cartridge')
...
cartridge.cfg({
  workdir = ...,
  advertise_uri = ...,
  roles = {'custom-role'},
})
...
```

   где `custom-role` (пользовательская роль) – это название загружаемого Lua-модуля.

The role module does not have required functions, but the cluster may execute the following ones during the *role's life cycle*:

- `init()` – это функция *инициализации* роли.

  Inside the function's body you can call any box functions: create spaces, indexes, grant permissions, etc. Here is what the initialization function may look like:

```
local function init(opts)
    -- The cluster passes an 'opts' Lua table containing an 'is_master' flag.
    if opts.is_master then
        local customer = box.schema.space.create('customer',
```

(continues on next page)

(продолжение с предыдущей страницы)

```
            { if_not_exists = true }
        )
        customer:format({
            {'customer_id', 'unsigned'},
            {'bucket_id', 'unsigned'},
            {'name', 'string'},
        })
        customer:create_index('customer_id', {
            parts = {'customer_id'},
            if_not_exists = true,
        })
    end
end
```

Примечание:

- Neither `vshard-router` nor `vshard-storage` manage spaces, indexes, or formats. You should do it within a *custom* role: add a `box.schema.space.create()` call to your first cluster role, as shown in the example above.
- Тело функции заключено в условный оператор, который позволяет вызывать функции `box` только на мастерах. Это предотвращает конфликты репликации, так как данные автоматически передаются на реплики.

- `stop()` is the role's *termination* function. Implement it if initialization starts a fiber that has to be stopped or does any job that needs to be undone on termination.
- `validate_config()` and `apply_config()` are functions that *validate* and *apply* the role's configuration. Implement them if some configuration data needs to be stored cluster-wide.

Next, get a grip on the *role's life cycle* to implement the functions you need.

Определение зависимостей для ролей

Можно заставить кластер применить некоторые другие роли, если включена пользовательская роль.

Например:

```
-- Role dependencies defined in app/roles/custom-role.lua
local role_name = 'custom-role'
...
return {
    role_name = role_name,
    dependencies = {'cartridge.roles.vshard-router'},
    ...
}
```

Здесь роль `vshard-router` будет инициализирована автоматически для каждого экземпляра, в котором включена роль `custom-role`.

Использование нескольких групп vshard storage

Для наборов реплик с ролью `vshard-storage` можно задавать *группы*. Например, группы `hot` и `cold` предназначены для независимой обработки горячих и холодных данных.

Группы указаны в конфигурации кластера:

```
-- Specify groups in init.lua
cartridge.cfg({
    vshard_groups = {'hot', 'cold'},
    ...
})
```

Если ни одна группа не указана, кластер предполагает, что все наборы реплик входят в группу `default` (по умолчанию).

Если включены несколько групп, каждый набор реплик с включенной ролью `vshard-storage` должен быть назначен в определенную группу. Эту настройку нельзя изменить впоследствии.

Есть еще одно ограничение – нельзя добавлять группы динамически (такая возможность появится в будущих версиях).

Finally, mind the syntax for router access. Every instance with a `vshard-router` role enabled initializes multiple routers. All of them are accessible through the role:

```
local router_role = cartridge.service_get('vshard-router')
router_role.get('hot'):call(...)
```

If you have no roles specified, you can access a static router as before (when Tarantool Cartridge was unaware of groups):

```
local vhsard = require('vshard')
vshard.router.call(...)
```

However, when using the current group-aware API, you must call a static router with a colon:

```
local router_role = cartridge.service_get('vshard-router')
local default_router = router_role.get() -- or router_role.get('default')
default_router:call(...)
```

Role's life cycle (and the order of function execution)

The cluster displays the names of all custom roles along with the built-in `vshard-*` roles in the *web interface*. Cluster administrators can enable and disable them for particular instances – either via the web interface or via the cluster public API. For example:

```
cartridge.admin.edit_replicaset('replicaset-uuid', {roles = {'vshard-router', 'custom-role'}})
```

If you enable multiple roles on an instance at the same time, the cluster first initializes the built-in roles (if any) and then the custom ones (if any) in the order the latter were listed in `cartridge.cfg()`.

If a custom role has dependent roles, the dependencies are registered and validated first, *prior* to the role itself.

Кластер вызывает функции роли в следующих случаях:

- Функция `init()` обычно выполняется один раз: либо когда администратор включает роль, либо при перезапуске экземпляра. Как правило, достаточно один раз включить роль.
- Функция `stop()` – только когда администратор отключает роль, а не во время завершения работы экземпляра.
- Функция `validate_config()`: сначала до автоматического вызова *box.cfg()*‘ (инициализация базы данных), а затем при каждом обновлении конфигурации.

- Функция `apply_config()` – при каждом обновлении конфигурации.

As a tryout, let's task the cluster with some actions and see the order of executing the role's functions:

- Присоединение экземпляра или создание набора реплик (в обоих случаях с включенной ролью):
  1. `validate_config()`
  2. `init()`
  3. `apply_config()`
- Перезапуск экземпляра с включенной ролью:
  1. `validate_config()`
  2. `init()`
  3. `apply_config()`
- Отключение роли: `stop()`.
- При вызове `cartridge.confapplier.patch_clusterwide()`:
  1. `validate_config()`
  2. `apply_config()`
- При запущенном восстановлении после отказа:
  1. `validate_config()`
  2. `apply_config()`

Учитывая вышеописанное поведение:

- Функция `init()` может:
  - Вызывать функции `box`.
  - Запускать файбер, и в таком случае функция `stop()` должна позаботиться о завершении работы файбера.
  - Настраивать встроенный *HTTP-сервер*.
  - Выполнять любой код, связанный с инициализацией роли.
- The `stop()` functions must undo any job that needs to be undone on role's termination.
- Функция `validate_config()` должна валидировать любые изменения конфигурации.
- Функция `apply_config()` может выполнять любой код, связанный с изменением конфигурации, например, следить за файбером `expirationd`.

The validation and application functions together allow you to change the cluster-wide configuration as described in the *next section*.

Конфигурация пользовательских ролей

Доступны следующие операции:

- Хранить настройки пользовательских ролей в виде разделов в конфигурации на уровне кластера, например:

```
# in YAML configuration file
my_role:
  notify_url: "https://localhost:8080"
```

```
-- in init.lua file
local notify_url = 'http://localhost'
function my_role.apply_config(conf, opts)
  local conf = conf['my_role'] or {}
  notify_url = conf.notify_url or 'default'
end
```

- Download and upload cluster-wide configuration using the *web interface* or API (via GET/PUT queries to `admin/config` endpoint like `curl localhost:8081/admin/config` and `curl -X PUT -d "{'my_parameter': 'value'}" localhost:8081/admin/config`).
- Utilize it in your role's `apply_config()` function.

Каждый экземпляр в кластере хранит копию конфигурационного файла в своей рабочей директории (которую можно задать с помощью `cartridge.cfg({workdir = ...})`):

- `/var/lib/tarantool/<instance_name>/config.yml` для экземпляров, развернутых из RPM-пакетов, под управлением `systemd`.
- `/home/<username>/tarantool_state/var/lib/tarantool/config.yml` for instances deployed from tar+gz archives.

The cluster's configuration is a Lua table, downloaded and uploaded as YAML. If some application-specific configuration data, e.g. a database schema as defined by DDL (data definition language), needs to be stored on every instance in the cluster, you can implement your own API by adding a custom section to the table. The cluster will help you spread it safely across all instances.

Such section goes in the same file with topology-specific and `vshard`-specific sections that the cluster generates automatically. Unlike the generated, the custom section's modification, validation, and application logic has to be defined.

Самый распространенный способ заключается в том, чтобы:

- `validate_config(conf_new, conf_old)` для валидации изменений, сделанных в новой конфигурации (`conf_new`) по отношению к старой конфигурации (`conf_old`).
- `apply_config(conf, opts)` для выполнения любого кода, связанного с изменениями конфигурации. Входными данными для этой функции будут применяемая конфигурация (`conf`, которая и есть новая конфигурация, проверенная чуть ранее с помощью `validate_config()`), а также параметры (аргумент `opts` включает в себя описываемый ниже логический флаг `is_master`).

Важно: Функция `validate_config()` должна обнаружить все проблемы конфигурации, которые могут привести к ошибкам `apply_config()`. Для получения дополнительной информации см. *следующий раздел*.

When implementing validation and application functions that call `box` ones for some reason, mind the following precautions:

- *Жизненный цикл роли* не предполагает, что кластер автоматически вызовет `box.cfg()` до вызова `validate_config()`.

  If the validation function calls any `box` functions (e.g., to check a format), make sure the calls are wrapped in a protective conditional statement that checks if `box.cfg()` has already happened:

```
-- Inside the validate_config() function:

if type(box.cfg) == 'table' then

    -- Here you can call box functions

end
```

- Unlike the validation function, `apply_config()` can call `box` functions freely as the cluster applies custom configuration after the automatic `box.cfg()` call.

  However, creating spaces, users, etc., can cause replication collisions when performed on both master and replica instances simultaneously. The appropriate way is to call such `box` functions *on masters only* and let the changes propagate to replicas automatically.

  По выполнении `apply_config(conf, opts)` кластер передает флаг `is_master` в таблице `opts`, который можно использовать для заключения функций из `box` в защитный условный оператор, если они могут вызвать конфликт:

```
-- Inside the apply_config() function:

if opts.is_master then

    -- Here you can call box functions

end
```

Пример пользовательской конфигурации

Рассмотрим следующий код как часть реализации модуля роли (`custom-role.lua`):

```
#!/usr/bin/env tarantool
-- Custom role implementation

local cartridge = require('cartridge')

local role_name = 'custom-role'

-- Modify the config by implementing some setter (an alternative to HTTP PUT)
local function set_secret(secret)
    local custom_role_cfg = cartridge.confapplier.get_deepcopy(role_name) or {}
    custom_role_cfg.secret = secret
    cartridge.confapplier.patch_clusterwide({
        [role_name] = custom_role_cfg,
    })
end
-- Validate
local function validate_config(cfg)
    local custom_role_cfg = cfg[role_name] or {}
    if custom_role_cfg.secret ~= nil then
        assert(type(custom_role_cfg.secret) == 'string', 'custom-role.secret must be a string')
    end
    return true
end
-- Apply
local function apply_config(cfg)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    local custom_role_cfg = cfg[role_name] or {}
    local secret = custom_role_cfg.secret or 'default-secret'
    -- Make use of it
end

return {
    role_name = role_name,
    set_secret = set_secret,
    validate_config = validate_config,
    apply_config = apply_config,
}
```

После настройки конфигурации выполните одно из следующих действий:

- продолжите разработку приложения, обращая особое внимание на *управление версиями*;
- (необязательно) *включите авторизацию* в веб-интерфейсе.
- если кластер уже развернут, *примените конфигурацию* для всего кластера.

Применение конфигурации пользовательской роли

With the implementation showed by the *example*, you can call the `set_secret()` function to apply the new configuration via the administrative console – or an HTTP endpoint if the role exports one.

Функция `set_secret()` вызывает `cartridge.confapplier.patch_clusterwide()`, которая производит двухфазную фиксацию транзакций:

1. Исправляет активную конфигурацию в памяти: копирует таблицу и заменяет раздел `"custom-role"` в копии на раздел, который задан функцией `set_secret()`.
2. The cluster checks if the new configuration can be applied on all instances except disabled and expelled. All instances subject to update must be healthy and `alive` according to the membership module.
3. (**Фаза подготовки**) Кластер передает исправленную конфигурацию. Каждый экземпляр валидирует ее с помощью функции `validate_config()` каждой зарегистрированной роли. В зависимости от результата валидации:
   - В случае успеха (то есть возврата значения `true`) экземпляр сохраняет новую конфигурацию во временный файл с именем `config.prepare.yml` в рабочей директории.
   - (**Abort phase**) Otherwise, the instance reports an error and all the other instances roll back the update: remove the file they may have already prepared.
4. (**Фаза фиксации**) После успешной подготовки всех экземпляров кластер фиксирует изменения. Каждый экземпляр:
   1. Создает жесткую ссылку активной конфигурации.
   2. Atomically replaces the active configuration file with the prepared one. The atomic replacement is indivisible – it can either succeed or fail entirely, never partially.
   3. Вызывает функцию `apply_config()` каждой зарегистрированной роли.

Если любой из этих шагов не будет выполнен, в веб-интерфейсе появится ошибка рядом с соответствующим экземпляром. Кластер не обрабатывает такие ошибки автоматически, их необходимо исправлять вручную.

Такого рода исправлений можно избежать, если функция `validate_config()` сможет обнаружить все проблемы конфигурации, которые могут привести к ошибкам в `apply_config()`.

Использование встроенного HTTP-сервера

Кластер запускает экземпляр `httpd`-сервера во время инициализации (`cartridge.cfg()`). Можно привязать порт к экземпляру через переменную окружения:

```
-- Get the port from an environmental variable or the default one:
local http_port = os.getenv('HTTP_PORT') or '8080'

local ok, err = cartridge.cfg({
   ...
   -- Pass the port to the cluster:
   http_port = http_port,
   ...
})
```

Чтобы использовать `httpd`-экземпляр, получите к нему доступ и настройте маршруты в рамках функции `init()` для какой-либо роли (например, для роли, которая предоставляет API через HTTP):

```
local function init(opts)

...

   -- Get the httpd instance:
   local httpd = cartridge.service_get('httpd')
   if httpd ~= nil then
       -- Configure a route to, for example, metrics:
       httpd:route({
               method = 'GET',
               path = '/metrics',
               public = true,
           },
           function(req)
               return req:render({json = stat.stat()})
           end
       )
   end
end
```

For more information on using Tarantool's HTTP server, see its documentation.

Реализация авторизации в веб-интерфейсе

To implement authorization in the web interface of every instance in a Tarantool cluster:

1. Используйте модуль, к примеру, `auth` с функцией `check_password`. Данная функция проверяет учетные данные любого пользователя, который пытается войти в веб-интерфейс.

   Функция `check_password` принимает имя пользователя и пароль и возвращает результат аутентификации: пройдена или нет.

   ```
   -- auth.lua

   -- Add a function to check the credentials
   local function check_password(username, password)

       -- Check the credentials any way you like

   ```

(continues on next page)

(продолжение с предыдущей страницы)

```
    -- Return an authentication success or failure
    if not ok then
        return false
    end
    return true
end
...
```

2. Передайте имя используемого модуля `auth` в качестве параметра для `cartridge.cfg()`, чтобы кластер мог использовать его:

```
-- init.lua

local ok, err = cartridge.cfg({
    auth_backend_name = 'auth',
    -- The cluster will automatically call 'require()' on the 'auth' module.
    ...
})
```

Это добавит кнопку **Log in** (Войти) в верхний правый угол в веб-интерфейсе, но все же позволит неавторизованным пользователям взаимодействовать с интерфейсом, что удобно для тестирования.

Примечание: Кроме того, для авторизации запросов к API кластера можно использовать базовый заголовок HTTP для авторизации.

---

3. Чтобы требовать авторизацию каждого пользователя в веб-интерфейсе даже до начальной загрузки кластера, добавьте следующую строку:

```
-- init.lua

local ok, err = cartridge.cfg({
    auth_backend_name = 'auth',
    auth_enabled = true,
    ...
})
```

С включенной аутентификацией при использовании модуля `auth` пользователь не сможет даже загрузить кластер без входа в систему. После успешного входа в систему и начальной загрузки можно включить и отключить аутентификацию для всего кластера в веб-интерфейсе, а параметр `auth_enabled` игнорируется.

Управление версиями приложения

Tarantool Cartridge understands semantic versioning as described at semver.org. When developing an application, create new Git branches and tag them appropriately. These tags are used to calculate version increments for subsequent packing.

Например, если версия вашего приложения – 1.2.1, пометьте текущую ветку тегом `1.2.1` (с аннотациями или без них).

Чтобы получить значение текущей версии из Git, выполните команду:

```
$ git describe --long --tags
1.2.1-12-g74864f2
```

Вывод показывает, что после версии 1.2.1 было 12 коммитов. Если мы соберемся упаковать приложение на данном этапе, его полная версия будет `1.2.1-12`, а пакет будет называться `<имя_приложения>-1.2.1-12.rpm`.

Запрещается использовать не семантические теги. Вы не сможете создать пакет из ветки, если последний тег не будет семантическим.

После *упаковки* приложения его версия сохраняется в файл `VERSION` в корневой каталог пакета.

Using .cartridge.ignore files

You can add a `.cartridge.ignore` file to your application repository to exclude particular files and/or directories from package builds.

For the most part, the logic is similar to that of `.gitignore` files. The major difference is that in `.cartridge.ignore` files the order of exceptions relative to the rest of the templates does not matter, while in `.gitignore` files the order does matter.

.cartridge.ignore entry	игнорирует все...
`target/`	**папки** (поскольку в конце стоит /) под названием `target` рекурсивно
`target`	**файлы или папки** под названием `target` рекурсивно
`/target`	**файлы или папки** под названием `target` в самой верхней директории (поскольку в начале стоит /)
`/target/`	**папки** под названием `target` в самой верхней директории (в начале и в конце стоит /)
`*.class`	**файлы или папки**, оканчивающиеся на `.class`, рекурсивно
`#comment`	ничего, это комментарий (первый символ – #)
`\#comment`	**файлы или папки** под названием `#comment` (\\ для выделения)
`target/logs/`	**папки** под названием `logs`, которые представляют собой поддиректорию папки под названием `target`
`target/*/logs/`	**папки** под названием `logs` на два уровня ниже папки под названием `target` (* не включает /)
`target/**/logs/`	**папки** под названием `logs` где угодно в пределах папки `target` (** включает /)
`*.py[co]`	**файлы или папки**, оканчивающиеся на `.pyc` или `.pyo`, но не на `.py`!
`*.py[!co]`	**файлы или папки**, оканчивающиеся на что угодно, кроме `c` или `o`
`*.file[0-9]`	**файлы или папки**, оканчивающиеся на цифру
`*.file[!0-9]`	**файлы или папки**, оканчивающиеся на что угодно, кроме цифры
`*`	**всё**
`/*`	**всё** в самой верхней директории (поскольку в начале стоит /)
`**/*.tar.gz`	файлы `*.tar.gz` или папки, которые находятся на **один или несколько** уровней ниже исходной папки
`!file`	**файлы и папки** будут проигнорированы, даже если они подходят под другие типы

Failover architecture

An important concept in cluster topology is appointing **a leader**. Leader is an instance which is responsible for performing key operations. To keep things simple, you can think of a leader as of the only writable master.

Every replica set has its own leader, and there's usually not more than one.

Which instance will become a leader depends on topology settings and failover configuration.

An important topology parameter is the **failover priority** within a replica set. This is an ordered list of instances. By default, the first instance in the list becomes a leader, but with the failover enabled it may be changed automatically if the first one is malfunctioning.

Instance configuration upon a leader change

When Cartridge configures roles, it takes into account the **leadership map** (consolidated in the `failover.lua` module). The leadership map is composed when the instance enters the `ConfiguringRoles` state for the first time. Later the map is updated according to the failover mode.

Every change in the leadership map is accompanied by instance re-configuration. When the map changes, Cartridge updates the `read_only` setting and calls the `apply_config` callback for every role. It also specifies the `is_master` flag (which actually means `is_leader`, but hasn't been renamed yet due to historical reasons).

It's important to say that we discuss a *distributed* system where every instance has its own opinion. Even if all opinions coincide, there still may be races between instances, and you (as an application developer) should take them into account when designing roles and their interaction.

Leader appointment rules

The logic behind leader election depends on the **failover mode**: disabled, eventual, or stateful.

Disabled mode

This is the simplest case. The leader is always the first instance in the failover priority. No automatic switching is performed. When it's dead, it's dead.

Eventual failover

In the `eventual` mode, the leader isn't elected consistently. Instead, every instance in the cluster thinks that the leader is the first **healthy** instance in the failover priority list, while instance health is determined according to the membership status (the SWIM protocol).

The member is considered healthy if both are true:

1. It reports either `ConfiguringRoles` or `RolesConfigured` state;
2. Its SWIM status is either `alive` or `suspect`.

A `suspect` member becomes `dead` after the `failover_timout` expires.

Leader election is done as follows. Suppose there are two replica sets in the cluster:

- a single router «R»,
- two storages, «S1» and «S2».

Then we can say: all the three instances (R, S1, S2) agree that S1 is the leader.

The SWIM protocol guarantees that *eventually* all instances will find a common ground, but it's not guaranteed for every intermediate moment of time. So we may get a conflict.

For example, soon after S1 goes down, R is already informed and thinks that S2 is the leader, but S2 hasn't received the gossip yet and still thinks he's not. This is a conflict.

Similarly, when S1 recovers and takes the leadership, S2 may be unaware of that yet. So, both S1 and S2 consider themselves as leaders.

Moreover, SWIM protocol isn't perfect and still can produce false-negative gossips (announce the instance is dead when it's not).

Stateful failover

Similarly to the eventual mode, every instance composes its own leadership map, but now the map is fetched from an **external state provider** (that's why this failover mode called «stateful»). Nowadays there are two state providers supported – `etcd` and `stateboard` (standalone Tarantool instance). State provider serves as a domain-specific key-value storage (simply `replicaset_uuid -> leader_uuid`) and a locking mechanism.

Changes in the leadership map are obtained from the state provider with the long polling technique.

All decisions are made by **the coordinator** – the one that holds the lock. The coordinator is implemented as a built-in Cartridge role. There may be many instances with the coordinator role enabled, but only one of them can acquire the lock at the same time. We call this coordinator the «active» one.

The lock is released automatically when the TCP connection is closed, or it may expire if the coordinator becomes unresponsive (in `stateboard` it's set by the stateboard's `--lock_delay` option, for `etcd` it's a part of clusterwide configuration), so the coordinator renews the lock from time to time in order to be considered alive.

The coordinator makes a decision based on the SWIM data, but the decision algorithm is slightly different from that in case of eventual failover:

- Right after acquiring the lock from the state provider, the coordinator fetches the leadership map.
- If there is no leader appointed for the replica set, the coordinator appoints the first leader according to the failover priority, regardless of the SWIM status.
- If a leader becomes `dead`, the coordinator makes a decision. A new leader is the first healthy instance from the failover priority list. If an old leader recovers, no leader change is made until the current leader down. Changing failover priority doesn't affect this.
- Every appointment (self-made or fetched) is immune for a while (controlled by the `IMMUNITY_TIMEOUT` option).

The case: external provider outage

In this case instances do nothing: the leader remains a leader, read-only instances remain read-only. If any instance restarts during an external state provider outage, it composes an empty leadership map: it doesn't know who actually is a leader and thinks there is none.

The case: coordinator outage

An active coordinator may be absent in a cluster either because of a failure or due to disabling the role everywhere. Just like in the previous case, instances do nothing about it: they keep fetching the leadership map from the state provider. But it will remain the same until a coordinator appears.

Manual leader promotion

It differs a lot depending on the failover mode.

In the disabled and eventual modes, you can only promote a leader by changing the failover priority (and applying a new clusterwide configuration).

In the stateful mode, the failover priority doesn't make much sense (except for the first appointment). Instead, you should use the promotion API (the Lua *cartridge.failover_promote* or the GraphQL `mutation {cluster{failover_promote()}}`) which pushes manual appointments to the state provider.

The stateful failover mode implies **consistent promotion**: before becoming writable, each instance performs the `wait_lsn` operation to sync up with the previous one.

Information about the previous leader (we call it a *vclockkeeper*) is also stored on the external storage. Even when the old leader is demoted, it remains the vclockkeeper until the new leader successfully awaits and persists its vclock on the external storage.

If replication is stuck and consistent promotion isn't possible, a user has two options: to revert promotion (to re-promote the old leader) or to force it inconsistently (all kinds of `failover_promote` API has `force_inconsistency` flag).

Consistent promotion doesn't work for replicasets with *all_rw* flag enabled and for single-instance replicasets. In these two cases an instance doesn't even try to query *vclockkeeper* and to perform *wait_lsn*. But the coordinator still appoints a new leader if the current one dies.

Fencing

Neither eventual nor stateful failover modes don't protect a replicaset from the presence of multiple leaders when the network is partitioned. But fencing does. It enforces at-most-one leader policy in a replicaset.

Fencing operates as a fiber that occasionally checks connectivity with the state provider and with replicas. Fencing fiber runs on vclockkeepers; it starts right after consistent promotion succeeds. Replicasets which don't need consistency (single-instance and `all_rw`) don't defense, though.

The condition for fencing actuation is the loss of both the state provider quorum and at least one replica. Otherwise, if either state provider is healthy or all replicas are alive, the fencing fiber waits and doesn't intervene.

When fencing is actuated, it generates a fake appointment locally and sets the leader to `nil`. Consequently, the instance becomes read-only. Subsequent recovery is only possible when the quorum reestablishes; replica connection isn't a must for recovery. Recovery is performed according to the rules of consistent switchover unless some other instance has already been promoted to a new leader.

Failover configuration

These are clusterwide parameters:

- `mode`: «disabled» / «eventual» / «stateful».
- `state_provider`: «tarantool» / «etcd».
- `failover_timeout` – time (in seconds) to mark `suspect` members as `dead` and trigger failover (default: 20).
- `tarantool_params`: `{uri = "...", password = "..."}`.
- `etcd2_params`: `{endpoints = {...}, prefix = "/", lock_delay = 10, username = "", password = ""}`.

- `fencing_enabled`: `true` / `false` (default: false).
- `fencing_timeout` – time to actuate fencing after the check fails (default: 10).
- `fencing_pause` – the period of performing the check (default: 2).

It's required that `failover_timeout > fencing_timeout >= fencing_pause`.

Lua API

See:

- *cartridge.failover_get_params*,
- *cartridge.failover_set_params*,
- *cartridge.failover_promote*.

GraphQL API

Use your favorite GraphQL client (e.g. Altair) for requests introspection:

- `query {cluster{failover_params{}}}`,
- `mutation {cluster{failover_params(){}}}`,
- `mutation {cluster{failover_promote()}}`.

Stateboard configuration

Like other Cartridge instances, the stateboard supports `cartridge.argprase` options:

- `listen`
- `workdir`
- `password`
- `lock_delay`

Similarly to other `argparse` options, they can be passed via command-line arguments or via environment variables, e.g.:

```
.rocks/bin/stateboard --workdir ./dev/stateboard --listen 4401 --password qwerty
```

Fine-tuning failover behavior

Besides failover priority and mode, there are some other private options that influence failover operation:

- `LONGPOLL_TIMEOUT` (`failover`) – the long polling timeout (in seconds) to fetch new appointments (default: 30);
- `NETBOX_CALL_TIMEOUT` (`failover/coordinator`) – stateboard client's connection timeout (in seconds) applied to all communications (default: 1);
- `RECONNECT_PERIOD` (`coordinator`) – time (in seconds) to reconnect to the state provider if it's unreachable (default: 5);

- `IMMUNITY_TIMEOUT` (`coordinator`) – minimal amount of time (in seconds) to wait before overriding an appointment (default: 15).

Конфигурация экземпляров

Cartridge orchestrates a distributed system of Tarantool instances – a cluster. One of the core concepts is **clusterwide configuration**. Every instance in a cluster stores a copy of it.

Clusterwide configuration contains options that must be identical on every cluster node, such as the topology of the cluster, failover and vshard configuration, authentication parameters and ACLs, and user-defined configuration.

Clusterwide configuration doesn't provide instance-specific parameters: ports, workdirs, memory settings, etc.

Configuration basics

Конфигурация экземпляра состоит из двух наборов параметров:

- *cartridge.cfg() parameters*;
- *box.cfg() parameters*.

Задать эти параметры можно:

1. В аргументах в командной строке.
2. В переменных окружения.
3. В конфигурационном файле формата YAML.
4. В файле `init.lua`.

Вышеуказанный порядок определяет приоритет: аргументы в командной строке замещают переменные окружения и т.д.

Независимо от того, как вы *запускаете экземпляры*, необходимо задать следующие параметры `cartridge.cfg()` для каждого экземпляра:

- `advertise_uri` – либо `<ХОСТ>:<ПОРТ>`, либо `<ХОСТ>:`, либо `<ПОРТ>`. Используется другими экземплярами для подключения. **НЕ** указывайте `0.0.0.0` – это должен быть внешний IP-адрес, а не привязка сокета.
- `http_port` – порт, который используется, чтобы открывать административный веб-интерфейс и API. По умолчанию: `8081`. Чтобы отключить, укажите `"http_enabled": False`.
- `workdir` – директория, где хранятся все данные: файлы снимка, журналы упреждающей записи и конфигурационный файл `cartridge`. По умолчанию: `.`.

Если вы запустите экземпляры, используя интерфейс командной строки `cartridge` или `systemctl`, сохраните конфигурацию в формате YAML, например:

```
my_app.router: {"advertise_uri": "localhost:3301", "http_port": 8080}
my_app.storage_A: {"advertise_uri": "localhost:3302", "http_enabled": False}
my_app.storage_B: {"advertise_uri": "localhost:3303", "http_enabled": False}
```

С помощью интерфейса командной строки `cartridge` вы можете передать путь к этому файлу в качестве аргумента командной строки `--cfg` для команды `cartridge start` – или же указать путь в конфигурации `cartridge` (в `./.cartridge.yml` или `~/.cartridge.yml`):

```
cfg: cartridge.yml
run_dir: tmp/run
apps_path: /usr/local/share/tarantool
```

С помощью `systemctl` сохраните файл в формате YAML в `/etc/tarantool/conf.d/` (по умолчанию путь `systemd`) или в место, указанное в переменной окружения `TARANTOOL_CFG`.

Если вы запускаете экземпляры с помощью `tarantool init.lua`, необходимо также передать другие параметры конфигурации в качестве параметров командной строки и переменных окружения, например:

```
$ tarantool init.lua --alias router --memtx-memory 100 --workdir "~/db/3301" --advertise_uri
↪"localhost:3301" --http_port "8080"
```

Internal representation of clusterwide configuration

In the file system, clusterwide configuration is represented by a **file tree**. Inside `workdir` of any configured instance you can find the following directory:

```
config/
├── auth.yml
├── topology.yml
└── vshard_groups.yml
```

This is the clusterwide configuration with three default **config sections** – `auth`, `topology`, and `vshard_groups`.

Due to historical reasons clusterwide configuration has two appearances:

- old-style single-file `config.yml` with all sections combined, and
- modern multi-file representation mentioned above.

Before cartridge v2.0 it used to look as follows, and this representation is still used in HTTP API and `luatest` helpers.

```
# config.yml
---
auth: {...}
topology: {...}
vshard_groups: {...}
...
```

Beyond these essential sections, clusterwide configuration may be used for storing some other role-specific data. Clusterwide configuration supports YAML as well as plain text sections. It can also be organized in nested subdirectories.

In Lua it's represented by the `ClusterwideConfig` object (a table with metamethods). Refer to the `cartridge.clusterwide-config` module documentation for more details.

Two-phase commit

Cartridge manages clusterwide configuration to be identical everywhere using the two-phase commit algorithm implemented in the `cartridge.twophase` module. Changes in clusterwide configuration imply applying it on every instance in the cluster.

Almost every change in cluster parameters triggers a two-phase commit: joining/expelling a server, editing replica set roles, managing users, setting failover and vshard configuration.

Two-phase commit requires all instances to be alive and healthy, otherwise it returns an error.

For more details, please, refer to the `cartridge.config_patch_clusterwide` API reference.

Managing role-specific data

Beside system sections, clusterwide configuration may be used for storing some other **role-specific data**. It supports YAML as well as plain text sections. And it can also be organized in nested subdirectories.

Role-specific sections are used by some third-party roles, i.e. sharded-queue and cartridge-extensions.

A user can influence clusterwide configuration in various ways. You can alter configuration using Lua, HTTP or GraphQL API. Also there are luatest helpers available.

HTTP API

It works with old-style single-file representation only. It's useful when there are only few sections needed.

Example:

```
cat > config.yml << CONFIG
---
custom_section: {}
...
CONFIG
```

Upload new config:

```
curl -v "localhost:8081/admin/config" -X PUT --data-binary @config.yml
```

Download it:

```
curl -v "localhost:8081/admin/config" -o config.yml
```

It's suitable for role-specific sections only. System sections (`topology`, `auth`, `vshard_groups`, `users_acl`) can be neither uploaded nor downloaded.

If authorization is enabled, use the `curl` option `--user username:password`.

GraphQL API

GraphQL API, by contrast, is only suitable for managing plain-text sections in the modern multi-file appearance. It is mostly used by WebUI, but sometimes it's also helpful in tests:

```
g.cluster.main_server:graphql({query = [[
    mutation($sections: [ConfigSectionInput!]) {
        cluster {
            config(sections: $sections) {
                filename
                content
            }
        }
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    }]],
    variables = {sections = {
      {
        filename = 'custom_section.yml',
        content = '---\n{}\n...',
      }
    }}
})
```

Unlike HTTP API, GraphQL affects only the sections mentioned in the query. All the other sections remain unchanged.

Similarly to HTTP API, GraphQL `cluster {config}` query isn't suitable for managing system sections.

Lua API

It's not the most convenient way to configure third-party role, but it may be useful for role development. Please, refer to the corresponding API reference:

- `cartridge.config_patch_clusterwide`
- `cartridge.config_get_deepcopy`
- `cartridge.config_get_readonly`

Example (from `sharded-queue`, simplified):

```
function create_tube(tube_name, tube_opts)
    local tubes = cartridge.config_get_deepcopy('tubes') or {}
    tubes[tube_name] = tube_opts or {}

    return cartridge.config_patch_clusterwide({tubes = tubes})
end

local function validate_config(conf)
    local tubes = conf.tubes or {}
    for tube_name, tube_opts in pairs(tubes) do
        -- validate tube_opts
    end
    return true
end

local function apply_config(conf, opts)
    if opts.is_master then
        local tubes = cfg.tubes or {}
        -- create tubes according to the configuration
    end
    return true
end
```

Luatest helpers

Cartridge test helpers provide methods for configuration management:

- `cartridge.test-helpers.cluster:upload_config`,

- `cartridge.test-helpers.cluster:download_config`.

Internally they wrap the HTTP API.

Example:

```
g.before_all(function()
    g.cluster = helpers.Cluster.new(...)
    g.cluster:upload_config({some_section = 'some_value'})
    t.assert_equals(
        g.cluster:download_config(),
        {some_section = 'some_value'}
    )
end)
```

Развертывание приложения

After you've developed your application locally, you can deploy it to a test or production environment.

«Deploy» includes packing the application into a specific distribution format, installing to the target system, and running the application.

Развернуть приложение Tarantool Cartridge можно четырьмя способами:

- в виде *rpm*-пакета (для эксплуатационной среды);
- в виде *deb*-пакета (для эксплуатационной среды);
- в виде архива *tar+gz* (для тестирования или для эксплуатационной среды, если отсутствует доступ уровня root).
- *из исходных файлов* (только для локального тестирования).

Развертывание приложения в виде пакета rpm или deb

The choice between DEB and RPM depends on the package manager of the target OS. For example, DEB is native for Debian Linux, and RPM – for CentOS.

1. Упакуйте файлы приложения в распространяемый пакет:

   ```
   $ cartridge pack rpm APP_NAME
   # -- OR --
   $ cartridge pack deb APP_NAME
   ```

   Будет создан RPM-пакет (например, `./my_app-0.1.0-1.rpm`) или же DEB-пакет (например, `./my_app-0.1.0-1.deb`).

2. Загрузите пакет на необходимые серверы с поддержкой `systemctl`.

3. Установите:

   ```
   $ yum install APP_NAME-VERSION.rpm
   # -- OR --
   $ dpkg -i APP_NAME-VERSION.deb
   ```

4. Configure the instance(s). Create a file called `/etc/tarantool/conf.d/instances.yml`. For example:

```
my_app:
  cluster_cookie: secret-cookie

my_app.instance-1:
  http_port: 8081
  advertise_uri: localhost:3301

my_app.instance-2:
  http_port: 8082
  advertise_uri: localhost:3302
```

See details here.

5. Запустите экземпляры Tarantool'а с соответствующими службами. Например, это можно сделать, используя *systemctl*:

```
# starts a single instance
$ systemctl start my_app

# starts multiple instances
$ systemctl start my_app@router
$ systemctl start my_app@storage_A
$ systemctl start my_app@storage_B
```

6. Если это приложение с поддержкой кластеров, далее переходите к *развертыванию кластера*.

Примечание: If you're migrating your application from local test environment to production, you can re-use your test configuration at this step:

1. In the cluster web interface of the test environment, click **Configuration files** > **Download** to save the test configuration.
2. In the cluster web interface of the production environment, click **Configuration files** > **Upload** to upload the saved configuration.

Развертывание архива tar+gz

1. Упакуйте файлы приложения в распространяемый пакет:

```
$ cartridge pack tgz APP_NAME
```

Будет создан архив tar+gz (например, `./my_app-0.1.0-1.tgz`).

2. Upload the archive to target servers, with `tarantool` and (optionally) *cartridge-cli* installed.
3. Распакуйте архив:

```
$ tar -xzvf APP_NAME-VERSION.tgz
```

4. Configure the instance(s). Create a file called `/etc/tarantool/conf.d/instances.yml`. For example:

```
my_app:
  cluster_cookie: secret-cookie

my_app.instance-1:
```

(continues on next page)

(продолжение с предыдущей страницы)

```
  http_port: 8081
  advertise_uri: localhost:3301

my_app.instance-2:
  http_port: 8082
  advertise_uri: localhost:3302
```

See details here.

5. Запустите экземпляры Tarantool'а. Это можно сделать, используя:
   - *tarantoolctl*, например:

     ```
     $ tarantool init.lua # starts a single instance
     ```

   - или *cartridge*, например:

     ```
     # in application directory
     $ cartridge start # starts all instances
     $ cartridge start .router_1 # starts a single instance

     # in multi-application environment
     $ cartridge start my_app # starts all instances of my_app
     $ cartridge start my_app.router # starts a single instance
     ```

6. Если это приложение с поддержкой кластеров, далее переходите к *развертыванию кластера*.

Примечание: If you're migrating your application from local test environment to production, you can re-use your test configuration at this step:

1. In the cluster web interface of the test environment, click **Configuration files** > **Download** to save the test configuration.
2. In the cluster web interface of the production environment, click **Configuration files** > **Upload** to upload the saved configuration.

Развертывание из исходных файлов

Такой метод развертывания предназначен только для локального тестирования.

1. Вытяните все зависимости в директорию `.rocks`:

   $ tarantoolctl rocks make

2. Configure the instance(s). Create a file called `/etc/tarantool/conf.d/instances.yml`. For example:

```
my_app:
  cluster_cookie: secret-cookie

my_app.instance-1:
  http_port: 8081
  advertise_uri: localhost:3301

my_app.instance-2:
  http_port: 8082
  advertise_uri: localhost:3302
```

See details here.

3. Запустите экземпляры Tarantool'а. Это можно сделать, используя:

   - *tarantoolctl*, например:

   ```
   $ tarantool init.lua # starts a single instance
   ```

   - или *cartridge*, например:

   ```
   # in application directory
   cartridge start # starts all instances
   cartridge start .router_1 # starts a single instance

   # in multi-application environment
   cartridge start my_app # starts all instances of my_app
   cartridge start my_app.router # starts a single instance
   ```

4. Если это приложение с поддержкой кластеров, далее переходите к *развертыванию кластера*.

Примечание: If you're migrating your application from local test environment to production, you can re-use your test configuration at this step:

1. In the cluster web interface of the test environment, click **Configuration files** > **Download** to save the test configuration.
2. In the cluster web interface of the production environment, click **Configuration files** > **Upload** to upload the saved configuration.

Запуск/остановка экземпляров

В зависимости от *способа развертывания* вы можете запускать/останавливать экземпляры, используя *tarantool*, *интерфейс командной строки cartridge* или *systemctl*.

Запуск/остановка с помощью `tarantool`

С помощью `tarantool` можно запустить только один экземпляр:

```
$ tarantool init.lua # the simplest command
```

Можно также задать дополнительные параметры в командной строке или в переменных окружения.

Чтобы остановить экземпляр, используйте Ctrl+C.

Запуск/остановка с помощью CLI в `cartridge`

С помощью интерфейса командной строки `cartridge`, можно запустить один или несколько экземпляров:

```
$ cartridge start [APP_NAME[.INSTANCE_NAME]] [options]
```

Возможные параметры:

`--script FILE` Точка входа в приложение. По умолчанию:

- TARANTOOL_SCRIPT, либо
- ./init.lua, если запуск идет из директории приложения, или же
- :путь_к_приложениям/:имя_приложения/init.lua в среде с несколькими приложениями.

--apps_path PATH Путь к директории с приложениями при запуске из среды с несколькими приложениями. По умолчанию: /usr/share/tarantool.

--run_dir DIR Директория с файлами pid и sock. По умолчанию: TARANTOOL_RUN_DIR или /var/run/tarantool.

--cfg FILE Cartridge instances YAML configuration file. Defaults to TARANTOOL_CFG or ./instances.yml. The instances.yml file contains cartridge.cfg() parameters described in the configuration section of this guide.

--foreground Не в фоне.

Например:

```
cartridge start my_app --cfg demo.yml --run_dir ./tmp/run --foreground
```

Это запустит все экземпляры Tarantool'a, указанные в файле cfg, не в фоновом режиме с принудительным использованием переменных окружения.

Если ИМЯ_ПРИЛОЖЕНИЯ не указано, cartridge выделит его из имени файла ./*.rockspec.

Если ИМЯ_ЭКЗЕМПЛЯРА не указывается, cartridge прочитает файл cfg и запустит все указанные экземпляры:

```
# in application directory
cartridge start # starts all instances
cartridge start .router_1 # start single instance

# in multi-application environment
cartridge start my_app # starts all instances of my_app
cartridge start my_app.router # start a single instance
```

Чтобы остановить экземпляры, выполните команду:

```
$ cartridge stop [APP_NAME[.INSTANCE_NAME]] [options]
```

Поддерживаются следующие параметры из команды *cartridge start*':

- --run_dir DIR
- --cfg FILE

Запуск/остановка с помощью systemctl

- Чтобы запустить отдельный экземпляр:

  ```
  $ systemctl start APP_NAME
  ```

  This will start a systemd service that will listen to the port specified in *instance configuration* (http_port parameter).

- Чтобы запустить несколько экземпляров на одном или нескольких серверах:

```
$ systemctl start APP_NAME@INSTANCE_1
$ systemctl start APP_NAME@INSTANCE_2
...
$ systemctl start APP_NAME@INSTANCE_N
```

где ИМЯ_ПРИЛОЖЕНИЯ@ЭКЗЕМПЛЯР_N – это имя экземпляра сервиса `systemd` с инкрементным числом N (уникальным для каждого экземпляра), которое следует добавить к порту 3300 для настройки прослушивания (например, 3301, 3302 и т.д.).

- Чтобы остановить все сервисы на сервере, используйте команду `systemctl stop` и укажите имена экземпляров по одному. Например:

```
$ systemctl stop APP_NAME@INSTANCE_1 APP_NAME@INSTANCE_2 ... APP_NAME@INSTANCE_<N>
```

When running instances with `systemctl`, keep these practices in mind:

- You can specify *instance configuration* in a YAML file.

  This file can contain these options; see an example here).

  Save this file to `/etc/tarantool/conf.d/` (the default `systemd` path) or to a location set in the `TARANTOOL_CFG` environment variable (if you've edited the application's `systemd` unit file). The file name doesn't matter: it can be `instances.yml` or anything else you like.

  Here's what `systemd` is doing further:

  – obtains `app_name` (and `instance_name`, if specified) from the name of the application's `systemd` unit file (e.g. `APP_NAME@default` or `APP_NAME@INSTANCE_1`);

  – sets default console socket (e.g. `/var/run/tarantool/APP_NAME@INSTANCE_1.control`), PID file (e.g. `/var/run/tarantool/APP_NAME@INSTANCE_1.pid`) and `workdir` (e.g. `/var/lib/tarantool/<APP_NAME>.<INSTANCE_NAME>`). Environment=TARANTOOL_WORKDIR=${workdir}.%i

  Finally, `cartridge` looks across all YAML files in `/etc/tarantool/conf.d` for a section with the appropriate name (e.g. `app_name` that contains common configuration for all instances, and `app_name.instance_1` that contain instance-specific configuration). As a result, Cartridge options `workdir`, `console_sock`, and `pid_file` in the YAML file cartridge.cfg become useless, because `systemd` overrides them.

- The default tool for querying logs is journalctl. For example:

```
# show log messages for a systemd unit named APP_NAME.INSTANCE_1
$ journalctl -u APP_NAME.INSTANCE_1

# show only the most recent messages and continuously print new ones
$ journalctl -f -u APP_NAME.INSTANCE_1
```

  If really needed, you can change logging-related `box.cfg` options in the YAML configuration file: see log and other related options.

Error handling guidelines

Almost all errors in Cartridge follow the `return nil, err` style, where `err` is an error object produced by Tarantool's errors module. Cartridge doesn't raise errors except for bugs and functions contracts mismatch. Developing new roles should follow these guidelines as well.

Error objects in Lua

Error classes help to locate the problem's source. For this purpose, an error object contains its class, stack traceback, and a message.

```
local errors = require('errors')
local DangerousError = errors.new_class("DangerousError")

local function some_fancy_function()

    local something_bad_happens = true

    if something_bad_happens then
        return nil, DangerousError:new("Oh boy")
    end

    return "success" -- not reachable due to the error
end

print(some_fancy_function())
```

```
nil DangerousError: Oh boy
stack traceback:
    test.lua:9: in function 'some_fancy_function'
    test.lua:15: in main chunk
```

For uniform error handling, `errors` provides the `:pcall` API:

```
local ret, err = DangerousError:pcall(some_fancy_function)
print(ret, err)
```

```
nil DangerousError: Oh boy
stack traceback:
    test.lua:9: in function <test.lua:4>
    [C]: in function 'xpcall'
    .rocks/share/tarantool/errors.lua:139: in function 'pcall'
    test.lua:15: in main chunk
```

`lua print(DangerousError:pcall(error, 'what could possibly go wrong?')) `

```
nil DangerousError: what could possibly go wrong?
stack traceback:
    [C]: in function 'xpcall'
    .rocks/share/tarantool/errors.lua:139: in function 'pcall'
    test.lua:15: in main chunk
```

For `errors.pcall` there is no difference between the `return nil, err` and `error()` approaches.

Note that `errors.pcall` API differs from the vanilla Lua pcall. Instead of `true` the former returns values returned from the call. If there is an error, it returns `nil` instead of `false`, plus an error message.

Remote `net.box` calls keep no stack trace from the remote. In that case, `errors.netbox_eval` comes to the rescue. It will find a stack trace from local and remote hosts and restore metatables.

```
> conn = require('net.box').connect('localhost:3301')
> print( errors.netbox_eval(conn, 'return nil, DoSomethingError:new("oops")') )
nil     DoSomethingError: oops
```

(continues on next page)

(продолжение с предыдущей страницы)

```
stack traceback:
        eval:1: in main chunk
during net.box eval on localhost:3301
stack traceback:
        [string "return print( errors.netbox_eval("]:1: in main chunk
        [C]: in function 'pcall'
```

However, `vshard` implemented in Tarantool doesn't utilize the `errors` module. Instead it uses its own errors. Keep this in mind when working with `vshard` functions.

Data included in an error object (class name, message, traceback) may be easily converted to string using the `tostring()` function.

GraphQL

GraphQL implementation in Cartridge wraps the `errors` module, so a typical error response looks as follows:

```
{
    "errors":[{
        "message":"what could possibly go wrong?",
        "extensions":{
            "io.tarantool.errors.stack":"stack traceback: ...",
            "io.tarantool.errors.class_name":"DangerousError"
        }
    }]
}
```

Read more about errors in the GraphQL specification.

If you're going to implement a GraphQL handler, you can add your own extension like this:

```
local err = DangerousError:new('I have extension')
err.graphql_extensions = {code = 403}
```

It will lead to the following response:

```
{
    "errors":[{
        "message":"I have extension",
        "extensions":{
            "io.tarantool.errors.stack":"stack traceback: ...",
            "io.tarantool.errors.class_name":"DangerousError",
            "code":403
        }
    }]
}
```

HTTP

In a nutshell, an `errors` object is a table. This means that it can be swiftly represented in JSON. This approach is used by Cartridge to handle errors via http:

```
local err = DangerousError:new('Who would have thought?')

local resp = req:render({
    status = 500,
    headers = {
        ['content-type'] = "application/json; charset=utf-8"
    },
    json = json.encode(err),
})
```

```
{
    "line":27,
    "class_name":"DangerousError",
    "err":"Who would have thought?",
    "file":".../app/roles/api.lua",
    "stack":"stack traceback:..."
}
```

4.3.3 Руководство администратора

В данном руководстве рассматривается развертывание и управление кластером Tarantool'а с помощью Tarantool Cartridge.

Примечание: Дополнительную информацию по управлению экземплярами Tarantool'а см. в разделе Администрирование серверной части.

Перед тем, как развертывать кластер, ознакомьтесь с понятием *кластерных ролей* и *разверните экземпляры Tarantool'а* в соответствии с предполагаемой топологией кластера.

Развертывание кластера

Чтобы развернуть кластер, сначала настройте все экземпляры Tarantool'а в соответствии с предполагаемой топологией кластера, например:

```
my_app.router: {"advertise_uri": "localhost:3301", "http_port": 8080, "workdir": "./tmp/router"}
my_app.storage_A_master: {"advertise_uri": "localhost:3302", "http_enabled": False, "workdir": "./
↪tmp/storage-a-master"}
my_app.storage_A_replica: {"advertise_uri": "localhost:3303", "http_enabled": False, "workdir": "./
↪tmp/storage-a-replica"}
my_app.storage_B_master: {"advertise_uri": "localhost:3304", "http_enabled": False, "workdir": "./
↪tmp/storage-b-master"}
my_app.storage_B_replica: {"advertise_uri": "localhost:3305", "http_enabled": False, "workdir": "./
↪tmp/storage-b-replica"}
```

Затем, *запустите экземпляры*, например, используя CLI в `cartridge`:

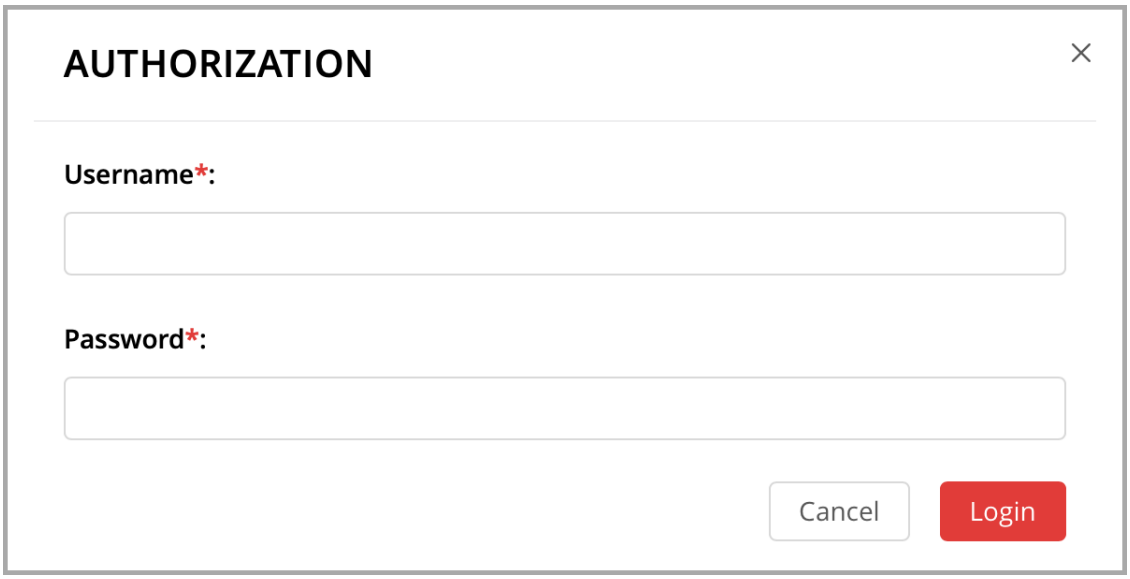
```
cartridge start my_app --cfg demo.yml --run_dir ./tmp/run --foreground
```

And bootstrap the cluster. You can do this via the Web interface which is available at `http://<instance_hostname>:<instance_http_port>` (in this example, `http://localhost:8080`).

В веб-интерфейсе выполните следующие действия:

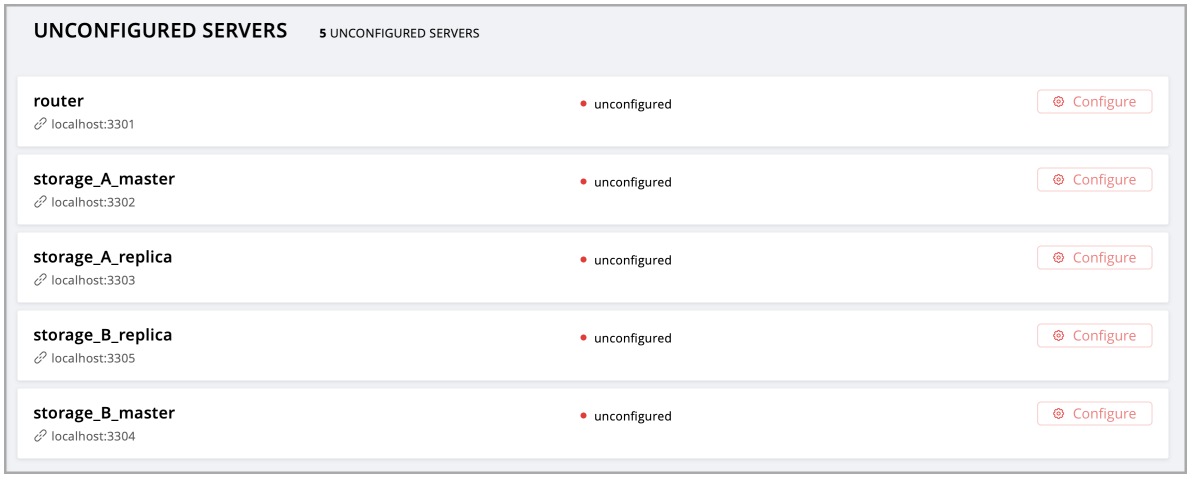
1. В зависимости от статуса аутентификации:

- Если аутентификация включена (в эксплуатационной среде), введите свои учетные данные и нажмите **Login** (Войти):



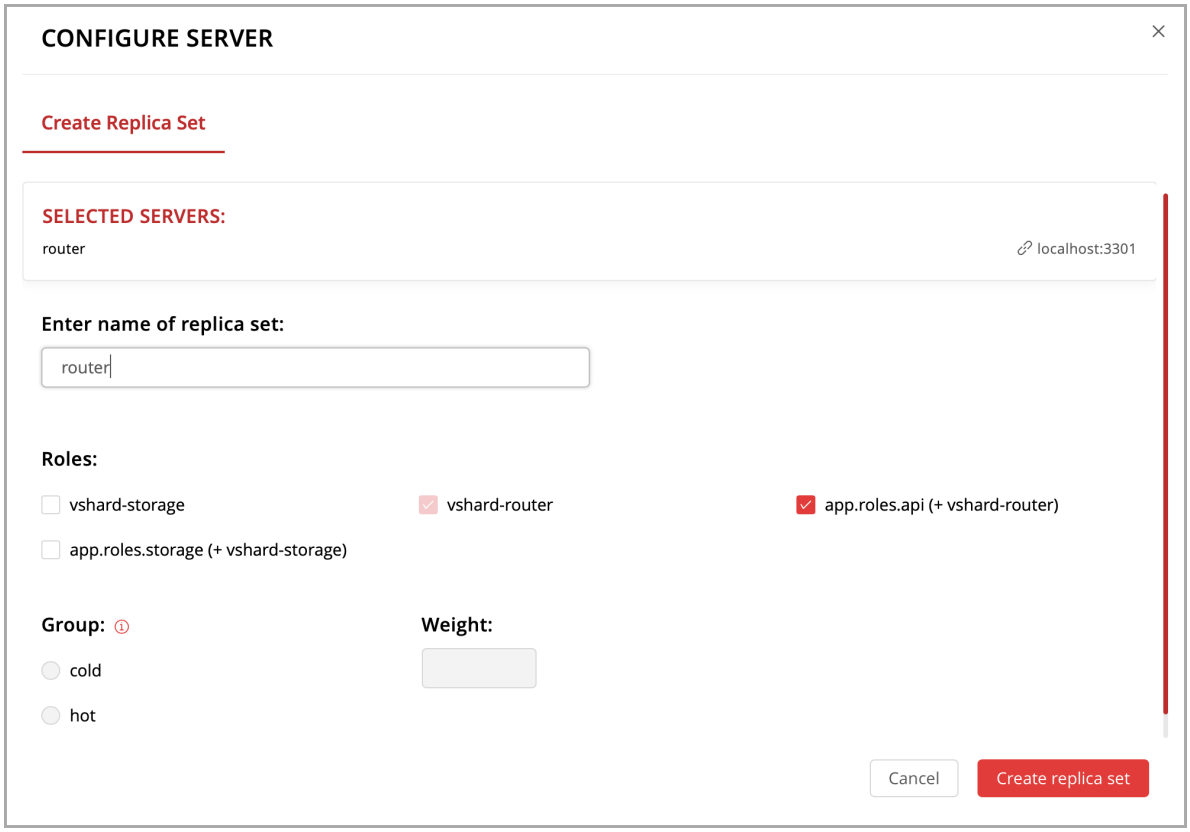
- Если отключен (для удобства тестирования), просто переходите к настройке кластера.

2. Нажмите **Configure** (Настроить) рядом с первым ненастроенным сервером, чтобы создать первый набор реплик исключительно для роутера (для обработки *ресурсоемких вычислений*).



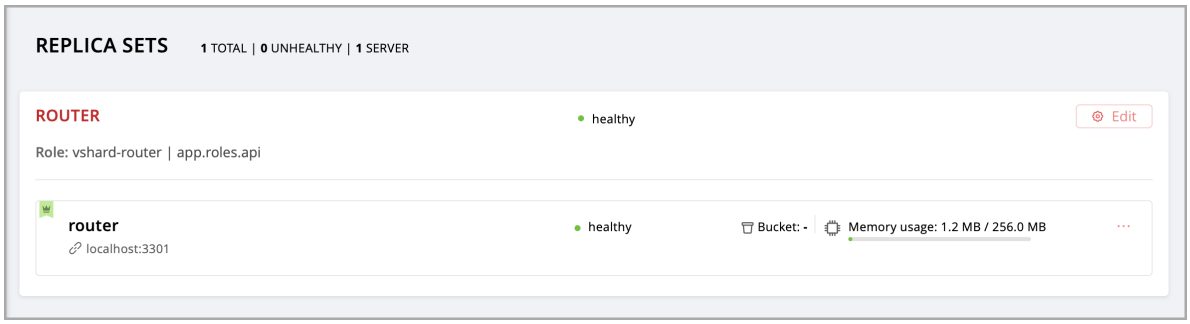
Во всплывающем окне отметьте флажок роли `vshard-router` или любой пользовательской роли, для которой роль `vshard-router` будет зависимой (в данном примере это пользовательская роль под названием `app.roles.api`).

(Необязательно) Укажите отображаемое имя для набора реплик, например `router`.



Примечание: Как описано в *разделе о встроенных ролях*, рекомендуется включать кластерные роли в зависимости от рабочей нагрузки на экземпляры, которые работают на физических серверах с аппаратным обеспечением, предназначенным для рабочей нагрузки определенного типа.

Нажмите **Create replica set** (Создать набор реплик), и созданный набор реплик отобразится в веб-интерфейсе



Предупреждение: Обратите внимание: после того, как экземпляр подключится к набору реплик, **НЕВОЗМОЖНО** это отменить или переподключить его к другому набору реплик.

3. Создайте новый набор реплик для мастер-узлов хранения данных (для обработки *большого ко-*

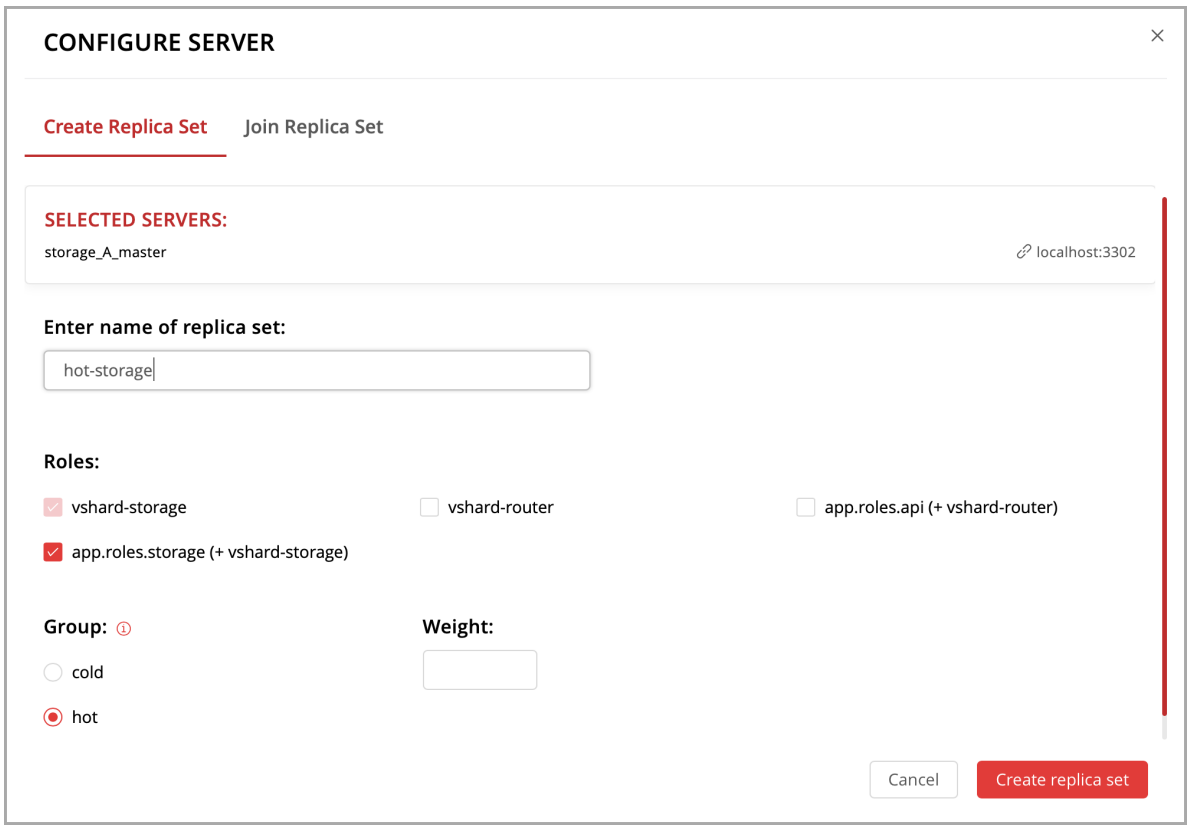
личества транзакций).

Отметьте флажок роли `vshard-storage` или любой пользовательской роли, для которой роль `vshard-storage` будет зависимой (в данном примере это пользовательская роль под названием `app.roles.storage`).

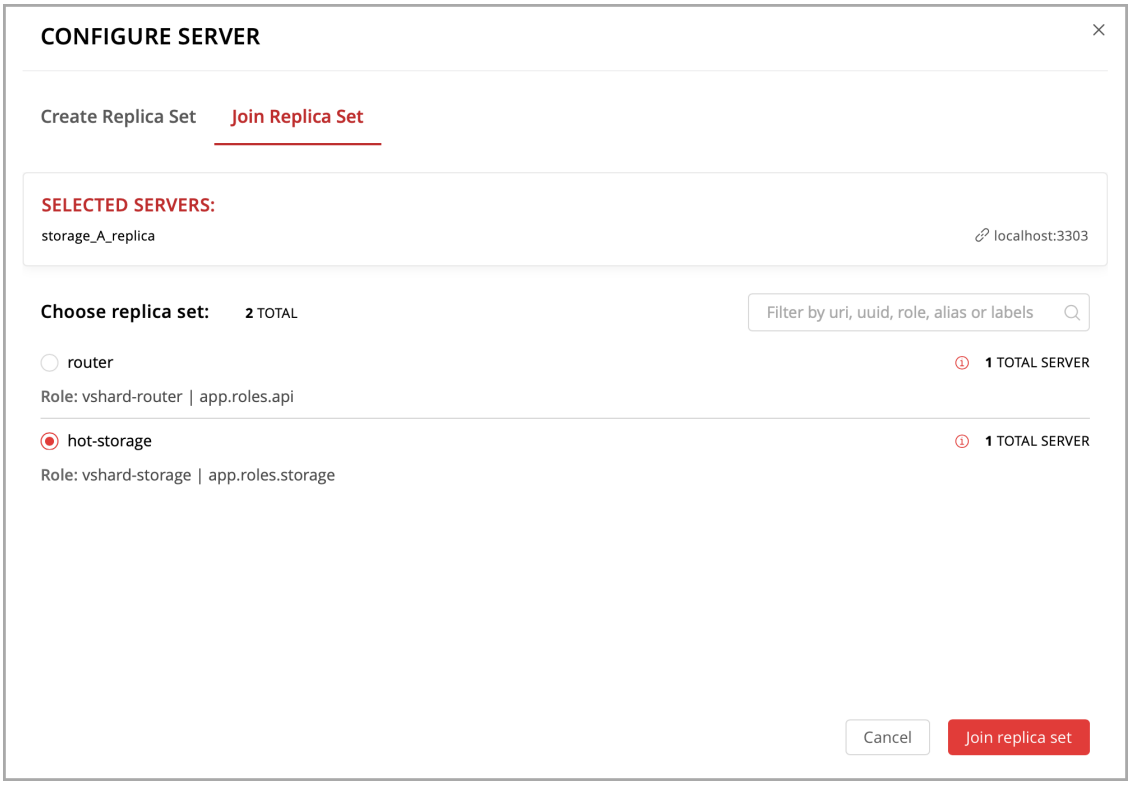
(Необязательно) Задайте определенную группу, например `hot` (горячие). Наборы реплик с ролями `vshard-storage` могут относиться к различным группам. В нашем примере группы `hot` и `cold` предназначены для независимой обработки горячих и холодных данных соответственно. Эти группы указаны в *конфигурационном файле* кластера. По умолчанию, кластер не входит ни в одну группу.

(Необязательно) Укажите отображаемое имя для набора реплик, например `hot-storage`.

Нажмите **Create replica set** (Создать набор реплик).



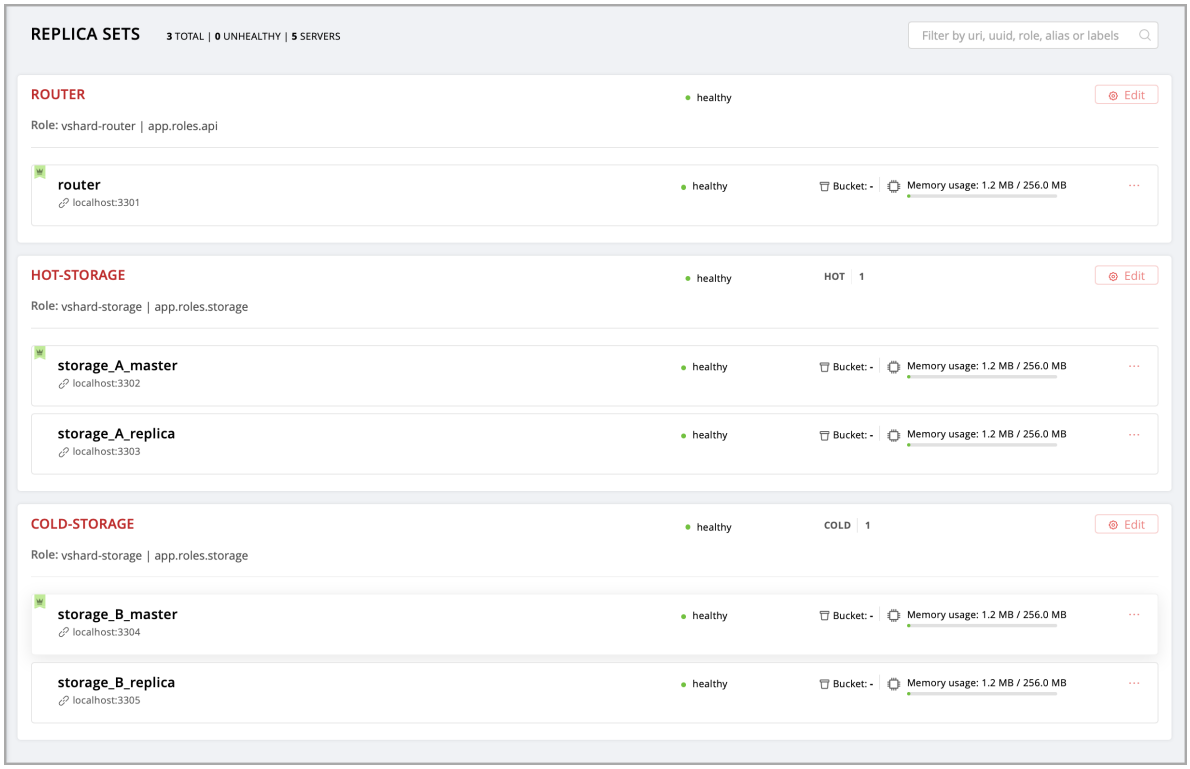
4. (Необязательно) Если этого требует топология, добавьте во второй набор реплик дополнительные хранилища:
    1. Нажмите **Configure** (Настроить) рядом с другим ненастроенным сервером, который выделен для рабочей нагрузки с *большим количеством транзакций*.
    2. Нажмите на вкладку **Join Replica Set** (Присоединиться к набору реплик).
    3. Выберите второй набор реплик и нажмите **Join replica set** (Присоединиться к набору реплик), чтобы добавить к нему сервер.



5. В зависимости от топологии кластера:

   - добавьте дополнительные экземпляры к первому или второму набору реплик, или же
   - создайте дополнительные наборы реплик и добавьте в них экземпляры для обработки определенной рабочей нагрузки (вычисления или транзакции).

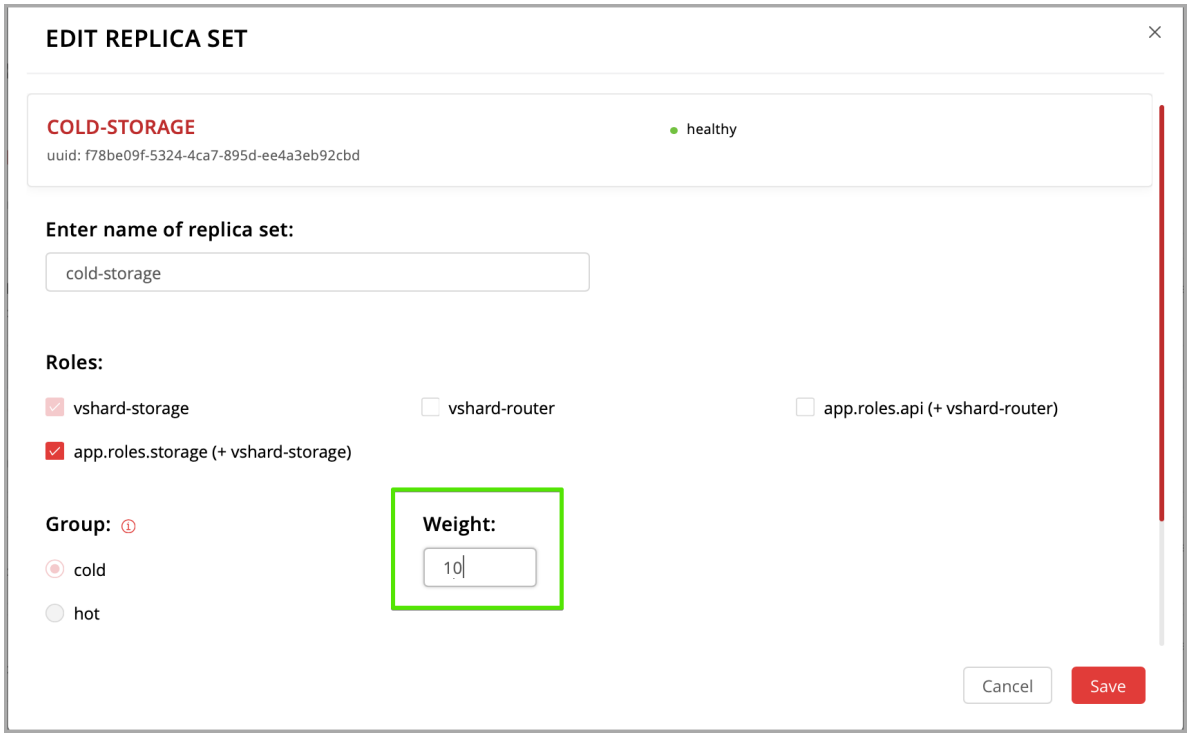
   Например:



6. (Необязательно) По умолчанию все новые наборы реплик `vshard-storage` получают вес, равный 1, до загрузки `vshard` в следующем шаге.

Примечание: Если вы добавите новый набор реплик после начальной загрузки `vshard`, как описано в *разделе об изменении топологии*, он по умолчанию получит вес 0.

Чтобы разные наборы реплик хранили разное количество сегментов, нажмите **Edit** (Изменить) рядом с набором реплик, измените значение веса по умолчанию и нажмите **Save** (Сохранить):



Для получения дополнительной информации о сегментах и весах набора реплик см. документацию по модулю vshard.

7. Загрузите `vshard`, нажав соответствующую кнопку или же выполнив команду `cartridge.admin.boostrap_vshard()` в административной консоли.

   Эта команда создает виртуальные сегменты и распределяет их по хранилищам.

   С этого момента всю настройку кластера можно выполнять через веб-интерфейс.

Обновление конфигурации

Конфигурация кластера задается в конфигурационном файле формата YAML. Этот файл включает в себя топологию кластера и описания ролей.

У всех экземпляров в кластере Tarantool'а одинаковые настройки. Для этого каждый экземпляр в кластере хранит копию конфигурационного файла, а кластер синхронизирует эти копии: как только вы подтверждаете обновление конфигурации в веб-интерфейсе, кластер валидирует ее (и отклоняет неприемлемые изменения) и передает ее *автоматически* по всему кластеру.

Чтобы обновить конфигурацию:

1. Нажмите на вкладку **Configuration files** (Конфигурационные файлы).
2. (Необязательно) Нажмите **Downloaded** (Загруженные), чтобы получить текущую версию конфигурационного файла.
3. Обновите конфигурационный файл.

   Можно добавлять/изменять/удалять любы разделы, кроме системных: `topology`, `vshard` и `vshard_groups`.

   Чтобы удалить раздел, просто удалите его из конфигурационного файла.

4. Создайте сжатую копию конфигурационного файла в виде архива в формате `.zip` и нажмите кнопку **Upload configuration** (Загрузить конфигурацию), чтобы загрузить ее.

   В нижней части экрана вы увидите сообщение об успешной загрузке конфигурации или ошибку, если новые настройки не были применены.

Управление кластером

В данной главе описывается, как:

- изменять топологию кластера,
- включать автоматическое восстановление после отказа,
- вручную менять мастера в наборе реплик,
- отключать наборы реплик,
- исключать экземпляры.

Изменение топологии кластера

При добавлении нового развернутого экземпляра в новый или уже существующий набор реплик:

1. Кластер валидирует обновление конфигурации, проверяя доступность нового экземпляра с помощью модуля membership.

   Примечание: Модуль `membership` работает по протоколу UDP и может производить операции до вызова функции `box.cfg`.

   Все узлы в кластере должны быть рабочими, чтобы валидация была пройдена.

2. Новый экземпляр ожидает, пока другой экземпляр в кластере не получит обновление конфигурации (оповещение реализовано с помощью того же модуля `membership`). На этом шаге у нового экземпляра еще нет своего UUID.

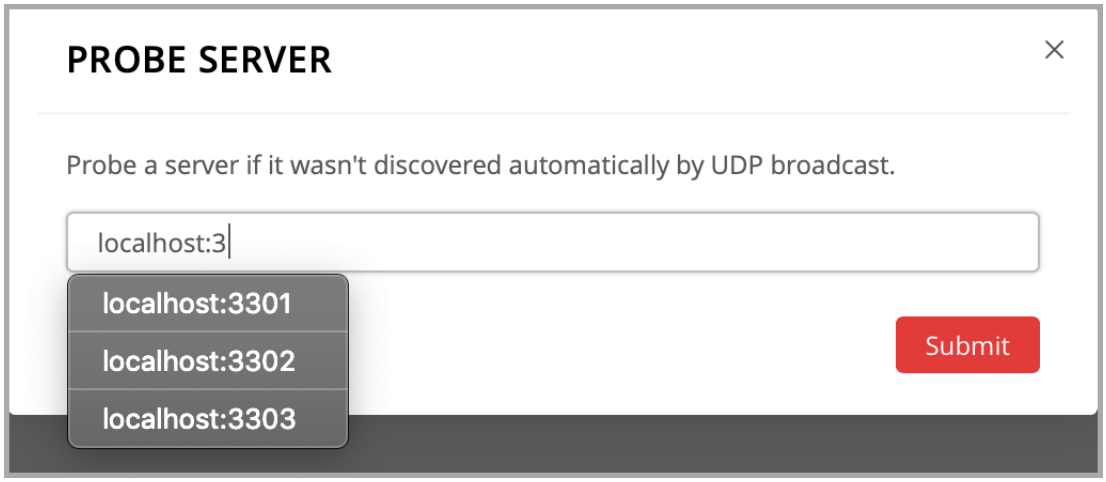
3. Как только новый экземпляр понимает, что кластер знает о нем, экземпляр вызывает функцию box.cfg и начинает работу.

Оптимальная стратегия подключения новых узлов к кластеру состоит в том, чтобы развертывать новые экземпляры в наборе реплик с нулевым весом для каждого экземпляра, а затем увеличивать вес. Как только вес обновится и все узлы кластера получат уведомление об изменении конфигурации, сегменты начинают мигрировать на новые узлы.

Чтобы добавить в кластер новые узлы, выполните следующие действия:

1. Разверните новые экземпляры Tarantool, как описано в *разделе по развертыванию*.

   Если новые узлы не появились в веб-интерфейсе, нажмите **Probe server** (Найти сервер) и укажите их URI вручную.

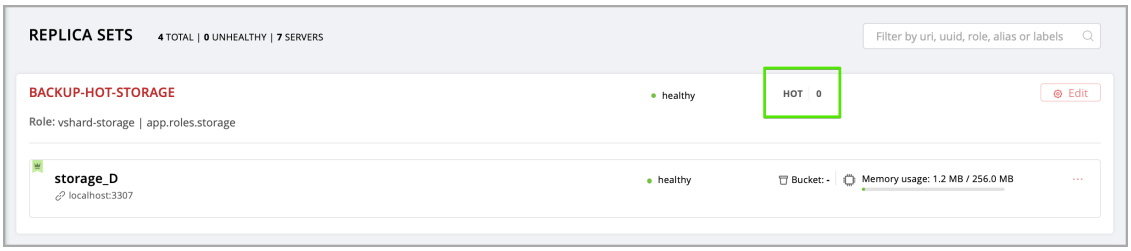


Если узел доступен, он появится в списке.

2. В веб-интерфейсе:

   - Создайте новый набор реплик с одним из новых экземпляров: нажмите **Configure** (Настроить) рядом с ненастроенным сервером, отметьте флажками необходимые роли и нажмите **Create replica set** (Создать набор реплик):

     Примечание: Если вы добавляете экземпляр `vshard-storage`, следует помнить, что вес всех таких экземпляров по умолчанию становится равным `0` после начальной загрузки `vshard`, которая происходит во время первоначального развертывания кластера.

   - Или добавьте дополнительные экземпляры к существующему набору реплик: нажмите **Configure** (Настроить) рядом с ненастроенным сервером, нажмите на вкладку **Join replica set** (Присоединиться к набору реплик), выберите набор реплик и нажмите **Join replica set**.

   При необходимости повторите действия для других экземпляров, чтобы достичь необходимого уровня резервирования.

3. При развертывании нового набора реплик `vshard-storage` заполните необходимую информацию: нажмите **Edit** (Изменить) рядом с необходимым набором реплик, увеличьте его вес и нажмите **Save** (Сохранить), чтобы начать *балансировку данных*.

Вместо веб-интерфейса можно использовать GraphQL для просмотра и изменения топологии кластера. Конечная точка кластера для выполнения запросов GraphQL – `/admin/api`. Можно пользоваться любыми сторонними клиентами GraphQL, такими как GraphiQL или Altair.

Примеры:

- вывод списка всех серверов в кластере:

```
query {
    servers { alias uri uuid }
}
```

- вывод списка всех наборов реплик с серверами:

```
query {
    replicasets {
        uuid
        roles
        servers { uri uuid }
    }
}
```

- подключение сервера к новому набору реплик с включенной ролью хранилища:

```
mutation {
    join_server(
        uri: "localhost:33003"
        roles: ["vshard-storage"]
    )
}
```

Балансировка данных

Балансировка (решардинг) запускается через определенные промежутки времени и при добавлении в кластер нового набора реплик с весом, отличным от нуля. Для получения дополнительной информации см. раздел по балансировке в документации по модулю `vshard`.

Самый удобный способ мониторинга процесса балансировки заключается в том, чтобы отслеживать количество активных сегментов на узлах хранения. Сначала в новом наборе реплик 0 активных сегментов. Через некоторое время фоновый процесс балансировки начинает переносить сегменты из других наборов реплик в новый. Балансировка продолжается до тех пор, пока данные не будут распределены равномерно по всем наборам реплик.

Чтобы отслеживать текущее количество сегментов, подключитесь к любому экземпляру Tarantool через *административную консоль* и выполните команду:

```
tarantool> vshard.storage.info().bucket
---
- receiving: 0
  active: 1000
  total: 1000
  garbage: 0
  sending: 0
...
```

Количество сегментов может увеличиваться или уменьшаться в зависимости от того, переносит ли балансировщик сегменты в узел хранения или из него.

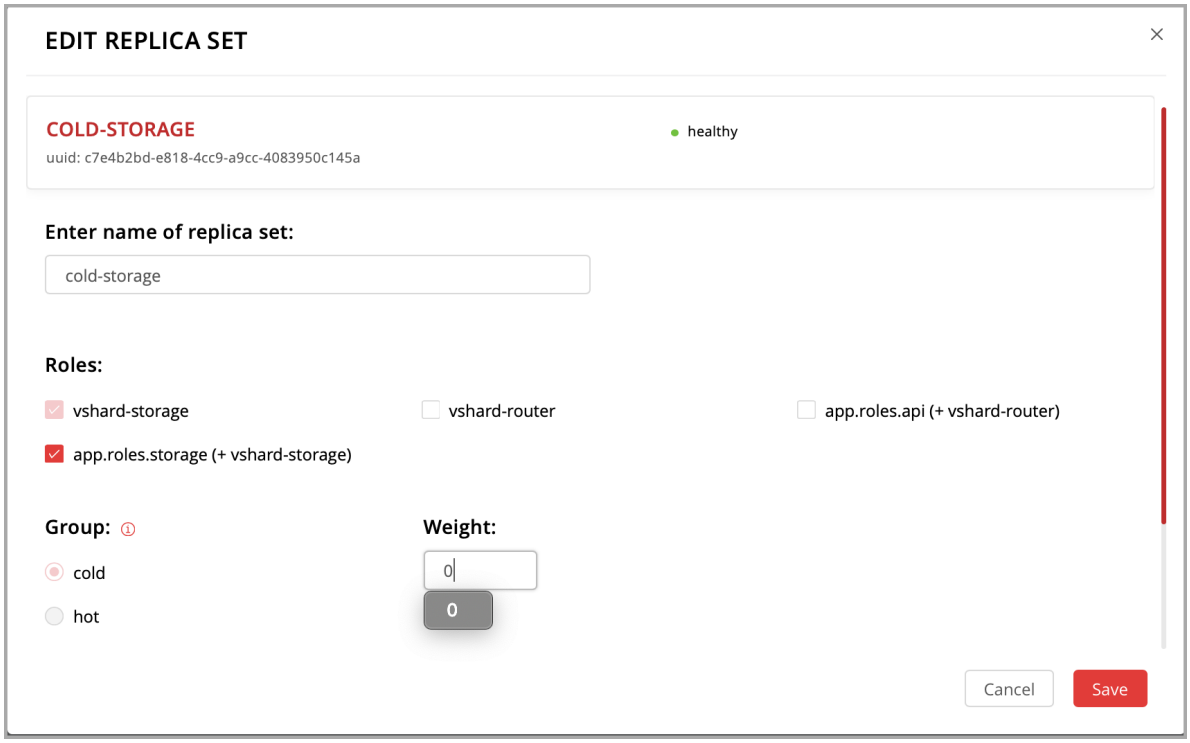
Для получения дополнительной информации о параметрах мониторинга см. *раздел по мониторингу хранилищ*.

Отключение наборов реплик

Под отключением всего набора реплик (например, для технического обслуживания) подразумевается перемещение всех его сегментов в другие наборы реплик.

Чтобы отключить набор реплик, выполните следующие действия:

1. Нажмите **Edit** (Изменить) рядом с необходимым набором реплик.
2. Укажите `0` как значение веса и нажмите **Save** (Сохранить):

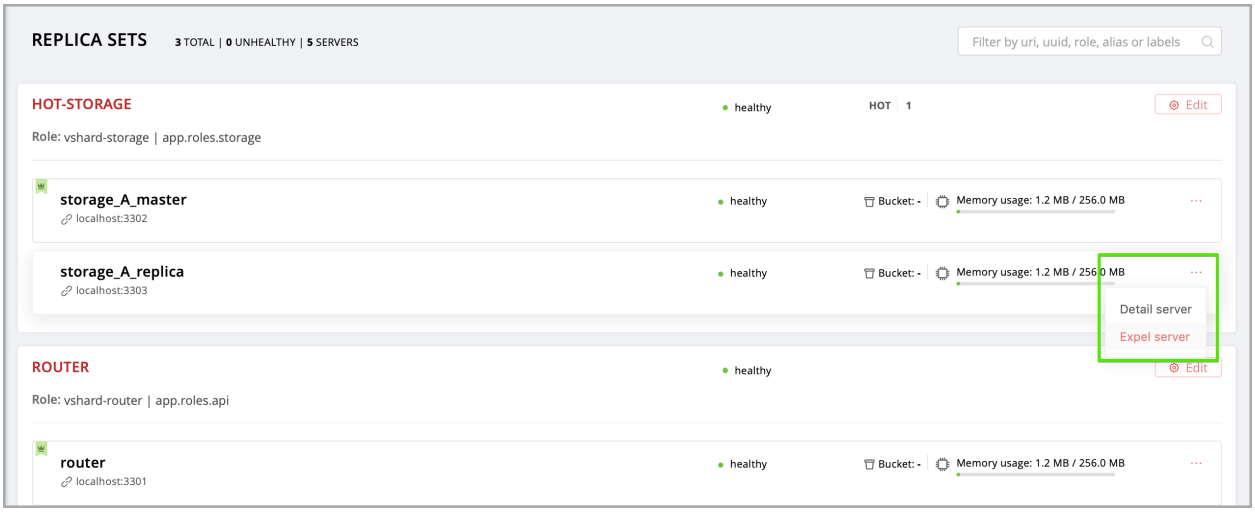


3. Подождите, пока процесс балансировки не завершит перенос всех сегментов набора. Можно отслеживать текущее количество сегментов, как описано в *разделе по балансировке данных*.

Исключение экземпляров

После того, как экземпляр будет *исключен* из кластера, он никогда не сможет снова участвовать в кластере, поскольку ни один экземпляр не будет принимать его.

Чтобы исключить экземпляр из кластера, нажмите **...** рядом с ним, затем нажмите **Expel server** (Исключить сервер) и **Expel**:



Примечание: Есть два ограничения:

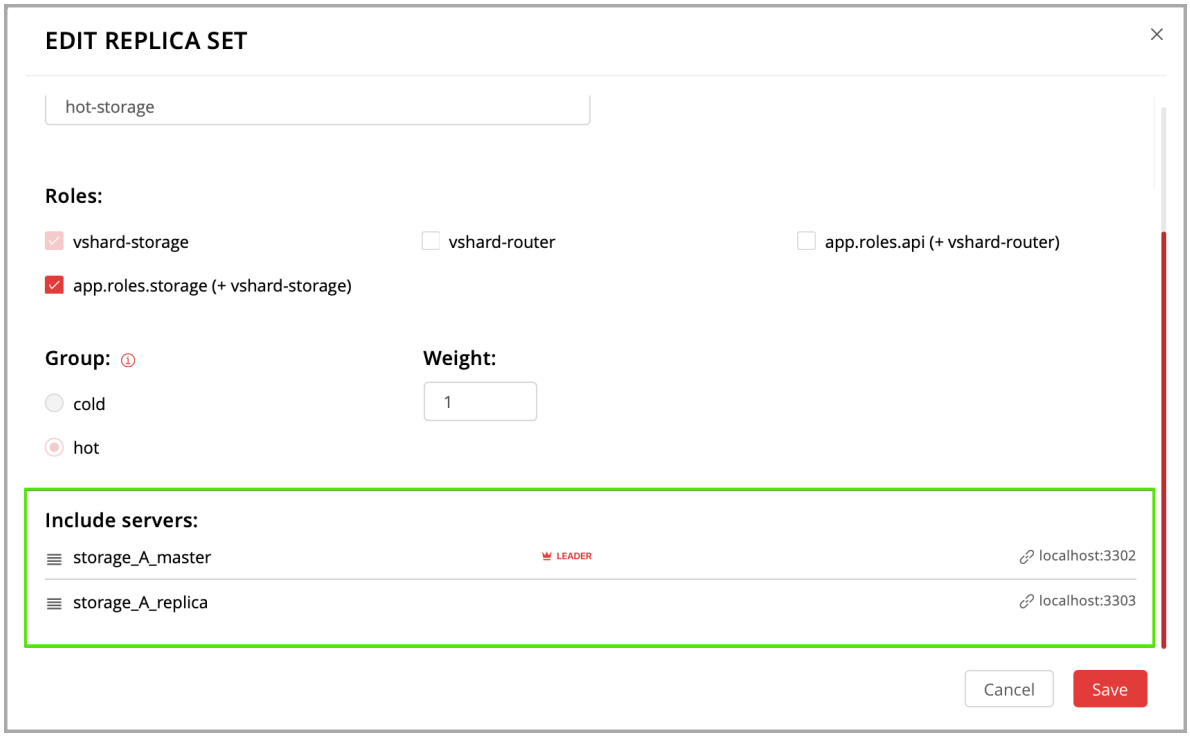
- Нельзя исключить лидера, если у него есть реплика. Сначала передайте роль лидера.
- Нельзя исключить vshard-хранилище, если оно хранит сегменты. Установите значение веса на 0 и дождитесь завершения ребалансировки.

Включение автоматического восстановления после отказа

В конфигурации кластера мастер-реплика с включенным автоматическим восстановлением после отказа, если происходит отказ указанного пользователем мастера из любого набора реплик, кластер автоматически выбирает следующую реплику из списка приоритетов и назначает ей роль активного мастера (чтение/запись). Когда вышедший из строя мастер возвращается к работе, его роль восстанавливается, и активный мастер снова становится репликой (только для чтения). Это работает для всех ролей.

Чтобы задать приоритет в наборе реплик:

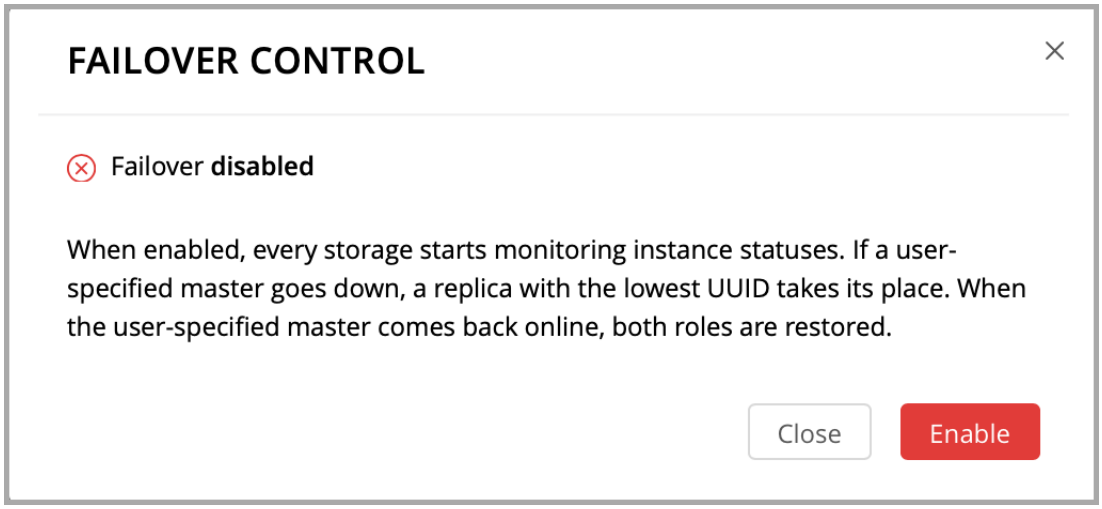
1. Нажмите **Edit** (Изменить) рядом с необходимым набором реплик.
2. Выполните прокрутку в окне **Edit replica set** (Изменить набор реплик), чтобы увидеть весь список серверов.
3. Перенесите реплики на необходимые места в списке приоритета и нажмите **Save** (Сохранить):



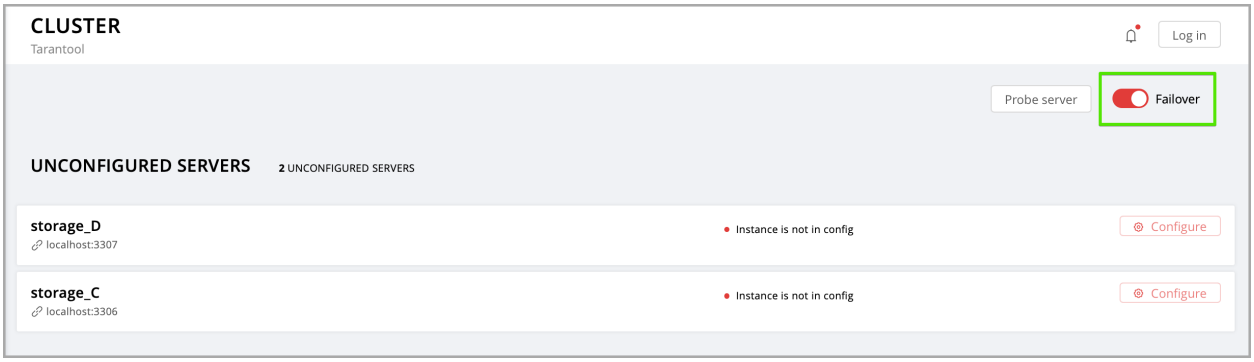
По умолчанию, автоматическое восстановление после отказа отключено. Чтобы включить его:

1. Нажмите **Failover** (Восстановление после отказа):

2. В окне **Failover control** (Управление восстановлением после отказа) нажмите **Enable** (Включить):



Статус восстановления после отказа изменится на enabled (включено):

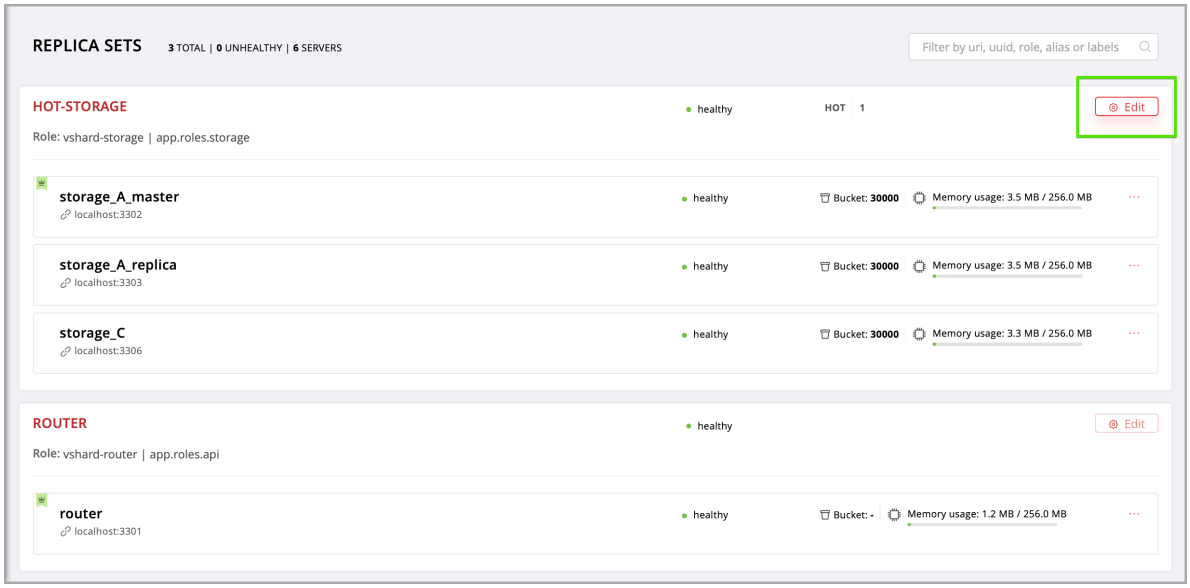


Для получения дополнительной информации см. раздел по репликации.

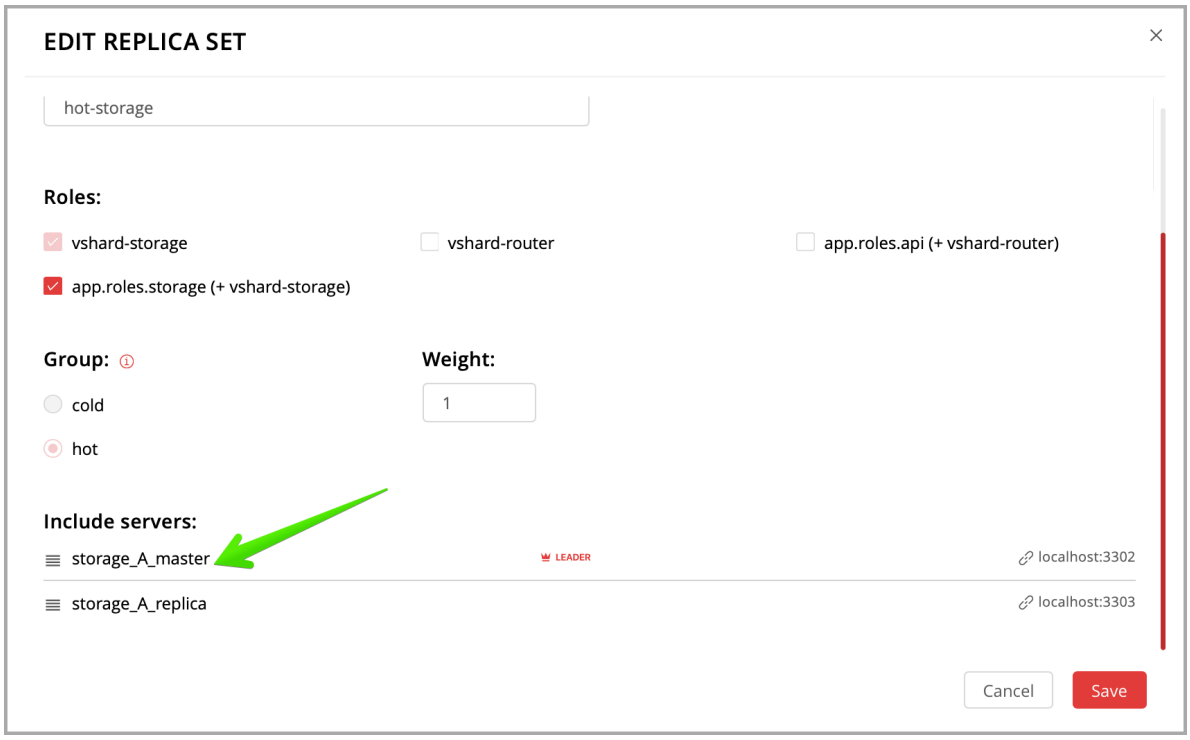
Смена мастера в наборе реплик

Чтобы вручную сменить мастера в наборе реплик:

1. Нажмите кнопку **Edit** (Изменить) рядом с необходимым набором реплик:



2. Выполните прокрутку в окне **Edit replica set** (Изменить набор реплик), чтобы увидеть весь список серверов. Мастером будет верхний сервер.



3. Перенесите необходимый сервер наверх и нажмите **Save** (Сохранить).

Новый мастер автоматически войдет в режим для чтения и записи, а предыдущий мастер будет использоваться только для чтения. Это работает для всех ролей.

Управление пользователями

На вкладке **Users** (Пользователи) можно включать и отключать аутентификацию, а также добавлять, удалять, изменять и просматривать пользователей, у которых есть доступ к веб-интерфейсу.

Обратите внимание, что вкладка **Users** (Пользователи) доступна только в том случае, если в веб-интерфейсе *реализована* авторизация.

Кроме того, некоторые функции (например, удаление пользователей) можно отключить в конфигурации кластера, что регулируется при помощи настройки auth_backend_name, которая передается в `cartridge.cfg()`.

Устранение конфликтов

В Tarantool встроен механизм асинхронной репликации. Как следствие, записи распределяются между репликами с задержкой, поэтому могут возникнуть конфликты.

Для предотвращения конфликтов используется специальный триггер `space.before_replace`. Он выполняется каждый раз перед внесением изменений в таблицу, для которой он был настроен. Функция триггера реализована на языке программирования Lua. Эта функция принимает в качестве аргументов исходные значения изменяемого кортежа и новые значения. Функция возвращает значение, которое используется для изменения результата операции: это будет новое значение измененного кортежа.

Для операций вставки старое значение отсутствует, поэтому в качестве первого аргумента передается нулевое значение `nil`.

Для операций удаления отсутствует новое значение, поэтому нулевое значение `nil` передается в качестве второго аргумента. Функция триггера также может возвращать нулевое значение `nil`, превращая эту операцию в удаление.

В примере ниже показано, как использовать триггер `space.before_replace`, чтобы предотвратить конфликты репликации. Предположим, у нас есть таблица `box.space.test`, которая изменяется в нескольких репликах одновременно. В этой таблице мы храним одно поле полезной нагрузки. Чтобы обеспечить согласованность, мы также сохраняем время последнего изменения в каждом кортеже этой таблицы и устанавливаем триггер `space.before_replace`, который отдает предпочтение новым кортежам. Ниже приведен код на Lua:

```
fiber = require('fiber')
-- define a function that will modify the function test_replace(tuple)
        -- add a timestamp to each tuple in the space
        tuple = box.tuple.new(tuple):update{{'!', 2, fiber.time()}}
        box.space.test:replace(tuple)
end
```

(continues on next page)

(продолжение с предыдущей страницы)

```
box.cfg{ } -- restore from the local directory
-- set the trigger to avoid conflicts
box.space.test:before_replace(function(old, new)
        if old ~= nil and new ~= nil and new[2] < old[2] then
                return old -- ignore the request
        end
        -- otherwise apply as is
end)
box.cfg{ replication = {...} } -- subscribe
```

Мониторинг кластера через CLI

В данном разделе описываются параметры, которые можно отслеживать в административной консоли.

Подключение к узлам через CLI

В каждом узле Tarantool (роутер/хранилище) есть административная консоль (интерфейс командной строки) для отладки, мониторинга и разрешения проблем. Консоль выступает в качестве интерпретатора Lua и отображает результат в удобном для восприятия формате YAML. Чтобы подключиться к экземпляру Tarantool через консоль, выполните команду:

```
$ tarantoolctl connect <instance_hostname>:<port>
```

где `<имя_хоста_экземпляра>:<порт>` – это URI данного экземпляра.

Мониторинг хранилищ

Для получения информации об узлах хранения данных используйте `vshard.storage.info()`.

Пример вывода

```
tarantool> vshard.storage.info()
---
- replicasets:
    <replicaset_2>:
    uuid: <replicaset_2>
    master:
        uri: storage:storage@127.0.0.1:3303
    <replicaset_1>:
    uuid: <replicaset_1>
    master:
        uri: storage:storage@127.0.0.1:3301
  bucket: <!-- buckets status
    receiving: 0 <!-- buckets in the RECEIVING state
    active: 2 <!-- buckets in the ACTIVE state
    garbage: 0 <!-- buckets in the GARBAGE state (are to be deleted)
    total: 2 <!-- total number of buckets
    sending: 0 <!-- buckets in the SENDING state
  status: 1 <!-- the status of the replica set
  replication:
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    status: disconnected <!-- the status of the replication
    idle: <idle>
  alerts:
  - ['MASTER_IS_UNREACHABLE', 'Master is unreachable: disconnected']
```

Список состояний

Код	Уровень критичности	Описание
0	Зеленый	Набор реплик работает в обычном режиме.
1	Желтый	Есть некоторые проблемы, но они не влияют на эффективность набора реплик (их стоит отметить, но они не требуют немедленного вмешательства).
2	Оранжевый	Набор реплик не восстановился после сбоя.
3	Красный	Набор реплик отключен.

Возможные проблемы

- `MISSING_MASTER` — В конфигурации набора реплик отсутствует мастер-узел.

  Уровень критичности: Оранжевый.

  Состояние кластера: Ухудшение работы запросов на изменение данных к набору реплик.

  Решение: Задайте мастер-узел для набора реплик, используя API.

- `UNREACHABLE_MASTER` — Отсутствует соединение между мастером и репликой.

  Уровень критичности:

  - Если значение бездействия не превышает порог T1 (1 с.) – Желтый,
  - Если значение бездействия не превышает порог T2 (5 с.) – Оранжевый,
  - Если значение бездействия не превышает порог T3 (10 с.) – Красный.

  Состояние кластера: При запросах на чтение из реплики данные могут быть устаревшими по сравнению с данными на мастере.

  Решение: Повторно подключитесь к мастеру: устраните проблемы с сетью, сбросьте текущий мастер, переключитесь на другой мастер.

- `LOW_REDUNDANCY` — У мастера есть доступ только к одной реплике.

  Уровень критичности: Желтый.

  Состояние кластера: Коэффициент избыточности хранения данных равен 2. Он ниже минимального рекомендуемого значения для использования в производстве.

  Решение: Проверить конфигурацию кластера:

  - Если в конфигурации указан только один мастер и одна реплика, рекомендуется добавить хотя бы еще одну реплику, чтобы коэффициент избыточности достиг 3.
  - Если в конфигурации указаны три или более реплик, проверьте статусы реплик и сетевое соединение между репликами.

- `INVALID_REBALANCING` — Нарушен инвариант балансировки. Во время миграции узел хранения может либо отправлять сегменты, либо получать их. Поэтому не должно быть так, чтобы набор реплик отправлял сегменты в один набор реплик и одновременно получал сегменты из другого набора реплик.

  Уровень критичности: Желтый.

  Состояние кластера: Балансировка приостановлена.

  Решение: Есть две возможные причины нарушения инварианта:

  – Отказ балансировщика.

  – Статус сегмента был изменен вручную.

  В любом случае обратитесь в техническую поддержку Tarantool'а.

- `HIGH_REPLICATION_LAG` — Отставание реплики превышает порог T1 (1 с.).

  Уровень критичности:

  – Если отставание не превышает порог T1 (1 с.) – Желтый;

  – Если отставание не превышает порог T2 (5 с.) – Оранжевый.

  Состояние кластера: При запросах только на чтение из реплики данные могут быть устаревшими по сравнению с данными на мастере.

  Решение: Проверьте статус репликации на реплике. Более подробные инструкции приведены в руководстве по разрешению проблем по.

- `OUT_OF_SYNC` — Произошла рассинхронизация. Отставание превышает порог T3 (10 с.).

  Уровень критичности: Красный.

  Состояние кластера: При запросах только на чтение из реплики данные могут быть устаревшими по сравнению с данными на мастере.

  Решение: Проверьте статус репликации на реплике. Более подробные инструкции приведены в руководстве по разрешению проблем по.

- `UNREACHABLE_REPLICA` — Одна или несколько реплик недоступны.

  Уровень критичности: Желтый.

  Состояние кластера: Коэффициент избыточности хранения данных для данного набора реплик меньше заданного значения. Если реплика стоит следующей в очереди на балансировку (в соответствии с настройками веса), запросы перенаправляются в реплику, которая все еще находится в очереди.

  Решение: Проверьте сообщение об ошибке и выясните, какая реплика недоступна. Если реплика отключена, включите ее. Если это не поможет, проверьте состояние сети.

- `UNREACHABLE_REPLICASET` — Все реплики, кроме текущей, недоступны. **Уровень критичности:** Красный.

  Состояние кластера: Реплика хранит устаревшие данные.

  Решение: Проверьте, включены ли другие реплики. Если все реплики включены, проверьте наличие сетевых проблем на мастере. Если реплики отключены, сначала проверьте их: возможно, мастер работает правильно.

Мониторинг роутеров

Для получения информации о роутерах используйте `vshard.router.info()`.

Пример вывода

```
tarantool> vshard.router.info()
---
- replicasets:
    <replica set UUID>:
      master:
        status: <available / unreachable / missing>
        uri: <!-- URI of master
        uuid: <!-- UUID of instance
      replica:
        status: <available / unreachable / missing>
        uri: <!-- URI of replica used for slave requests
        uuid: <!-- UUID of instance
      uuid: <!-- UUID of replica set
    <replica set UUID>: ...
    ...
  status: <!-- status of router
  bucket:
    known: <!-- number of buckets with the known destination
    unknown: <!-- number of other buckets
  alerts: [<alert code>, <alert description>], ...
```

Список состояний

Код	Уровень критичности	Описание
0	Зеленый	Роутер работает в обычном режиме.
1	Желтый	Некоторые реплики недоступны, что влияет на скоость выполнения запросов на чтение.
2	Оранжевый	Работа запросов на изменение данных ухудшена.
3	Красный	Работа запросов на чтение данных ухудшена.

Возможные проблемы

Примечание: В зависимости от характера проблемы используйте либо UUID реплики, либо UUID набора реплик.

- `MISSING_MASTER` — В конфигурации одного или нескольких наборов реплик не указан мастер.

  Уровень критичности: Оранжевый.

  Состояние кластера: Частичное ухудшение работы запросов на изменение данных.

  Решение: Укажите мастера в конфигурации.

- `UNREACHABLE_MASTER` — Роутер потерял соединение с мастером одного или нескольких наборов реплик.

  Уровень критичности: Оранжевый.

  Состояние кластера: Частичное ухудшение работы запросов на изменение данных.

  Решение: Восстановите соединение с мастером. Сначала проверьте, включен ли мастер. Если он включен, проверьте состояние сети.

- `SUBOPTIMAL_REPLICA` — Восстановите соединение с мастером. Сначала проверьте, включен ли мастер. Если он включен, проверьте состояние сети.

  Уровень критичности: Желтый.

  Состояние кластера: Запросы только на чтение направляются на резервную реплику.

  Решение: Проверьте статус оптимальной реплики и ее сетевого подключения.

- `UNREACHABLE_REPLICASET` — Набор реплик недоступен как для запросов только на чтение, так и для запросов на изменение данных.

  Уровень критичности: Красный.

  Состояние кластера: Частичное ухудшение работы запросов на изменение данных и на чтение данных.

  Решение: В наборе реплик недоступны мастер и реплика. Проверьте сообщение об ошибке, чтобы найти этот набор реплик. Исправьте ошибку, как описано в решении ошибки *UNREACHABLE_REPLICA*.

Обновление схемы

Во время перехода на более новую версию Tarantool, пожалуйста, не забудьте:

1. Остановить кластер
2. Убедиться, что включена *опция* `upgrade_schema`
3. Затем снова запустить кластер

Это автоматически запустит box.schema.upgrade() на лидере в соответствии с приоритетом (failover priority) в настройках набора реплик.

Аварийное восстановление

См. раздел Аварийное восстановление в руководстве по Tarantool.

Резервное копирование

См. раздел Резервное копирование в руководстве по Tarantool.

4.3.4 Troubleshooting

First of all, see a similar guide in the Tarantool manual. Below you can find other Cartridge-specific problems considered.

Editing clusterwide configuration in WebUI returns an error

Examples:

- `NetboxConnectError: "localhost:3302": Connection refused`;
- `Prepare2pcError: Instance state is OperationError, can't apply config in this state`.

The root problem: all cluster instances are equal, and all of them store a copy of clusterwide configuration, which must be the same. If an instance degrades (can't accept new configuration) – the quorum is lost. This prevents further configuration modifications to avoid inconsistency.

But sometimes inconsistency is needed to repair the system, at least partially and temporarily. It can be achieved by disabling degraded instances.

Solution:

1. Connect to the console of the alive instance.

```
tarantoolctl connect unix/:/var/run/tarantool/<app-name>.<instance-name>.control
```

2. Inspect what's going on.

```
cartridge = require('cartridge')
report = {}
for _, srv in pairs(cartridge.admin_get_servers()) do
    report[srv.uuid] = {uri = srv.uri, status = srv.status, message = srv.message}
end
return report
```

3. If you're ready to proceed, run the following snippet. It'll disable all instances which are not healthy. After that, you can use the WebUI as usual.

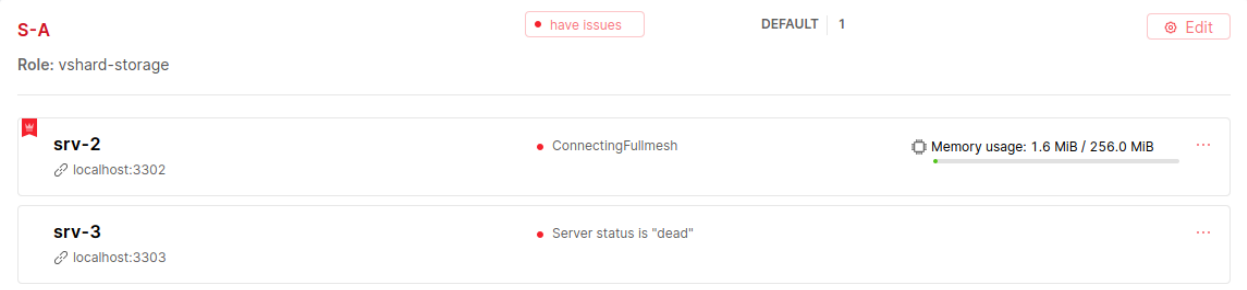
```
disable_list = {}
for uuid, srv in pairs(report) do
    if srv.status ~= 'healthy' then
       table.insert(disable_list, uuid)
    end
end
return cartridge.admin_disable_servers(disable_list)
```

4. When it's necessary to bring disabled instances back, re-enable them in a similar manner:

```
cartridge = require('cartridge')
enable_list = {}
for _, srv in pairs(cartridge.admin_get_servers()) do
    if srv.disabled then
       table.insert(enable_list, srv.uuid)
    end
end
return cartridge.admin_enable_servers(enable_list)
```

An instance is stuck in the ConnectingFullmesh state upon restart

Example:



The root problem: after restart, the instance tries to connect to all its replicas and remains in the `ConnectingFullmesh` state until it succeeds. If it can't (due to replica URI unavailability or for any other reason) – it's stuck forever.

Solution:

Set the replication_connect_quorum option to zero. It may be accomplished in two ways:

- By restarting it with the corresponding option set (in environment variables or in the *instance configuration file*);
- Or without restart – by running the following one-liner:

```
echo "box.cfg({replication_connect_quorum = 0})" | tarantoolctl connect \
unix/:/var/run/tarantool/<app-name>.<instance-name>.control
```

I want to run an instance with a new advertise_uri

The root problem: `advertise_uri` parameter is persisted in the clusterwide configuration. Even if it changes upon restart, the rest of the cluster keeps using the old one, and the cluster may behave in an odd way.

Solution:

The clusterwide configuration should be updated.

1. Make sure all instances are running and not stuck in the ConnectingFullmesh state (see *above*).
2. Make sure all instances have discovered each other (i.e. they look healthy in the WebUI).
3. Run the following snippet in the Tarantool console. It'll prepare a patch for the clusterwide configuration.

```
cartridge = require('cartridge')
members = require('membership').members()

edit_list = {}
changelog = {}
for _, srv in pairs(cartridge.admin_get_servers()) do
    for _, m in pairs(members) do
        if m.status == 'alive'
        and m.payload.uuid == srv.uuid
        and m.uri ~= srv.uri
        then
            table.insert(edit_list, {uuid = srv.uuid, uri = m.uri})
            table.insert(changelog, string.format('%s -> %s (%s)', srv.uri, m.uri, m.payload.
↪alias))
            break
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        end
    end
end
return changelog
```

As a result you'll see a brief summary like the following one:

```
localhost:3301> return changelog
---
- - localhost:13301 -> localhost:3301 (srv-1)
  - localhost:13302 -> localhost:3302 (srv-2)
  - localhost:13303 -> localhost:3303 (srv-3)
  - localhost:13304 -> localhost:3304 (srv-4)
  - localhost:13305 -> localhost:3305 (srv-5)
...
```

4. Finally, apply the patch:

```
cartridge.admin_edit_topology({servers = edit_list})
```

The cluster is doomed, I've edited the config manually. How do I reload it?

Предупреждение: Please be aware that it's quite risky and you know what you're doing. There's some useful information about clusterwide configuration anatomy and «normal» management API.

But if you're still determined to reload the configuration manually, you can do (in the Tarantool console):

```
function reload_clusterwide_config()
    local changelog = {}

    local ClusterwideConfig = require('cartridge.clusterwide-config')
    local confapplier = require('cartridge.confapplier')

    -- load config from filesystem
    table.insert(changelog, 'Loading new config...')

    local cfg, err = ClusterwideConfig.load('./config')
    if err ~= nil then
        return changelog, string.format('Failed to load new config: %s', err)
    end

    -- check instance state
    table.insert(changelog, 'Checking instance config state...')

    local roles_configured_state = 'RolesConfigured'
    local connecting_fullmesh_state = 'ConnectingFullmesh'

    local state = confapplier.wish_state(roles_configured_state, 10)

    if state == connecting_fullmesh_state then
        return changelog, string.format(
            'Failed to reach %s config state. Stuck in %s. ' ..
                'Call "box.cfg({replication_connect_quorum = 0})" in instance console and try again
↪',
```

(continues on next page)

(продолжение с предыдущей страницы)

```
            roles_configured_state, state
        )
    end

    if state ~= roles_configured_state then
        return changelog, string.format(
            'Failed to reach %s config state. Stuck in %s',
            roles_configured_state, state
        )
    end

    -- apply config changes
    table.insert(changelog, 'Applying config changes...')

    cfg:lock()
    local ok, err = confapplier.apply_config(cfg)
    if err ~= nil then
        return changelog, string.format('Failed to apply new config: %s', err)
    end

    table.insert(changelog, 'Cluster-wide configuration was successfully updated')

    return changelog
end

reload_clusterwide_config()
```

This snippet reloads the configuration on a single instance. All other instances continue operating as before.

Примечание: If further configuration modifications are made with a two-phase commit (e.g. via the WebUI or with the Lua API), the active configuration of an active instance will be spread across the cluster.

Repairing cluster using Cartridge CLI *repair* command

Cartridge CLI has repair command since version 2.3.0.

It can be used to get current topology, remove instance from cluster, change repicaset leader or change instance advertise URI.

Примечание: `cartridge repair` patches the cluster-wide configuration files of application instances placed ON THE LOCAL MACHINE. It means that running `cartridge repair` on all machines is user responsibility.

Примечание: It's not enough to apply new configuration: the configuration should be reloaded by the instance. If your application uses `cartridge >= 2.0.0`, you can simply use `--reload` flag to reload configuration. Otherwise, you need to restart instances or reload configuration manually.

Changing instance advertise URI

To change instance advertise URI you have to perform these actions:

1. Start instance with a new advertise URI. The easiest way is to change `advertise_uri` value in the *instance configuration file*).
2. Make sure instances are running and not stuck in the ConnectingFullmesh state (see *above*).
3. Get instance UUID: * open `server details` tab in WebUI; * call `cartridge repair list-topology --name <app-name>` and find desired instance UUID: * get instance `box.info().uuid`:

```
echo "return box.info().uuid" | tarantoolctl connect \
unix/:/var/run/tarantool/<app-name>.<instance-name>.control
```

4. Now we need to update instance advertise URI in all instances cluster-wide configuration files on each machine. Run `cartridge repair set-advertise-uri` with `--dry-run` flag on each machine to check cluster-wide config changes computed by `cartridge-cli`:

```
cartridge repair set-advertise-uri \
  --name myapp \
  --dry-run \
  <instance-uuid> <new-advertise-uri>
```

5. Run `cartridge repair set-advertise-uri` without `--dry-run` flag on each machine to apply config changes computed by `cartridge-cli`. If your application uses `cartridge >= 2.0.0`, you can specify `--reload` flag to load new cluter-wide configuration on instances. Otherwise, you need to restart instances or reload configuration manually.

```
cartridge repair set-advertise-uri \
  --name myapp \
  --verbose \
  --reload \
  <instance-uuid> <new-advertise-uri>
```

Changing replicaset leader

You can change replicaset leader using `cartridge repair` command.

1. Get replicaset UUID and new leader UUID (in WebUI or by calling `cartridge repair list-topology --name <app-name>`).
2. Now we need to update cluster-wide config for all instances on each machine. Run `cartridge repair set-leader` with `--dry-run` flag on each machine to check cluster-wide config changes computed by “ cartridge-cli“:

```
cartridge repair set-leader \
  --name myapp \
  --dry-run \
  <replicaset-uuid> <instance-uuid>
```

3. Run `cartridge repair set-advertise-uri` without `--dry-run` flag on each machine to apply config changes computed by `cartridge-cli`. If your application uses `cartridge >= 2.0.0`, you can specify `--reload` flag to load new cluter-wide configuration on instances. Otherwise, you need to restart instances or reload configuration manually.

```
cartridge repair set-leader \
  --name myapp \
  --verbose \
  --reload \
  <replicaset-uuid> <instance-uuid>
```

Removing instance from the cluster

You can remove instance from cluster using `cartridge repair` command.

1. Get instance UUID: * open `server details` tab in WebUI; * call `cartridge repair list-topology --name <app-name>` and find desired instance UUID: * get instance `box.info().uuid`:

```
echo "return box.info().uuid" | tarantoolctl connect \
unix/:/var/run/tarantool/<app-name>.<instance-name>.control
```

2. Now we need to update cluster-wide config for all instances on each machine. Run `cartridge repair remove-instance` with `--dry-run` flag on each machine to check cluster-wide config changes computed by `cartridge-cli`:

```
cartridge repair remove-instance \
  --name myapp \
  --dry-run \
  <replicaset-uuid>
```

3. Run `cartridge repair remove-instance` without `--dry-run` flag on each machine to apply config changes computed by `cartridge-cli`. If your application uses `cartridge >= 2.0.0`, you can specify `--reload` flag to load new cluter-wide configuration on instances. Otherwise, you need to restart instances or reload configuration manually.

```
cartridge repair set-leader \
  --name myapp \
  --verbose \
  --reload \
  <replicaset-uuid> <instance-uuid>
```

4.3.5 Table of contents

Module *cartridge*

Tarantool framework for distributed applications development.

Cartridge provides you a simple way to manage distributed applications operations. The cluster consists of several Tarantool instances acting in concert. Cartridge does not care about how the instances start, it only cares about the configuration of already running processes.

Cartridge automates vshard and replication configuration, simplifies custom configuration and administrative tasks.

Functions

cfg (opts, box_opts)

Initialize the cartridge module.

After this call, you can operate the instance via Tarantool console. Notice that this call does not initialize the database - `box.cfg` is not called yet. Do not try to call `box.cfg` yourself: `cartridge` will do it when it is time.

Both *cartridge.cfg* and `box.cfg` options can be configured with command-line arguments or environment variables.

Parameters:

- *opts:* Available options are:
  - *workdir:* (optional string) a directory where all data will be stored: snapshots, wal logs and cartridge config file.(default: «.», overridden byenv `TARANTOOL_WORKDIR` ,args `--workdir`)
  - *advertise_uri:* (optional string) either «<*HOST*>:<*PORT*>» or «<*HOST*>:» or «<*PORT*>».Used by other instances to connect to the current one.When <*HOST*> isn't specified, it's detected as the only non-local IP address.If there is more than one IP address available - defaults to «localhost».When <*PORT*> isn't specified, it's derived as follows:If the *TARANTOOL_INSTANCE_NAME* has numeric suffix _<*N*>, then <*PORT*> = *3300*+<*N*>.Otherwise default <*PORT*> = *3301* is used.
  - *cluster_cookie:* (optional string) secret used to separate unrelated applications, whichprevents them from seeing each other during broadcasts.Also used as admin password in HTTP and binary connections and forencrypting internal communications.Allowed symbols are `[a-zA-Z0-9_.~-]` .(default: «secret-cluster-cookie», overridden byenv `TARANTOOL_CLUSTER_COOKIE` ,args `--cluster-cookie`)
  - *swim_broadcast:* (optional **boolean**) Announce own *advertise_uri* over UDP broadcast.Cartridge health-checks are governed by SWIM protocol. To simplifyinstances discovery on start it can UDP broadcast all networksknown from *getifaddrs()* C call. The broadcast is sent to severalports: default 3301, the <*PORT*> from the *advertise_uri* option,and its neighbours <*PORT*>*+1* and <*PORT*>*-1*.(**Added** in v2.3.0-23,default: true, overridden byenv *TARANTOOL_SWIM_BROADCAST*,args *–swim-broadcast*)
  - *bucket_count:* (optional **number**) bucket count for vshard cluster. See vshard doc for more details.(default: 30000, overridden byenv `TARANTOOL_BUCKET_COUNT` ,args `--bucket-count`)
  - *vshard_groups:* (optional {[string]=VshardGroup,...}) vshard storage groups, table keys used as names
  - *http_enabled:* (optional **boolean**) whether http server should be started(default: true, overridden byenv *TARANTOOL_HTTP_ENABLED*,args *–http-enabled*)
  - *http_port:* (string or **number**) port to open administrative UI and API on(default: 8081, derived from'TARANTOOL_INSTANCE_NAME',overridden byenv *TARANTOOL_HTTP_PORT*,args *–http-port*)
  - *alias:* (optional string) human-readable instance name that will be available in administrative UI(default: argparse instance name, overridden byenv *TARANTOOL_ALIAS*,args *–alias*)
  - *roles:* (table) list of user-defined roles that will be availableto enable on the instance_uuid
  - *auth_enabled:* (optional **boolean**) toggle authentication in administrative UI and API(default: false)
  - *auth_backend_name:* (optional string) user-provided set of callbacks related to authentication

  - *console_sock:* (optional string) Socket to start console listening on.(default: nil, overridden byenv `TARANTOOL_CONSOLE_SOCK` ,args `--console-sock`)
  - *webui_blacklist:* (optional {string,...}) List of pages to be hidden in WebUI.(**Added** in v2.0.1-54, default: `{}`)
  - *upgrade_schema:* (optional **boolean**) Run schema upgrade on the leader instance.(**Added** in v2.0.2-3,default: `false` , overridden byenv `TARANTOOL_UPGRADE_SCHEMA` args `--upgrade-schema`)
  - *roles_reload_allowed:* (optional **boolean**) Allow calling *cartridge.reload_roles*.(**Added** in v2.3.0-73, default: `false`)
- *box_opts:* (optional table) tarantool extra box.cfg options (e.g. memtx_memory),that may require additional tuning

Returns:

true

Or

(**nil**)

(table) Error description

reload_roles ()

Perform hot-reload of cartridge roles code.

This is an experimental feature, it's only allowed if the application enables it explicitly: `cartridge.cfg({roles_reload_allowed = true})` .

Reloading starts by stopping all roles and restoring the initial state. It's supposed that a role cleans up the global state when stopped, but even if it doesn't, cartridge kills all fibers and removes global variables and HTTP routes.

All Lua modules that were loaded during *cartridge.cfg* are unloaded, including supplementary modules required by a role. Modules, loaded before *cartridge.cfg* aren't affected.

Instance performs roles reload in a dedicated state `ReloadingRoles` . If reload fails, the instance enters the `ReloadError` state, which can later be retried. Otherwise, if reload succeeds, instance proceeds to the `ConfiguringRoles` state and initializes them as usual with `validate_config()` , `init()` , and `apply_config()` callbacks.

Returns:

(**boolean**) true

Or

(**nil**)

(table) Error description

is_healthy ()

Check the cluster health. It is healthy if all instances are healthy.

The function is designed mostly for testing purposes.

Returns:

(**boolean**) true / false

Tables

VshardGroup

Vshard storage group configuration.

Every vshard storage must be assigned to a group.

Fields:

- *bucket_count:* (**number**) Bucket count for the storage group.

Global functions

_G.cartridge_get_schema ()

Get clusterwide DDL schema.

(**Added** in v1.2.0-28)

Returns:

(string) Schema in YAML format

Or

(**nil**)

(table) Error description

_G.cartridge_set_schema (schema)

Apply clusterwide DDL schema.

(**Added** in v1.2.0-28)

Parameters:

- *schema:* (string) in YAML format

Returns:

(string) The same new schema

Or

(**nil**)

(table) Error description

Clusterwide DDL schema

get_schema ()

Get clusterwide DDL schema. It's like **_G.cartridge_get_schema**, but isn't a global variable.

(**Added** in v2.0.1-54)

Returns:

(string) Schema in YAML format

Or

(**nil**)

(table) Error description

set_schema (schema)

Apply clusterwide DDL schema. It's like **_G.cartridge_set_schema**, but isn't a global variable.

(**Added** in v2.0.1-54)

Parameters:

- *schema:* (string) in YAML format

Returns:

(string) The same new schema

Or

(**nil**)

(table) Error description

Cluster administration

ServerInfo

Instance general information.

Fields:

- *alias:* (string) Human-readable instance name.
- *uri:* (string)
- *uuid:* (string)
- *disabled:* (**boolean**)
- *status:* (string) Instance health.
- *message:* (string) Auxilary health status.
- *replicaset:* (*ReplicasetInfo*) Circular reference to a replicaset.
- *priority:* (**number**) Leadership priority for automatic failover.
- *clock_delta:* (**number**) Difference between remote clock and the current one (inseconds), obtained from the membership module (SWIM protocol).Positive values mean remote clock are ahead of local, and viceversa.
- *zone:* (string)

ReplicasetInfo

Replicaset general information.

Fields:

- *uuid:* (string) The replicaset UUID.
- *roles:* ({string,...}) Roles enabled on the replicaset.
- *status:* (string) Replicaset health.
- *master:* (*ServerInfo*) Replicaset leader according to configuration.
- *active_master:* (*ServerInfo*) Active leader.
- *weight:* (**number**) Vshard replicaset weight.Matters only if vshard-storage role is enabled.
- *vshard_group:* (string) Name of vshard group the replicaset belongs to.
- *all_rw:* (**boolean**) A flag indicating that all servers in the replicaset should be read-write.
- *alias:* (string) Human-readable replicaset name.
- *servers:* (*{ServerInfo,...}*) Circular reference to all instances in the replicaset.

admin_get_servers ([uuid])

Get servers list. Optionally filter out the server with the given uuid.

Parameters:

- *uuid:* (string) (optional)

Returns:

(*{ServerInfo,...}*)

Or

(**nil**)

(table) Error description

admin_get_replicasets ([uuid])

Get replicasets list. Optionally filter out the replicaset with given uuid.

Parameters:

- *uuid:* (string) (optional)

Returns:

(*{ReplicasetInfo,...}*)

Or

(**nil**)

(table) Error description

admin_probe_server (uri)

Discover an instance.

Parameters:

- *uri:* (string)

admin_enable_servers (uuids)

Enable nodes after they were disabled.

Parameters:

- *uuids:* ({string,...})

Returns:

(*{ServerInfo,...}*)

Or

(**nil**)

(table) Error description

admin_disable_servers (uuids)

Temporarily diable nodes.

Parameters:

- *uuids:* ({string,...})

Returns:

(*{ServerInfo,...}*)

Or

(**nil**)

(table) Error description

admin_bootstrap_vshard ()

Call `vshard.router.bootstrap()` . This function distributes all buckets across the replica sets.

Returns:

(**boolean**) *true*

Or

(**nil**)

(table) Error description

Automatic failover management

FailoverParams

Failover parameters.

(**Added** in v2.0.2-2)

Fields:

- *mode:* (string) Supported modes are «disabled», «eventual» and «stateful»
- *state_provider:* (optional string) Supported state providers are «tarantool» and «etcd2».
- *failover_timeout:* (**number**) (added in v2.3.0-52)Timeout (in seconds), used by membership tomark `suspect` members as `dead` (default: 20)
- *tarantool_params:* (added in v2.0.2-2)
  - *uri:* (string)
  - *password:* (string)
- *etcd2_params:* (added in v2.1.2-26)
  - *prefix:* (string) Prefix used for etcd keys: <*prefix*>/*lock* and‘<prefix>/leaders‘
  - *lock_delay:* (optional **number**) Timeout (in seconds), determines lock's time-to-live (default: 10)
  - *endpoints:* (optional table) URIs that are used to discover and to access etcd cluster instances.(default: `{'http://localhost:2379', 'http://localhost:4001'}`)
  - *username:* (optional string) (default: «»)
  - *password:* (optional string) (default: «»)
- *fencing_enabled:* (**boolean**) (added in v2.3.0-57)Abandon leadership when both the state provider quorum and atleast one replica are lost (suitable in stateful mode only,default: false)
- *fencing_timeout:* (**number**) (added in v2.3.0-57)Time (in seconds) to actuate fencing after the check fails(default: 10)
- *fencing_pause:* (**number**) (added in v2.3.0-57)The period (in seconds) of performing the check(default: 2)

failover_get_params ()

Get failover configuration.

(**Added** in v2.0.2-2)

Returns:

(*FailoverParams*)

failover_set_params (opts)

Configure automatic failover.

(**Added** in v2.0.2-2)

Parameters:

- *opts:*
  - *mode:* (optional string)
  - *state_provider:* (optional string)
  - *failover_timeout:* (optional **number**) (added in v2.3.0-52)
  - *tarantool_params:* (optional table)
  - *etcd2_params:* (optional table) (added in v2.1.2-26)
  - *fencing_enabled:* (optional **boolean**) (added in v2.3.0-57)
  - *fencing_timeout:* (optional **number**) (added in v2.3.0-57)
  - *fencing_pause:* (optional **number**) (added in v2.3.0-57)

Returns:

(**boolean**) *true* if config applied successfully

Or

(**nil**)

(table) Error description

failover_promote (replicaset_uuid[, opts])

Promote leaders in replicasets.

Parameters:

- *replicaset_uuid:* (table)] = leader_uuid }
- *opts:*
  - *force_inconsistency:* (optional **boolean**) (default: **false**)

Returns:

(**boolean**) true On success

Or

(**nil**)

(table) Error description

admin_get_failover ()

Get current failover state.

(**Deprecated** since v2.0.2-2)

admin_enable_failover ()

Enable failover. (**Deprecated** since v2.0.1-95 in favor of *cartridge.failover_set_params*)

admin_disable_failover ()

Disable failover. (**Deprecated** since v2.0.1-95 in favor of *cartridge.failover_set_params*)

Managing cluster topology

admin_edit_topology (args)

Edit cluster topology. This function can be used for:

- bootstrapping cluster from scratch
- joining a server to an existing replicaset
- creating new replicaset with one or more servers
- editing uri/labels of servers
- disabling and expelling servers

(**Added** in v1.0.0-17)

Parameters:

- *args:*
  - *servers:* (optional *{EditServerParams,..}*)
  - *replicasets:* (optional *{EditReplicasetParams,..}*)

EditReplicasetParams

Replicatets modifications.

Fields:

- *uuid:* (optional string)
- *alias:* (optional string)
- *roles:* (optional {string,...})
- *all_rw:* (optional **boolean**)
- *weight:* (optional **number**)
- *failover_priority:* (optional {string,...}) array of uuids specifying servers failover priority
- *vshard_group:* (optional string)
- *join_servers:* (optional *{JoinServerParams,...}*)

EditServerParams

Servers modifications.

Fields:

- *uri:* (optional string)
- *uuid:* (string)

- *zone:* (optional string)
- *labels:* (optional table)
- *disabled:* (optional **boolean**)
- *expelled:* (optional **boolean**) Expelling an instance is permanent and can't be undone.It's suitable for situations when the hardware is destroyed,snapshots are lost and there is no hope to bring it back to life.

JoinServerParams

Parameters required for joining a new server.

Fields:

- *uri:* (string)
- *uuid:* (optional string)
- *zone:* (optional string) (**Added** in v2.4.0-14)
- *labels:* (optional table)

Clusterwide configuration

config_get_readonly ([section_name])

Get a read-only view on the clusterwide configuration.

Returns either `conf[section_name]` or entire `conf` . Any attempt to modify the section or its children will raise an error.

Parameters:

- *section_name:* (string) (optional)

Returns:

(table)

config_get_deepcopy ([section_name])

Get a read-write deep copy of the clusterwide configuration.

Returns either `conf[section_name]` or entire `conf` . Changing it has no effect unless it's used to patch clusterwide configuration.

Parameters:

- *section_name:* (string) (optional)

Returns:

(table)

config_patch_clusterwide (patch)

Edit the clusterwide configuration. Top-level keys are merged with the current configuration. To remove a top-level section, use `patch_clusterwide{key = box.NULL}` .

The function uses a two-phase commit algorithm with the following steps:

I. Patches the current configuration.

II. Validates topology on the current server.

III. Executes the preparation phase (`prepare_2pc`) on every server excluding expelled and disabled servers.

IV. If any server reports an error, executes the abort phase (`abort_2pc`). All servers prepared so far are rolled back and unlocked.

V. Performs the commit phase (`commit_2pc`). In case the phase fails, an automatic rollback is impossible, the cluster should be repaired manually.

Parameters:

- *patch:* (table)

Returns:

(**boolean**) true

Or

(**nil**)

(table) Error description

config_force_reapply (uuids)

Forcefully apply config to the given instances.

In particular:

- Abort two-phase commit (remove `config.prepare` lock)
- Upload the active config from the current instance.
- Apply it (reconfigure all roles)

(**Added** in v2.3.0-68)

Parameters:

- *uuids:* ({string,...})

Returns:

(**boolean**) true

Or

(**nil**)

(table) Error description

Inter-role interaction

service_get (module_name)

Get a module from registry.

Parameters:

- *module_name:* (string)

Returns:

(**nil**)

Or

(table) instance

service_set (module_name, instance)

Put a module into registry or drop it. This function typically doesn't need to be called explicitly, the cluster automatically sets all the initialized roles.

Parameters:

- *module_name:* (string)
- *instance:* (**nil** or table)

Returns:

(**nil**)

Cross-instance calls

rpc_call (role_name, fn_name[, args[, opts]])

Perform a remote procedure call. Find a suitable healthy instance with an enabled role and perform a [`net.box conn:call`](https://tarantool.io/en/doc/latest/reference/reference_lua/net_box/#net-box-call) on it. `rpc.call()` can only be used for functions defined in role return table unlike `net.box conn:call()` , which is used for global functions as well.

Parameters:

- *role_name:* (string)
- *fn_name:* (string)
- *args:* (table) (optional)
- *opts:*
  - *prefer_local:* (optional **boolean**) Don't perform a remote call if possible. When the role is enabledlocally and current instance is healthy the remote netbox call issubstituted with a local Lua function call. When the option isdisabled it never tries to perform call locally and always usesnetbox connection, even to connect self.(default: **true**)
  - *leader_only:* (optional **boolean**) Perform a call only on the replica set leaders.(default: **false**)

- *uri:* (optional string) Force a call to be performed on this particular uri.Disregards member status and `opts.prefer_local` .Conflicts with `opts.leader_only = true` .(added in v1.2.0-63)
- *remote_only:* (*deprecated*) Use `prefer_local` instead.
- *timeout:* passed to `net.box conn:call` options.
- *buffer:* passed to `net.box conn:call` options.
- *on_push:* passed to `net.box conn:call` options.
- *on_push_ctx:* passed to `net.box conn:call` options.

Returns:

conn:call() result

Or

(**nil**)

(table) Error description

Usage:

```
-- myrole.lua
return {
    role_name = 'myrole',
    add = function(a, b) return a + b end,
}
```

```
-- call it as follows:
cartridge.rpc_call('myrole', 'add', {2, 2}) -- returns 4
```

rpc_get_candidates (role_name[, opts])

List instances suitable for performing a remote call.

Parameters:

- *role_name:* (string)
- *opts:*
  - *leader_only:* (optional **boolean**) Filter instances which are leaders now.(default: **false**)
  - *healthy_only:* (optional **boolean**) The member is considered healthy ifit reports either `ConfiguringRoles` or `RolesConfigured` stateand its SWIM status is either `alive` or `suspect` (added in v1.1.0-11, default: **true**)

Returns:

({string,...}) URIs

Authentication and authorization

http_authorize_request (request)

Authorize an HTTP request.

Get username from cookies or basic HTTP authentication.

(**Added** in v1.1.0-4)

Parameters:

- *request:* (table)

Returns:

(**boolean**) Access granted

http_render_response (response)

Render HTTP response.

Inject set-cookie headers into response in order to renew or reset the cookie.

(**Added** in v1.1.0-4)

Parameters:

- *response:* (table)

Returns:

(table) The same response with cookies injected

http_get_username ()

Get username for the current HTTP session.

(**Added** in v1.1.0-4)

Returns:

(string or **nil**)

Deprecated functions

admin_edit_replicaset (args)

Edit replicaset parameters (*deprecated*).

(**Deprecated** since v1.0.0-17 in favor of *cartridge.admin_edit_topology*)

Parameters:

- *args:*
  - *uuid:* (string)
  - *alias:* (string)
  - *roles:* (optional {string,...})
  - *master:* (optional {string,...}) Failover order
  - *weight:* (optional **number**)
  - *vshard_group:* (optional string)
  - *all_rw:* (optional **boolean**)

Returns:

(**boolean**) true

Or

(**nil**)

(table) Error description

admin_edit_server (args)

Edit an instance (*deprecated*).

(**Deprecated** since v1.0.0-17 in favor of *cartridge.admin_edit_topology*)

Parameters:

- *args:*
  - *uuid:* (string)
  - *uri:* (optional string)
  - *labels:* (optional {[string]=string,...})

Returns:

(**boolean**) true

Or

(**nil**)

(table) Error description

admin_join_server (args)

Join an instance to the cluster (*deprecated*).

(**Deprecated** since v1.0.0-17 in favor of *cartridge.admin_edit_topology*)

Parameters:

- *args:*
  - *uri:* (string)
  - *instance_uuid:* (optional string)
  - *replicaset_uuid:* (optional string)
  - *roles:* (optional {string,...})
  - *timeout:* (optional **number**)
  - *zone:* (optional string) (**Added** in v2.4.0-14)
  - *labels:* (optional {[string]=string,...})
  - *vshard_group:* (optional string)
  - *replicaset_alias:* (optional string)
  - *replicaset_weight:* (optional **number**)

Returns:

(**boolean**) true

Or

(**nil**)

(table) Error description

admin_expel_server (uuid)

Expel an instance (*deprecated*). Forever.

(**Deprecated** since v1.0.0-17 in favor of *cartridge.admin_edit_topology*)

Parameters:

- *uuid:* (string)

Returns:

(**boolean**) true

Or

(**nil**)

(table) Error description

Module *cartridge.auth*

Administrators authentication and authorization.

Local Functions

set_enabled (enabled)

Allow or deny unauthenticated access to the administrator's page. (*Changed* in v0.11)

This function affects only the current instance. It can't be used after the cluster was bootstrapped. To modify clusterwide config use `set_params` instead.

Parameters:

- *enabled:* (**boolean**)

Returns:

(**boolean**) *true*

Or

(**nil**)

(table) Error description

get_enabled ()

Check if unauthenticated access is forbidden. (*Added* in v0.7)

Returns:

(**boolean**) enabled

init ()

Initialize the authentication HTTP API.

Set up `login` and `logout` HTTP endpoints.

set_callbacks (callbacks)

Set authentication callbacks.

Parameters:

- *callbacks:*
  - *add_user:* (**function**)
  - *get_user:* (**function**)
  - *edit_user:* (**function**)
  - *list_users:* (**function**)
  - *remove_user:* (**function**)
  - *check_password:* (**function**)

Returns:

(**boolean**) *true*

get_callbacks ()

Get authentication callbacks.

Returns:

(table) callbacks

Configuration

set_params (opts)

Modify authentication params. (*Changed* in v0.11)

Can't be used before the bootstrap. Affects all cluster instances. Triggers `cluster.config_patch_clusterwide` .

Parameters:

- *opts:*

- *enabled:* (optional **boolean**) (*Added* in v0.11)
- *cookie_max_age:* (optional **number**)
- *cookie_renew_age:* (optional **number**) (*Added* in v0.11)

Returns:

(**boolean**) *true*

Or

(**nil**)

(table) Error description

get_params ()

Retrieve authentication params.

Returns:

(*AuthParams*)

AuthParams

Authentication params.

Fields:

- *enabled:* (**boolean**) Wether unauthenticated access is forbidden
- *cookie_max_age:* (**number**) Number of seconds until the authentication cookie expires
- *cookie_renew_age:* (**number**) Update provided cookie if it's older then this age (in seconds)

Authorizarion

set_lsid_cookie (user)

Create session for current user.

Creates session for user with specified username and user version or clear it if no arguments passed.

(**Added** in v2.2.0-43)

Parameters:

- *user:* (table)

get_session_username ()

Get username for the current HTTP session.

(**Added** in v1.1.0-4)

Returns:

(string or **nil**)

authorize_request (request)

Authorize an HTTP request.

Get username from cookies or basic HTTP authentication.

(**Added** in v1.1.0-4)

Parameters:

- *request:* (table)

Returns:

(**boolean**) Access granted

render_response (response)

Render HTTP response.

Inject set-cookie headers into response in order to renew or reset the cookie.

(**Added** in v1.1.0-4)

Parameters:

- *response:* (table)

Returns:

(table) The same response with cookies injected

User management

UserInfo

User information.

Fields:

- *username:* (string)
- *fullname:* (optional string)
- *email:* (optional string)
- *version:* (optional **number**)

add_user (username, password, fullname, email)

Trigger registered add_user callback.

The callback is triggered with the same arguments and must return a table with fields conforming to `UserInfo`. Unknown fields are ignored.

Parameters:

- *username:* (string)
- *password:* (string)

- *fullname:* (optional string)
- *email:* (optional string)

Returns:

(*UserInfo*)

Or

(**nil**)

(table) Error description

get_user (username)

Trigger registered get_user callback.

The callback is triggered with the same arguments and must return a table with fields conforming to `UserInfo`. Unknown fields are ignored.

Parameters:

- *username:* (string)

Returns:

(*UserInfo*)

Or

(**nil**)

(table) Error description

edit_user (username, password, fullname, email)

Trigger registered edit_user callback.

The callback is triggered with the same arguments and must return a table with fields conforming to `UserInfo`. Unknown fields are ignored.

Parameters:

- *username:* (string)
- *password:* (optional string)
- *fullname:* (optional string)
- *email:* (optional string)

Returns:

(*UserInfo*)

Or

(**nil**)

(table) Error description

list_users ()

Trigger registered list_users callback.

The callback is triggered without any arguments. It must return an array of `UserInfo` objects.

Returns:

(*{UserInfo,...}*)

Or

(**nil**)

(table) Error description

remove_user (username)

Trigger registered remove_user callback.

The callback is triggered with the same arguments and must return a table with fields conforming to `UserInfo` , which was removed. Unknown fields are ignored.

Parameters:

- *username:* (string)

Returns:

(*UserInfo*)

Or

(**nil**)

(table) Error description

Module *cartridge.roles*

Role management (internal module).

The module consolidates all the role management functions: `cfg` , some getters, `validate_config` and `apply_config` .

The module is almost stateless, it's only state is a collection of registered roles.

(**Added** in v1.2.0-20)

Functions

reload ()

Perform hot-reload of cartridge roles code.

This is an experimental feature, it's only allowed if the application enables it explicitly: `cartridge.cfg({roles_reload_allowed = true})` .

Reloading starts by stopping all roles and restoring the initial state. It's supposed that a role cleans up the global state when stopped, but even if it doesn't, cartridge kills all fibers and removes global variables and HTTP routes.

All Lua modules that were loaded during *cartridge.cfg* are unloaded, including supplementary modules required by a role. Modules, loaded before *cartridge.cfg* aren't affected.

Instance performs roles reload in a dedicated state `ReloadingRoles` . If reload fails, the instance enters the `ReloadError` state, which can later be retried. Otherwise, if reload succeeds, instance proceeds to the `ConfiguringRoles` state and initializes them as usual with `validate_config()` , `init()` , and `apply_config()` callbacks.

Returns:

(**boolean**) true

Or

(**nil**)

(table) Error description

Local Functions

cfg (module_names)

Load modules and register them as Cartridge Roles.

This function is internal, it's called as a part of *cartridge.cfg*.

Parameters:

- *module_names:* ({string,...})

Returns:

(**boolean**) true

Or

(**nil**)

(table) Error description

get_all_roles ()

List all registered roles.

Hidden and permanent roles are listed too.

Returns:

({string,..})

get_known_roles ()

List registered roles names.

Hidden roles are not listed as well as permanent ones.

Returns:

({string,..})

get_enabled_roles (roles)

Roles to be enabled on the server. This function returns all roles that will be enabled including their dependencies (bot hidden and not) and permanent roles.

Parameters:

- *roles:* ({string,...} or {[string]=boolean,...})

Returns:

({[string]=boolean,...})

get_role_dependencies (role_name)

List role dependencies. Including sub-dependencies.

Parameters:

- *role_name:* (string)

Returns:

({string,..})

validate_config (conf_new, conf_old)

Validate configuration by all roles.

Parameters:

- *conf_new:* (table)
- *conf_old:* (table)

Returns:

(**boolean**) true

Or

(**nil**)

(table) Error description

apply_config (conf, opts, is_master)

Apply the role configuration.

Parameters:

- *conf:* (table)
- *opts:* (table)
- *is_master:* (**boolean**)

Returns:

(**boolean**) true

Or

(**nil**)

(table) Error description

Module *cartridge.issues*

Monitor issues across cluster instances.

Cartridge detects the following problems:

Replication:

- «Replication from ... to ... isn't running» - when `box.info.replication.upstream == nil` ;
- «Replication from ... to ... is stopped/orphan/etc. (...)»;
- «Replication from ... to ...: high lag» - when `upstream.lag > box.cfg.replication_sync_lag` ;
- «Replication from ... to ...: long idle» - when `upstream.idle > 2 * box.cfg.replication_timeout` ;

Failover:

- «Can't obtain failover coordinator (...)»;
- «There is no active failover coordinator»;
- «Failover is stuck on ...: Error fetching appointments (...)»;
- «Failover is stuck on ...: Failover fiber is dead» - this is likely a bug;

Tables

limits

Thresholds for issuing warnings. All settings are local, not clusterwide. They can be changed with corresponding environment variables (`TARANTOOL_*`) or command-line arguments. See *cartridge.argparse* module for details.

Fields:

- *fragmentation_threshold_critical:* (**number**) *default*: 0.9.
- *fragmentation_threshold_warning:* (**number**) *default*: 0.6.
- *clock_delta_threshold_warning:* (**number**) *default*: 5.

Module *cartridge.argparse*

Gather configuration options.

The module tries to read configuration options from multiple sources and then merge them together according to the priority of the source:

- *–<VARNAME>* command line arguments
- *TARANTOOL_<VARNAME>* environment variables
- configuration files

You can specify a configuration file using the *–cfg <CONFIG_FILE>* option or the *TARANTOOL_CFG=<CONFIG_FILE>* environment variable.

Configuration files are *yaml* files, divided into sections like the following:

```
default:
  memtx_memory: 10000000
  some_option: "default value"
myapp.router:
  memtx_memory: 1024000000
  some_option: "router specific value"
```

Within the configuration file, *argparse* looks for multiple matching sections:

- The section named *<APP_NAME>.<INSTANCE_NAME>* is parsed first. Application name is derived automatically from the rockspec filename in the project directory. Or it can be can be specified manually with the `--app-name` command line argument or the `TARANTOOL_APP_NAME` environment variable. Instance name can be specified the same way, either as *–instance-name* or `TARANTOOL_INSTANCE_NAME` .
- The common *<APP_NAME>* section is parsed next.
- Finally, the section `[default]` with global configuration is parsed with the lowest priority.

Functions

parse ()

Parse command line arguments, environment variables, and configuration files.

Returns:

({argname=value,...})

get_opts (filter)

Filter the results of parsing and cast variables to a given type.

From all configuration options gathered by `parse` , select only those specified in the filter.

For example, running an application as following:

```
./init.lua --alias router --memtx-memory 100
```

results in:

```
parse()            -> {memtx_memory = "100", alias = "router"}
get_cluster_opts() -> {alias = "router"} -- a string
get_box_opts()     -> {memtx_memory = 100} -- a number
```

Parameters:

- *filter:* **({argname=type,...})**

Returns:

({argname=value,...})

get_box_opts ()

Shorthand for `get_opts(box_opts)` .

get_cluster_opts ()

Shorthand for `get_opts(cluster_opts)` .

Tables

cluster_opts

Common *cartridge.cfg* options.

Options which are not listed below (like `roles`) can't be modified with `argparse` and should be configured in code.

Fields:

- *alias:* **string**
- *workdir:* **string**
- *http_port:* **number**
- *http_enabled:* **boolean**
- *advertise_uri:* **string**
- *cluster_cookie:* **string**
- *console_sock:* **string**
- *auth_enabled:* **boolean**
- *bucket_count:* **number**
- *upgrade_schema:* **boolean**
- *swim_broadcast:* **boolean**

box_opts

Common [box.cfg](https://www.tarantool.io/en/doc/latest/reference/configuration/) tuning options.

Fields:

- *listen:* **string**
- *memtx_memory:* **number**
- *strip_core:* **boolean**
- *memtx_min_tuple_size:* **number**
- *memtx_max_tuple_size:* **number**
- *memtx_use_mvcc_engine:* **boolean**
- *slab_alloc_factor:* **number**

- *work_dir:* **string** (**deprecated**)
- *memtx_dir:* **string**
- *wal_dir:* **string**
- *vinyl_dir:* **string**
- *vinyl_memory:* **number**
- *vinyl_cache:* **number**
- *vinyl_max_tuple_size:* **number**
- *vinyl_read_threads:* **number**
- *vinyl_write_threads:* **number**
- *vinyl_timeout:* **number**
- *vinyl_run_count_per_level:* **number**
- *vinyl_run_size_ratio:* **number**
- *vinyl_range_size:* **number**
- *vinyl_page_size:* **number**
- *vinyl_bloom_fpr:* **number**
- *log:* **string**
- *log_nonblock:* **boolean**
- *log_level:* **number**
- *log_format:* **string**
- *io_collect_interval:* **number**
- *readahead:* **number**
- *snap_io_rate_limit:* **number**
- *too_long_threshold:* **number**
- *wal_mode:* **string**
- *rows_per_wal:* **number**
- *wal_max_size:* **number**
- *wal_dir_rescan_delay:* **number**
- *force_recovery:* **boolean**
- *replication:* **string**
- *instance_uuid:* **string**
- *replicaset_uuid:* **string**
- *custom_proc_title:* **string**
- *pid_file:* **string**
- *background:* **boolean**
- *username:* **string**
- *coredump:* **boolean**

- *checkpoint_interval:* **number**
- *checkpoint_wal_threshold:* **number**
- *checkpoint_count:* **number**
- *read_only:* **boolean**
- *hot_standby:* **boolean**
- *worker_pool_threads:* **number**
- *replication_timeout:* **number**
- *replication_sync_lag:* **number**
- *replication_sync_timeout:* **number**
- *replication_connect_timeout:* **number**
- *replication_connect_quorum:* **number**
- *replication_skip_conflict:* **boolean**
- *replication_synchro_quorum:* **number**
- *replication_synchro_timeout:* **number**
- *feedback_enabled:* **boolean**
- *feedback_host:* **string**
- *feedback_interval:* **number**
- *net_msg_max:* **number**

Module *cartridge.twophase*

Clusterwide configuration propagation two-phase algorithm.

(**Added** in v1.2.0-19)

Functions

patch_clusterwide (patch)

Edit the clusterwide configuration. Top-level keys are merged with the current configuration. To remove a top-level section, use `patch_clusterwide{key = box.NULL}` .

The function uses a two-phase commit algorithm with the following steps:

I. Patches the current configuration.

II. Validates topology on the current server.

III. Executes the preparation phase (`prepare_2pc`) on every server excluding expelled and disabled servers.

IV. If any server reports an error, executes the abort phase (`abort_2pc`). All servers prepared so far are rolled back and unlocked.

V. Performs the commit phase (`commit_2pc`). In case the phase fails, an automatic rollback is impossible, the cluster should be repaired manually.

Parameters:

- *patch:* (table)

Returns:

(**boolean**) true

Or

(**nil**)

(table) Error description

force_reapply (uuids)

Forcefully apply config to the given instances.

In particular:

- Abort two-phase commit (remove `config.prepare` lock)
- Upload the active config from the current instance.
- Apply it (reconfigure all roles)

(**Added** in v2.3.0-68)

Parameters:

- *uuids:* ({string,... })

Returns:

(**boolean**) true

Or

(**nil**)

(table) Error description

get_schema ()

Get clusterwide DDL schema.

(**Added** in v1.2.0-28)

Returns:

(string) Schema in YAML format

Or

(**nil**)

(table) Error description

set_schema (schema)

Apply clusterwide DDL schema.

(**Added** in v1.2.0-28)

Parameters:

- *schema:* (string) in YAML format

Returns:

(string) The same new schema

Or

(**nil**)

(table) Error description

on_patch (trigger_new, trigger_old)

Set up trigger for for patch_clusterwide.

It will be executed **before** new new config applied.

If the parameters are `(nil, old_trigger)` , then the old trigger is deleted.

The trigger function is called with two argument: `- conf_new` (`ClusterwideConfig`) `- conf_old` (`ClusterWideConfig`)

It is allowed to modify `conf_new` , but not `conf_old` . Return values are ignored. If calling a trigger raises an error, `patch_clusterwide` returns it as `nil, err` .

(**Added** in v2.1.0-4)

Parameters:

- *trigger_new:* (**function**)
- *trigger_old:* (**function**)

Usage:

```
local function inject_data(conf_new, _)
    local data_yml = yaml.encode({foo = 'bar'})
    conf_new:set_plaintext('data.yml', data_yml)
end)

twophase.on_patch(inject_data) -- set custom patch modifier trigger
twophase.on_patch(nil, inject_data) -- drop trigger
```

Local Functions

prepare_2pc (data)

Two-phase commit - preparation stage.

Validate the configuration and acquire a lock setting local variable and writing «config.prepare.yml» file. If the validation fails, the lock isn't acquired and doesn't have to be aborted.

Parameters:

- *data:* (table) clusterwide config content

Returns:

(**boolean**) true

Or

(**nil**)

(table) Error description

commit_2pc ()

Two-phase commit - commit stage.

Back up the active configuration, commit changes to filesystem by renaming prepared file, release the lock, and configure roles. If any errors occur, configuration is not rolled back automatically. Any problem encountered during this call has to be solved manually.

Returns:

(**boolean**) true

Or

(**nil**)

(table) Error description

abort_2pc ()

Two-phase commit - abort stage.

Release the lock for further commit attempts.

Returns:

(**boolean**) true

Module *cartridge.failover*

Gather information regarding instances leadership.

Failover can operate in two modes:

- In *disabled* mode the leader is the first server configured in `topology.replicasets[].master` array.
- In `eventual` mode the leader isn't elected consistently. Instead, every instance in cluster thinks the leader is the first **healthy** server in replicaset, while instance health is determined according to membership status (the SWIM protocol).
- In *stateful* mode leaders appointments are polled from the external storage. (**Added** in v2.0.2-2)

This module behavior depends on the instance state.

From the very beginning it reports *is_rw() == false*, `is_leader() == false` , `get_active_leaders() == {}` .

The module is configured when the instance enters *ConfiguringRoles* state for the first time. From that moment it reports actual values according to the mode set in clusterwide config.

(**Added** in v1.2.0-17)

Functions

get_coordinator ()

Get current stateful failover coordinator

Returns:

(table) coordinator

Or

(**nil**)

(table) Error description

Local Functions

schedule_clear ()

Cancel all pending reconfigure_all tasks.

schedule_add ()

Schedule new reconfigure_all task.

_get_appointments_disabled_mode ()

Generate appointments according to clusterwide configuration. Used in „disabled“ failover mode.

_get_appointments_eventual_mode ()

Generate appointments according to membership status. Used in „eventual“ failover mode.

_get_appointments_stateful_mode ()

Get appointments from external storage. Used in „stateful“ failover mode.

accept_appointments (replicaset_uuid)

Accept new appointments.

Get appointments wherever they come from and put them into cache. Cached active_leaders table is never modified, but overriden by it's modified copy (if necessary).

Parameters:

- *replicaset_uuid:* ({[string]=string}) to leader_uuid map

Returns:

(**boolean**) Whether leadership map has changed

fencing_check ()

Perform the fencing healthcheck.

Fencing is actuated when the instance disconnects from both the state provider and a replica, i.e. the check returns false.

Returns:

(**boolean**) true / false

failover_loop ()

Repeatedly fetch new appointments and reconfigure roles.

cfg ()

Initialize the failover module.

get_active_leaders ()

Get map of replicaset leaders.

Returns:

{[replicaset_uuid] = instance_uuid,...}

is_leader ()

Check current instance leadership.

Returns:

(**boolean**) true / false

is_rw ()

Check current instance writability.

Returns:

(**boolean**) true / false

is_vclockkeeper ()

Check if current instance has persisted his vclock.

Returns:

(**boolean**) true / false

consistency_needed ()

Check if current configuration implies consistent switchover.

Returns:

(**boolean**) true / false

force_inconsistency (replicaset_uuid)

Force inconsistent leader switching. Do it by resetting vclockkepers in state provider.

Parameters:

- *replicaset_uuid:* ({[string]=string,...}) to leader_uuid mapping

Returns:

(**boolean**) true

Or

(**nil**)

(table) Error description

wait_consistency (replicaset_uuid)

Wait when promoted instances become vclockkepers.

Parameters:

- *replicaset_uuid:* ({[string]=string,...}) to leader_uuid mapping

Returns:

(**boolean**) true

Or

(**nil**)

(table) Error description

Module *cartridge.topology*

Topology validation and filtering.

Functions

cluster_is_healthy ()

Check the cluster health. It is healthy if all instances are healthy.

The function is designed mostly for testing purposes.

Returns:

(**boolean**) true / false

Local Functions

get_leaders_orded (topology_cfg, replicaset_uuid, new_order)

Get full list of replicaset leaders.

Full list is composed of:

- New order array
- Initial order from topology_cfg (with no repetitions)
- All other servers in the replicaset, sorted by uuid, ascending

Neither `topology_cfg` nor `new_order` tables are modified. New order validity is ignored too.

Parameters:

- *topology_cfg:* (table)
- *replicaset_uuid:* (string)
- *new_order:* (optional table)

Returns:

({string,...}) array of leaders uuids

validate (topology_new, topology_old)

Validate topology configuration.

Parameters:

- *topology_new:* (table)
- *topology_old:* (table)

Returns:

(**boolean**) true

Or

(**nil**)

(table) Error description

find_server_by_uri (topology_cfg, uri)

Find the server in topology config.

(**Added** in v1.2.0-17)

Parameters:

- *topology_cfg:* (table)
- *uri:* (string)

Returns:

(**nil** or string) *instance_uuid* found

refine_servers_uri (topology_cfg)

Merge servers URIs form topology_cfg with fresh membership status.

This function sustains cartridge operability in case of advertise_uri change. The uri map is composed basing on topology_cfg, but if some of them turns out to be dead, the member with corresponding payload.uuid is searched beyond.

(**Added** in v2.3.0-7)

Parameters:

- *topology_cfg:* (table)

Returns:

({**[uuid]**) = uri} with all servers except expelled ones.

probe_missing_members (servers)

Send UDP ping to servers missing from membership table.

Parameters:

- *servers:* (table)

Returns:

(**boolean**) true

Or

(**nil**)

(table) Error description

get_fullmesh_replication (topology_cfg, replicaset_uuid)

Get replication config to set up full mesh.

(**Added** in v1.2.0-17)

Parameters:

- *topology_cfg:* (table)
- *replicaset_uuid:* (string)

Returns:

(table)

Module *cartridge.clusterwide-config*

The abstraction, representing clusterwide configuration.

Clusterwide configuration is more than just a lua table. It's an object in terms of OOP paradigm.

On filesystem clusterwide config is represented by a file tree.

In Lua it's represented as an object which holds both plaintext files content and unmarshalled lua tables. Unmarshalling is implicit and performed automatically for the sections with `.yml` file extension.

To access plaintext content there are two functions: *get_plaintext* and `set_plaintext` .

Unmarshalled lua tables are accessed without `.yml` extension by `get_readonly` and `get_deepcopy` . Plaintext serves for accessing unmarshalled representation of corresponding sections.

To avoid ambiguity it's prohibited to keep both *<FILENAME>* and *<FILENAME>.yml* in the configuration. An attempt to do so would result in *return nil, err* from *new()* and *load()*, and an attempt to call `get_readonly/deepcopy` would raise an error. Nevertheless one can keep any other extensions because they aren't unmarshalled implicitly.

(**Added** in v1.2.0-17)

Usage:

```
tarantool> cfg = ClusterwideConfig.new({
         >     -- two files
         >     ['forex.yml'] = '{EURRUB_TOM: 70.33, USDRUB_TOM: 63.18}',
         >     ['text'] = 'Lorem ipsum dolor sit amet',
         > })
---
...

tarantool> cfg:get_plaintext()
---
- text: Lorem ipsum dolor sit amet
  forex.yml: '{EURRUB_TOM: 70.33, USDRUB_TOM: 63.18}'
...

tarantool> cfg:get_readonly()
---
- forex.yml: '{EURRUB_TOM: 70.33, USDRUB_TOM: 63.18}'
  forex:
    EURRUB_TOM: 70.33
    USDRUB_TOM: 63.18
  text: Lorem ipsum dolor sit amet
...
```

Functions

new ([data])

Create new object.

Parameters:

- *data:* ({string=string,...}) Plaintext content (optional)

Returns:

(**ClusterwideConfig**)

Or

(**nil**)

(table) Error description

save (clusterwide_config, filename)

Write configuration to filesystem.

Write atomicity is achieved by splitting it into two phases: 1. Configuration is saved with a random filename in the same directory 2. Temporal filename is renamed to the destination

Parameters:

- *clusterwide_config:* (**ClusterwideConfig**)
- *filename:* (string)

Returns:

(**boolean**) true

Or

(**nil**)

(table) Error description

load (filename)

Load object from filesystem.

This function handles both old-style single YAML and new-style directory with a file tree.

Parameters:

- *filename:* (string)

Returns:

(**ClusterwideConfig**)

Or

(**nil**)

(table) Error description

Local Functions

load_from_file (filename)

Load old-style config from YAML file.

Parameters:

- *filename:* (string) Filename to load.

Returns:

(**ClusterwideConfig**)

Or

(**nil**)

(table) Error description

load_from_dir (path)

Load new-style config from a directory.

Parameters:

- *path:* (string) Path to the config.

Returns:

(**ClusterwideConfig**)

Or

(**nil**)

(table) Error description

remove (string)

Remove config from filesystem atomically.

The atomicity is achieved by splitting it into two phases: 1. Configuration is saved with a random filename in the same directory 2. Temporal filename is renamed to the destination

Parameters:

- *string:* (**path**) Directory path to remove.

Returns:

(**boolean**) true

Or

(**nil**)

(table) Error description

Module *cartridge.rpc*

Remote procedure calls between cluster instances.

Functions

get_candidates (role_name[, opts])

List instances suitable for performing a remote call.

Parameters:

- *role_name:* (string)
- *opts:*
  - *leader_only:* (optional **boolean**) Filter instances which are leaders now.(default: **false**)
  - *healthy_only:* (optional **boolean**) The member is considered healthy ifit reports either `ConfiguringRoles` or `RolesConfigured` stateand its SWIM status is either `alive` or `suspect` (added in v1.1.0-11, default: **true**)

Returns:

({string,...}) URIs

call (role_name, fn_name[, args[, opts]])

Perform a remote procedure call. Find a suitable healthy instance with an enabled role and perform a [`net.box conn:call`](https://tarantool.io/en/doc/latest/reference/reference_lua/net_box/#net-box-call) on it. `rpc.call()` can only be used for functions defined in role return table unlike `net.box conn:call()` , which is used for global functions as well.

Parameters:

- *role_name:* (string)
- *fn_name:* (string)
- *args:* (table) (optional)
- *opts:*
  - *prefer_local:* (optional **boolean**) Don't perform a remote call if possible. When the role is enabledlocally and current instance is healthy the remote netbox call issubstituted with a local Lua function call. When the option isdisabled it never tries to perform call locally and always usesnetbox connection, even to connect self.(default: **true**)
  - *leader_only:* (optional **boolean**) Perform a call only on the replica set leaders.(default: **false**)
  - *uri:* (optional string) Force a call to be performed on this particular uri.Disregards member status and `opts.prefer_local` .Conflicts with `opts.leader_only = true` .(added in v1.2.0-63)
  - *remote_only:* (*deprecated*) Use `prefer_local` instead.
  - *timeout:* passed to `net.box conn:call` options.
  - *buffer:* passed to `net.box conn:call` options.
  - *on_push:* passed to `net.box conn:call` options.
  - *on_push_ctx:* passed to `net.box conn:call` options.

Returns:

conn:call() result

Or

(**nil**)

(table) Error description

Usage:

```
-- myrole.lua
return {
    role_name = 'myrole',
    add = function(a, b) return a + b end,
}
```

```
-- call it as follows:
cartridge.rpc_call('myrole', 'add', {2, 2}) -- returns 4
```

Local Functions

get_connection (role_name[, opts])

Connect to an instance with an enabled role.

Parameters:

- *role_name:* (string)
- *opts:*
  - *prefer_local:* (optional **boolean**)
  - *leader_only:* (optional **boolean**)

Returns:

`net.box` connection

Or

(**nil**)

(table) Error description

Module *cartridge.tar*

Handle basic tar format.

<http://www.gnu.org/software/tar/manual/html_node/Standard.html>

While an archive may contain many files, the archive itself is a single ordinary file. Physically, an archive consists of a series of file entries terminated by an end-of-archive entry, which consists of two 512 blocks of zero bytes. A file entry usually describes one of the files in the archive (an archive member), and consists of a file header and the contents of the file. File headers contain file names and statistics, checksum information which tar uses to detect file corruption, and information about file types.

A tar archive file contains a series of blocks. Each block contains exactly 512 (*BLOCKSIZE*) bytes:

```
+---------+-------+-------+-------+---------+-------+-----
| header1 | file1 |  ...  |  ...  | header2 | file2 | ...
+---------+-------+-------+-------+---------+-------+-----
```

All characters in header blocks are represented by using 8-bit characters in the local variant of ASCII. Each field within the structure is contiguous; that is, there is no padding used within the structure. Each character on the archive medium is stored contiguously. Bytes representing the contents of files (after the header block of each file) are not translated in any way and are not constrained to represent characters in any character set. The tar format does not distinguish text files from binary files, and no translation of file contents is performed.

Functions

pack (files)

Create TAR archive.

Parameters:

- *files:* ({string=string})

Returns:

(string) The archive

Or

(**nil**)

(table) Error description

unpack (tar)

Parse TAR archive.

Only regular files are extracted, directories are ommitted.

Parameters:

- *tar:* (string)

Returns:

({string=string}) Extracted files (their names and content)

Or

(**nil**)

(table) Error description

Module *cartridge.pool*

Connection pool.

Reuse tarantool net.box connections with ease.

Functions

connect (uri[, opts])

Connect a remote or get cached connection. Connection is established using `net.box.connect()` .

Parameters:

- *uri:* (string)
- *opts:*
  - *wait_connected:* (**boolean** or **number**) by default, connection creation is blocked until theconnection is established, but passing `wait_connected=false` makes it return immediately. Also, passing a timeout makes itwait before returning (e.g. `wait_connected=1.5` makes it waitat most 1.5 seconds).
  - *connect_timeout:* (optional **number**) (*deprecated*)Use *wait_connected* instead
  - *user:* (*deprecated*) don't use it
  - *password:* (*deprecated*) don't use it
  - *reconnect_after:* (*deprecated*) don't use it

Returns:

`net.box` connection

Or

(**nil**)

(table) Error description

Local Functions

format_uri (uri)

Enrich URI with credentials. Suitable to connect other cluster instances.

Parameters:

- *uri:* (string) *host:port*

Returns:

(string) *username:password@host:port*

map_call (fn_name[, args[, opts]])

Perform a remote call to multiple URIs and map results.

(**Added** in v1.2.0-17)

Parameters:

- *fn_name:* (string)
- *args:* (table) function arguments (optional)
- *opts:*
  - *uri_list:* ({string,...}) array of URIs for performing remote call
  - *timeout:* (optional **number**) passed to `net.box conn:call()`

Returns:

(**{URI=value,...}**) Call results mapping for every URI.

(table) United error object, gathering errors for every URI that failed.

Module *cartridge.confapplier*

Configuration management primitives.

Implements the internal state machine which helps to manage cluster operation and protects from invalid state transitions.

Functions

get_active_config ()

Get current ClusterwideConfig object of instance

Returns:

cartridge.clusterwide-config or nil, if instance not bootstrapped.

get_readonly ([section_name])

Get a read-only view on the clusterwide configuration.

Returns either `conf[section_name]` or entire `conf` . Any attempt to modify the section or its children will raise an error.

Parameters:

- *section_name:* (string) (optional)

Returns:

(table)

get_deepcopy ([section_name])

Get a read-write deep copy of the clusterwide configuration.

Returns either `conf[section_name]` or entire `conf` . Changing it has no effect unless it's used to patch clusterwide configuration.

Parameters:

- *section_name:* (string) (optional)

Returns:

(table)

Local Functions

set_state (state[, err])

Perform state transition.

Parameters:

- *state:* (string) New state
- *err:* (optional)

Returns:

(**nil**)

wish_state (state[, timeout])

Make a wish for meeting desired state.

Parameters:

- *state:* (string) Desired state.
- *timeout:* (**number**) (optional)

Returns:

(string) Final state, may differ from desired.

validate_config (clusterwide_config_new)

Validate configuration by all roles.

Parameters:

- *clusterwide_config_new:* (table)

Returns:

(**boolean**) true

Or

(**nil**)

(table) Error description

apply_config (clusterwide_config)

Apply the role configuration.

Parameters:

- *clusterwide_config:* (table)

Returns:

(**boolean**) true

Or

(**nil**)

(table) Error description

Module *cartridge.test-helpers*

Helpers for integration testing.

This module extends `luatest.helpers` with cartridge-specific classes and helpers.

Fields

Server

Extended luatest.server class to run tarantool instance.

See also:

- cartridge.test-helpers.server

Cluster

Class to run and manage multiple tarantool instances.

See also:

- cartridge.test-helpers.cluster

Etcd

Class to run and manage etcd node.

See also:

- cartridge.test-helpers.etcd

Module *cartridge.remote-control*

Tarantool remote control server.

Allows to control an instance over TCP by `net.box` `call` and `eval` . The server is designed as a partial replacement for the **iproto** protocol. It's most useful when `box.cfg` wasn't configured yet.

Other `net.box` features aren't supported and will never be.

(**Added** in v0.10.0-2)

Local Functions

bind (host, port)

Init remote control server.

Bind the port but don't start serving connections yet.

Parameters:

- *host:* (string)
- *port:* (string or **number**)

Returns:

(**boolean**) true

Or

(**nil**)

(table) Error description

accept (credentials)

Start remote control server. To connect the server use regular `net.box` connection.

Access is restricted to the user with specified credentials, which can be passed as `net_box.connect('username:password@host:port')` .

Parameters:

- *credentials:*
  - *username:* (string)
  - *password:* (string)

unbind ()

Stop the server.

It doesn't interrupt any existing connections.

drop_connections ()

Explicitly drop all established connections.

Module *cartridge.service-registry*

Inter-role interaction.

These functions make different roles interact with each other.

The registry stores initialized modules and accesses them within the one and only current instance. For cross-instance access, use the *cartridge.rpc* module.

Functions

set (module_name, instance)

Put a module into registry or drop it. This function typically doesn't need to be called explicitly, the cluster automatically sets all the initialized roles.

Parameters:

- *module_name:* (string)
- *instance:* (**nil** or table)

Returns:

(**nil**)

get (module_name)

Get a module from registry.

Parameters:

- *module_name:* (string)

Returns:

(**nil**)

Or

(table) instance

Module *custom-role*

User-defined role API.

If you want to implement your own role it must conform this API.

Functions

init (opts)

Role initialization callback. Called when role is enabled on an instance. Caused either by editing topology or instance restart.

Parameters:

- *opts:*
  - *is_master:* (**boolean**)

stop (opts)

Role shutdown callback. Called when role is disabled on an instance.

Parameters:

- *opts:*
  - *is_master:* (**boolean**)

validate_config (conf_new, conf_old)

Validate clusterwide configuration callback.

Parameters:

- *conf_new:* (table)
- *conf_old:* (table)

apply_config (conf, opts)

Apply clusterwide configuration callback.

Parameters:

- *conf:* (table) Clusterwide configuration
- *opts:*
  - *is_master:* (**boolean**)

Fields

role_name

Displayed role name. When absent, module name is used instead.

- *role_name:* (string)

hidden

Hidden role flag. aren't listed in `cartridge.admin_get_replicasets().roles` and therefore in WebUI. Hidden roled are supposed to be a dependency for another role.

- *hidden:* (**boolean**)

permanent

Permanent role flag. Permanent roles will be enabled on every instance in cluster. Implies `hidden = true` .

- *permanent:* (**boolean**)

Module *cartridge.lua-api.stat*

Administration functions (`box.slab.info` related).

Local Functions

get_stat (uri)

Retrieve `box.slab.info` of a remote server.

Parameters:

- *uri:* (string)

Returns:

(table)

Or

(**nil**)

(table) Error description

Module *cartridge.lua-api.boxinfo*

Administration functions (`box.info` related).

Local Functions

get_info (uri)

Retrieve `box.cfg` and `box.info` of a remote server.

Parameters:

- *uri:* (string)

Returns:

(table)

Or

(**nil**)

(table) Error description

Module *cartridge.lua-api.get-topology*

Administration functions (`get-topology` implementation).

Tables

ReplicasetInfo

Replicaset general information.

Fields:

- *uuid:* (string) The replicaset UUID.
- *roles:* ({string,...}) Roles enabled on the replicaset.
- *status:* (string) Replicaset health.
- *master:* (*ServerInfo*) Replicaset leader according to configuration.
- *active_master:* (*ServerInfo*) Active leader.
- *weight:* (**number**) Vshard replicaset weight.Matters only if vshard-storage role is enabled.
- *vshard_group:* (string) Name of vshard group the replicaset belongs to.
- *all_rw:* (**boolean**) A flag indicating that all servers in the replicaset should be read-write.
- *alias:* (string) Human-readable replicaset name.
- *servers:* (*{ServerInfo,...}*) Circular reference to all instances in the replicaset.

ServerInfo

Instance general information.

Fields:

- *alias:* (string) Human-readable instance name.
- *uri:* (string)
- *uuid:* (string)
- *disabled:* (**boolean**)
- *status:* (string) Instance health.
- *message:* (string) Auxilary health status.
- *replicaset:* (*ReplicasetInfo*) Circular reference to a replicaset.
- *priority:* (**number**) Leadership priority for automatic failover.
- *clock_delta:* (**number**) Difference between remote clock and the current one (inseconds), obtained from the membership module (SWIM protocol).Positive values mean remote clock are ahead of local, and viceversa.
- *zone:* (string)

Local Functions

get_topology ()

Get servers and replicasets lists.

Returns:

(**{servers={ServerInfo,...},replicasets={ReplicasetInfo,...}}**)

Or

(**nil**)

(table) Error description

Module *cartridge.lua-api.edit-topology*

Administration functions (`edit-topology` implementation).

Editing topology

edit_topology (args)

Edit cluster topology. This function can be used for:

- bootstrapping cluster from scratch
- joining a server to an existing replicaset
- creating new replicaset with one or more servers

- editing uri/labels of servers
- disabling and expelling servers

(**Added** in v1.0.0-17)

Parameters:

- *args:*
  - *servers:* (optional *{EditServerParams,..}*)
  - *replicasets:* (optional *{EditReplicasetParams,..}*)

EditReplicasetParams

Replicatets modifications.

Fields:

- *uuid:* (optional string)
- *alias:* (optional string)
- *roles:* (optional {string,...})
- *all_rw:* (optional **boolean**)
- *weight:* (optional **number**)
- *failover_priority:* (optional {string,...}) array of uuids specifying servers failover priority
- *vshard_group:* (optional string)
- *join_servers:* (optional *{JoinServerParams,...}*)

JoinServerParams

Parameters required for joining a new server.

Fields:

- *uri:* (string)
- *uuid:* (optional string)
- *zone:* (optional string) (**Added** in v2.4.0-14)
- *labels:* (optional table)

EditServerParams

Servers modifications.

Fields:

- *uri:* (optional string)
- *uuid:* (string)
- *zone:* (optional string)
- *labels:* (optional table)

- *disabled:* (optional **boolean**)
- *expelled:* (optional **boolean**) Expelling an instance is permanent and can't be undone.It's suitable for situations when the hardware is destroyed,snapshots are lost and there is no hope to bring it back to life.

Module *cartridge.lua-api.topology*

Administration functions (topology related).

Functions

get_servers ([uuid])

Get servers list. Optionally filter out the server with the given uuid.

Parameters:

- *uuid:* (string) (optional)

Returns:

(**{ServerInfo,...}**)

Or

(**nil**)

(table) Error description

get_replicasets ([uuid])

Get replicasets list. Optionally filter out the replicaset with given uuid.

Parameters:

- *uuid:* (string) (optional)

Returns:

(**{ReplicasetInfo,...}**)

Or

(**nil**)

(table) Error description

probe_server (uri)

Discover an instance.

Parameters:

- *uri:* (string)

enable_servers (uuids)

Enable nodes after they were disabled.

Parameters:

- *uuids:* ({string,...})

Returns:

(**{ServerInfo,...}**)

Or

(**nil**)

(table) Error description

disable_servers (uuids)

Temporarily diable nodes.

Parameters:

- *uuids:* ({string,...})

Returns:

(**{ServerInfo,...}**)

Or

(**nil**)

(table) Error description

Local Functions

get_self ()

Get alias, uri and uuid of current instance.

Returns:

(table)

Module *cartridge.lua-api.failover*

Administration functions (failover related).

Functions

get_params ()

Get failover configuration.

(**Added** in v2.0.2-2)

Returns:

(*FailoverParams*)

set_params (opts)

Configure automatic failover.

(**Added** in v2.0.2-2)

Parameters:

- *opts:*
  - *mode:* (optional string)
  - *state_provider:* (optional string)
  - *failover_timeout:* (optional **number**) (added in v2.3.0-52)
  - *tarantool_params:* (optional table)
  - *etcd2_params:* (optional table) (added in v2.1.2-26)
  - *fencing_enabled:* (optional **boolean**) (added in v2.3.0-57)
  - *fencing_timeout:* (optional **number**) (added in v2.3.0-57)
  - *fencing_pause:* (optional **number**) (added in v2.3.0-57)

Returns:

(**boolean**) *true* if config applied successfully

Or

(**nil**)

(table) Error description

get_failover_enabled ()

Get current failover state.

(**Deprecated** since v2.0.2-2)

set_failover_enabled (enabled)

Enable or disable automatic failover.

(**Deprecated** since v2.0.2-2)

Parameters:

- *enabled:* (**boolean**)

Returns:

(**boolean**) New failover state

Or

(**nil**)

(table) Error description

promote (replicaset_uuid[, opts])

Promote leaders in replicasets.

Parameters:

- *replicaset_uuid:* (table)] = leader_uuid }
- *opts:*
  - *force_inconsistency:* (optional **boolean**) (default: **false**)

Returns:

(**boolean**) true On success

Or

(**nil**)

(table) Error description

Tables

FailoverParams

Failover parameters.

(**Added** in v2.0.2-2)

Fields:

- *mode:* (string) Supported modes are «disabled», «eventual» and «stateful»
- *state_provider:* (optional string) Supported state providers are «tarantool» and «etcd2».
- *failover_timeout:* (**number**) (added in v2.3.0-52)Timeout (in seconds), used by membership tomark `suspect` members as `dead` (default: 20)
- *tarantool_params:* (added in v2.0.2-2)
  - *uri:* (string)
  - *password:* (string)
- *etcd2_params:* (added in v2.1.2-26)
  - *prefix:* (string) Prefix used for etcd keys: <*prefix*>/*lock* and‘<prefix>/leaders‘
  - *lock_delay:* (optional **number**) Timeout (in seconds), determines lock's time-to-live (default: 10)
  - *endpoints:* (optional table) URIs that are used to discover and to access etcd cluster instances.(default: `{'http://localhost:2379', 'http://localhost:4001'}`)
  - *username:* (optional string) (default: «»)
  - *password:* (optional string) (default: «»)
- *fencing_enabled:* (**boolean**) (added in v2.3.0-57)Abandon leadership when both the state provider quorum and atleast one replica are lost (suitable in stateful mode only,default: false)

- *fencing_timeout:* (**number**) (added in v2.3.0-57)Time (in seconds) to actuate fencing after the check fails(default: 10)
- *fencing_pause:* (**number**) (added in v2.3.0-57)The period (in seconds) of performing the check(default: 2)

Module *cartridge.lua-api.vshard*

Administration functions (vshard related).

Functions

bootstrap_vshard ()

Call `vshard.router.bootstrap()` . This function distributes all buckets across the replica sets.

Returns:

(**boolean**) *true*

Or

(**nil**)

(table) Error description

Module *cartridge.lua-api.deprecated*

Administration functions (deprecated).

Deprecated functions

join_server (args)

Join an instance to the cluster (*deprecated*).

(**Deprecated** since v1.0.0-17 in favor of *cartridge.admin_edit_topology*)

Parameters:

- *args:*
  - *uri:* (string)
  - *instance_uuid:* (optional string)
  - *replicaset_uuid:* (optional string)
  - *roles:* (optional {string,...})
  - *timeout:* (optional **number**)
  - *zone:* (optional string) (**Added** in v2.4.0-14)
  - *labels:* (optional {[string]=string,...})
  - *vshard_group:* (optional string)
  - *replicaset_alias:* (optional string)

  - *replicaset_weight:* (optional **number**)

Returns:

(**boolean**) true

Or

(**nil**)

(table) Error description

edit_server (args)

Edit an instance (*deprecated*).

(**Deprecated** since v1.0.0-17 in favor of *cartridge.admin_edit_topology*)

Parameters:

- *args:*
  - *uuid:* (string)
  - *uri:* (optional string)
  - *labels:* (optional {[string]=string,...})

Returns:

(**boolean**) true

Or

(**nil**)

(table) Error description

expel_server (uuid)

Expel an instance (*deprecated*). Forever.

(**Deprecated** since v1.0.0-17 in favor of *cartridge.admin_edit_topology*)

Parameters:

- *uuid:* (string)

Returns:

(**boolean**) true

Or

(**nil**)

(table) Error description

edit_replicaset (args)

Edit replicaset parameters (*deprecated*).

(**Deprecated** since v1.0.0-17 in favor of *cartridge.admin_edit_topology*)

Parameters:

- *args:*
  - *uuid:* (string)
  - *alias:* (string)
  - *roles:* (optional {string,...})
  - *master:* (optional {string,...}) Failover order
  - *weight:* (optional **number**)
  - *vshard_group:* (optional string)
  - *all_rw:* (optional **boolean**)

Returns:

(**boolean**) true

Or

(**nil**)

(table) Error description

Class *cartridge.test-helpers.cluster*

Class to run and manage multiple tarantool instances.

Functions

Cluster:new (object)

Build cluster object.

Parameters:

- *object:*
  - *datadir:* (string) Data directory for all cluster servers.
  - *server_command:* (string) Command to run server.
  - *cookie:* (string) Cluster cookie.
  - *base_http_port:* (**int**) Value to calculate server's http_port. (optional)
  - *base_advertise_port:* (**int**) Value to calculate server's advertise_port. (optional)
  - *use_vshard:* (**bool**) bootstrap vshard after server is started. (optional)
  - *replicasets:* (**tab**) Replicasets configuration. List of *replicaset_config*

Returns:

object

Cluster:server (alias)

Find server by alias.

Parameters:

- *alias:* (string)

Returns:

cartridge.test-helpers.server

Cluster:apply_topology ()

Execute `edit_topology` GraphQL request to setup replicasets, apply roles join servers to replicasets.

Cluster:start ()

Bootstraps cluster if it wasn't bootstrapped before. Otherwise starts servers.

Cluster:stop ()

Stop all servers.

Cluster:join_server (server)

Register running server in the cluster.

Parameters:

- *server:* (**Server**) Server to be registered.

Cluster:wait_until_healthy (server)

Blocks fiber until `cartridge.is_healthy()` returns true on main_server.

Parameters:

- *server:*

Cluster:upload_config (config)

Upload application config, shortcut for `cluster.main_server:upload_config(config)` .

Parameters:

- *config:*

See also:

- cartridge.test-helpers.server.Server:upload_config

Cluster:download_config ()

Download application config, shortcut for `cluster.main_server:download_config()` .

See also:

- cartridge.test-helpers.server.Server:download_config

Cluster:retrying (config, fn[, ...])

Keeps calling fn until it returns without error. Throws last error if config.timeout is elapsed.

Parameters:

- *config:* (**tab**) Options for `luatest.helpers.retrying` .
- *fn:* (**func**) Function to call
- *...:* Args to run fn with. (optional)

Tables

cartridge.test-helpers.cluster.replicaset_config

Replicaset config.

Fields:

- *alias:* (string) Prefix to generate server alias automatically. (optional)
- *uuid:* (string) Replicaset uuid. (optional)
- *roles:* ({string}) List of roles for servers in the replicaset.
- *vshard_group:* (optional string) Name of vshard group.
- *all_rw:* (optional **boolan**) Make all replicas writable.
- *servers:* (table or **number**) List of objects to build `Server` s with or.. code-block:: lua number of servers in replicaset.

Class *cartridge.test-helpers.server*

Extended luatest.Server class to run tarantool instance.

Functions

Server:build_env ()

Generates environment to run process with. The result is merged into os.environ().

Returns:

map

Server:start ()

Start the server.

Server:stop ()

Stop server process.

Server:graphql (request, http_options)

Perform GraphQL request.

Parameters:

- *request:*
  - *query:* (string) grapqhl query
  - *variables:* (optional table) variables for graphql query
  - *raise:* (optional **boolean**) raise if response contains an error(default: **true**)
- *http_options:* (table) passed to `http_request` options. (optional)

Returns:

(table) parsed response JSON.

Raises:

- HTTPRequest error
- GraphQL error

Server:join_cluster (main_server[, options])

Advertise this server to the cluster.

Parameters:

- *main_server:* Server to perform GraphQL request on.
- *options:*
  - *timeout:* request timeout

Server:setup_replicaset (config)

Update server's replicaset config.

Parameters:

- *config:*
  - *uuid:* replicaset uuid
  - *roles:* list of roles
  - *master:*

- *weight:*

Server:upload_config (config)

Upload application config.

Parameters:

- *config:* (string or table) * table will be encoded as yaml and posted to /admin/config.

Server:download_config ()

Download application config.

Methods

cartridge.test-helpers.server:new (object)

Build server object.

Parameters:

- *object:*
  - *command:* (string) Command to start server process.
  - *workdir:* (string) Value to be passed in `TARANTOOL_WORKDIR` .
  - *chdir:* (**bool**) Path to cwd before starting a process. (optional)
  - *env:* (**tab**) Table to pass as env variables to process. (optional)
  - *args:* (**tab**) Args to run command with. (optional)
  - *http_port:* (**int**) Value to be passed in `TARANTOOL_HTTP_PORT` and used to perform HTTP requests. (optional)
  - *advertise_port:* (**int**) Value to generate `TARANTOOL_ADVERTISE_URI` and used for net_box connection.
  - *net_box_port:* (**int**) Alias for `advertise_port` . (optional)
  - *net_box_credentials:* (**tab**) Override default net_box credentials. (optional)
  - *alias:* (string) Instance alias.
  - *cluster_cookie:* (string) Value to be passed in `TARANTOOL_CLUSTER_COOKIE` and used as default net_box password.
  - *instance_uuid:* (string) Server identifier. (optional)
  - *replicaset_uuid:* (string) Replicaset identifier. (optional)

Returns:

input object

Class *cartridge.test-helpers.etcd*

Class to run and manage etcd node.

Functions

Etcd:new (object)

Build etcd node object.

Parameters:

- *object:*
  - *name:* (string) Human-readable node name.
  - *workdir:* (string) Path to the data directory.
  - *etcd_path:* (string) Path to the etcd executable.
  - *peer_url:* (string) URL to listen on for peer traffic.
  - *client_url:* (string) URL to listen on for client traffic.
  - *env:* (**tab**) Environment variables passed to the process. (optional)
  - *args:* (**tab**) Command-line arguments passed to the process. (optional)

Returns:

object

Etcd:start ()

Start the node.

Etcd:stop ()

Stop the node.

4.3.6 Cartridge Command Line Interface

Содержание

- *Cartridge Command Line Interface*
  - *Installation*
  - *Quick start*
  - *Command-line completion*
    - *Linux*
    - *OS X*
  - *Usage*
    - *An application lifecycle*

* *Creating an application from template*
* *Building an application*
* *Starting/stopping an application locally*
  - `start`
  - *Options*
  - *Environment variables*
  - *Overriding default options*
  - `stop`
  - `status`
  - `log`
  - `clean`
* *Packing an application*
  - *Build directory*
  - *Distribution directory*
  - *Stage 1. Cleaning up the application directory*
  - *Stage 2. Building the application*
  - *Stage 3. Cleaning up the files before packing*
* *Repairing a cluster*
  - *Repair commands*
  - *Topology summary*
  - *Remove instance*
  - *Set leader*
  - *Set advertise URI*
* *TGZ*
* *RPM and DEB*
  - *Usage example*
  - *Package details*
* *Docker*
  - *Usage example*
  - *Runtime image tag*
  - *Build and runtime images*
  - *Tarantool Enterprise SDK*
  - *Customizing the application build in Docker*
  - *Using the runtime image*
* *Special files*

- *Example: cartridge.pre-build*
- *Example: cartridge.post-build*

Installation

1. Install third-party software:
   - Install `git`, a version control system.
   - Install the `unzip` utility.
   - Install the `gcc` compiler.
   - Install the `cmake` and `make` tools.
2. Install Tarantool 1.10 or higher.

   You can:
   - Install it from a package (see https://www.tarantool.io/en/download/ for OS-specific instructions).
   - Build it from sources (see https://www.tarantool.io/en/download/os-installation/building-from-source/).
3. [On all platforms except MacOS X] If you built Tarantool from sources, you need to manually set up the Tarantool packages repository:

   ```
   curl -L https://tarantool.io/installer.sh | sudo -E bash -s -- --repo-only
   ```
4. Install the `cartridge-cli` package:
   - for CentOS, Fedora, ALT Linux (RPM package):

     ```
     sudo yum install cartridge-cli
     ```
   - for Debian, Ubuntu (DEB package):

     ```
     sudo apt-get install cartridge-cli
     ```
   - for MacOS X (Homebrew formula):

     ```
     brew install cartridge-cli
     ```
5. Check the installation:

   ```
   cartridge version
   ```

Now you can create and start your first application!

Quick start

To create your first application:

```
cartridge create --name myapp
```

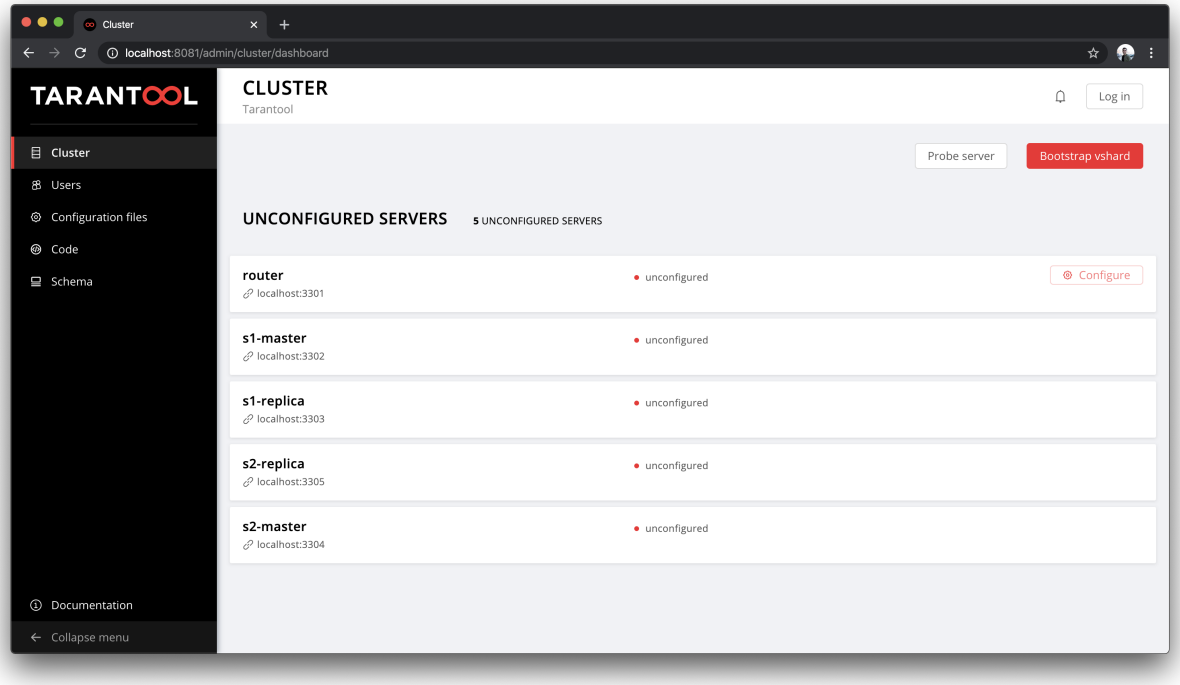
Let's go inside:

```
cd myapp
```

Now build the application and start it:

```
cartridge build
cartridge start
```

That's it! Now you can visit http://localhost:8081 and see your application's Admin Web UI:



You can find more details in this README document or you can start with the getting started guide.

Command-line completion

Linux

RPM and DEB `cartridge-cli` packages contain `/etc/bash_completion.d/cartridge` Bash completion script. To enable completion after `cartridge-cli` installation start a new shell or source `/etc/bash_completion.d/cartridge` completion file. Make sure that you have bash completion installed.

To install Zsh completion, say

```
cartridge gen completion --skip-bash --zsh="${fpath[1]}/_cartridge"
```

To enable shell completion:

```
echo "autoload -U compinit; compinit" >> ~/.zshrc
```

OS X

If you install `cartridge-cli` from `brew`, it automatically installs both Bash and Zsh completions.

Usage

For more details, say:

```
cartridge --help
```

The following commands are supported:

- `create` — create a new application from template;
- `build` — build the application for local development and testing;
- `start` — start a Tarantool instance(s);
- `stop` — stop a Tarantool instance(s);
- `status` — get current instance(s) status;
- `log` — get logs of instance(s);
- `clean` - clean instance(s) files;
- `pack` — pack the application into a distributable bundle;
- `repair` — patch cluster configuration files;
- admin - call an admin function provided by the application;
- replicasets - manage cluster replica sets running locally;
- enter and connect - connect to running instance.

The following global flags are supported:

- `verbose` — verbose mode, additional log messages are shown as well as commands/docker output (such as *tarantoolctl rocks make* or *docker build* output);
- `debug` — debug mode (the same as verbose, but temporary files and directories aren't removed);
- `quiet` — the mode that hides all logs; only errors are shown.

An application lifecycle

In a nutshell:

1. *Create* an application (e.g. `myapp`) from template:

   ```
   cartridge create --name myapp
   cd ./myapp
   ```

2. *Build* the application for local development and testing:

   ```
   cartridge build
   ```

3. *Run* instances locally:

```
cartridge start
cartridge stop
```

4. *Pack* the application into a distributable (e.g. into an RPM package):

```
cartridge pack rpm
```

Creating an application from template

To create an application from the Cartridge template, say this in any directory:

```
cartridge create [PATH] [flags]
```

The following options (`[flags]`) are supported:

- `--name strin` is an application name.
- `--from DIR` is a path to the application template (see details below).
- `--template string` is a name of application template to be used. Currently only `cartridge` template is supported.

Application is created in the `<path>/<app-name>/` directory.

By default, `cartridge` template is used. It contains a simple Cartridge application with:

- one custom role with an HTTP endpoint;
- sample tests and basic test helpers;
- files required for development (like `.luacheckrc`).

If you have `git` installed, this will also set up a Git repository with the initial commit, tag it with version 0.1.0, and add a `.gitignore` file to the project root.

Let's take a closer look at the files inside the `<app_name>/` directory:

- application files:
  - `app/roles/custom-role.lua` a sample custom role with simple HTTP API; can be enabled as `app.roles.custom`
  - `<app_name>-scm-1.rockspec` file where you can specify application dependencies
  - `init.lua` file which is the entry point for your application
  - `stateboard.init.lua` file which is the entry point for the application stateboard
- *special files* (used to build and pack the application):
  - `cartridge.pre-build`
  - `cartridge.post-build`
  - `Dockerfile.build.cartridge`
  - `Dockerfile.cartridge`
- development files:
  - `deps.sh` script that resolves the dependencies from the `.rockspec` file and installs test dependencies (like `luatest`)
  - `instances.yml` file with instances configuration (used by `cartridge start`)

  - `.cartridge.yml` file with Cartridge configuration (used by `cartridge start`)
  - `tmp` directory for temporary files (used as a run dir, see `.cartridge.yml`)
  - `.git` file necessary for a Git repository
  - `.gitignore` file where you can specify the files for Git to ignore
  - `env.lua` file that sets common rock paths so that the application can be started from any directory.
- test files (with sample tests):

```
test
├── helper
│   ├── integration.lua
│   └── unit.lua
│   ├── helper.lua
│   ├── integration
│   │   └── api_test.lua
│   └── unit
│       └── sample_test.lua
```

- configuration files:
  - `.luacheckrc`
  - `.luacov`
  - `.editorconfig`

You can create your own application template and use it with `cartridge create` with `--from` flag.

If template directory is a git repository, the *.git/* files would be ignored on instantiating template. In the created application a new git repo is initialized.

Template application shouldn't contain *.rocks* directory. To specify application dependencies use rockspec and *cartridge.pre-build* files.

Filenames and content can contain text templates.

Available variables are:

- `Name` — the application name;
- `StateboardName` — the application stateboard name (`<app-name>-stateboard`);
- `Path` - an absolute path to the application.

For example:

```
my-template
├── {{ .Name }}-scm-1.rockspec
└── init.lua
└── stateboard.init.lua
└── test
    └── sample_test.lua
```

`init.lua`:

```
print("Hi, I am {{ .Name }} application")
print("I also have a stateboard named {{ .StateboardName }}")
```

Building an application

To build your application locally (for local testing), say this in any directory:

```
cartridge build [PATH] [flags]
```

This command requires one argument — the path to your application directory (i.e. to the build source). The default path is `.` (the current directory).

This command runs:

1. `cartridge.pre-build` if the *pre-build file* exists. This builds the application in the `[PATH]` directory.
2. `tarantoolctl rocks make` if the *rockspec file* exists. This installs all Lua rocks to the `[PATH]` directory.

During step 1 of the `cartridge build` command, `cartridge` builds the application inside the application directory – unlike when building the application as part of the `cartridge pack` command, when the application is built in a temporary *build directory* and no build artifacts remain in the application directory.

During step 2 – the key step here – `cartridge` installs all dependencies specified in the rockspec file (you can find this file within the application directory created from template).

(An advanced alternative would be to specify build logic in the rockspec as `cmake` commands, like we do it for `cartridge`.)

If your application depends on closed-source rocks, or if the build should contain rocks from a project added as a submodule, then you need to **install** all these dependencies before calling `tarantoolctl rocks make`. You can do it using the file `cartridge.pre-build` in your application root (again, you can find this file within the application directory created from template). In this file, you can specify all rocks to build (e.g. `tarantoolctl rocks make --chdir ./third_party/proj`). For details, see *special files*.

As a result, in the application's `.rocks` directory you will get a fully built application that you can start locally from the application's directory.

Starting/stopping an application locally

`start`

Now, after the application is *built*, you can run it locally:

```
cartridge start [INSTANCE_NAME...] [flags]
```

where `[INSTANCE_NAME...]` means that several instances can be specified.

If no `INSTANCE_NAME` is provided, all the instances from the Cartridge instances configuration file are taken as arguments (see the `--cfg` option below).

We also need an application name (`APP_NAME`) to pass it to the instances while started and to define paths to the instance files (for example, `<run-dir>/<APP_NAME>.<INSTANCE_NAME>.pid`). By default, the `APP_NAME` is taken from the application rockspec in the current directory, but also it can be defined explicitly via the `--name` option (see description below).

Options

The following options (`[flags]`) are supported:

- `--script FILE` is the application's entry point. It should be a relative path to the entry point in the project directory or an absolute path. Defaults to `init.lua` (or to the value of the «script» parameter in the Cartridge *configuration file*).
- `--run-dir DIR` is the directory where PID and socket files are stored. Defaults to `./tmp/run` (or to the value of the «run-dir» parameter in the Cartridge *configuration file*).
- `--data-dir DIR` is the directory where instances" data is stored. Each instance's working directory is `<data-dir>/<app-name>.<instance-name>`. Defaults to `./tmp/data` (or to the value of the «data-dir» parameter in the Cartridge *configuration file*).
- `--log-dir DIR` is the directory to store instances logs when running in background. Defaults to `./tmp/log` (or to the value of the «log-dir» parameter in the Cartridge *configuration file*).
- `--cfg FILE` is the configuration file for Cartridge instances. Defaults to `./instances.yml` (or to the value of the «cfg» parameter in the Cartridge *configuration file*).
- `--daemonize, -d` starts the instance in background. With this option, Tarantool also waits until the application's main script is finished. For example, it is useful if the `init.lua` requires time-consuming startup from snapshot, and Tarantool waits for the startup to complete. This is also useful if the application's main script generates errors, and Tarantool can handle them.
- `--stateboard` starts the application stateboard as well as instances. Ignored if `--stateboard-only` is specified.
- `--stateboard-only` starts only the application stateboard. If specified, `INSTANCE_NAME...` are ignored.
- `--name string` defines the application name. By default, it is taken from the application rockspec.
- `--timeout string` Time to wait for instance(s) start in background. Can be specified in seconds or in the duration form (`72h3m0.5s`). Timeout can't be negative. Timeout `0` means no timeout (wait for instance(s) start forever). The default timeout is 60 seconds (`1m0s`).

Environment variables

The `cartridge start` command starts a Tarantool instance with enforced **environment variables**:

```
TARANTOOL_APP_NAME="<instance-name>"
TARANTOOL_INSTANCE_NAME="<app-name>"
TARANTOOL_CFG="<cfg>"
TARANTOOL_PID_FILE="<run-dir>/<app-name>.<instance-name>.pid"
TARANTOOL_CONSOLE_SOCK="<run-dir>/<app-name>.<instance-name>.control"
TARANTOOL_WORKDIR="<data-dir>/<app-name>.<instance-name>.control"
```

When started in background, a notify socket path is passed additionally:

```
NOTIFY_SOCKET="<data-dir>/<app-name>.<instance-name>.notify"
```

`cartridge.cfg()` uses `TARANTOOL_APP_NAME` and `TARANTOOL_INSTANCE_NAME` to read the instance's configuration from the file provided in `TARANTOOL_CFG`.

Overriding default options

You can override default options for the `cartridge` command in the `./.cartridge.yml` configuration file.

Here is an example of `.cartridge.yml`:

```
run-dir: my-run-dir
cfg: my-instances.yml
script: my-init.lua
```

stop

To stop one or more running instances, say:

```
cartridge stop [INSTANCE_NAME...] [flags]
```

By default, SIGTERM is sent to instances.

The following options (`[flags]`) are supported:

- `-f, --force` indicates if instance(s) stop should be forced (sends SIGKILL).

The following *options* from the `start` command are supported:

- `--run-dir DIR`
- `--cfg FILE`
- `--stateboard`
- `--stateboard-only`

Примечание: `run-dir` should be exactly the same as used in the `cartridge start` command. PID files stored there are used to stop the running instances.

---

status

To check the current instance status, use the `status` command:

```
cartridge status [INSTANCE_NAME...] [flags]
```

The following *options* from the `start` command are supported:

- `--run-dir DIR`
- `--cfg FILE`
- `--stateboard`
- `--stateboard-only`

log

To get logs of the instance running in background, use the `log` command:

```
cartridge log [INSTANCE_NAME...] [flags]
```

The following options (`[flags]`) are supported:

- `-f, --follow` outputs appended data as the log grows.
- `-n, --lines int` is the number of lines to output (from the end). Defaults to 15.

The following *options* from the `start` command are supported:

- `--log-dir DIR`
- `--run-dir DIR`
- `--cfg FILE`
- `--stateboard`
- `--stateboard-only`

clean

To remove instance(s) files (log, workdir, console socket, PID-file and notify socket), use the `clean` command:

```
cartridge clean [INSTANCE_NAME...] [flags]
```

cartridge clean for running instance(s) causes an error.

The following *options* from the `start` command are supported:

- `--log-dir DIR`
- `--data-dir DIR`
- `--run-dir DIR`
- `--cfg FILE`
- `--stateboard`
- `--stateboard-only`

Packing an application

To pack your application, say this in any directory:

```
cartridge pack TYPE [PATH] [flags]
```

where:

- `TYPE` (required) is the distribution type. Supported types:
  - *TGZ*
  - *RPM*
  - *DEB*
  - *Docker*
- `PATH` (optional) is the path to the application directory to pack. Defaults to `.` (the current directory).

Примечание: If you pack application into RPM or DEB on MacOS without *–use-docker* flag, the result artifact is broken - it contains rocks and executables that can't be used on Linux. In this case packing fails.

The options (`[flags]`) are as follows:

- `--name string` (common for all distribution types) is the application name. It coincides with the package name and the systemd-service name. The default name comes from the `package` field in the rockspec file.
- `--version string` (common for all distribution types) is the application's package version. The expected pattern is `major.minor.patch[-count][-commit]`: if you specify `major.minor.patch`, it is normalized to `major.minor.patch-count`. The default version is determined as the result of `git describe --tags --long`. If the application is not a git repository, you need to set the `--version` option explicitly.
- `--suffix string` (common for all distribution types) is the result file (or image) name suffix.
- `--unit-template string` (used for `rpm` and `deb`) is the path to the template for the `systemd` unit file.
- `--instantiated-unit-template string` (used for `rpm` and `deb`) is the path to the template for the `systemd` instantiated unit file.
- `--stateboard-unit-template string` (used for `rpm` and `deb`) is the path to the template for the stateboard `systemd` unit file.
- `--use-docker` (enforced for `docker`) forces to build the application in Docker.
- `--tag strings` (used for `docker`) is the tag(s) of the Docker image that results from `pack docker`.
- `--from string` (used for `docker`) is the path to the base Dockerfile of the runtime image. Defaults to `Dockerfile.cartridge` in the application root.
- `--build-from string` (common for all distribution types, used for building in Docker) is the path to the base Dockerfile of the build image. Defaults to `Dockerfile.build.cartridge` in the application root.
- `--no-cache` creates build and runtime images with `--no-cache` docker flag.
- `--cache-from strings` images to consider as cache sources for both build and runtime images. See `--cache-from` flag for `docker build` command.
- `--sdk-path string` (common for all distribution types, used for building in Docker) is the path to the SDK to be delivered in the result artifact. Alternatively, you can pass the path via the `TARANTOOL_SDK_PATH` environment variable (this variable is of lower priority).
- `--sdk-local` (common for all distribution types, used for building in Docker) is a flag that indicates if the SDK from the local machine should be delivered in the result artifact.

For Tarantool Enterprise, you must specify one (and only one) of the `--sdk-local` and `--sdk-path` options.

For `rpm`, `deb`, and `tgz`, we also deliver rocks modules and executables specific for the system where the `cartridge pack` command is running.

For `docker`, the resulting runtime image will contain rocks modules and executables specific for the base image (`centos:8`).

Next, we dive deeper into the packaging process.

Build directory

The first step of the packaging process is to *build the application*.

By default, application build is done in a temporary directory in `~/.cartridge/tmp/`, so the packaging process doesn't affect the contents of your application directory.

You can specify a custom build directory for your application in the `CARTRIDGE_TEMPDIR` environment variable. If this directory doesn't exists, it will be created, used for building the application, and then removed.

If you specify an existing directory in the `CARTRIDGE_TEMPDIR` environment variable, the `CARTRIDGE_TEMPDIR/cartridge.tmp` directory will be used for build and then removed. This directory will be cleaned up before building the application.

Distribution directory

For each distribution type, a temporary directory with application source files is created (further on we address it as *application directory*). This includes 3 stages.

Stage 1. Cleaning up the application directory

On this stage, some files are filtered out of the application directory:

- First, `git clean -X -d -f` removes all untracked and ignored files (it works for submodules, too).
- After that, `.rocks` and `.git` directories are removed.

Files permissions are preserved, and the code files owner is set to `root:root` in the resulting package.

All application files should have at least `a+r` permissions (`a+rx` for directories). Otherwise, `cartridge pack` command raises an error.

Stage 2. Building the application

On this stage, `cartridge` *builds* the application in the cleaned up application directory.

Stage 3. Cleaning up the files before packing

On this stage, `cartridge` runs `cartridge.post-build` (if it exists) to remove junk files (like `node_modules`) generated during application build.

See an *example* in *special files*.

Repairing a cluster

To repair a running application, you can use the `cartridge repair` command.

There are several simple rules you need to know before using this command:

- Rule #1 of `repair` is: you do not use it if you aren't sure that it's exactly what you need.
- Rule #2: always use `--dry-run` before running `repair`.
- Rule #3: do not hesitate to use the `--verbose` option.
- Rule #4: do not use the `--force` option if you aren't sure that it's exactly what you need.

Please, pay attention to the troubleshooting documentation before using `repair`.

What does `repair` actually do?

It patches the cluster-wide configuration files of application instances placed on the local machine. Note that it's not enough to *apply* new configuration: the configuration should be *reloaded* by the instance.

`repair` was created to be used on production (but it still can be used for local development). So, it requires the application name option `--name`. Moreover, remember that the default data directory is `/var/lib/tarantool` and the default run directory is `/var/run/tarantool` (both of them can be rewritten by options).

In default mode, `repair` walks across all cluster-wide configurations placed in `<data-dir>/<app-name>.*` directories and patches all found configuration files.

If the `--dry-run` flag is specified, files aren't patched, and only a computed configuration diff is shown.

If configuration files are diverged between instances on the local machine, `repair` raises an error. But you can specify the `--force` option to patch different versions of configuration independently.

`repair` can also reload configuration for all instances if the `--reload` flag is specified (only if the application uses `cartridge >= 2.0.0`). Configuration will be reloaded for all instances that are placed in the new configuration using console sockets that are placed in the run directory. Make sure that you specified the right run directory when using `--reload` flag.

```
cartridge repair [command]
```

The following `repair` commands are available (see *details* below):

- `list-topology` - shows the current topology summary;
- `remove-instance` - removes an instance from the cluster;
- `set-leader` - changes a replica set leader;
- `set-uri` - changes an instance's advertise URI.

All repair commands have these flags:

- `--name` (required) is an application name.
- `--data-dir` is a directory where the instances" data is stored (defaults to `/var/lib/tarantool`).

All commands, except `list-topology`, have these flags:

- `--run-dir` is a directory where PID and socket files are stored (defaults to `/var/run/tarantool`).
- `--dry-run` runs the `repair` command in the dry-run mode (shows changes but doesn't apply them).
- `--reload` is a flag that enables reloading configuration on instances after the patch.

Repair commands

Topology summary

```
cartridge repair list-topology [flags]
```

Takes no arguments. Prints the current topology summary.

Remove instance

```
cartridge repair remove-instance UUID [flags]
```

Removes an instance with the specified UUID from cluster. If the specified instance isn't found, raises an error.

Set leader

```
cartridge repair set-leader REPLICASET-UUID INSTANCE-UUID [flags]
```

Sets the specified instance as the leader of the specified replica set. Raises an error if:

- a replica set or instance with the specified UUID doesn't exist;
- the specified instance doesn't belong to the specified replica set;
- the specified instance is disabled or expelled.

Set advertise URI

```
cartridge repair set-uri INSTANCE-UUID URI-TO [flags]
```

Rewrites the advertise URI for the specified instance. If the specified instance isn't found or is expelled, raises an error.

TGZ

`cartridge pack tgz ./myapp` creates a .tgz archive. It contains all files from the *distribution directory* (i.e. the application source code and rocks modules described in the application rockspec).

The result artifact name is `<name>-<version>[-<suffix>].tar.gz`.

RPM and DEB

`cartridge pack rpm|deb ./myapp` creates an RPM or DEB package.

The result artifact name is `<name>-<version>[-<suffix>].{rpm,deb}`.

Usage example

After package installation you need to specify configuration for instances to start.

For example, if your application is named `myapp` and you want to start two instances, put the `myapp.yml` file into the `/etc/tarantool/conf.d` directory.

```
myapp:
  cluster_cookie: secret-cookie

myapp.instance-1:
  http_port: 8081
  advertise_uri: localhost:3301

myapp.instance-2:
  http_port: 8082
  advertise_uri: localhost:3302
```

For more details about instances configuration see the documentation.

Now, start the configured instances:

```
systemctl start myapp@instance-1
systemctl start myapp@instance-2
```

If you use stateful failover, you need to start application stateboard.

(Remember that your application should contain `stateboard.init.lua` in its root.)

Add the `myapp-stateboard` section to `/etc/tarantool/conf.d/myapp.yml`:

```
myapp-stateboard:
  listen: localhost:3310
  password: passwd
```

Then, start the stateboard service:

```
systemctl start myapp-stateboard
```

Package details

The installed package name will be `<name>` no matter what the artifact name is.

It contains meta information: the package name (which is the application name), and the package version.

If you use an opensource version of Tarantool, the package has a `tarantool` dependency (version >= `<major>.<minor>` and < `<major+1>`, where `<major>.<minor>` is the version of Tarantool used for packing the application). You should enable the Tarantool repo to allow your package manager install this dependency correctly:

- for both RPM and DEB:

  ```
  curl -L https://tarantool.io/installer.sh | VER=${TARANTOOL_VERSION} bash
  ```

The package contents is as follows:

- the contents of the distribution directory, placed in the `/usr/share/tarantool/<app-name>` directory (for Tarantool Enterprise, this directory also contains `tarantool` and `tarantoolctl` binaries);
- unit files for running the application as a `systemd` service: `/etc/systemd/system/<app-name>.service` and `/etc/systemd/system/<app-name>@.service`;
- application stateboard unit file: `/etc/systemd/system/<app-name>-stateboard.service` (will be packed only if the application contains `stateboard.init.lua` in its root);
- the file `/usr/lib/tmpfiles.d/<app-name>.conf` that allows the instance to restart after server restart.

The following directories are created:

- `/etc/tarantool/conf.d/` — directory for instances configuration;
- `/var/lib/tarantool/` — directory to store instances snapshots;
- `/var/run/tarantool/` — directory to store PID-files and console sockets.

See the documentation for details about deploying a Tarantool Cartridge application.

To start the `instance-1` instance of the `myapp` service, say:

```
systemctl start myapp@instance-1
```

To start the application stateboard service, say:

```
systemctl start myapp-stateboard
```

This instance will look for its configuration across all sections of the YAML file(s) stored in `/etc/tarantool/conf.d/*`.

Use the options `--unit-template`, `--instantiated-unit-template` and `--stateboard-unit-template` to customize standard unit files.

You may need it first of all for DEB packages, if your build platform is different from the deployment platform. In this case, `ExecStartPre` may contain an incorrect path to *mkdir*. As a hotfix, we suggest customizing the unit files.

Example of an instantiated unit file:

```
[Unit]
Description=Tarantool Cartridge app {{ .Name }}@%i
After=network.target

[Service]
Type=simple
ExecStartPre=/bin/sh -c 'mkdir -p {{ .InstanceWorkDir }}'
ExecStart={{ .Tarantool }} {{ .AppEntrypointPath }}
Restart=on-failure
RestartSec=2
User=tarantool
Group=tarantool

Environment=TARANTOOL_APP_NAME={{ .Name }}
Environment=TARANTOOL_WORKDIR={{ .InstanceWorkDir }}
Environment=TARANTOOL_CFG={{ .ConfPath }}
Environment=TARANTOOL_PID_FILE={{ .InstancePidFile }}
Environment=TARANTOOL_CONSOLE_SOCK={{ .InstanceConsoleSock }}
Environment=TARANTOOL_INSTANCE_NAME=%i

LimitCORE=infinity
# Disable OOM killer
OOMScoreAdjust=-1000
# Increase fd limit for Vinyl
LimitNOFILE=65535

# Systemd waits until all xlogs are recovered
TimeoutStartSec=86400s
# Give a reasonable amount of time to close xlogs
TimeoutStopSec=10s

[Install]
WantedBy=multi-user.target
Alias={{ .Name }}.%i
```

Supported variables:

- `Name` — the application name;
- `StateboardName` — the application stateboard name (`<app-name>-stateboard`);
- `DefaultWorkDir` — default instance working directory (`/var/lib/tarantool/<app-name>.default`);
- `InstanceWorkDir` — application instance working directory (`/var/lib/tarantool/<app-name>.<instance-name>`);

- `StateboardWorkDir` — stateboard working directory (`/var/lib/tarantool/<app-name>-stateboard`);
- `DefaultPidFile` — default instance pid file (`/var/run/tarantool/<app-name>.default.pid`);
- `InstancePidFile` — application instance pid file (`/var/run/tarantool/<app-name>.<instance-name>.pid`);
- `StateboardPidFile` — stateboard pid file (`/var/run/tarantool/<app-name>-stateboard.pid`);
- `DefaultConsoleSock` — default instance console socket (`/var/run/tarantool/<app-name>.default.control`);
- `InstanceConsoleSock` — application instance console socket (`/var/run/tarantool/<app-name>.<instance-name>.control`);
- `StateboardConsoleSock` — stateboard console socket (`/var/run/tarantool/<app-name>-stateboard.control`);
- `ConfPath` — path to the application instances config (`/etc/tarantool/conf.d`);
- `AppEntrypointPath` — path to the application entrypoint (`/usr/share/tarantool/<app-name>/init.lua`);
- `StateboardEntrypointPath` — path to the stateboard entrypoint (`/usr/share/tarantool/<app-name>/stateboard.init.lua`);

Docker

`cartridge pack docker ./myapp` builds a Docker image where you can start one instance of the application.

Usage example

To start the `instance-1` instance of the `myapp` application, say:

```
docker run -d \
                --name instance-1 \
                -e TARANTOOL_INSTANCE_NAME=instance-1 \
                -e TARANTOOL_ADVERTISE_URI=3302 \
                -e TARANTOOL_CLUSTER_COOKIE=secret \
                -e TARANTOOL_HTTP_PORT=8082 \
                -p 127.0.0.1:8082:8082 \
                myapp:1.0.0
```

By default, `TARANTOOL_INSTANCE_NAME` is set to `default`.

To check the instance logs, say:

```
docker logs instance-1
```

Runtime image tag

The result image is tagged as follows:

- `<name>:<detected_version>[-<suffix>]`: by default;
- `<name>:<version>[-<suffix>]`: if the `--version` parameter is specified;

- <tag>: if the --tag parameter is specified.

Build and runtime images

In fact, two images are created during the packing process: build image and runtime image.

First, the build image is used to perform application build. The build stages here are exactly the same as for other distribution types:

- *Stage 1. Cleaning up the application directory*
- *Stage 2. Building the application* (the build is always done **'in Docker <Building in Docker _>'_**)
- *Stage 3. Cleaning up the files before packaging*

Second, the files are copied to the resulting (runtime) image, similarly to packing an application as an archive. This image is exactly the result of running `cartridge pack docker`).

Both images are based on `centos:8`.

All packages required for the default `cartridge` application build (`git`, `gcc`, `make`, `cmake`, `unzip`) are installed on the build image.

A proper version of Tarantool is provided on the runtime image:

- For opensource, Tarantool of the same version as the one used for local development is installed to the image.
- For Tarantool Enterprise, the bundle with Tarantool Enterprise binaries is copied to the image.

If your application requires some other applications for build or runtime, you can specify base layers for build and runtime images:

- build image: `Dockerfile.build.cartridge` (default) or `--build-from`;
- runtime image: `Dockerfile.cartridge` (default) or `--from`.

The Dockerfile of the base image should be started with the `FROM centos:8` or `FROM centos:7` line (except comments).

For example, if your application requires `gcc-c++` for build and `zip` for runtime, customize the Dockerfiles as follows:

- `Dockerfile.cartridge.build`:

```
FROM centos:8
RUN yum install -y gcc-c++
# Note that git, gcc, make, cmake, unzip packages
# will be installed anyway
```

- *Dockerfile.cartridge*:

```
FROM centos:8
RUN yum install -y zip
```

Tarantool Enterprise SDK

If you use Tarantool Enterprise, you should explicitly specify the Tarantool SDK to be delivered on the runtime image.

If you want to use the SDK from your local machine, just pass the `--sdk-local` flag to the `cartridge pack docker` command.

Alternatively, you can specify a local path to another SDK using the `--sdk-path` option (or the environment variable `TARANTOOL_SDK_PATH`, which has lower priority).

Customizing the application build in Docker

You can pass `--cache-from` and `--no-cache` options of `docker build` command on building application in docker.

Using the runtime image

The application code is placed in the `/usr/share/tarantool/<app-name>` directory. An opensource version of Tarantool is installed to the image.

The run directory is `/var/run/tarantool/<app-name>`, the workdir is `/var/lib/tarantool/<app-name>`.

The runtime image also contains the file `/usr/lib/tmpfiles.d/<app-name>.conf` that allows the instance to restart after container restart.

It is the user's responsibility to set up a proper advertise URI (`<host>:<port>`) if the containers are deployed on different machines. The problem here is that an instance's advertise URI must be the same on all machines, because it will be used by all the other instances to connect to this one. For example, if you start an instance with an advertise URI set to `localhost:3302`, and then address it as `<instance-host>:3302` from other instances, this won't work: the other instances will be recognizing it only as `localhost:3302`.

If you specify only a port, `cartridge` will use an auto-detected IP, so you need to configure Docker networks to set up inter-instance communication.

You can use Docker volumes to store instance snapshots and xlogs on the host machine. To start an image with a new application code, just stop the old container and start a new one using the new image.

Special files

You can put these files in your application root to control the application packaging process (see examples below):

- `cartridge.pre-build`: a script to be run before `tarantoolctl rocks make`. The main purpose of this script is to build some non-standard rocks modules (for example, from a submodule). Should be executable.
- `cartridge.post-build`: a script to be run after `tarantoolctl rocks make`. The main purpose of this script is to remove build artifacts from result package. Should be executable.

Example: cartridge.pre-build

```
#!/bin/sh

# The main purpose of this script is to build some non-standard rocks modules.
# It will be run before `tarantoolctl rocks make` on application build

tarantoolctl rocks make --chdir ./third_party/my-custom-rock-module
```

Example: cartridge.post-build

```
#!/bin/sh

# The main purpose of this script is to remove build artifacts from resulting package.
# It will be ran after `tarantoolctl rocks make` on application build.

rm -rf third_party
rm -rf node_modules
rm -rf doc
```

4.3.7 Tarantool Cartridge on Kubernetes

This guide covers the full life cycle of a Tarantool Cartridge app–from developing the app to operating it on Kubernetes.

Содержание

- *Tarantool Cartridge on Kubernetes*
  - *Installation tools*
  - *Creating an application*
    - *Building the application*
  - *Creating a Kubernetes cluster*
    - *Using* minikube
    - *Using* kind
  - *Launch the application*
    - *Tarantool Kubernetes operator*
    - *Deploying a Tarantool Cartridge application*
  - *Cluster management*
    - *Adding a new replica*
    - *Adding a shard (replica set)*
    - *Updating application version*
    - *Running multiple Tarantool Cartridge clusters in different namespaces*
    - *Deleting a cluster*
  - *Troubleshooting*
    - *Insufficient CPU*
    - *Insufficient disk space*
  - *Customization*
    - *Sidecar containers*
  - *Installation in an internal network*

* *Delivery of tools*
* *Installing the Tarantool Kubernetes operator*
* *Installing the Tarantool Cartridge app*

Installation tools

The following tools are needed:

1. **cartridge-cli** is a utility for managing Cartridge applications. We need the version 2.3.0 or higher. Installation instructions are available here. If the installation is successful, the *cartridge* utility will be available in the system.

```
$ cartridge version
---
Tarantool Cartridge CLI v2.3.0 linux/amd64 commit: 06a5dad
```

2. **kubectl** is a Kubernetes cluster management tool. We need the version 1.16 or higher. Installation instructions can be found here.

```
$ kubectl version --client
---
Client Version: version.Info{Major:"1", Minor:"16", GitVersion:"v1.16.0", GitCommit:
↪"2bd9643cee5b3b3a5ecbd3af49d09018f0773c77", GitTreeState:"clean", BuildDate:"2019-09-
↪18T14:36:53Z",     GoVersion:"go1.12.9", Compiler:"gc", Platform:"linux/amd64"}
```

3. **helm** is a package manager for Kubernetes apps. We need the version 3.3.x. Installation instructions can be found here.

```
$ helm version
---
version.BuildInfo{Version:"v3.3.1", GitCommit:"249e5215cde0c3fa72e27eb7a30e8d55c9696144",␣
↪GitTreeState:"clean", GoVersion:"go1.14.7"}
```

4. **minikube** is a tool for creating a local Kubernetes cluster. We need the version 1.12 or higher. Installation instructions can be found here.

```
$ minikube version
---
minikube version: v1.12.3
commit: 2243b4b97c131e3244c5f014faedca0d846599f5-dirty
```

5. **kind** (optional) is another tool for creating a local cluster. It can be used instead of the minicube. Installation instructions can be found here.

```
$ kind version
---
kind v0.9.0 go1.15.2 linux/amd64
```

Creating an application

Let's create a Cartridge application named `test-app` using `cartridge-cli`:

```
$ cartridge create --name test-app
---
   • Create application test-app
   • Generate application files
   • Initialize application git repository
   • Application "test-app" created successfully
```

In the `test-app` directory, we get the app created from a template:

```
$ ls test-app
---
...

instances.yml
test-app-scm-1.rockspec
...
```

The app is fully functional and can respond to the HTTP GET request `/hello`.

Примечание: Check the cartridge version in *test-app-scm-1.rockspec*:

```
dependencies = {
    ...
    'cartridge == 2.3.0-1',
    ...
}
```

The version of Cartridge must be >= **2.3.0**. Starting from this version, Cartridge waits for an instance to become available on its DNS address during the instance start. This is required for correct operations on Kubernetes. For versions below 2.3.0, an application must be customized independently. See the example of how to do this.

Building the application

Let's create a Docker image using `cartridge-cli`:

```
$ cartridge pack docker --tag vanyarock01/test-app:0.1.0-0-g68f6117
---
...
Running in 0ffbd57a0edf
Removing intermediate container 0ffbd57a0edf
 ---> aceef7a3be63
 ---> aceef7a3be63
Successfully built aceef7a3be63
Successfully tagged test-app:0.1.0-0-g68f6117
   • Created result image test-app:0.1.0-0-g68f6117
   • Application was successfully packed
```

Upload the image to the Docker registry:

```
$ docker push vanyarock01/test-app:0.1.0-0-g68f6117
---
The push refers to repository [docker.io/vanyarock01/test-app]
b327b35afe0a: Pushed
```

(continues on next page)

(продолжение с предыдущей страницы)

```
de30ed3f758d: Pushed
3c8808fbd85d: Pushed
291f6e44771a: Pushed
0.1.0-0-g275baa8: digest: sha256:5b3b92a615b34c7f132e72e2d61f692cf2091ca28be27bbbfed98106398d1c19␣
↪size: 1160
```

Примечание: You must be logged in via `docker login` and have access rights to the target registry.

Creating a Kubernetes cluster

If you have a ready-made cluster in the cloud, you can use it. If not, we suggest two ways of how to create a local cluster:

- using *minikube*
- using *kind*.

Using *minikube*

Create a Kubernetes cluster of version 1.16.4 with 4GB of RAM (recommended):

```
$ minikube start --kubernetes-version v1.16.4 --memory 4096
---
  minikube v1.12.3 on Ubuntu 18.10
  Automatically selected the docker driver. Other choices: kvm2, virtualbox
  Starting control plane node minikube in cluster minikube
  Creating docker container (CPUs=2, Memory=4096MB) ...
  Preparing Kubernetes v1.16.4 on Docker 19.03.8 ...
  Verifying Kubernetes components...
  Enabled addons: default-storageclass, storage-provisioner
  Done! kubectl is now configured to use "minikube"
```

Wait for the cluster state to be *Ready*:

```
$ kubectl get nodes
---
NAME       STATUS   ROLES    AGE   VERSION
minikube   Ready    master   21m   v1.16.4
```

Using *kind*

Create a Kubernetes cluster of version 1.16.4 by using the *kind* utility as an alternative to *minikube*:

```
$ kind create cluster --image kindest/node:v1.16.4
---
Creating cluster "kind" ...
 ✓ Ensuring node image (kindest/node:v1.16.4)
 ✓ Preparing nodes
 ✓ Writing configuration
 ✓ Starting control-plane
```

(continues on next page)

(продолжение с предыдущей страницы)

```
 ✓ Installing CNI
 ✓ Installing StorageClass
Set kubectl context to "kind-kind"
You can now use your cluster with:

kubectl cluster-info --context kind-kind

Not sure what to do next?  Check out https://kind.sigs.k8s.io/docs/user/quick-start/
```

Let's check the cluster status:

```
$ kubectl get nodes
---
NAME                 STATUS   ROLES    AGE   VERSION
kind-control-plane   Ready    master   48s   v1.16.4
```

Launch the application

To install the Tarantool Kubernetes operator and deploy the cluster, we will use the `helm` utility. Charts are published in our repository. Let's add it:

```
$ helm repo add tarantool https://tarantool.github.io/tarantool-operator
```

Two charts are available in the repository:

```
$ helm search repo tarantool
---
NAME                            CHART VERSION   APP VERSION DESCRIPTION
tarantool/tarantool-operator    0.0.8           1.16.0      kubernetes tarantool operator
tarantool/cartridge             0.0.8           1.0         A Helm chart for tarantool
```

The `tarantool/tarantool-operator` chart installs and configures the operator that manages Tarantool Cartridge clusters.

The `tarantool/cartridge` chart is a template for creating Tarantool Cartridge clusters. With the default settings, this chart deploys an example application consisting of 3 instances. The chart works only in conjunction with the Tarantool Kubernetes operator.

Примечание: Use the same version with both charts. If you set the `tarantool-operator` chart to version 0.0.8, set the `cartridge` chart to the same version 0.0.8.

Install *tarantool-operator* in the *tarantool* namespace:

```
$ helm install tarantool-operator tarantool/tarantool-operator --namespace tarantool --create-
↪namespace --version 0.0.8
---
NAME: tarantool-operator
LAST DEPLOYED: Sun Sep 13 23:29:28 2020
NAMESPACE: tarantool
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

Let's wait until a pod with the operator is ready to work:

```
$ kubectl get pods -n tarantool
---
NAME                                READY   STATUS    RESTARTS   AGE
tarantool-operator-xxx-yyy          0/1     Pending   0          3s
```

In the meantime, let's talk about what the Tarantool operator is and why it is needed.

Tarantool Kubernetes operator

This is a Kubernetes application that can manage Tarantool Cartridge resources.

What does this mean for us?

We don't need to know how to perform administrative actions such as joining a node or creating a replica set. The operator knows how to do this better, and if you set the value for its desired system configuration, it begins to bring the cluster to the desired state.

The Tarantool Kubernetes operator itself is an implementation of the Kubernetes Operator design pattern. It offers the automation of work with user resources using controllers that respond to various events and changes.

The following links can help you understand this pattern:

- Official description on kubernetes.io;
- Overview from the creators of the pattern (CoreOS);
- Post on Habr from Lamoda about the development of the operator.

In the meantime, our pod with `tarantool-operator` went into a *Running* state. The next step is to install the app using the `tarantool/cartridge` helm chart. To do this, prepare a description of the desired system.

Deploying a Tarantool Cartridge application

After you have deployed the cluster and installed the operator, you can move to the next step–launching the app.

We will deploy the app using the `tarantool/cartridge` chart. This is a template. Run it with the default settings and get our example application that has 3 instances. If you define your own settings, you can deploy any application of any topology using the Tarantool Cartridge.

Let's have a look at the settings in the `values.yaml` file. Comments provide a description of each parameter:

```
# Environment name and cluster name
ClusterEnv: "dev"
ClusterName: "test-app"

# Docker image of the application
image:
  repository: "vanyarock01/test-app"
  tag: "0.1.0-0-g68f6117"
  pullPolicy: "IfNotPresent"

# The cluster topology includes a description of the number and
# characteristics of replicasets and is described in the RoleConfig section.

# For example, we want to create a cluster containing two types of replicasets:
```

(continues on next page)

(продолжение с предыдущей страницы)

```
# routers and storages:
RoleConfig:
  - RoleName: "routers" # Name of the replicaset type
    ReplicaCount: 1     # Number of replicas in the replicaset
    ReplicaSetCount: 1  # Number of replicasets for this role
    DiskSize: "1Gi"     # Persistent storage size
    CPUallocation: 0.1  # Part of vCPUs allocated for each container
    MemtxMemoryMB: 256  # Size of RAM allocated for each container
    RolesToAssign:      # Cartridge roles
      - "app.roles.custom"
      - "vshard-router"

  - RoleName: "storages"
    ReplicaCount: 2
    ReplicaSetCount: 1
    DiskSize: "1Gi"
    CPUallocation: 0.1
    MemtxMemoryMB: 256
    RolesToAssign:
      - "app.roles.custom"
      - "vshard-storage"
```

With this configuration we will get the following:

- A Tarantool Cartridge cluster called `test-app`.
- Two replica sets in the cluster: `routers` and `storages`.
- One Tarantool instance in the `routers` replica set.
- Two instances, master and replica, in the `storages` replica set.
- Each replica set performs the roles listed in the `RolesToAssign` parameter.

Install the app:

```
$ helm install -f values.yaml test-app tarantool/cartridge --namespace tarantool --version 0.0.8
---
NAME: test-app
LAST DEPLOYED: Mon Sep 14 10:46:50 2020
NAMESPACE: tarantool
STATUS: deployed
REVISION: 1
```

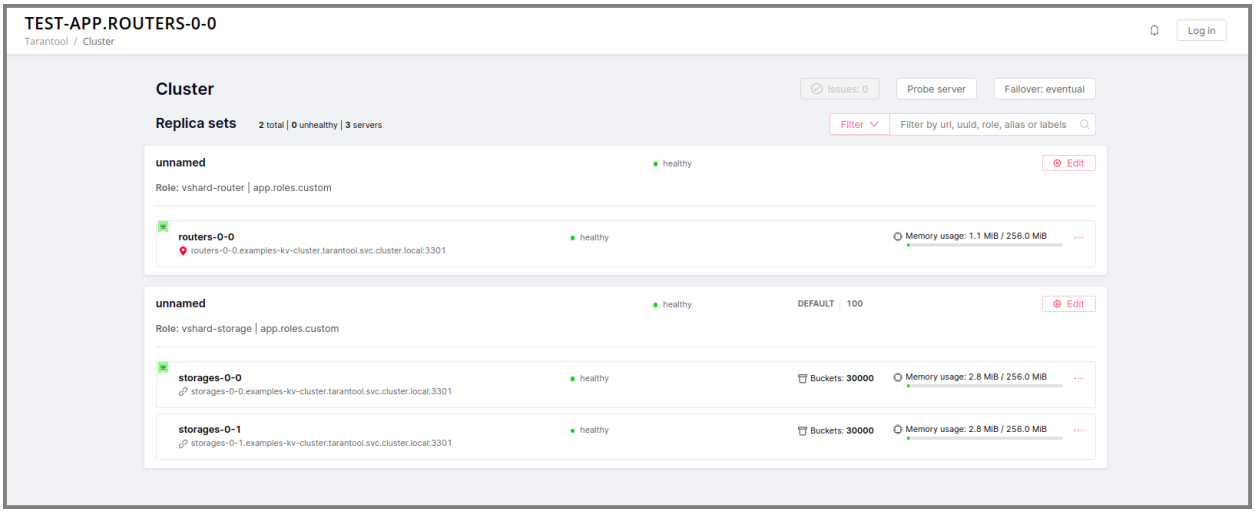
Let's wait for all the pods to launch:

```
$ kubectl -n tarantool get pods
NAME                          READY   STATUS    RESTARTS   AGE
routers-0-0                   0/1     Running   0          10s
storages-0-0                  1/1     Running   0          10s
...
tarantool-operator-xxx-yyy    1/1     Running   0          2m
```

To check the cluster, we forward ports from one of the pods and go to the Cartridge dashboard:

```
$ kubectl port-forward -n tarantool routers-0-0 8081:8081
```

Now the Tarantool Cartridge Web UI is available at `http://localhost:8081`.



Cluster management

Adding a new replica

To increase the number of replicas in a replica set:

1. Change the configuration in the `values.yaml` file.
2. Update the app using the `helm upgrade` command.

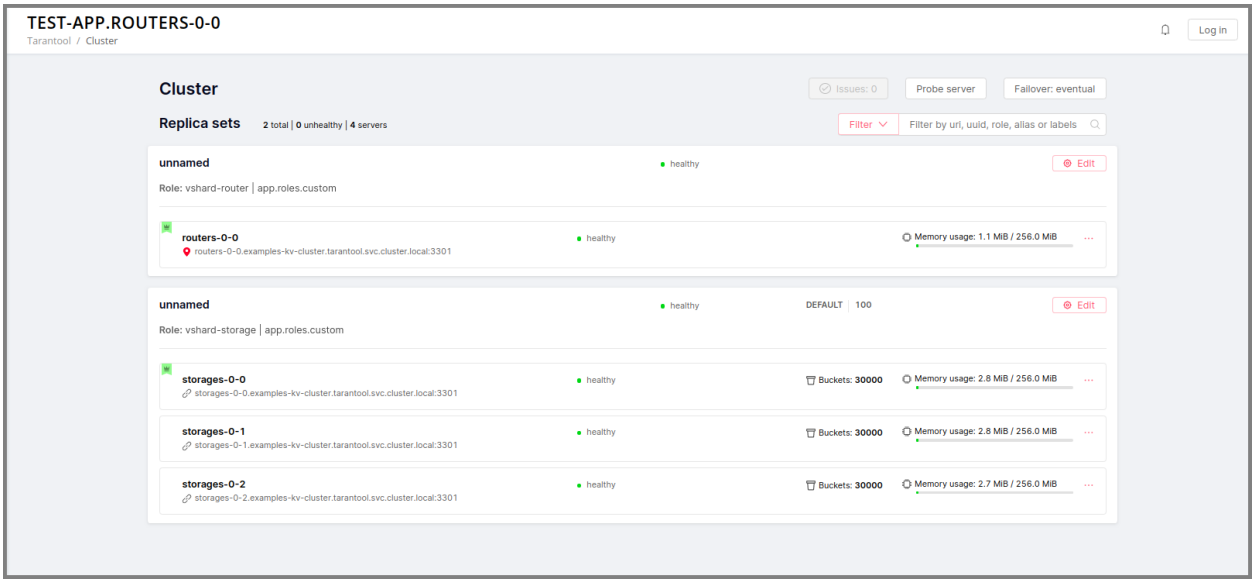
The `ReplicaCount` parameter is responsible for the number of instances in a replica set. Set it to 3 for the `storages` replica set:

```
- RoleName: "storages"
  ReplicaCount: 3
  ReplicaSetCount: 1
  DiskSize: "1Gi"
  CPUallocation: 0.10
  MemtxMemoryMB: 256
  RolesToAssign: "custom.vshard-storage"
```

Update the app:

```
$ helm upgrade -f values.yaml test-app tarantool/cartridge --namespace tarantool
---
Release "test-app" has been upgraded. Happy Helming!
NAME: test-app
LAST DEPLOYED: Tue Sep 15 10:35:55 2020
NAMESPACE: tarantool
STATUS: deployed
REVISION: 2
```

Let's wait until all the new pods go into the **Running** state and are displayed in the Cartridge Web UI.



The `storages` replica set has 3 instances: 1 master and 2 replicas.

Adding a shard (replica set)

The `ReplicaSetCount` parameter defines the number of replicas of the same type.

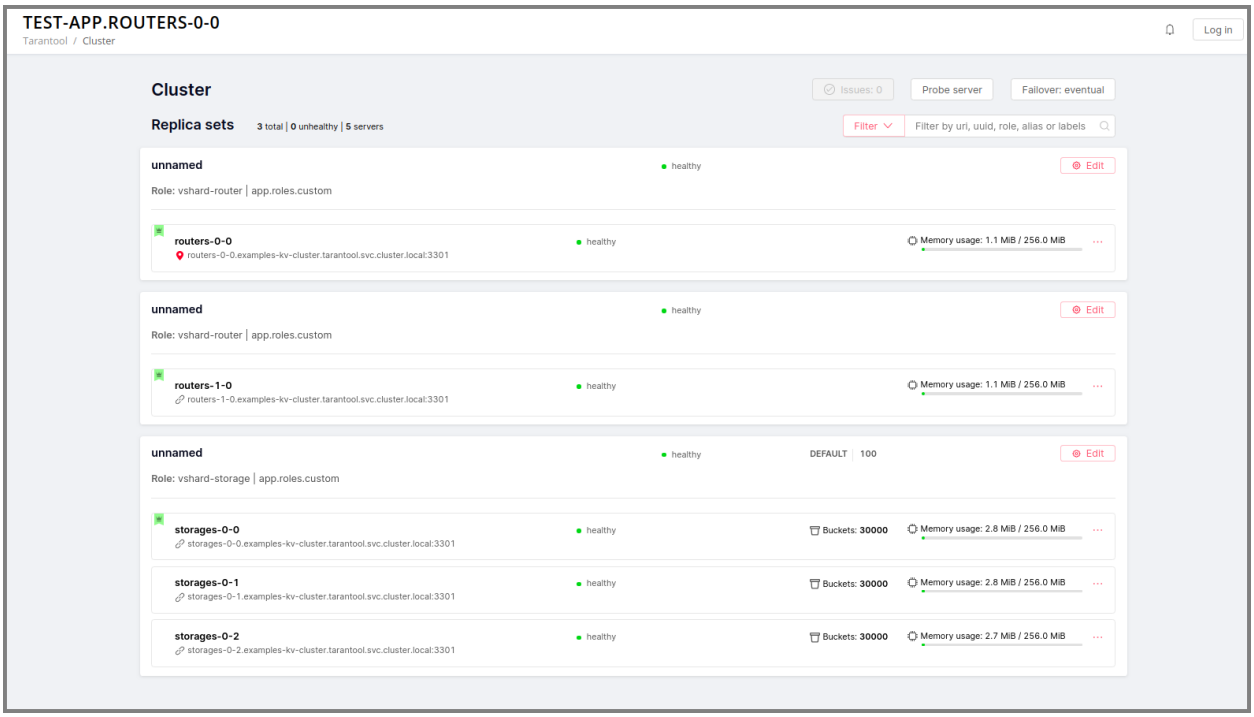
Let's increase the number of the `routers` replica sets to 2:

```
- RoleName: "routers"
  ReplicaCount: 1
  ReplicaSetCount: 2
  DiskSize: "1Gi"
  CPUallocation: 0.10
  MemtxMemoryMB: 256
  RolesToAssign: "custom.vshard-router"
```

Update the app:

```
$ helm upgrade -f values.yaml test-app tarantool/cartridge --namespace tarantool
---
Release "test-app" has been upgraded. Happy Helming!
NAME: test-app
LAST DEPLOYED: Tue Sep 15 10:37:57 2020
NAMESPACE: tarantool
STATUS: deployed
REVISION: 3
```

Let's wait for the new pod to start:



Updating application version

Currently, the app logic contains one HTTP endpoint `/hello` that returns the string `Hello world!` in response to a GET request.

To check this out, let's forward the ports to the desired node:

```
$ kubectl port-forward -n tarantool routers-0-0 8081:8081
---
Forwarding from 127.0.0.1:8081 -> 8081
Forwarding from [::1]:8081 -> 8081
```

And then execute the request:

```
$ curl http://localhost:8081/hello
---
Hello world!
```

Let's add another endpoint that will return the string «Hello world, new version of the app!». To do this, add another `httpd:route` in the `init` function in the `app/roles/custom.lua` role:

```
local function init(opts) -- luacheck: no unused args
    ...
    -- new endpoint
    httpd:route({method = 'GET', path = '/v2/hello'}, function()
        return {body = 'Hello world, new version of the app!'}
    end)

    ...
end
```

Pack the new version of the app:

```
$ cartridge pack docker --tag vanyarock01/test-app:0.1.0-1-g4577716

---

...
Successfully tagged vanyarock01/test-app:0.1.0-1-g4577716
   • Created result image vanyarock01/test-app:0.1.0-1-g4577716
   • Application was successfully packed
```

Upload the new image version to the Docker registry:

```
$ docker push vanyarock01/test-app:0.1.0-1-g4577716
```

Update the `values.yaml` configuration file by specifying a new `image.tag`:

```
image:
  repository: "vanyarock01/test-app"
  tag: "0.1.0-1-g4577716"
  pullPolicy: "IfNotPresent"
```

Update the app on Kubernetes:

```
$ helm upgrade -f values.yaml test-app tarantool/cartridge --namespace tarantool

---

Release "test-app" has been upgraded. Happy Helming!
NAME: test-app
LAST DEPLOYED: Tue Sep 15 10:45:53 2020
NAMESPACE: tarantool
STATUS: deployed
REVISION: 4
```

Tarantool Kubernetes operator uses the **OnDelete** update policy. This means that the update has reached the cluster, but the pods will update the app image only after a restart:

```
$ kubectl delete pods -l tarantool.io/cluster-id=test-app -n tarantool
---
pod "routers-0-0" deleted
pod "routers-1-0" deleted
pod "storages-0-0" deleted
pod "storages-0-1" deleted
pod "storages-0-2" deleted
```

Lets wait for the pods to start again and check the update:

```
$ kubectl port-forward -n tarantool routers-0-0 8081:8081
---
Forwarding from 127.0.0.1:8081 -> 8081
Forwarding from [::1]:8081 -> 8081
...
```

```
curl http://localhost:8081/v2/hello
---
Hello world, new version of the app!
```

Running multiple Tarantool Cartridge clusters in different namespaces

Tarantool Kubernetes operator can manage Tarantool Cartridge clusters only in its own namespace. Therefore, to deploy multiple Cartridge clusters in different namespaces you need to deploy an operator in each of them.

To install an operator in several namespaces, just specify the required namespace during installation:

```
$ helm install tarantool-operator tarantool/tarantool-operator --namespace NS_1 --create-namespace␣
↪--version 0.0.8

$ helm install tarantool-operator tarantool/tarantool-operator --namespace NS_2 --create-namespace␣
↪--version 0.0.8
```

These commands set the operator to the namespace `NS_1` and the namespace `NS_2`. Then, in each of them, you can run a Tarantool Cartridge cluster.

```
$ helm install -f values.yaml cartridge tarantool/cartridge --namespace NS_1 --version 0.0.8

$ helm install -f values.yaml cartridge tarantool/cartridge --namespace NS_2 --version 0.0.8
```

Finally, we have two namespaces. Each has an operator and a Tarantool Cartridge cluster.

Deleting a cluster

To remove a cluster, execute the following command:

```
$ helm uninstall test-app --namespace tarantool
---
release "test-app" uninstalled
```

After a while, all the pods of our application will disappear. Among the pods in the `tarantool` namespace, only the Tarantool Kubernetes operator will remain.

```
$ kubectl get pods -n tarantool
---
NAME                                READY   STATUS    RESTARTS   AGE
tarantool-operator-xxx-yyy          1/1     Running   0          9m45s
```

If you need to remove the Tarantool Kubernetes operator, execute:

```
$ helm uninstall tarantool-operator --namespace tarantool
---
release "tarantool-operator" uninstalled
```

Примечание: `helm uninstall` does not remove persistent volumes. To remove them, you need to additionally perform the following:

```
$ kubectl delete pvc --all -n tarantool
---
persistentvolumeclaim "www-routers-0-0" deleted
persistentvolumeclaim "www-routers-1-0" deleted
persistentvolumeclaim "www-storages-0-0" deleted
```

Troubleshooting

When creating, updating, or scaling a cluster, errors may occur due to lack of physical resources.

Let's examine possible error indications, root causes and solutions.

Insufficient CPU

After executing `helm install / upgrade` the pods remain in the **Pending** state.

It looks like this:

```
$ kubectl get pods -n tarantool
---
NAME                                READY   STATUS    RESTARTS   AGE
routers-0-0                         0/1     Pending   0          20m
routers-1-0                         0/1     Pending   0          20m
storages-0-0                        0/1     Pending   0          20m
tarantool-operator-xxx-yyy          1/1     Running   0          23m
```

Let's take a look at the events of one of the pending pods:

```
$ kubectl -n tarantool describe pods routers-0-0
---
Events:
  Type     Reason             Age                   From               Message
  ----     ------             ----                  ----               -------
  Warning  FailedScheduling   34m                   default-scheduler  0/2 nodes are available:␣
↪2 Insufficient cpu.
  Warning  FailedScheduling   34m                   default-scheduler  0/2 nodes are available:␣
↪2 Insufficient cpu.
  Normal   NotTriggerScaleUp  3m33s (x175 over 34m)  cluster-autoscaler  pod didn't trigger scale-
↪up (it wouldn't fit if a new node is added):
```

It is now clear that we don't have enough CPU. You can reduce the allocated CPU size in the `values.yaml` configuration file–the `CPUallocation` parameter.

Insufficient disk space

After executing `helm install/upgrade` the pods remain in the **ContainerCreating** state. Let's take a look at the events:

```
$ kubectl -n tarantool describe pods routers-0-0
---
Events:
  Type     Reason                  Age                  From                                                 ␣
↪       Message
  ----     ------                  ----                 ----                                                 ␣
↪       -------
  Warning  FailedScheduling        7m44s                default-scheduler                                    ␣
↪       pod has unbound immediate PersistentVolumeClaims
  Warning  FailedScheduling        7m44s                default-scheduler                                    ␣
↪       pod has unbound immediate PersistentVolumeClaims
  Normal   Scheduled               7m42s                default-scheduler                                    ␣
↪       Successfully assigned tarantool/routers-0-0 to kubernetes-cluster-3010-default-group-0
```

(continues on next page)

(продолжение с предыдущей страницы)

```
  Normal   SuccessfulAttachVolume  7m37s                attachdetach-controller
↪       AttachVolume.Attach succeeded for     volume "pvc-e0d3f30a-7dcc-4a67-a69d-4670dc77d556"
  Warning  FailedMount             67s (x9 over 7m5s)   kubelet, kubernetes-cluster-3010-default-
↪group-0  MountVolume.MountDevice failed for volume "pvc-e0d3f30a-7dcc-4a67-a69d-4670dc77d556" :
↪rpc error: code = Internal desc = Unable to find Device path for volume
  Warning  FailedMount             66s (x3 over 5m38s)  kubelet, kubernetes-cluster-3010-default-
↪group-0  Unable to attach or mount volumes: unmounted volumes=[www], unattached volumes=[www
↪default-token-jrz94]: timed out waiting for the condition
```

Such events indicate that there is not enough disk space to create storages. You can change the size of the allocated memory using the `DiskSize` parameter in the *values.yaml* file for replica sets. The error can also be resolved by increasing the size of the physical cluster disk.

Customization

For most cases, the `tarantool/cartridge` helm chart is enough for you. However, if customization is required, you can continue to use the chart by making your own changes. You can also `deployment.yaml` and `kubectl` instead of `helm`.

Sidecar containers

What are they? With Kubernetes, it is possible to create several containers inside one pod that share common resources such as disk storage and network interfaces. Such containers are called sidecar.

Learn more about this architectural pattern here.

For implementation on Kubernetes, it is necessary to expand the container park in the description of the required resource. Let's try to add another service container with `nginx` to each pod containing a container with a Tarantool instance based on this article.

To do this, you will need to change the `tarantool/cartridge` chart. You can find it here. Add a new container with `nginx` to the `ReplicasetTemplate` which can be found in the `templates/deployment.yaml` file.

```
containers:
- name: "pim-storage"
  image: "{{ $.Values.image.repository }}:{{ $.Values.image.tag }}"
  ...
- name: "nginx-container"
  image: "nginx"
  volumeMounts:
    - name: "www"
      mountPath: "/data"
```

Примечание: It is important to describe additional containers strictly after the pim-storage container. Otherwise, problems may occur when updating the version of the application.

By default, the Tarantool Kubernetes operator chooses the first one in the list as the application container.

Now, let's start the installation specifying the path to the directory with the customized chart:

```
$ helm install -f values.yaml test-app tarantool-operator/examples/kv/helm-chart/ --namespace␣
↪tarantool
---
NAME: test-app
LAST DEPLOYED: Wed Sep 30 11:25:12 2020
NAMESPACE: tarantool
STATUS: deployed
REVISION: 1
```

If everything goes well, it will be visible in the pod list:

```
$ kubectl -n tarantool get pods
---
NAME                                READY   STATUS    RESTARTS   AGE
routers-0-0                         2/2     Running   0          113s
routers-1-0                         2/2     Running   0          113s
storages-0-0                        2/2     Running   0          113s
tarantool-operator-xxx-yyy          1/1     Running   0          30m
```

`READY 2/2` means that 2 containers are ready inside the pod.

Installation in an internal network

Delivery of tools

We need to bring the `tarantool-cartridge` and `tarantool-operator` charts and the image of your application inside the internal network.

You can download the charts from the following links:

- tarantool-operator v0.0.8
- cartridge v0.0.8.

Next, you need to pack a Docker image with the `tarantool-operator`. First, let's pull the required version from the Docker Hub:

```
$ docker pull tarantool/tarantool-operator:0.0.8
---
0.0.8: Pulling from tarantool/tarantool-operator
3c72a8ed6814: Pull complete
e6ffc8cffd54: Pull complete
cb731cdf9a11: Pull complete
a42b002f4072: Pull complete
Digest: sha256:e3b46c2a0231bd09a8cdc6c86eac2975211b2c597608bdd1e8510ee0054a9854
Status: Downloaded newer image for tarantool/tarantool-operator:0.0.8
docker.io/tarantool/tarantool-operator:0.0.8
```

And pack it into the archive:

```
$ docker save tarantool/tarantool-operator:0.0.8 | gzip > tarantool-operator-0.0.8.tar.gz
```

After delivering the archive with the container to the target location, you need to load the image to your Docker:

```
$ docker load < tarantool-operator-0.0.8.tar.gz
---
Loaded image: tarantool/tarantool-operator:0.0.8
```

All that remains is to push the image to the internal Docker registry. We will use an example Docker registry hosted on `localhost:5000`:

```
$ docker tag tarantool/tarantool-operator:0.0.8 localhost:5000/tarantool-operator:0.0.8

$ docker push localhost:5000/tarantool-operator:0.0.8
---
The push refers to repository [localhost:5000/tarantool-operator]
febd47bb69b9: Pushed
bacec9f8c1dd: Pushed
d1d164c2f681: Pushed
291f6e44771a: Pushed
0.0.8: digest: sha256:e3b46c2a0231bd09a8cdc6c86eac2975211b2c597608bdd1e8510ee0054a9854 size: 1155
```

Примечание: You can deliver the image with the application using the method described above.

Installing the Tarantool Kubernetes operator

Let's describe the custom operator values in the `operator_values.yaml` file:

```
image:
  # internal Docker repository
  repository: "localhost:5000/tarantool-operator"
  tag: "0.0.8"
  pullPolicy: "IfNotPresent"
```

And install the operator specifying the path to the archive with chart:

```
$ helm install tarantool-operator -f operator_values.yaml ./tarantool-operator-0.0.8.tgz --
↪namespace tarantool --create-namespace
---
NAME: tarantool-operator
LAST DEPLOYED: Tue Dec  1 14:53:47 2020
NAMESPACE: tarantool
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

Check the installation:

```
$ kubectl -n tarantool get pods
---
NAME                                READY   STATUS    RESTARTS   AGE
tarantool-operator-xxx-yyy          1/1     Running   0          7s
```

Installing the Tarantool Cartridge app

We have pushed the app image to the local Docker registry beforehand. What remains is to customize the `values.yaml` file by specifying the available repository:

```
...
image:
  repository: "localhost:5000/test-app"
  tag: "0.1.0-0-g68f6117"
  pullPolicy: "IfNotPresent"
...
```

The complete configuration of the `values.yaml` can be found in the instructions for installing the Tarantool Cartridge application described in the guide earlier.

It remains to unpack the Cartridge chart:

```
$ tar -xzf tarantool-operator-cartridge-0.0.8.tar.gz
```

And run the installation by specifying the path to the chart:

```
$ helm install -f values.yaml test-app tarantool-operator-cartridge-0.0.8/examples/kv/helm-chart/ -
↪-namespace tarantool
---
NAME: test-app
LAST DEPLOYED: Tue Dec  1 15:52:41 2020
NAMESPACE: tarantool
STATUS: deployed
REVISION: 1
```

Let's take a look at the pods to make sure the installation is successful:

```
$ kubectl -n tarantool get pods
---
NAME                                READY   STATUS    RESTARTS   AGE
routers-0-0                         1/1     Running   0          8m30s
storages-0-0                        1/1     Running   0          8m30s
storages-1-0                        1/1     Running   0          8m30s
tarantool-operator-xxx-yyy          1/1     Running   0          67m
```

4.3.8 Changelog

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog and this project adheres to Semantic Versioning.

[Unreleased]

Added

- New GraphQL API: `{cluster {suggestions {force_apply {uuid reasons}}}}` to heal the cluster in case of config errors like `Configuration checksum mismatch`, `Configuration is prepared and locked`, and sometimes `OperationError`.
- Show an issue when `ConfiguringRoles` state stucks for more than 5s.

Fixed

- Properly handle etcd index updates while polling stateful failover updates. The problem affected long-running clusters and resulted in flooding logs with the «Etcd cluster id mismatch» warnings.
- Allow specifying server zone in `join_server` API.

[2.4.0] - 2020-12-29

Added

Zones and zone distances:

- Add support of replica weights and zones via a clusterwide config new section `zone_distances` and a server parameter `zone`.

Fencing:

- Implement a fencing feature. It protects a replicaset from the presence of multiple leaders when the network is partitioned and forces the leader to become read-only.
- New failover parameter `failover_timout` specifies the time (in seconds) used by membership to mark `suspect` members as `dead` which triggers failover.
- Fencing parameters `fencing_enabled`, `fencing_pause`, `fencing_timeout` are available for customization via Lua and GraphQL API, and in WebUI too.

Issues and suggestions:

- New GraphQL API: `{cluster {suggestions {refine_uri {}}}}` to heal the cluster after relocation of servers `advertise_uri`.
- New Lua API `cartridge.config_force_reapply()` and similar GraphQL mutation `cluster { config_force_reapply() }` to heal several operational errors:
  - «Prepare2pcError: Two-phase commit is locked»;
  - «SaveConfigError: .../config.prepare: Directory not empty»;
  - «Configuration is prepared and locked on ...» (an issue);
  - «Configuration checksum mismatch on ...» (an issue).

  It'll unlock two-phase commit (remove `config.prepare` lock), upload the active config from the current instance and reconfigure all roles.

Hot-reload:

- New feature for hot reloading roles code without restarting an instance – `cartridge.reload_roles`. The feature is experimental and should be enabled explicitly: `cartridge.cfg({roles_reload_allowed = true})`.

Miscellanous:

- New `cartridge.cfg` option `swim_broadcast` to manage instances auto-discovery on start. Default: true.
- New argparse options support for tarantool 2.5+: `replication_synchro_quorum`, `replication_synchro_timeout`, `memtx_use_mvcc_engine`.

Changed

- Default value of `failover_timeout` increased from 3 to 20 seconds **(important change)**.
- RPC functions now consider `suspect` members as healthy to be in agreement with failover **(important change)**.

Fixed

- Don't stuck in `ConnectingFullmesh` state when instance is restarted with a different `advertise_uri`. Also keep «Server details» dialog in WebUI operable in this case.
- Allow applying config when instance is in `OperationError`. It doesn't cause loss of quorum anymore.
- Stop vshard fibers when the corresponding role is disabled.
- Make `console.listen` error more clear when `console_sock` exceeds `UNIX_PATH_MAX` limit.
- Fix `upstream.idle` issue tolerance to avoid unnecessary warnings «Replication: long idle ($1 > 1$)».
- Allow removing spaces from DDL schema for the sake of `drop` migrations.
- Make DDL schema validation stricter. Forbid redundant keys in schema top-level and make `spaces` mandatory.

Enhanced is WebUI

- Update server details modal, add support for server zones.
- Properly display errors on WebUI pages «Users» and «Code».
- Indicate config checksum mismatch in issues list.
- Indicate the change of `arvertise_uri` in issues list.
- Show an issue if the clusterwide config is locked on an instance.
- Refresh interval and stat refresh period variables can be customized via frontend-core's `set_variable` feature or at runtime.

[2.3.0] - 2020-08-26

Added

- When failover mode is stateful, all manual leader promotions will be consistent: every instance before becoming writable performs `wait_lsn` operation to sync with previous one. If consistency couldn't be reached due to replication failure, a user could either revert it (promote previous leader), or force promotion to be inconsistent.
- Early logger initialization (for Tarantool $>$ 2.5.0-100, which supports it).
- Add `probe_uri_timeout` argparse option responsible for retrying «Can't ping myself» error on startup.
- New test helper: `cartridge.test-helpers.etcd`.
- Support `on_push` and `on_push_ctx` options for `cartridge.rpc_call()`.
- Changing users password invalidates HTTP cookie.
- Support GraphQL default variables.

Fixed

- Eventual failover may miss an event while roles are being reconfigured.
- Compatibility with pipe logging, see tarantool/tarantool#5220.
- Non-informative assertion when instance is bootstrapped with a distinct `advertise_uri`.
- Indexing `nil` value in `get_topology()` query.
- Initialization race of vshard storage which results in `OperationError`.
- Lack of vshard router attempts to reconnect to the replicas.
- Make GraphQL syntax errors more clear.
- Better `errors.pcall()` performance, `errors` rock updated to v2.1.4.

Enhanced is WebUI

- Show instance names in issues list.
- Show app name in window title.
- Add the «Force leader promotion» button in the stateful failover mode.
- Indicate consistent switchover problems with a yellow leader flag.

[2.2.0] - 2020-06-23

Added

- When running under systemd use `<APP_NAME>.<INSTANCE_NAME>` as default syslog identity.
- Support `etcd` as state provider for stateful failover.

Changed

- Improve rocks detection for feedback daemon. Besides cartridge version it now parses manifest file from the `.rocks/` directory and collects rocks versions.
- Make `uuid` parameters optional for test helpers. Make `servers` option accept number of servers in replicaset.

Enhanced in WebUI

- Prettier errors displaying.
- Enhance replicaset filtering by role / status.
- Error stacktrace received from the backend is shown in notifications.

[2.1.2] - 2020-04-24

Fixed

- Avoid trimming `console_sock` if it's name is too long.
- Fix file descriptors leak during box recovery.
- Support `console_sock` option in stateboard as well as notify socket and other box options similar to regular cartridge instances.

[2.1.1] - 2020-04-20

Fixed

- Frontend core update: fix route mapping

[2.1.0] - 2020-04-16

Added

- Implement stateful failover mode. You can read more in «Failover architecture» documentation topic.
- Respect `box.cfg` options `wal_dir`, `memtx_dir`, `vinyl_dir`. They can be either absolute or relative - in the later case it's calculated relative to `cartridge.workdir`.
- New option in `cartridge.cfg({upgrade_schema=...})` to automatically upgrade schema to modern tarantool version (only for leader). It also has been added for `argparse`.
- Extend GraphQL `issues` API with various topics: `replication`, `failover`, `memory`, `clock`. Make thresholds configurable via argparse.

Changed

- Make GraphQL validation stricter: scalar values can't have sub-selections; composite types must have sub-selections; omitting non-nullable arguments in variable list is forbidden. Your code **may be affected** if it doesn't conform GraphQL specification.
- GraphQL query `auth_params` returns «fullname» (if it was specified) instead of «username».
- Update `errors` dependency to 2.1.3.
- Update `ddl` dependency to 1.1.0.

Deprecated

Lua API:

- `cartridge.admin_get_failover` -> `cartridge.failover_get_params`
- `cartridge.admin_enable/disable_failover` -> `cartridge.failover_set_params`

GraphQL API:

- `query {cluster {failover} }` -> `query {cluster {failover_params {...} } }`

- `mutation {cluster {failover()} }` -> `mutation {cluster {failover_params() {...} } }`

Fixed

- Properly handle nested input object in GraphQL:

```
mutation($uuid: String!) {
  cluster { edit_topology(servers: [{uuid: $uuid ...}]) {} }
}
```

- Show WebUI notification on successful config upload.
- Repair GraphQL queries `add_user`, `issues` on uninitialized instance.

Enhanced in WebUI

- Show «You are here» marker.
- Show application and instance names in app title.
- Indicate replication and failover issues.
- Fix bug with multiple menu items selected.
- Refactor pages filtering, forbid opening blacklisted pages.
- Enable JS chunks caching.

[2.0.2] - 2020-03-17

Added

- Expose membership options in `argparse` module (edit them with environment variables and command-line arguments).
- New internal module to handle `.tar` files.

Lua API:

- `cartridge.cfg({webui_blacklist = {'/admin/code', ...}})`: blacklist certain WebUI pages.
- `cartridge.get_schema()` referencing older `_G.cartridge_get_schema`.
- `cartridge.set_schema()` referencing older `_G.cartridge_set_schema`.

GraphQL API:

- Make use of GraphQL error extensions: provide additional information about `class_name` and `stack` of original error.
- `cluster{ issues{ level message ... }}`: obtain more details on replication status
- `cluster{ self {...} }`: new fields `app_name`, `instance_name`.
- `servers{ boxinfo { cartridge {...} }}`: new fields `version`, `state`, `error`.

Test helpers:

- Allow specifying `all_rw` replicaset flag in luatest helpers.
- Add `cluster({env = ...})` option for specifying clusterwide environment variables.

Changed

- Remove redundant topology availability checks from two-phase commit.
- Prevent instance state transition from `ConnectingFullmesh` to `OperationError` if replication fails to connect or to sync. Since now such fails result in staying in `ConnectingFullmesh` state until it succeeds.
- Specifying `pool.connect()` options `user`, `password`, `reconnect_after` are deprecated and ignored, they never worked as intended and will never do. Option `connect_timeout` is deprecated, but for backward compatibility treated as `wait_connected`.

Fixed

- Fix DDL failure if `spaces` field is `null` in input schema.
- Check content of `cluster_cookie` for absence of special characters so it doesn't break the authorization. Allowed symbols are `[a-zA-Z0-9_.~-]`.
- Drop remote-control connections after full-featured `box.cfg` becomes available to prevent clients from using limited functionality for too long. During instance recovery remote-control won't accept any connections: clients wait for box.cfg to finish recovery.
- Update errors rock dependency to 2.1.2: eliminate duplicate stack trace from `error.str` field.
- Apply `custom_proc_title` setting without waiting for `box.cfg`.
- Make GraphQL compatible with `req:read_cached()` call in httpd hooks.
- Avoid «attempt to index nil value» error when using rpc on an uninitialized instance.

Enhanced in WebUI

- Add an ability to hide certain WebUI pages.
- Validate YAML in code editor WebUI.
- Fix showing errors in Code editor page.
- Remember last open file in Code editor page. Open first file when local storage is empty.
- Expand file tree in Code editor page by default.
- Show Cartridge version in server info dialog.
- Server alias is clickable in replicaset list.
- Show networking errors in splash panel instead of notifications.
- Accept float values for vshard-storage weight.

[2.0.1] - 2020-01-15

Added

- Expose `TARANTOOL_DEMO_URI` environment variable in GraphQL query `cluster{ self{demo_uri} }` for demo purposes.

Fixed

- Notifications in schema editor WebUI.
- Fix GraphQL `servers` query compatibility with old cartridge versions.
- Two-phase commit backward compatibility with v1.2.0.

[2.0.0] - 2019-12-27

Added

- Use for frontend part single point of configuration HTTP handlers. As example: you can add your own client HTTP middleware for auth.
- Built-in DDL schema management. Schema is a part of clusterwide configuration. It's applied to every instance in cluster.
- DDL schema editor and code editor pages in WebUI.
- Instances now have internal state machine which helps to manage cluster operation and protect from invalid state transitions.
- WebUI checkbox to specify `all_rw` replicaset property.
- GraphQL API for clusterwide configuration management.
- Measure clock difference across instances and provide `clock_delta` in GraphQL `servers` query and in `admin.get_servers()` Lua API.
- New option in `rpc_call(..., {uri=...})` to perform a call on a particular uri.

Changed

- `cartridge.rpc_get_candidates()` doesn't return error «No remotes with role available» anymore, empty table is returned instead. **(incompatible change)**
- Base advertise port in luatest helpers changed from 33000 to 13300, which is outside `ip_local_port_range`. Using port from local range usually caused tests failing with an error «address already in use». (*incompatible change*, but affects tests only)
- Whole new way to bootstrap instances. Instead of polling membership for getting clusterwide config the instance now start Remote Control Server (with limited iproto protocol functionality) on the same port. Two-phase commit is then executed over net.box connection. (**major change**, but still compatible)
- Failover isn't triggered on `suspect` instance state anymore
- Functions `admin.get_servers`, `get_replicasets` and similar GraphQL queries now return an error if the instance handling the request is in state `InitError` or `BootError`.
- Clusterwide configuration is now represented with a file tree. All sections that were tables are saved to separate `.yml` files. Compatibility with the old-style configuration is preserved. Accessing unmarshalled sections with `get_readonly/deepcopy` methods is provided without `.yml` extension as earlier. (**major change**, but still compatible)
- After an old leader restarts it'll try to sync with an active one before taking the leadership again so that failover doesn't switch too early before leader finishes recovery. If replication setup fails the instance enters the `OperationError` state, which can be avoided by explicitly specifying `replication_connect_quorum = 1` (or 0). **(major change)**

- Option {`prefer_local = false`} in `rpc_call` makes it always use netbox connection, even to connect self. It never tries to perform call locally.
- Update `vshard` dependency to 0.1.14.

Removed

- Function `cartridge.bootstrap` is removed. Use `admin_edit_topology` interad. **(incompatible change)**
- Misspelled role callback `validate` is now removed completely. Keep using `validate_config`.

Fixed

- Arrange proper failover triggering: don't miss events, don't trigger if nothing changed. Fix races in calling `apply_config` between failover and two-phase commit.
- Race condition when creating working directory.
- Hide users page in WebUI when auth backend implements no user management functions. Enable auth switcher is displayed on main cluster page in this case.
- Displaying boolean values in server details.
- Add deduplication for WebUI notifications: no more spam.
- Automatically choose default vshard group in create and edit replicaset modals.
- Enhance WebUI modals scrolling.

[1.2.0] - 2019-10-21

Added

- „Auto" placeholder to weight input in the Replicaset forms.
- „Select all" and „Deselect all" buttons to roles field in Replicaset add and edit forms.
- Refresh replicaset list in UI after topology edit actions: bootstrap, join, expel, probe, replicaset edit.
- New Lua API `cartridge.http_authorize_request()` suitable for checking HTTP request headers.
- New Lua API `cartridge.http_render_response()` for generating HTTP response with proper `Set-Cookie` headers.
- New Lua API `cartridge.http_get_username()` to check authorization of active HTTP session.
- New Lua API `cartridge.rpc_get_candidates()` to get list of instances suitable for performing a remote call.
- Network error notification in UI.
- Allow specifying vshard storage group in test helpers.

Changed

- Get UI components from Tarantool UI-Kit
- When recovering from snapshot, instances are started read-only. It is still possible to override it by argparse (command line arguments or environment variables)

Fixed

- Editing topology with `failover_priority` argument.
- Now `cartridge.rpc.get_candidates()` returns value as specified in doc. Also it accepts new option `healthy_only` to filter instances which have membership status healthy.
- Replicaset weight tooltip in replicasets list
- Total buckets count in buckets tooltip
- Validation error in user edit form
- Leader flag in server details modal
- Human-readable error for invalid GrqphQL queries: `Field "x" is not defined on type "String"`
- User management error «attempt to index nil value» when one of users has empty e-mail value
- Catch `rpc_call` errors when they are performed locally

[1.1.0] - 2019-09-24

Added

- New Lua API `admin_edit_topology` has been added to unite multiple others: `admin_edit_replicaset`, `admin_edit_server`, `admin_join_server`, `admin_expel_server`. It's suitable for editing multiple servers/replicasets at once. It can be used for bootstrapping cluster from scratch, joining a server to an existing replicaset, creating new replicaset with one or more servers, editing uri/labels of servers, disabling or expelling servers.
- Similar API is implemented in a GraphQL mutation `cluster{edit_topology()}`.
- New GraphQL mutation `cluster { edit_vshard_options }` is suitable for fine-tuning vshard options: `rebalancer_max_receiving`, `collect_lua_garbage`, `sync_timeout`, `collect_bucket_garbage_interval`, `rebalancer_disbalance_threshold`.

Changed

- Both bootstrapping from scratch and patching topology in clusterwide config automatically probe servers, which aren't added to membership yet (earlier it influenced join_server mutation only). This is a prerequisite for multijoin api implementation.
- WebUI users page is hidden if auth_backend doesn't provide list_users callback.

Deprecated

Lua API:

- `cartridge.admin_edit_replicaset()`
- `cartridge.admin_edit_server()`
- `cartridge.admin_join_server()`
- `cartridge.admin_expel_server()`

GraphQL API:

- `mutation{ edit_replicaset() }`
- `mutation{ edit_server() }`
- `mutation{ join_server() }`
- `mutation{ expel_server() }`

Fixed

- Protect `users_acl` and `auth` sections when downloading clusterwide config. Also forbid uploading them.

[1.0.0] - 2019-08-29

Added

- New parameter `topology.replicasets[].all_rw` in clusterwide config for configuring all instances in the replicaset as `read_only = false`. It can be managed with both GraphQL and Lua API `edit_replicaset`.
- Remote Control server - a replacement for the `box.cfg({listen})`, with limited functionality, independent on `box.cfg`. The server is only to be used internally for bootstrapping new instances.
- New module `argparse` for gathering configuration options from command-line arguments, environment variables, and configuration files. It is used internally and overrides `cluster.cfg` and `box.cfg` options.
- Auth parameter `cookie_max_age` is now configurable with GraphQL API. Also now it's stored in clusterwide config, so changing it on a single server will affect all others in cluster.
- Detect that we run under systemd and switch to syslog logging from stderr. This allows to filter log messages by severity with `journalctl`
- Redesign WebUI

Changed

- The project renamed to **cartridge**. Use `require('cartridge')` instead of `require('cluster')`. All submodules are renamed too. **(incompatible change)**
- Submodule `cluster.test_helpers` renamed to `cartridge.test-helpers` for consistency. **(incompatible change)**
- Modifying auth params with GraphQL before the cluster was bootstrapped is now forbidden and returns an error.

- Introducing a new auth parameter `cookie_renew_age`. When cluster handles an HTTP request with the cookie, whose age in older then specified, it refreshes the cookie. It may be useful to set `cookie_max_age` to a small value (for example 10 minutes), so the user will be logged out after `cookie_max_age` seconds of inactivity. Otherwise, if he's active, the cookie will be updated every `cookie_renew_age` seconds and the session will not be interrupted.
- Changed configuration options for `cluster.cfg()`: `roles` now is a mandatory table, `workdir` is optional now (defaults to «.»)
- Parameter `advertise_uri` is optional now, default value is derived as follows. `advertise_uri` is a compound of `<HOST>` and `<PORT>`. When `<HOST>` isn't specified, it's detected as the only non-local IP address. If it can't be determined or there is more than one IP address available it defaults to `"localhost"`. When `<PORT>` isn't specified, it's derived from numeric suffix `_<N>` of `TARANTOOL_INSTANCE_NAME`: `<PORT> = 3300+<N>`. Otherwise default `<PORT> = 3301` is used.
- Parameter `http_port` is derived from instance name too. If it can't be derived it defaults to 8081. New parameter `http_enabled = false` is used to disable it (by default it's enabled).
- Removed user `cluster`, which was used internally for orchestration over netbox. Tarantool built-in user `admin` is used instead now. It can also be used for HTTP authentication to access WebUI. Cluster cookie is used as a password in both cases. **(incompatible change)**

Removed

Two-layer table structure in API, which was deprecated earlier, is now removed completely:

- `cartridge.service_registry.*`
- `cartridge.confapplier.*`
- `cartridge.admin.*`

Instead you can use top-level functions:

- `cartridge.config_get_readonly`
- `cartridge.config_get_deepcopy`
- `cartridge.config_patch_clusterwide`
- `cartridge.service_get`
- `cartridge.admin_get_servers`
- `cartridge.admin_get_replicasets`
- `cartridge.admin_probe_server`
- `cartridge.admin_join_server`
- `cartridge.admin_edit_server`
- `cartridge.admin_expel_server`
- `cartridge.admin_enable_servers`
- `cartridge.admin_disable_servers`
- `cartridge.admin_edit_replicaset`
- `cartridge.admin_get_failover`
- `cartridge.admin_enable_failover`
- `cartridge.admin_disable_failover`

[0.10.0] - 2019-08-01

Added

- Cluster can now operate without vshard roles (if you don't need sharding). Deprecation warning about implicit vshard roles isn't issued any more, they aren't registered unless explicitly specified either in `cluster.cfg({roles=...})` or in `dependencies` to one of user-defined roles.
- New role flag `hidden = true`. Hidden roles aren't listed in `cluster.admin.get_replicasets().roles` and therefore in WebUI. Hidden roles are supposed to be a dependency for another role, yet they still can be enabled with `edit_replicaset` function (both Lua and GraphQL).
- New role flag: `permanent = true`. Permanent roles are always enabled. Also they are hidden implicitly.
- New functions in cluster test_helpers - `Cluster:upload_config(config)` and `Cluster:download_config()`

Fixed

- `cluster.call_rpc` used to return „Role unavailable" error as a first argument instead of `nil, err`. It can appear when role is specified in clusterwide config, but wasn't initialized properly. There are two reasons for that: race condition, or prior error in either role `init` or `apply_config` methods.

[0.9.2] - 2019-07-12

Fixed

- Update frontend-core dependency which used to litter `package.loaded` with tons of JS code

[0.9.1] - 2019-07-10

Added

- Support for vshard groups in WebUI

Fixed

- Uniform handling vshard group „default" when multiple groups aren't configured
- Requesting multiple vshard groups info before the cluster was bootstrapped

[0.9.0] - 2019-07-02

Added

- User management page in WebUI
- Configuring multiple isolated vshard groups in a single cluster
- Support for joining multiple instances in a single call to config_patch_clusterwide
- Integration tests helpers

Changed

- GraphQL API `known_roles` format now includes roles dependencies
- `cluster.rpc_call` option `remote_only` renamed to `prefer_local` with the opposite meaning

Fixed

- Don't display renamed or removed roles in webui
- Uploading config without a section removes it from clusterwide config

[0.8.0] - 2019-05-20

Added

- Specifying role dependencies
- Set read-only option for slave nodes
- Labels for servers

Changed

- Admin http endpoint changed from /graphql to /admin/api
- Graphql output now contains null values for empty objects
- Deprecate implicity of vshard roles `'cluster.roles.vshard-storage'`, `'cluster.roles.vshard-router'`. Now they should be specified explicitly in `cluster.cfg({roles = ...})`
- `cluster.service_get('vshard-router')` now returns `cluster.roles.vshard-router` module instead of `vshard.router` **(incompatible change)**
- `cluster.service_get('vshard-storage')` now returns `cluster.roles.vshard-storage` module instead of *vshard.storage‘* **(incompatible change)**
- `cluster.admin.bootstrap_vshard` now can be called on any instance

Fixed

- Operating vshard-storage roles before vshard was bootstrapped

[0.7.0] - 2019-04-05

Added

- Failover priority configuration using WebUI
- Remote calls across cluster instances using `cluster.rpc` module
- Displaying `box.cfg` and `box.info` in WebUI
- Authorization for HTTP API and WebUI

- Configuration download/upload via WebUI
- Lua API documentation, which you can read with `tarantoolctl rocks doc cluster` command.

Changed

- Instance restart now triggers config validation before roles initialization
- Update WebUI design
- Lua API changed (old functions still work, but issue warnings): - `cluster.confapplier.*` -> `cluster.config_*` - `cluster.service_registry.*` -> `cluster.service_*`

[0.6.3] - 2019-02-08

Fixed

- Cluster used to call „validate()“ role method instead of documented „validate_config()“, so it was added. The undocumented „validate()“ still may be used for the sake of compatibility, but issues a warning that it was deprecated.

[0.6.2] - 2019-02-07

Fixed

- Minor internal corner cases

[0.6.1] - 2019-02-05

Fixed

- UI/UX: Replace «bootstrap vshard» button with a noticable panel
- UI/UX: Replace failover panel with a small button

[0.6.0] - 2019-01-30

Fixed

- Ability to disable vshard-storage role when zero-weight rebalancing finishes
- Active master indication during failover
- Other minor improvements

Changed

- New frontend core
- Dependencies update
- Call to `join_server` automatically does `probe_server`

Added

- Servers filtering by roles, uri, alias in WebUI

[0.5.1] - 2018-12-12

Fixed

- WebUI errors

[0.5.0] - 2018-12-11

Fixed

- Graphql mutations order

Changed

- Callbacks in user-defined roles are called with `is_master` parameter, indicating state of the instance
- Combine `cluster.init` and `cluster.register_role` api calls in single `cluster.cfg`
- Eliminate raising exceptions
- Absorb http server in `cluster.cfg`

Added

- Support of vshard replicaset weight parameter
- `join_server()` `timeout` parameter to make call synchronous

[0.4.0] - 2018-11-27

Fixed/Improved

- Uncaught exception in WebUI
- Indicate when backend is unavailable
- Sort servers in replicaset, put master first
- Cluster mutations are now synchronous, except joining new servers

Added

- Lua API for temporarily disabling servers
- Lua API for implementing user-defined roles

[0.3] - 2018-10-30

Changed

- Config structure **incompatible** with v0.2

Added

- Explicit vshard master configuration
- Automatic failover (switchable)
- Unit tests

[0.2] - 2018-10-01

Changed

- Allow vshard bootstrapping from ui
- Several stability improvements

[0.1] - 2018-09-25

Added

- Basic functionality
- Integration tests
- Luarock-based packaging
- Gitlab CI integration

4.4 Сервер приложений

В данной главе мы рассмотрим основы работы с Tarantool'ом в качестве сервера приложений на языке Lua.

Эта глава состоит из следующих разделов:

4.4.1 Запуск приложения

Используя Tarantool в качестве сервера приложений, вы можете написать собственные приложения. Собственный язык Tarantool'а для приложений – Lua, поэтому типовое приложение представляет собой файл, который содержит Lua-скрипт. Однако вы также можете писать приложения на C или C++.

Примечание: Если вы только осваиваете Lua, рекомендуем выполнить практическое задание по Tarantool'у до работы с данной главой. Для запуска практического задания, выполните команду `tutorial()` в консоли Tarantool'а:

```
tarantool> tutorial()
---
- |
 Tutorial -- Screen #1 -- Hello, Moon
 ====================================

 Welcome to the Tarantool tutorial.
 It will introduce you to Tarantool's Lua application server
 and database server, which is what's running what you're seeing.
 This is INTERACTIVE -- you're expected to enter requests
 based on the suggestions or examples in the screen's text.
 <...>
```

Создадим и запустим первое приложение на языке Lua для Tarantool'а – самое простое приложение, старую добрую программу «Hello, world!»:

```
#!/usr/bin/env tarantool
print('Hello, world!')
```

Сохраним приложение в файле. Пусть это будет `myapp.lua` в текущей директории.

Теперь рассмотрим, как можно запустить наше приложение с Tarantool'ом.

Запуск в Docker

Если мы запустим Tarantool в *Docker-контейнере*, Tarantool начнет работу без какого-либо приложения после следующей команды:

```
$ # создать временный контейнер и запустить его в интерактивном режиме
$ docker run --rm -t -i tarantool/tarantool:1
```

Чтобы запустить Tarantool с нашим приложением, можно выполнить команду:

```
$ # создать временный контейнер и
$ # запустить Tarantool с нашим приложением
$ docker run --rm -t -i \
             -v `pwd`/myapp.lua:/opt/tarantool/myapp.lua \
             -v /data/dir/on/host:/var/lib/tarantool \
             tarantool/tarantool:1 tarantool /opt/tarantool/myapp.lua
```

Здесь два ресурса подключаются к серверу в контейнере:

- наш файл с приложением (myapp.lua) и
- каталог данных Tarantool'а (`/data/dir/on/host`).

Традиционно в контейнере директория `/opt/tarantool` используется для кода приложения Tarantool'а, а директория `/var/lib/tarantool` используется для данных.

Запуск бинарной программы

При запуске Tarantool'а из *пакета* или при *сборке из исходников*, можно запустить наше приложение:

- в режиме скрипта,
- как серверное приложение или

- как демон службы.

Самый простой способ – передать имя файла в Tarantool при запуске:

```
$ tarantool myapp.lua
Hello, world!
$
```

Tarantool запускается, выполняет наш скрипт в **режиме скрипта** и завершает работу.

Теперь превратим этот скрипт в **серверное приложение**. Используем *box.cfg* из встроенного в Tarantool Lua-модуля, чтобы:

- запустить базу данных (данные в базе находятся в персистентном состоянии на диске, которое следует восстановить после запуска приложения) и
- настроить Tarantool как сервер, который принимает запросы по TCP-порту.

Также добавим простую логику для базы данных, используя *space.create()* и *create_index()* для создания спейса с первичным индексом. Используем функцию *box.once()*, чтобы обеспечить единовременное выполнение логики после первоначальной инициализации базы данных, поскольку мы не хотим создавать уже существующий спейс или индекс при каждом обращении к скрипту:

```
#!/usr/bin/env tarantool
-- настроить базу данных
box.cfg {
   listen = 3301
}
box.once("bootstrap", function()
   box.schema.space.create('tweedledum')
   box.space.tweedledum:create_index('primary',
       { type = 'TREE', parts = {1, 'unsigned'}})
end)
```

Далее запустим наше приложение, как делали ранее:

```
$ tarantool myapp.lua
Hello, world!
2017-08-11 16:07:14.250 [41436] main/101/myapp.lua C> version 2.1.0-429-g4e5231702
2017-08-11 16:07:14.250 [41436] main/101/myapp.lua C> log level 5
2017-08-11 16:07:14.251 [41436] main/101/myapp.lua I> mapping 1073741824 bytes for tuple arena...
2017-08-11 16:07:14.255 [41436] main/101/myapp.lua I> recovery start
2017-08-11 16:07:14.255 [41436] main/101/myapp.lua I> recovering from `./00000000000000000000.snap'
2017-08-11 16:07:14.271 [41436] main/101/myapp.lua I> recover from `./00000000000000000000.xlog'
2017-08-11 16:07:14.271 [41436] main/101/myapp.lua I> done `./00000000000000000000.xlog'
2017-08-11 16:07:14.272 [41436] main/102/hot_standby I> recover from `./00000000000000000000.xlog'
2017-08-11 16:07:14.274 [41436] iproto/102/iproto I> binary: started
2017-08-11 16:07:14.275 [41436] iproto/102/iproto I> binary: bound to [::]:3301
2017-08-11 16:07:14.275 [41436] main/101/myapp.lua I> done `./00000000000000000000.xlog'
2017-08-11 16:07:14.278 [41436] main/101/myapp.lua I> ready to accept requests
```

На этот раз Tarantool выполняет скрипт и продолжает работать в качестве сервера, принимая TCP-запросы на порт 3301. Можно увидеть Tarantool в списке процессов текущей сессии:

```
$ ps | grep "tarantool"
  PID TTY           TIME CMD
41608 ttys001       0:00.47 tarantool myapp.lua <running>
```

Однако экземпляр Tarantool'а завершит работу, если мы закроем окно командной строки. Чтобы отделить Tarantool и приложение от окна командной строки, можно запустить **режим демона**. Для этого

добавим некоторые параметры в `box.cfg{}`:

- *background* = `true`, который собственно заставит Tarantool работать в качестве демона,
- *log* = `'dir-name'`, который укажет, где демон Tarantool'а будет сохранять файл журнала (другие настройки журнала находятся в модуле Tarantool'а *log* module), а также
- *pid_file* = `'file-name'`, который укажет, где демон Tarantool'а будет сохранять файл журнала pid-файл.

Например:

```
box.cfg {
   listen = 3301,
   background = true,
   log = '1.log',
   pid_file = '1.pid'
}
```

Запустим наше приложение, как делали ранее:

```
$ tarantool myapp.lua
Hello, world!
$
```

Tarantool выполняет наш скрипт, отделяется от текущей сессии (он не отображается при вводе `ps | grep "tarantool"`) и продолжает работать в фоновом режиме в качестве демона, прикрепленного к общей сессии (с SID = 0):

```
$ ps -ef | grep "tarantool"
  PID SID     TIME  CMD
42178   0  0:00.72 tarantool myapp.lua <running>
```

Рассмотрев создание и запуск Lua-приложения для Tarantool'а, перейдем к углубленному изложению методик программирования.

4.4.2 Создание приложения

Далее мы пошагово разберем ключевые методики программирования, что послужит хорошим началом для написания Lua-приложений для Tarantool'а. Для интереса возьмем историю реализации... настоящего микросервиса на основе Tarantool'а! Мы реализуем бэкенд для упрощенной версии Pokémon Go, игры на основе определения местоположения дополненной реальности, выпущенной в середине 2016 года. В этой игре игроки используют GPS-возможности мобильных устройств, чтобы находить, захватывать, сражаться и тренировать виртуальных существ, или покемонов, которые появляются на экране, как если бы они находились в том же реальном месте, как и игрок.

Чтобы не выходить за рамки пошагового примера, ограничим оригинальный сюжет игры. У нас есть карта с местами появления покемонов. Далее у нас есть несколько игроков, которые могут отправлять запросы на поимку покемона на сервер (где работает микросервис Tarantool'а). Сервер отвечает, пойман ли покемон, увеличивает счетчик покемонов, если пойман, и вызывает метод респауна покемона, который через некоторое время создает нового покемона на том же самом месте.

Мы вынесем клиентские приложения за рамки рассказа. Но в конце обещаем небольшую демонстрацию с моделированием настоящих пользователей, чтобы немного поразвлечься. :-)

* * *

Для начала как лучше всего предоставить микросервис?

Модули и приложения

Чтобы наша логическая схема игры была доступна другим разработчикам и Lua-приложениям, поместим ее в Lua-модуль.

Модуль (который называется «rock» в Lua) – это дополнительная библиотека, которая расширяет функции Tarantool'а. Поэтому можно установить нашу логическую схему в виде модуля в Tarantool и использовать ее из любого Tarantool-приложения или модуля. Как и приложения, модули в Tarantool'е могут быть написаны на Lua (rocks), C или C++.

Модули хороши для двух целей:

- облегченное **управление кодом** (переиспользование, подготовка к развертыванию, версионирование) и
- горячая **перезагрузка кода** без перезапуска экземпляра Tarantool'а.

В техническом смысле, модуль - это файл с исходным кодом, который экспортирует свои функции в API. Например, вот Lua-модуль под названием `mymodule.lua`, который экспортирует одну функцию под названием `myfun`:

```
local exports = {}
exports.myfun = function(input_string)
   print('Hello', input_string)
end
return exports
```

Чтобы запустить функцию `myfun()` – из другого модуля, из Lua-приложения или из самого Tarantool'а – необходимо сохранить этот модуль в виде файла, а затем загрузить этот модуль с директивой `require()` и вызвать экспортированную функцию.

Например, вот Lua-приложение, которое использует функцию `myfun()` из модуля `mymodule.lua`:

```
-- загрузка модуля
local mymodule = require('mymodule')

-- вызов myfun() из функции test
local test = function()
  mymodule.myfun()
end
```

Здесь важно запомнить, что директива `require()` берет пути загрузки к Lua-модулям из переменной `package.path`. Она представляет собой строку с разделителями в виде точки с запятой, где знак вопроса используется для вставки имени модуля. По умолчанию, эта переменная содержит пути в системе и рабочую директорию. Но если мы поместим наши модули в особую папку (например, `scripts/`), необходимо будет добавить эту папку в `package.path` до вызова `require()`:

```
package.path = 'scripts/?.lua;' .. package.path
```

Для нашего микросервиса простым и удобным решением будет разместить все методы в Lua-модуле (скажем, `pokemon.lua`) и написать Lua-приложение (скажем, `game.lua`), которое запустит игровое окружение и цикл игры.

* * *

Теперь приступим к деталям реализации. В игре нам необходимы три сущности:

- **карта**, которая представляет собой массив покемонов с координатами мест респауна; в данной версии игры пусть местом будет прямоугольник, установленный по двум точкам, верхней левой и нижней правой;
- **игрок**, у которого есть ID, имя и координаты местонахождения игрока;
- **покемон**, у которого такие же поля, как и у игрока, плюс статус (активный/неактивный, то есть находится ли на карте) и возможность поимки (давайте уж дадим нашим покемонам шанс сбежать :-))

Эти данные будем хранить как кортежи в спейсах Tarantool'а. Но чтобы бэкенд-приложение работало как микросервис, правильно будет отправлять/получать данные в универсальном формате JSON, используя Tarantool в качестве системы хранения документов.

Avro-схемы

Чтобы хранить JSON-данные в виде кортежей, используем продвинутую методику, которая уменьшит отпечаток данных и обеспечит пригодность всех сохраняемых документов. Будем использовать Tarantool-модуль avro-schema, который проверяет схему JSON-документа и конвертирует его в кортеж Tarantool'а. Кортеж будет содержать только значения полей, таким образом, занимая меньше места, чем оригинальный документ. С точки зрения avro-схемы, конвертация JSON-документов в кортежи – «flattening» (конвертация в плоские файлы), а восстановление оригинальных документов – «unflattening» (конвертация из плоских файлов).

Для начала необходимо установить модуль с помощью команды `tarantoolctl rocks install avro-schema`.

Использовать модуль достаточно просто:

(1) Для каждой сущности необходимо определить схему в синтаксисе схемы Apache Avro, где мы перечисляем поля сущности с их наименованиями и типами данных по Avro.

(2) При инициализации мы вызываем функцию `avro-schema.create()`, которая создает объекты в памяти для всех сущностей схемы, а также функцию `compile()`, которая создает методы flatten/unflatten (конвертация в плоские файлы и обратно) для каждой сущности.

(3) Далее мы просто вызываем методы flatten/unflatten для соответствующей сущности при получении/отправке данных об этой сущности.

Вот как будут выглядеть определения схемы для сущностей игрока и покемона:

```
local schema = {
    player = {
        type="record",
        name="player_schema",
        fields={
            {name="id", type="long"},
            {name="name", type="string"},
            {
                name="location",
                type= {
                    type="record",
                    name="player_location",
                    fields={
                        {name="x", type="double"},
                        {name="y", type="double"}
                    }
                }
            }
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        }
    },
    pokemon = {
        type="record",
        name="pokemon_schema",
        fields={
            {name="id", type="long"},
            {name="status", type="string"},
            {name="name", type="string"},
            {name="chance", type="double"},
            {
                name="location",
                type= {
                    type="record",
                    name="pokemon_location",
                    fields={
                        {name="x", type="double"},
                        {name="y", type="double"}
                    }
                }
            }
        }
    }
}
```

А вот как мы создадим и скомпилируем наши сущности при инициализации:

```
-- загрузить модуль avro-schema с директивой require()
local avro = require('avro_schema')

-- создать модели
local ok_m, pokemon = avro.create(schema.pokemon)
local ok_p, player = avro.create(schema.player)
if ok_m and ok_p then
    -- скомпилировать модели
    local ok_cm, compiled_pokemon = avro.compile(pokemon)
    local ok_cp, compiled_player = avro.compile(player)
    if ok_cm and ok_cp then
        -- начать игру
        <...>
    else
        log.error('Schema compilation failed')
    end
else
    log.info('Schema creation failed')
end
return false
```

Что касается сущности карты, вводить для нее схему будет перебор, потому что в игре всего одна карта, у нее мало полей, и – что самое главное – мы используем карту только внутри нашей логики, не показывая ее внешним пользователям.

* * *

Далее нам нужны методы для реализации игровой логики. Чтобы смоделировать объектно-ориентированное программирование в нашем Lua-коде, будем хранить все Lua-функции и общие пере-

менные в одной внутренней переменной (назовем ее game). Это позволит нам обращаться к функциям или переменным из нашего модуля с помощью self.func_name или self.var_name следующим образом:

```
local game = {
    -- локальная переменная
    num_players = 0,

    -- метод, который выводит локальную переменную
    hello = function(self)
      print('Hello! Your player number is ' .. self.num_players .. '.')
    end,

    -- метод, который вызывает другой метод и возвращает локальную переменную
    sign_in = function(self)
      self.num_players = self.num_players + 1
      self:hello()
      return self.num_players
    end
}
```

В терминах ООП сейчас мы можем рассматривать внутренние переменные внутри переменной game как поля объекта, а внутренние функции – как методы объекта.

Примечание: Обратите внимание, что в текущей документации в примерах Lua-кода используются *локальные* переменные. Используйте *глобальные* переменные аккуратно, поскольку пользователи ваших модулей могут не знать об этих переменных.

Чтобы включить/отключить использование необъявленных глобальных переменных в вашем коде на языке Lua, используйте модуль Tarantool'а *strict*.

Таким образом, в модуле игры будут следующие методы:

- catch() (поймать) для расчета, когда был пойман покемон (помимо координат как игрока, так и покемона, этот метод будет использовать коэффициент вероятности, чтобы в пределах досягаемости игрока можно было поймать не каждого покемона);
- respawn() (респаун) для добавления отсутствующих покемонов на карту, скажем, каждые 60 секунд (предположим, что испуганный покемон убегает, поэтому мы убираем покемона с карты при любой попытке поймать его и через некоторое время добавляем обратно на карту);
- notify() (уведомить) для записи информации о пойманных покемонах (например, «Игрок 1 поймал покемона A»);
- start() (начать) для инициализации игры (метод создаст спейсы в базе данных, создаст и скомпилирует avro-схемы, а также запустит метод respawn()).

Кроме того, было бы удобно завести методы для работы с хранилищем Tarantool'а. Например:

- add_pokemon() (добавить покемона) для добавления покемона в базу данных и
- map() (карта) для заполнения карты всеми покемонами, которые хранятся в Tarantool'е.

Эти два метода будут главным образом использоваться во время инициализации нашей игры, но их также можно вызывать позднее, например для тестирования кода.

Настройка базы данных

Обсудим инициализацию игры. В методе `start()` нам нужно заполнить спейсы Tarantool'а данными о покемонах. Почему бы не хранить все игровые данные в памяти? Зачем нужна база данных? Ответ на это: *персистентность*. Без базы данных мы рискуем потерять данные при отключении электроэнергии, например. Но если мы храним данные в in-memory базе данных, Tarantool позаботится о том, чтобы обеспечить постоянное хранение данных при их изменении. Это дает дополнительное преимущество: быстрая загрузка в случае отказа. *Умный алгоритм* Tarantool'а быстро загружает все данные с диска в память при начале работы, так что подготовка к работе не займет много времени.

Мы будем использовать функции из встроенного модуля Tarantool'а *box*:

- `box.schema.create_space('pokemons')` для создания спейса под названием `pokemon` (покемон), чтобы хранить информацию о покемонах (мы не создаем аналогичный спейс по игрокам, потому что планируем только отправлять и получать информацию об игроках с помощью вызовов API, так что нет необходимости хранить ее);
- `box.space.pokemons:create_index('primary', {type = 'hash', parts = {1, 'unsigned'}})` для создания первичного HASH-индекса по ID покемона;
- `box.space.pokemons:create_index('status', {type = 'tree', parts = {2, 'str'}})` для создания вторичного TREE-индекса по статусу покемона.

Обратите внимание на аргумент `parts =` в спецификации индекса. ID покемона – это первое поле в кортеже Tarantool'а, потому что это первый элемент соответствующего типа Avro. То же относится к статусу покемона. В самом JSON-файле поля ID или статуса могут быть в любом положении на JSON-карте.

Реализация метода `start()` выглядит следующим образом:

```
-- создать игровой объект
start = function(self)
    -- создать спейсы и индексы
    box.once('init', function()
        box.schema.create_space('pokemons')
        box.space.pokemons:create_index(
            "primary", {type = 'hash', parts = {1, 'unsigned'}}
        )
        box.space.pokemons:create_index(
            "status", {type = "tree", parts = {2, 'str'}}
        )
    end)

    -- создать модели
    local ok_m, pokemon = avro.create(schema.pokemon)
    local ok_p, player = avro.create(schema.player)
    if ok_m and ok_p then
        -- скомпилировать модели
        local ok_cm, compiled_pokemon = avro.compile(pokemon)
        local ok_cp, compiled_player = avro.compile(player)
        if ok_cm and ok_cp then
            -- начать игру
            <...>
        else
            log.error('Schema compilation failed')
        end
    else
        log.info('Schema creation failed')
    end
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    return false
end
```

ГИС

Теперь обсудим метод `catch()`, который является основным в логике нашей игры.

Здесь мы получаем координаты игрока и номер ID искомого покемона, а нужен нам ответ на вопрос, поймал ли игрок покемона (помните, что у каждого покемона есть шанс убежать).

Для начала проверим полученные данные об игроке по *Avro-схеме*. Также проверим, есть ли такой покемон в базе данных, и отображается ли он на карте (у покемона должен быть активный статус):

```
catch = function(self, pokemon_id, player)
    -- проверить данные игрока
    local ok, tuple = self.player_model.flatten(player)
    if not ok then
        return false
    end
    -- получить данные покемона
    local p_tuple = box.space.pokemons:get(pokemon_id)
    if p_tuple == nil then
        return false
    end
    local ok, pokemon = self.pokemon_model.unflatten(p_tuple)
    if not ok then
        return false
    end
    if pokemon.status ~= self.state.ACTIVE then
        return false
    end
    -- логика поимки будет дополняться
    <...>
end
```

Далее вычисляем ответ: пойман или нет.

Чтобы работать с географическими координатами, используем модуль Tarantool'а gis.

Чтобы не усложнять, не будем загружать какую-то особую карту, допуская, что рассматриваем карту мира. Также не будет проверять поступающие координаты, снова допуская, что все места находятся на планете Земля.

Используем две географические переменные:

- `wgs84`, что означает последнюю редакцию стандарта Мировой геодезической системы координат, WGS84. В целом, она представляет собой стандартную систему координат Земли и изображает Землю как эллипсоид.
- `nationalmap`, что означает Государственный атлас США в равновеликой проекции (US National Atlas Equal Area). Это система спроецированных координат на основании WGS84. Она дает основу для проецирования мест и позволяет определить местоположение наших игроков и покемонов в метрах.

Обе системы указаны в Реестре геодезических параметров EPSG, где каждой системе присвоен уникальный номер. Мы назначим эти числа соответствующим переменным в нашем коде:

```
wgs84 = 4326,
nationalmap = 2163,
```

Для игровой логики необходима еще одна переменная `catch_distance`, которая определяет, насколько близко игрок должен подойти к покемону, чтобы попытаться поймать его. Определим это расстояние в 100 метров.

```
catch_distance = 100,
```

Теперь можно рассчитать ответ. Необходимо спроецировать текущее местоположение как игрока (`p_pos`), так и покемона (`m_pos`) на карте, проверить, достаточно ли близко к покемону находится игрок (с помощью `catch_distance`), и рассчитать, поймал ли игрок покемона (здесь мы генерируем случайное значение, и покемон убегает, если случайное значение оказывается меньше, чем 100 минус случайная величина покемона):

```
-- спроецировать местоположение
 local m_pos = gis.Point(
     {pokemon.location.x, pokemon.location.y}, self.wgs84
 ):transform(self.nationalmap)
 local p_pos = gis.Point(
     {player.location.x, player.location.y}, self.wgs84
 ):transform(self.nationalmap)

 -- проверить условие близости игрока
 if p_pos:distance(m_pos) > self.catch_distance then
     return false
 end
 -- попытаться поймать покемона
 local caught = math.random(100) >= 100 - pokemon.chance
 if caught then
     -- обновить и сообщить об успехе
     box.space.pokemons:update(
         pokemon_id, {{'=', self.STATUS, self.state.CAUGHT}}
     )
     self:notify(player, pokemon)
 end
 return caught
```

Итератор с индексом

По сюжету игры все пойманные покемоны возвращаются на карту. Метод `respawn()` обеспечивает это для всех покемонов на карте каждые 60 секунд. Мы выполняем перебор покемонов по статусу с помощью функции Tarantool'а итератора с индексом *index:pairs* и сбрасываем статусы всех «пойманных» покемонов обратно на «активный» с помощью `box.space.pokemons:update()`.

```
respawn = function(self)
    fiber.name('Respawn fiber')
    for _, tuple in box.space.pokemons.index.status:pairs(
            self.state.CAUGHT) do
        box.space.pokemons:update(
            tuple[self.ID],
            {{'=', self.STATUS, self.state.ACTIVE}}
        )
    end
 end
```

Для удобства введем именованные поля:

ID = 1, STATUS = 2,

Реализация метода `start()` полностью теперь выглядит так:

```
-- создать игровой объект
start = function(self)
    -- создать спейсы и индексы
    box.once('init', function()
       box.schema.create_space('pokemons')
       box.space.pokemons:create_index(
           "primary", {type = 'hash', parts = {1, 'unsigned'}}
       )
       box.space.pokemons:create_index(
           "status", {type = "tree", parts = {2, 'str'}}
       )
    end)

    -- создать модели
    local ok_m, pokemon = avro.create(schema.pokemon)
    local ok_p, player = avro.create(schema.player)
    if ok_m and ok_p then
        -- скомпилировать модели
        local ok_cm, compiled_pokemon = avro.compile(pokemon)
        local ok_cp, compiled_player = avro.compile(player)
        if ok_cm and ok_cp then
            -- начать игру
            self.pokemon_model = compiled_pokemon
            self.player_model = compiled_player
            self.respawn()
            log.info('Started')
            return true
         else
            log.error('Schema compilation failed')
         end
    else
        log.info('Schema creation failed')
    end
    return false
end
```

Файберы

Но подождите! Если мы запустим функцию `self.respawn()`, как показано выше, то она запустится только один раз, как и остальные методы. А нам необходимо запускать `respawn()` каждые 60 секунд. Tarantool заставляет логику приложения непрерывно работать в фоновом режиме с помощью *файбера*.

Файбер предназначен для выполнения последовательностей команд, но это не поток. Ключевое отличие в том, что потоки используют многозадачность с реализацией приоритетов, тогда как файберы используют кооперативную многозадачность. Это дает файберам два преимущества над потоками:

- Улучшенная управляемость. Потоки часто зависят от планировщика потока ядра в вопросе вытеснения занятого потока и возобновления другого потока, поэтому вытеснение может быть непредвиденным. Файберы передают управление самостоятельно другому файберу во время работы, поэтому управление файберами осуществляется логикой приложения.
- Повышенная производительность. Потокам необходимо больше ресурсов для вытеснения, по-

скольку они обращаются к ядру системы. Файберы легче и быстрее, поскольку для передачи управления им не нужно обращаться к ядру.

Однако у файберов есть определенные ограничения, по сравнению с потоками, основное из которых – отсутствие режима работы с многоядерной системой. Все файберы в приложении относятся к одному потоку, поэтому они используют то же ядро процессора, что и родительский поток. В то же время, это ограничение незначительно для приложений Tarantool'а, поскольку узкое место Tarantool'а – жесткий диск, а не ЦП.

У файбера есть все возможности сопрограммы на языке Lua, и все принципы программирования, которые применяются к сопрограммам на Lua, применимы и к файберам. Однако Tarantool расширил возможности файберов для внутреннего использования. Поэтому, несмотря на возможность и поддержку использования сопрограмм, рекомендуется использовать файберы.

Производительность или управляемость не слишком важны в нашем случае. Запустим `respawn()` в файбере для непрерывной работы в фоновом режиме. Для этого необходимо изменить `respawn()`:

```
respawn = function(self)
    -- назовем наш файбер;
    -- это выполнит чистый вывод в fiber.info()
    fiber.name('Respawn fiber')
    while true do
        for _, tuple in box.space.pokemons.index.status:pairs(
                self.state.CAUGHT) do
            box.space.pokemons:update(
                tuple[self.ID],
                {{'=', self.STATUS, self.state.ACTIVE}}
            )
        end
        fiber.sleep(self.respawn_time)
    end
end
```

и назвать его файбером в `start()`:

```
start = function(self)
    -- создать спейсы и индексы
        <...>
    -- создать модели
        <...>
    -- скомпилировать модели
        <...>
    -- начать игру
       self.pokemon_model = compiled_pokemon
       self.player_model = compiled_player
       fiber.create(self.respawn, self)
       log.info('Started')
    -- ошибки, если создание схемы или компиляция не работает
       <...>
end
```

Запись в журнал

В `start()` мы использовали еще одну полезную функцию – `log.info()` из *модуля log* Tarantool'а . Эта функция также понадобится в `notify()` для добавления записи в файл журнала при каждой успешной поимке:

```
-- уведомление о событии
notify = function(self, player, pokemon)
    log.info("Player '%s' caught '%s'", player.name, pokemon.name)
end
```

Мы используем стандартные *настройки журнала* Tarantool'а, поэтому увидим вывод записей журнала в консоли, когда запустим приложение в режиме скрипта.

* * *

Отлично! Мы обсудили все методики программирования, используемые в нашем Lua-модуле (см. pokemon.lua).

Теперь подготовим среду тестирования. Как и планировалось, напишем приложение на языке Lua (см. game.lua), чтобы инициализировать модуль базы данных Tarantool'а, инициализировать нашу игру, вызвать цикл игры и смоделировать пару запросов от игроков.

Чтобы запустить микросервис, поместим модуль `pokemon.lua` и приложение `game.lua` в текущую директорию, установим все внешние модули и запустим экземпляр Tarantool'а с работают приложением `game.lua` (это пример для Ubuntu):

```
$ ls
game.lua  pokemon.lua
$ sudo apt-get install tarantool-gis
$ sudo apt-get install tarantool-avro-schema
$ tarantool game.lua
```

Tarantool запускает и инициализирует базу данных. Затем Tarantool выполняет демо-логику из `game.lua`: добавляет покемона под названием Пикачу (Pikachu) (шанс его поимки очень высок – 99,1), отображает текущую карту (на ней расположен один активный покемон, Пикачу) и обрабатывает запросы поимки от двух игроков. Player1 (Игрок 1) находится очень близко к одинокому покемону Пикачу, а Player2 (Игрок 2) находится очень далеко от него. Как предполагается, результаты поимки в таком выводе будут «true» для Player1 и «false» для Player2. Наконец, Tarantool отображает текущую карту, которая пуста, потому что Пикачу пойман и временно неактивен:

```
$ tarantool game.lua
2017-01-09 20:19:24.605 [6282] main/101/game.lua C> version 1.7.3-43-gf5fa1e1
2017-01-09 20:19:24.605 [6282] main/101/game.lua C> log level 5
2017-01-09 20:19:24.605 [6282] main/101/game.lua I> mapping 1073741824 bytes for tuple arena...
2017-01-09 20:19:24.609 [6282] main/101/game.lua I> initializing an empty data directory
2017-01-09 20:19:24.634 [6282] snapshot/101/main I> saving snapshot `./00000000000000000000.snap.
↪inprogress'
2017-01-09 20:19:24.635 [6282] snapshot/101/main I> done
2017-01-09 20:19:24.641 [6282] main/101/game.lua I> ready to accept requests
2017-01-09 20:19:24.786 [6282] main/101/game.lua I> Started
---
- {'id': 1, 'status': 'active', 'location': {'y': 2, 'x': 1}, 'name': 'Pikachu', 'chance': 99.1}
...

2017-01-09 20:19:24.789 [6282] main/101/game.lua I> Player 'Player1' caught 'Pikachu'
true
false
--- []
...

2017-01-09 20:19:24.789 [6282] main C> entering the event loop
```

nginx

В реальной жизни такой микросервис работал бы по HTTP. Добавим веб-сервер nginx в нашу среду и сделаем аналогичный пример. Но как вызывать методы Tarantool'а с помощью REST API? Мы используем nginx с модулем Tarantool nginx upstream и создадим еще один скрипт на Lua (app.lua), который экспортирует три наших игровых метода – `add_pokemon()`, `map()` и `catch()` – в качестве конечных точек обработки запросов REST модуля nginx upstream:

```
local game = require('pokemon')
box.cfg{listen=3301}
game:start()

-- функции add, map и catch по REST API
function add(request, pokemon)
    return {
        result=game:add_pokemon(pokemon)
    }
end

function map(request)
    return {
        map=game:map()
    }
end

function catch(request, pid, player)
    local id = tonumber(pid)
    if id == nil then
        return {result=false}
    end
    return {
        result=game:catch(id, player)
    }
end
```

Чтобы с легкостью настроить и запустить nginx, необходимо создать Docker-контейнер на основе Docker-образа с уже установленными nginx и модулем upstream (см. http/Dockerfile). Берем стандартный nginx.conf, где определяем upstream с работающим бэкендом Tarantool'а (это еще один Docker-контейнер, см. нижеприведенную информацию):

```
upstream tnt {
      server pserver:3301 max_fails=1 fail_timeout=60s;
      keepalive 250000;
}
```

и добавляем специальные параметры для Tarantool'а (см. описание в файле README модуля upstream):

```
server {
  server_name tnt_test;

  listen 80 default deferred reuseport so_keepalive=on backlog=65535;

  location = / {
      root /usr/local/nginx/html;
  }
```

(continues on next page)

(продолжение с предыдущей страницы)

```
  location /api {
    # ответы проверяют бесконечное время ожидания
    tnt_read_timeout 60m;
    if ( $request_method = GET ) {
       tnt_method "map";
    }
    tnt_http_rest_methods get;
    tnt_http_methods all;
    tnt_multireturn_skip_count 2;
    tnt_pure_result on;
    tnt_pass_http_request on parse_args;
    tnt_pass tnt;
  }
}
```

Аналогичным образом, поместим Tarantool-сервер и всю игровую логику в другой Docker-контейнер на основе официального образа Tarantool'а 1.9 (см. src/Dockerfile) и установим `tarantool app.lua` в качестве стандартной команды для контейнера. Это бэкенд.

Неблокирующий ввод-вывод

Чтобы протестировать REST API, создадим новый скрипт (client.lua), который похож на наше приложение `game.lua`, но отправляет запросы HTTP POST и GET, а не вызывает Lua-функции:

```
local http = require('curl').http()
local json = require('json')
local URI = os.getenv('SERVER_URI')
local fiber = require('fiber')

local player1 = {
    name="Player1",
    id=1,
    location = {
        x=1.0001,
        y=2.0003
    }
}
local player2 = {
    name="Player2",
    id=2,
    location = {
        x=30.123,
        y=40.456
    }
}

local pokemon = {
    name="Pikachu",
    chance=99.1,
    id=1,
    status="active",
    location = {
        x=1,
        y=2
    }
```

(continues on next page)

(продолжение с предыдущей страницы)

```
}

function request(method, body, id)
    local resp = http:request(
        method, URI, body
    )
    if id ~= nil then
        print(string.format('Player %d result: %s',
            id, resp.body))
    else
        print(resp.body)
    end
end

local players = {}
function catch(player)
    fiber.sleep(math.random(5))
    print('Catch pokemon by player ' .. tostring(player.id))
    request(
        'POST', '{"method": "catch",
        "params": [1, '..json.encode(player)..']}',
        tostring(player.id)
    )
    table.insert(players, player.id)
end

print('Create pokemon')
request('POST', '{"method": "add",
    "params": ['..json.encode(pokemon)..']}')
request('GET', '')

fiber.create(catch, player1)
fiber.create(catch, player2)

-- подождать игроков
while #players ~= 2 do
    fiber.sleep(0.001)
end

request('GET', '')
os.exit()
```

При запуске этого скрипта вы заметите, что у обоих игроков одинаковые шансы сделать первую попытку поимки покемона. В классическом Lua-скрипте сетевой вызов блокирует скрипт, пока он не будет выполнен, поэтому первым попытаться поймать может тот игрок, который раньше зашел в игру. В Tarantool'е оба игрока играют одновременно, поскольку все модули объединены в *кооперативной многозадачности* и используют неблокирующий ввод-вывод.

Действительно, когда Player1 посылает первый REST-вызов, скрипт не блокируется. Файбер, выполняющий функцию `catch()` от Player1, посылает неблокирующий вызов в операционную систему и передает управление на следующий файбер, которым оказывается файбер от Player2. Файбер от Player2 делает то же самое. Когда получен сетевой ответ, файбер от Player1 активируется с помощью кооперативного планировщика Tarantool'а и возобновляет работу. Все *модули* Tarantool'а используют неблокирующий ввод-вывод и интегрированы с кооперативным планировщиком Tarantool'а. Разработчикам модулей Tarantool предоставляет *API*.

Для HTTP-теста создадим третий контейнер на основе официального образа Tarantool'а 1.9 (см.

client/Dockerfile) установим `tarantool client.lua` в качестве стандартной команды для контейнера.

* * *

Чтобы запустить тест локально, скачайте наш проект покемон из GitHub и вызовите:

```
$ docker-compose build
$ docker-compose up
```

Docker Compose собирает и запускает все три контейнера: `pserver` (бэкенд Tarantool'a), `phttp` (nginx) и"pclient" (демо-клиент). ВЫ можете увидеть все сообщения журнала из всех этих контейнеров в консоли. pclient выведет, что сделал HTTP-запрос на создание покемона, два запроса на поимку покемона, запросил карту (пустая, поскольку покемон пойман и временно неактивен) и завершил работу:

```
pclient_1  | Create pokemon
<...>
pclient_1  | {"result":true}
pclient_1  | {"map":[{"id":1,"status":"active","location":{"y":2,"x":1},"name":"Pikachu","chance
↪":99.100000}]}
pclient_1  | Catch pokemon by player 2
pclient_1  | Catch pokemon by player 1
pclient_1  | Player 1 result: {"result":true}
pclient_1  | Player 2 result: {"result":false}
pclient_1  | {"map":[]}
pokemon_pclient_1 exited with code 0
```

Поздравляем! Вот мы и закончили наш пошаговый пример. Для дальнейшего изучения рекомендуем *установку* и *добавление* модуля.

См. также справочник по *модулям Tarantool'a* и *C API* и не пропустите наши *рекомендации по разработке на Lua*.

4.4.3 Установка модуля

Модули на Lua и C от разработчиков Tarantool'a и сторонних разработчиков доступны здесь:

- Tarantool modules repository (see *below*)
- Tarantool deb/rpm repositories (see *below*)

Установка модуля из репозитория

Для получения подробной информации см. README в репозитории tarantool/rocks.

Установка модуля из deb/rpm

Выполните следующие действия:

1. Установите Tarantool в соответствии с рекомендациями на странице загрузки.
2. Установите необходимый модуль. Найдите имя модуля на странице со сторонними библиотеками Tarantool'a и введите префикс «tarantool-» перед названием модуля во избежание неоднозначности:

```
$ # для Ubuntu/Debian:
$ sudo apt-get install tarantool-<module-name>

$ # для RHEL/CentOS/Amazon:
$ sudo yum install tarantool-<module-name>
```

Например, чтобы установить модуль shard на Ubuntu, введите:

```
$ sudo apt-get install tarantool-shard
```

Теперь можно:

- загружать любой модуль с помощью

```
tarantool> name = require('module-name')
```

например:

```
tarantool> shard = require('shard')
```

- локально находить установленные модули с помощью `package.path` (Lua) или `package.cpath` (C):

```
tarantool> package.path
---
- ./?.lua;./?/init.lua; /usr/local/share/tarantool/?.lua;/usr/local/share/
tarantool/?/init.lua;/usr/share/tarantool/?.lua;/usr/share/tarantool/?/ini
t.lua;/usr/local/share/lua/5.1/?.lua;/usr/local/share/lua/5.1/?/init.lua;/
usr/share/lua/5.1/?.lua;/usr/share/lua/5.1/?/init.lua;
...

tarantool> package.cpath
---
- ./?.so;/usr/local/lib/x86_64-linux-gnu/tarantool/?.so;/usr/lib/x86_64-li
nux-gnu/tarantool/?.so;/usr/local/lib/tarantool/?.so;/usr/local/lib/x86_64
-linux-gnu/lua/5.1/?.so;/usr/lib/x86_64-linux-gnu/lua/5.1/?.so;/usr/local/
lib/lua/5.1/?.so;
...
```

Примечание: Знаки вопроса стоят вместо имени модуля, которое было указано ранее при вызове `require('module-name')`.

4.4.4 Добавление собственного модуля

Мы уже обсуждали, *как создать простой модуль на языке Lua для локального использования*.Теперь давайте обсудим, как создать модуль более продвинутого уровня для Tarantool'а, а затем разместить его на странице модулей Tarantool'а <http://tarantool.org/rocks.html>‘_ и включить его в официальные образы Tarantool'а для Docker.

Чтобы помочь разработчикам, мы создали modulekit, набор шаблонов для создания Tarantool-модулей на Lua и C.

Примечание: Чтобы использовать `modulekit`, необходимо предварительно установить пакет `tarantool-dev`. Например, в Ubuntu выполните команду:

```
$ sudo apt-get install tarantool-dev
```

Добавление собственного модуля на Lua

Подробную информацию и примеры см. в README в ветке «luakit» репозитория tarantool/modulekit.

Добавление собственного модуля на C

В некоторых случаях может потребоваться создание Tarantool-модуля на C, а не на Lua, например, для работы со специальным оборудованием или низкоуровневыми системными интерфейсами.

Подробную информацию и примеры см. в README в ветке «ckit» репозитория tarantool/modulekit.

Примечание: Вы можете аналогичным образом создавать модули на C++ при условии, что в их коде не будут выбрасываться исключения.

4.4.5 Перезагрузка модуля

Любое приложение или модуль Tarantool'а можно перезагрузить с нулевым временем простоя.

Перезагрузка модуля на Lua

Ниже представлен пример, который иллюстрирует наиболее типичный случай – «обновление и перезагрузка».

Примечание: В этом примере используются рекомендованные *методики администрирования* на основании *файлов экземпляров* и утилиты *tarantoolctl*.

1. Обновите файлы приложения.

   Например, модуль в `/usr/share/tarantool/app.lua`:

```
local function start()
  -- начальная версия
  box.once("myapp:v1.0", function()
    box.schema.space.create("somedata")
    box.space.somedata:create_index("primary")
    ...
  end)

  -- код миграции с 1.0 на 1.1
  box.once("myapp:v1.1", function()
    box.space.somedata.index.primary:alter(...)
    ...
  end)

  -- код миграции с 1.1 на 1.2
  box.once("myapp:v1.2", function()
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    box.space.somedata.index.primary:alter(...)
    box.space.somedata:insert(...)
    ...
  end)
end

-- запустить файберы в фоновом режиме, если необходимо

local function stop()
  -- остановить все файберы, работающие в фоновом режиме, и очистить ресурсы
end

local function api_for_call(xxx)
  -- сделать что-то
end

return {
  start = start,
  stop = stop,
  api_for_call = api_for_call
}
```

2. Обновить *файл экземпляра*.

   Например, `/etc/tarantool/instances.enabled/my_app.lua`:

```
#!/usr/bin/env tarantool
--
-- пример горячей перезагрузки кода
--

box.cfg({listen = 3302})

-- ВНИМАНИЕ: правильно выполните разгрузку!
local app = package.loaded['app']
if app ~= nil then
  -- остановите старую версию приложения
  app.stop()
  -- разгрузите приложение
  package.loaded['app'] = nil
  -- разгрузите все зависимости
  package.loaded['somedep'] = nil
end

-- загрузите приложение
log.info('require app')
app = require('app')

-- запустите приложение
app.start({some app options controlled by sysadmins})
```

   Самое главное – правильно разгрузить приложение и его зависимости.

3. Вручную перезагрузите файл приложения.

   Например, используя `tarantoolctl`:

```
$ tarantoolctl eval my_app /etc/tarantool/instances.enabled/my_app.lua
```

Перезагрузка модуля на C

После компиляции новой версии модуля на C (библиотека общего пользования `*.so`), вызовите функцию *box.schema.func.reload(„module-name“)* из Lua-скрипта для перезагрузки модуля.

4.4.6 Разработка с IDE

Для разработки и отладки Lua-приложений для Tarantool'а можно использовать IntelliJ IDEA в качестве интегрированной среды разработки (IDE).

1. Загрузите и установите IDE с официального сайта.

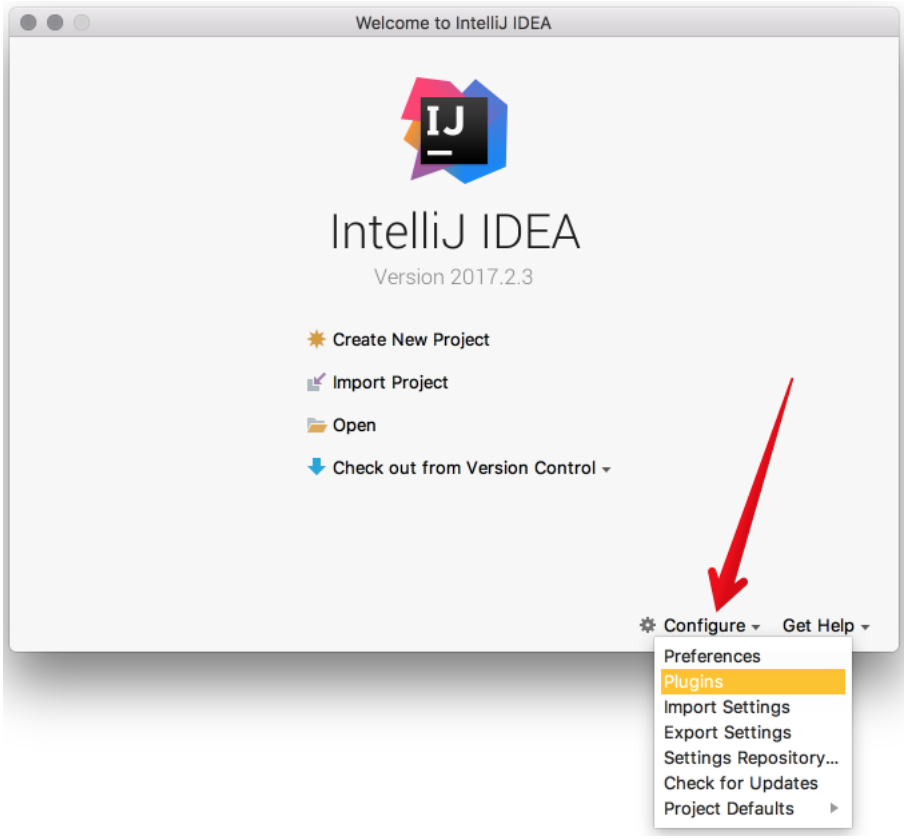
   JetBrains предоставляет специализированные версии для разных языков программирования: IntelliJ IDEA (Java), PHPStorm (PHP), PyCharm (Python), RubyMine (Ruby), CLion (C/C++), WebStorm (Web) и другие. Поэтому загрузите версию, которая подходит предпочитаемому языку.

   Для всех версий поддерживается интеграция с Tarantool'ом.

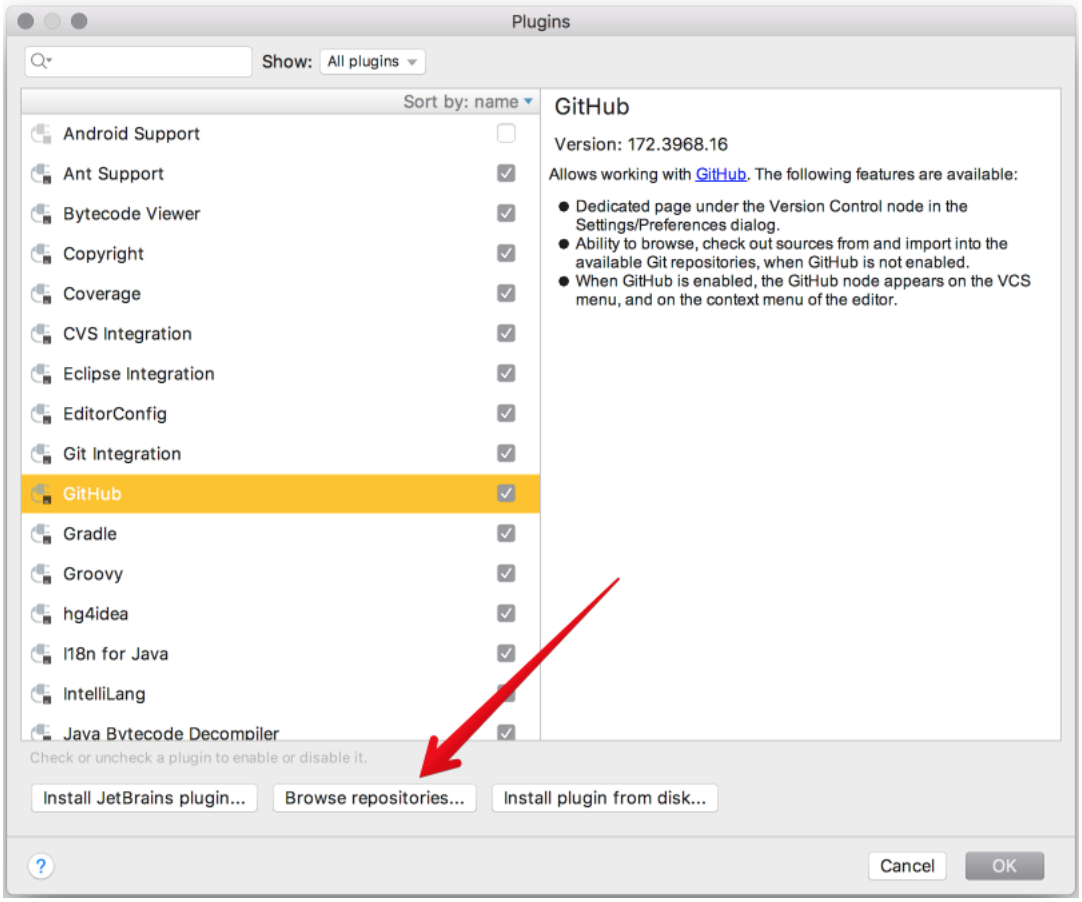
2. Настройте IDE:

   a. Запустите IntelliJ IDEA.

   b. Нажмите кнопку `Configure` и выберите `Plugins`.

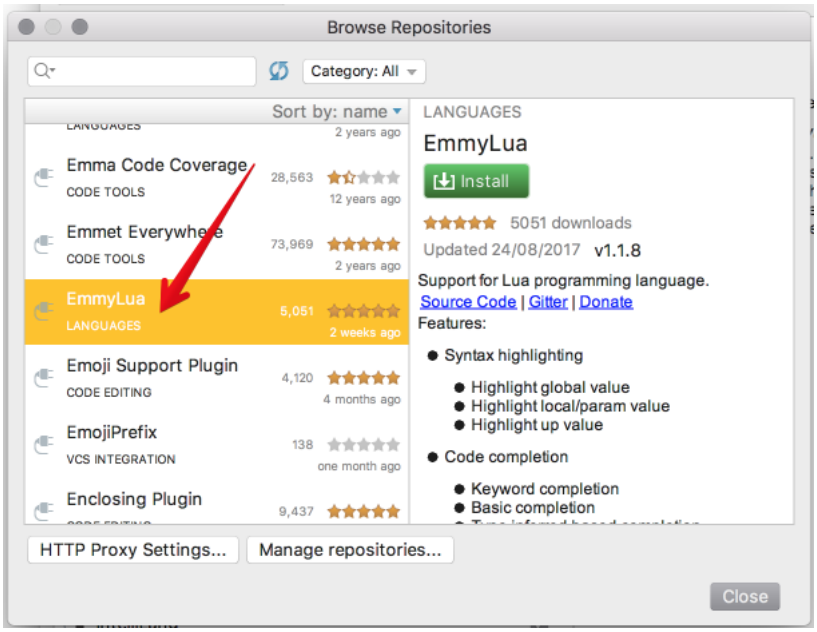


c. Нажмите `Browse repositories`.



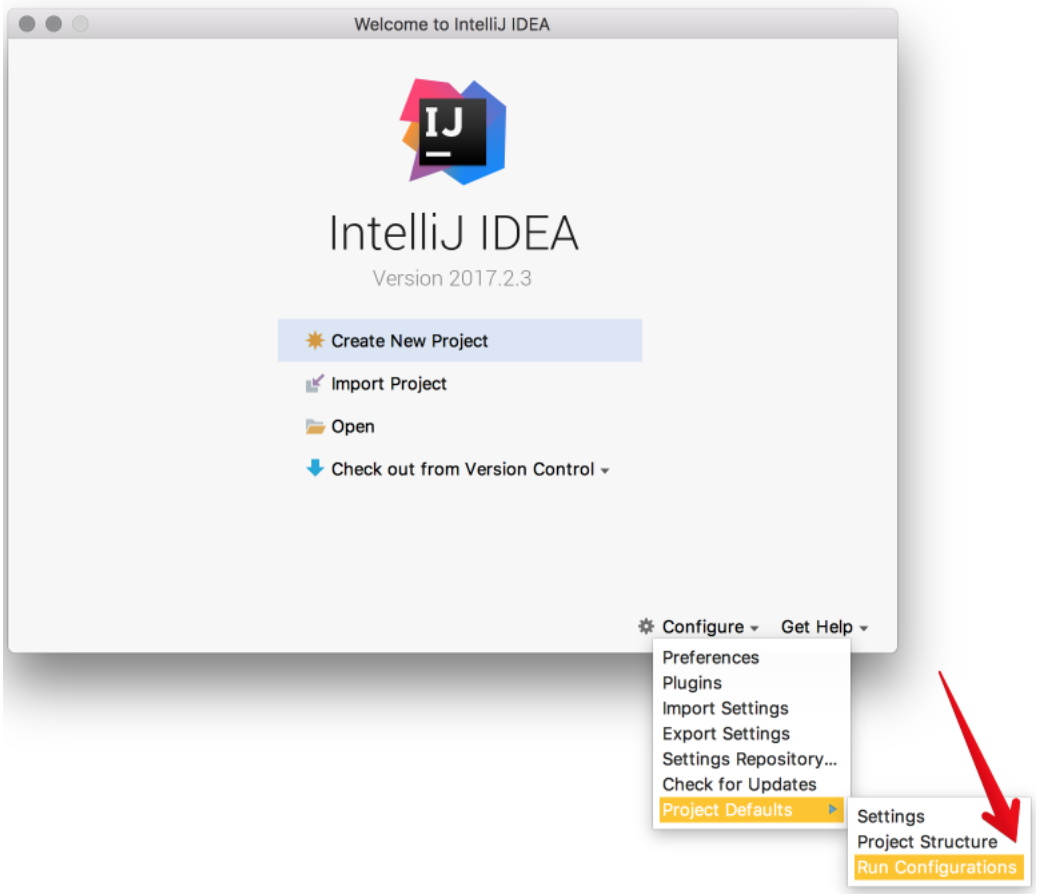
d. Установите плагин `EmmyLua`.

Примечание: Не путайте с плагином `Lua`, у которого меньше возможностей, чем у `EmmyLua`.



e. Перезапустите IntelliJ IDEA.

f. Нажмите `Configure`, выберите `Project Defaults`, а затем `Run Configurations`.

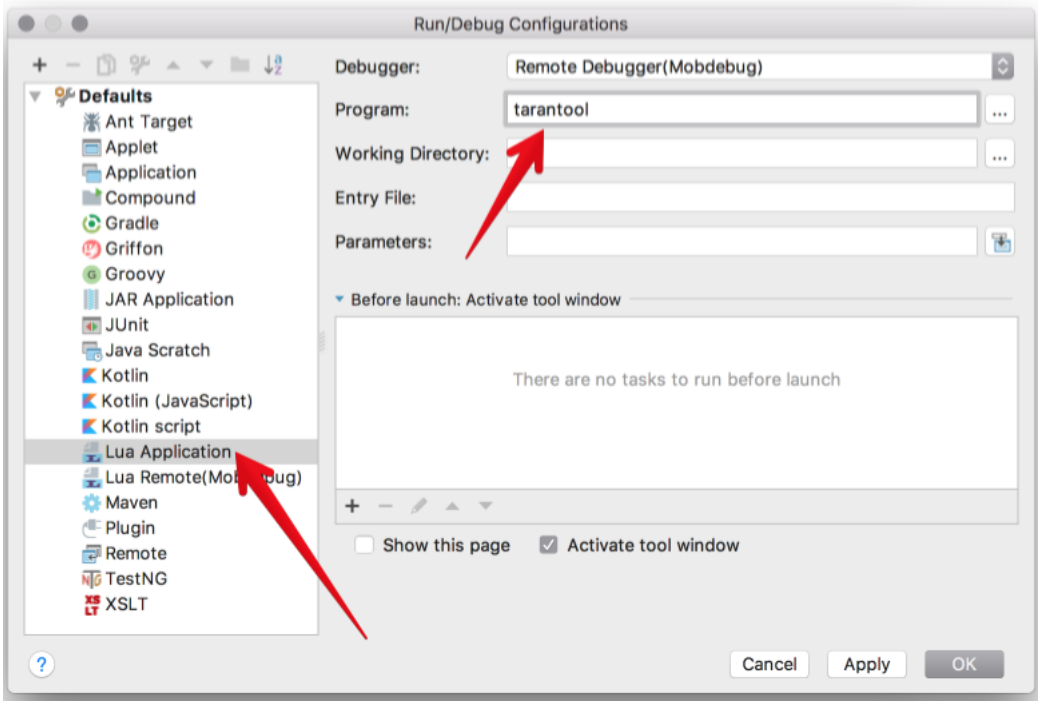


g. Найдите `Lua Application` в боковой панели слева.

h. В `Program` введите путь к установленному бинарному файлу `tarantool`.

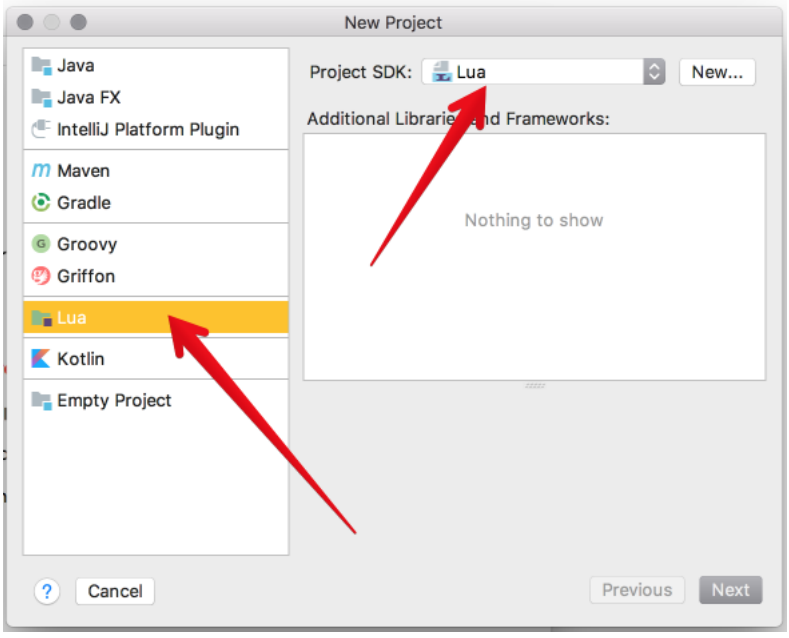
По умолчанию, это `tarantool` или `/usr/bin/tarantool` на большинстве платформ.

Если вы установили `tarantool` из источников в другую директорию, укажите здесь правильный путь.

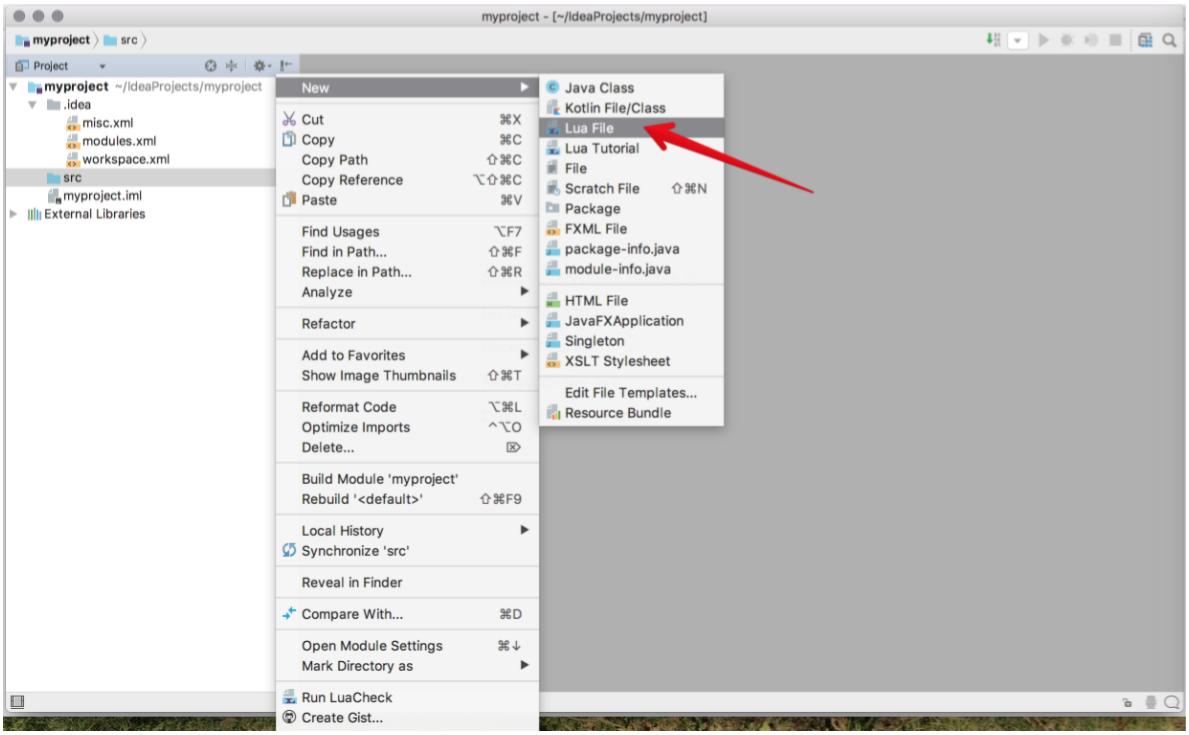


Теперь IntelliJ IDEA можно использовать с Tarantool'ом.

3. Создайте новый проект на Lua.

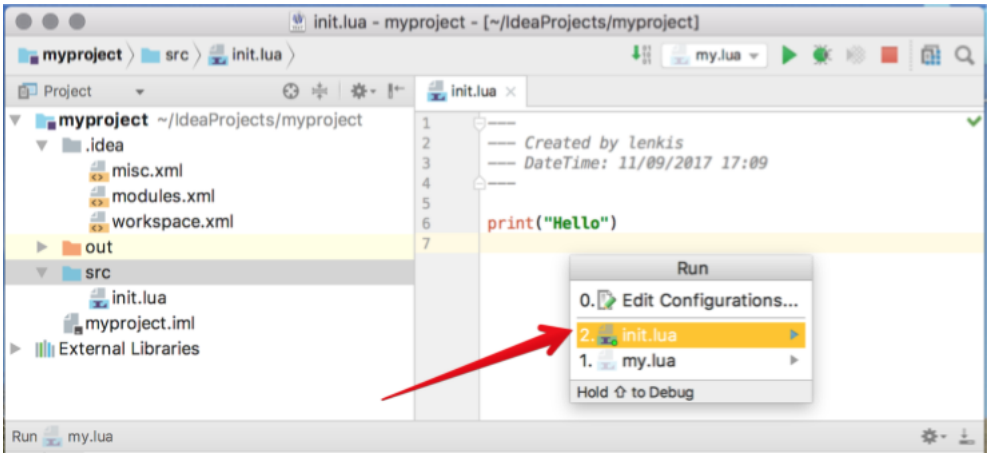


4. Добавьте новый Lua-файл, например, `init.lua`.



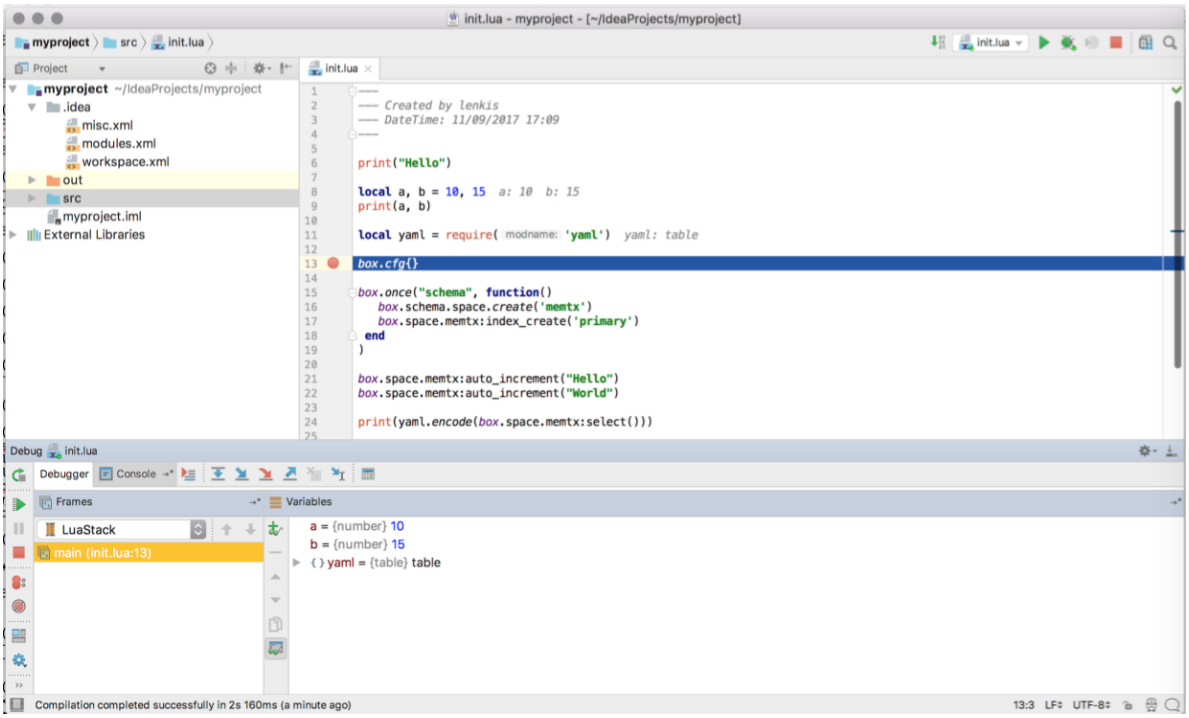
5. Разработайте код, сохраните файл.

6. Чтобы запустить приложение, нажмите `Run -> Run` в основном меню и выберите исходный файл из списка.



Или нажмите `Run -> Debug` для начала отладки.

Примечание: Чтобы использовать Lua-отладчик, обновите Tarantool до версии 1.7.5-29-gbb6170e4b или более поздней версии.



4.4.7 Примеры и рекомендации по разработке

Ниже представлены дополнения в виде Lua-программ для часто встречающихся или сложных случаев.

Любую из этих программ можно выполнить, скопировав код в `.lua-файл`, а затем выполнив в командной строке `chmod +x ./имя-программы.lua` и :samp :*./{имя-программы}.lua*.

Первая строка – это шебанг:

```
#!/usr/bin/env tarantool
```

Он запускает сервер приложений Tarantool'а на языке Lua, который должен быть в пути выполнения.

В этом разделе собраны следующие рецепты:

- *hello_world.lua*
- *console_start.lua*
- *fio_read.lua*
- *fio_write.lua*
- *ffi_printf.lua*
- *ffi_gettimeofday.lua*
- *ffi_zlib.lua*
- *ffi_meta.lua*
- *ffi_varbinary_insert.lua*
- *print_arrays.lua*

- *count_array.lua*
- *count_array_with_nils.lua*
- *count_array_with_nulls.lua*
- *count_map.lua*
- *swap.lua*
- *class.lua*
- *garbage.lua*
- *fiber_producer_and_consumer.lua*
- *socket_tcpconnect.lua*
- *socket_tcp_echo.lua*
- *getaddrinfo.lua*
- *socket_udp_echo.lua*
- *http_get.lua*
- *http_send.lua*
- *http_server.lua*
- *http_generate_html.lua*
- *select_all.go*

Можно использовать свободно.

hello_world.lua

Стандартный пример простой программы.

```
#!/usr/bin/env tarantool

 print('Hello, World!')
```

console_start.lua

Для инициализации базы данных (создания спейсов) используйте *box.once()*, если сервер запускается впервые. Затем используйте *console.start()*, чтобы запустить интерактивный режим.

```
#!/usr/bin/env tarantool

-- Настроить базу данных
box.cfg {
    listen = 3313
}

box.once("bootstrap", function()
    box.schema.space.create('tweedledum')
    box.space.tweedledum:create_index('primary',
        { type = 'TREE', parts = {1, 'unsigned'}})
```

(continues on next page)

(продолжение с предыдущей страницы)

```
end)

require('console').start()
```

fio_read.lua

Используйте *Модуль fio*, чтобы открыть, прочитать и закрыть файл.

```
#!/usr/bin/env tarantool

local fio = require('fio')
local errno = require('errno')
local f = fio.open('/tmp/xxxx.txt', {'O_RDONLY' })
if not f then
    error("Failed to open file: "..errno.strerror())
end
local data = f:read(4096)
f:close()
print(data)
```

fio_write.lua

Используйте *Модуль fio*, чтобы открыть, записать данные и закрыть файл.

```
#!/usr/bin/env tarantool

local fio = require('fio')
local errno = require('errno')
local f = fio.open('/tmp/xxxx.txt', {'O_CREAT', 'O_WRONLY', 'O_APPEND'},
    tonumber('0666', 8))
if not f then
    error("Failed to open file: "..errno.strerror())
end
f:write("Hello\n");
f:close()
```

ffi_printf.lua

Используйте Библиотеку LuaJIT FFI, чтобы вызвать встроенную в C функцию: printf(). (Чтобы лучше понимать FFI, см. Учебное пособие по FFI.)

```
#!/usr/bin/env tarantool

local ffi = require('ffi')
ffi.cdef[[
    int printf(const char *format, ...);
]]

ffi.C.printf("Hello, %s\n", os.getenv("USER"));
```

ffi_gettimeofday.lua

Используйте Библиотеку LuaJIT FFI, чтобы вызвать встроенную в C функцию: gettimeofday(). Она позволяет получить значение времени с точностью в миллисекундах, в отличие от функции времени в Tarantool'е *Модуль clock*.

```
#!/usr/bin/env tarantool

local ffi = require('ffi')
ffi.cdef[[
    typedef long time_t;
    typedef struct timeval {
    time_t tv_sec;
    time_t tv_usec;
} timeval;
    int gettimeofday(struct timeval *t, void *tzp);
]]

local timeval_buf = ffi.new("timeval")
local now = function()
    ffi.C.gettimeofday(timeval_buf, nil)
    return tonumber(timeval_buf.tv_sec * 1000 + (timeval_buf.tv_usec / 1000))
end
```

ffi_zlib.lua

Используйте Библиотеку LuaJIT FFI, чтобы вызвать библиотечную функцию в C. (Чтобы лучше понимать FFI, см. Учебное пособие по FFI.)

```
#!/usr/bin/env tarantool

local ffi = require("ffi")
ffi.cdef[[
    unsigned long compressBound(unsigned long sourceLen);
    int compress2(uint8_t *dest, unsigned long *destLen,
    const uint8_t *source, unsigned long sourceLen, int level);
    int uncompress(uint8_t *dest, unsigned long *destLen,
    const uint8_t *source, unsigned long sourceLen);
]]
local zlib = ffi.load(ffi.os == "Windows" and "zlib1" or "z")

-- Надстройка Lua для функции compress2()
local function compress(txt)
    local n = zlib.compressBound(#txt)
    local buf = ffi.new("uint8_t[?]", n)
    local buflen = ffi.new("unsigned long[1]", n)
    local res = zlib.compress2(buf, buflen, txt, #txt, 9)
    assert(res == 0)
    return ffi.string(buf, buflen[0])
end

-- Надстройка Lua для функции uncompress
local function uncompress(comp, n)
    local buf = ffi.new("uint8_t[?]", n)
    local buflen = ffi.new("unsigned long[1]", n)
    local res = zlib.uncompress(buf, buflen, comp, #comp)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    assert(res == 0)
    return ffi.string(buf, buflen[0])
end

-- Простой код теста
local txt = string.rep("abcd", 1000)
print("Uncompressed size: ", #txt)
local c = compress(txt)
print("Compressed size: ", #c)
local txt2 = uncompress(c, #txt)
assert(txt2 == txt)
```

ffi_meta.lua

Используйте Библиотеку LuaJIT FFI, чтобы получить доступ к объекту в C с помощью метаметода (метод, который определен метатаблицей).

```
#!/usr/bin/env tarantool

local ffi = require("ffi")
ffi.cdef[[
typedef struct { double x, y; } point_t;
]]

local point
local mt = {
  __add = function(a, b) return point(a.x+b.x, a.y+b.y) end,
  __len = function(a) return math.sqrt(a.x*a.x + a.y*a.y) end,
  __index = {
    area = function(a) return a.x*a.x + a.y*a.y end,
  },
}
point = ffi.metatype("point_t", mt)

local a = point(3, 4)
print(a.x, a.y)  --> 3  4
print(#a)        --> 5
print(a:area())  --> 25
local b = a + point(0.5, 8)
print(#b)        --> 12.5
```

ffi_varbinary_insert.lua

Используйте библиотеку LuaJIT ffi для вставки кортежа, имеющего поле VARBINARY.

Обратите внимание, что это разрешено только внутри memtx-транзакции: когда `box_insert()` не передает управление.

Lua не имеет прямой поддержки VARBINARY, поэтому использование C является одним из способов вставить данные, которые в MessagePack хранятся в виде bin (MP_BIN). Если кортеж будет получен позже, то поле «b» будет иметь тип = «cdata».

```
#!/usr/bin/env tarantool

```

(continues on next page)

(продолжение с предыдущей страницы)

```
-- box.cfg{} should be here

s = box.schema.space.create('withdata')
s:format({{"b", "varbinary"}})
s:create_index('pk', {parts = {1, "varbinary"}})

buffer = require('buffer')
ffi = require('ffi')

function varbinary_insert(space, bytes)
    local tmpbuf = buffer.ibuf()
    local p = tmpbuf:alloc(3 + #bytes)
    p[0] = 0x91 -- MsgPack code for "array-1"
    p[1] = 0xC4 -- MsgPack code for "bin-8" so up to 256 bytes
    p[2] = #bytes
    for i, c in pairs(bytes) do p[i + 3 - 1] = c end
    ffi.cdef[[int box_insert(uint32_t space_id,
                             const char *tuple,
                             const char *tuple_end,
                             box_tuple_t **result);]]
    ffi.C.box_insert(space.id, tmpbuf.rpos, tmpbuf.wpos, nil)
    tmpbuf:recycle()
end

varbinary_insert(s, {0xDE, 0xAD, 0xBE, 0xAF})
varbinary_insert(s, {0xFE, 0xED, 0xFA, 0xCE})

-- if successful, Tarantool enters the event loop now
```

print_arrays.lua

Используйте, чтобы создать Lua-таблицы и вывести их. Следует отметить, что для таблицы типа массива (array) функция-итератор будет `ipairs()`, а для таблицы типа ассоциативного массива (map) функция-итератор – `pairs()`. (`ipairs()` быстрее, чем `pairs()`, но `pairs()` рекомендуется для ассоциативных массивов или смешанных таблиц.) Результат будет выглядеть следующим образом: «1 Apple | 2 Orange | 3 Grapefruit | 4 Banana | k3 v3 | k1 v1 | k2 v2».

```
#!/usr/bin/env tarantool

array = { 'Apple', 'Orange', 'Grapefruit', 'Banana'}
for k, v in ipairs(array) do print(k, v) end

map = { k1 = 'v1', k2 = 'v2', k3 = 'v3' }
for k, v in pairs(map) do print(k, v) end
```

count_array.lua

Используйте оператор „#“, чтобы получить количество элементов в Lua-таблице типа массива. У этой операции сложность O(log(N)).

```
#!/usr/bin/env tarantool

array = { 1, 2, 3}
print(#array)
```

count_array_with_nils.lua

Отсутствующие элементы в массивах, которые Lua рассматривает как nil, заставляют простой оператор „#“ выдавать неправильные результаты. Команда «print(#t)» выведет «4», команда «print(counter)» выведет «3», а команда «print(max)» – «10». Другие табличные функции, такие как *table.sort()*, также сработают неправильно при наличии нулевых значений nil.

```
#!/usr/bin/env tarantool

local t = {}
t[1] = 1
t[4] = 4
t[10] = 10
print(#t)
local counter = 0
for k,v in pairs(t) do counter = counter + 1 end
print(counter)
local max = 0
for k,v in pairs(t) do if k > max then max = k end end
print(max)
```

count_array_with_nulls.lua

Используйте явные значения‘‘NULL‘‘, чтобы избежать проблем, вызванных nil в Lua == поведение с пропущенными значениями. Хотя `json.NULL == nil` является `true`, все команды вывода в данной программе выведут правильное значение: 10.

```
#!/usr/bin/env tarantool

local json = require('json')
local t = {}
t[1] = 1; t[2] = json.NULL; t[3]= json.NULL;
t[4] = 4; t[5] = json.NULL; t[6]= json.NULL;
t[6] = 4; t[7] = json.NULL; t[8]= json.NULL;
t[9] = json.NULL
t[10] = 10
print(#t)
local counter = 0
for k,v in pairs(t) do counter = counter + 1 end
print(counter)
local max = 0
for k,v in pairs(t) do if k > max then max = k end end
print(max)
```

count_map.lua

Программа используется для получения количества элементов в таблице типа ассоциативного массива.

```
#!/usr/bin/env tarantool

local map = { a = 10, b = 15, c = 20 }
local size = 0
for _ in pairs(map) do size = size + 1; end
print(size)
```

swap.lua

Программа использует особенность Lua менять местами две переменные без необходимости использования третьей переменной.

```
#!/usr/bin/env tarantool

local x = 1
local y = 2
x, y = y, x
print(x, y)
```

class.lua

Используется для создания класса, метатаблицы для класса, экземпляра класса. Другой пример можно найти в http://lua-users.org/wiki/LuaClassesWithMetatable.

```
#!/usr/bin/env tarantool

-- определить объекты класса
local myclass_somemethod = function(self)
    print('test 1', self.data)
end

local myclass_someothermethod = function(self)
    print('test 2', self.data)
end

local myclass_tostring = function(self)
    return 'MyClass <'..self.data..'>'
end

local myclass_mt = {
    __tostring = myclass_tostring;
    __index = {
        somemethod = myclass_somemethod;
        someothermethod = myclass_someothermethod;
    }
}

-- создать новый объект своего класса myclass
local object = setmetatable({ data = 'data'}, myclass_mt)
print(object:somemethod())
print(object.data)
```

garbage.lua

Запустите сборщик мусора в Lua с помощью функции collectgarbage.

```
#!/usr/bin/env tarantool

collectgarbage('collect')
```

fiber_producer_and_consumer.lua

Запустите один файбер для производителя и один файбер для потребителя. Используйте *fiber.channel()* для обмена данных и синхронизации. Можно настроить ширину канала (`ch_size` в программном коде) для управления количеством одновременных задач к обработке.

```
#!/usr/bin/env tarantool

local fiber = require('fiber')
local function consumer_loop(ch, i)
    -- инициализировать потребитель синхронно или выдать ошибку()
    fiber.sleep(0) -- позволить fiber.create() продолжать
    while true do
        local data = ch:get()
        if data == nil then
            break
        end
        print('consumed', i, data)
        fiber.sleep(math.random()) -- моделировать работу
    end
end

local function producer_loop(ch, i)
    -- инициализировать потребитель синхронно или выдать ошибку()
    fiber.sleep(0) -- allow fiber.create() to continue
    while true do
        local data = math.random()
        ch:put(data)
        print('produced', i, data)
    end
end

local function start()
    local consumer_n = 5
    local producer_n = 3

    -- создать канал
    local ch_size = math.max(consumer_n, producer_n)
    local ch = fiber.channel(ch_size)

    -- запустить потребители
    for i=1, consumer_n,1 do
        fiber.create(consumer_loop, ch, i)
    end

    -- запустить производители
    for i=1, producer_n,1 do
        fiber.create(producer_loop, ch, i)
    end
end

start()
print('started')
```

socket_tcpconnect.lua

Используйте *socket.tcp_connect()* для подключения к удаленному серверу по TCP. Можно отобразить информацию о подключении и результат запроса GET.

```
#!/usr/bin/env tarantool

local s = require('socket').tcp_connect('google.com', 80)
print(s:peer().host)
print(s:peer().family)
print(s:peer().type)
print(s:peer().protocol)
print(s:peer().port)
print(s:write("GET / HTTP/1.0\r\n\r\n"))
print(s:read('\r\n'))
print(s:read('\r\n'))
```

socket_tcp_echo.lua

Используйте *socket.tcp_connect()* для настройки простого TCP-сервера путем создания функции, которая обрабатывает запросы и отражает их, а затем передачи функции на *socket.tcp_server()*. Данная программа была протестирована на 100 000 клиентов, каждый из которых получил отдельный файбер.

```
#!/usr/bin/env tarantool

local function handler(s, peer)
    s:write("Welcome to test server, " .. peer.host .."\n")
    while true do
        local line = s:read('\n')
        if line == nil then
            break -- ошибка или конец файла
        end
        if not s:write("pong: "..line) then
            break -- ошибка или конец файла
        end
    end
end

local server, addr = require('socket').tcp_server('localhost', 3311, handler)
```

getaddrinfo.lua

Используйте *socket.getaddrinfo()*, чтобы провести неблокирующее разрешение имен DNS, получая как AF_INET6, так и AF_INET информацию для „google.com“. Данная техника не всегда необходима для TCP-соединений, поскольку *socket.tcp_connect()* выполняет *socket.getaddrinfo* с точки зрения внутреннего устройства до попытки соединения с первым доступным адресом.

```
#!/usr/bin/env tarantool

local s = require('socket').getaddrinfo('google.com', 'http', { type = 'SOCK_STREAM' })
print('host=',s[1].host)
print('family=',s[1].family)
print('type=',s[1].type)
print('protocol=',s[1].protocol)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
print('port=',s[1].port)
print('host=',s[2].host)
print('family=',s[2].family)
print('type=',s[2].type)
print('protocol=',s[2].protocol)
print('port=',s[2].port)
```

socket_udp_echo.lua

В данный момент в Tarantool нет функции *udp_server*, поэтому socket_udp_echo.lua – более сложная программа, чем socket_tcp_echo.lua. Ее можно реализовать с помощью сокетов и файберов.

```
#!/usr/bin/env tarantool

local socket = require('socket')
local errno = require('errno')
local fiber = require('fiber')

local function udp_server_loop(s, handler)
    fiber.name("udp_server")
    while true do
        -- попытка прочитать сначала датаграмму
        local msg, peer = s:recvfrom()
        if msg == "" then
            -- сокет был закрыт с помощью s:close()
            break
        elseif msg ~= nil then
            -- получена новая датаграмма
            handler(s, peer, msg)
        else
            if s:errno() == errno.EAGAIN or s:errno() == errno.EINTR then
                -- сокет не готов
                s:readable() -- передача управления, epoll сообщит, когда будут новые данные
            else
                -- ошибка сокета
                local msg = s:error()
                s:close() -- сохранить ресурсы и не ждать сборку мусора
                error("Socket error: " .. msg)
            end
        end
    end
end

local function udp_server(host, port, handler)
    local s = socket('AF_INET', 'SOCK_DGRAM', 0)
    if not s then
        return nil -- проверить номер ошибки errno:strerror()
   end
    if not s:bind(host, port) then
        local e = s:errno() -- сохранить номер ошибки errno
        s:close()
        errno(e) -- восстановить номер ошибки errno
        return nil -- проверить номер ошибки errno:strerror()
    end
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    fiber.create(udp_server_loop, s, handler) -- запустить новый файбер в фоновом режиме
    return s
end
```

Функция для клиента, который подключается к этому серверу, может выглядеть следующим образом:

```
local function handler(s, peer, msg)
    -- Необязательно ждать, пока сокет будет готов отправлять UDP
    -- s:writable()
    s:sendto(peer.host, peer.port, "Pong: " .. msg)
end

local server = udp_server('127.0.0.1', 3548, handler)
if not server then
    error('Failed to bind: ' .. errno.strerror())
end

print('Started')

require('console').start()
```

http_get.lua

Используйте *Модуль HTTP* для получения данных по HTTP.

```
#!/usr/bin/env tarantool

local http_client = require('http.client')
local json = require('json')
local r = http_client.get('https://api.frankfurter.app/latest?to=USD%2CRUB')
if r.status ~= 200 then
    print('Failed to get currency ', r.reason)
    return
end
local data = json.decode(r.body)
print(data.base, 'rate of', data.date, 'is', data.rates.RUB, 'RUB or', data.rates.USD, 'USD')
```

http_send.lua

Используйте *Модуль HTTP* для отправки данных по HTTP.

```
#!/usr/bin/env tarantool

local http_client = require('http.client')
local json = require('json')
local data = json.encode({ Key = 'Value'})
local headers = { Token = 'xxxx', ['X-Secret-Value'] = '42' }
local r = http_client.post('http://localhost:8081', data, { headers = headers})
if r.status == 200 then
    print 'Success'
end
```

http_server.lua

Используйте сторонний модуль http (который необходимо предварительно установить), чтобы превратить Tarantool в веб-сервер.

```
#!/usr/bin/env tarantool

local function handler(self)
    return self:render{ json = { ['Your-IP-Is'] = self:peer().host } }
end

local server = require('http.server').new(nil, 8080) -- listen *:8080
local router = require('http.router').new({charset = "utf8"})
server:set_router(router)
router:route({ path = '/' }, handler)
server:start()
-- connect to localhost:8080 and see json
```

http_generate_html.lua

Используйте сторонний модуль http (который необходимо предварительно установить) для создания HTML-страниц из шаблонов. В модуле http достаточно простой движок шаблонов, который позволяет выполнять регулярный код на Lua в текстовых блоках (как в PHP). Таким образом, нет необходимости в изучении новых языков, чтобы написать шаблоны.

```
#!/usr/bin/env tarantool

local function handler(self)
local fruits = { 'Apple', 'Orange', 'Grapefruit', 'Banana'}
    return self:render{ fruits = fruits }
end

local server = require('http.server').new(nil, 8080) -- nil means '*'
local router = require('http.router').new({charset = "utf8"})
server:set_router(router)
router:route({ path = '/', file = 'index.html.lua' }, handler)
server:start()
```

HTML-файл для этого сервера, включая Lua, может выглядеть следующим образом (он выведет «1 Apple | 2 Orange | 3 Grapefruit | 4 Banana»).

```
<html>
<body>
    <table border="1">
        % for i,v in pairs(fruits) do
        <tr>
            <td><%= i %></td>
            <td><%= v %></td>
        </tr>
        % end
    </table>
</body>
</html>
```

select_all.go

На языке Go выборка содержимого всего спейса не является тривиальной задачей, которая решается в одну строчку. Ниже мы приводим пример программы, которая осуществляет полную выборку из спейса „tester“. Эту программу нужно вызвать на том экземпляре, с которым вы собираетесь установить соединение через Go-коннектор.

```
package main

import (
        "fmt"
        "log"

        "github.com/tarantool/go-tarantool"
)

/*
box.cfg{listen = 3301}
box.schema.user.passwd('pass')

s = box.schema.space.create('tester')
s:format({
    {name = 'id', type = 'unsigned'},
    {name = 'band_name', type = 'string'},
    {name = 'year', type = 'unsigned'}
})
s:create_index('primary', { type = 'hash', parts = {'id'} })
s:create_index('scanner', { type = 'tree', parts = {'id', 'band_name'} })

s:insert{1, 'Roxette', 1986}
s:insert{2, 'Scorpions', 2015}
s:insert{3, 'Ace of Base', 1993}
*/

func main() {
        conn, err := tarantool.Connect("127.0.0.1:3301", tarantool.Opts{
                User: "admin",
                Pass: "pass",
        })

        if err != nil {
                log.Fatalf("Connection refused")
        }
        defer conn.Close()

        spaceName := "tester"
        indexName := "scanner"
        idFn := conn.Schema.Spaces[spaceName].Fields["id"].Id
        bandNameFn := conn.Schema.Spaces[spaceName].Fields["band_name"].Id

        var tuplesPerRequest uint32 = 2
        cursor := []interface{}{}

        for {
                resp, err := conn.Select(spaceName, indexName, 0, tuplesPerRequest, tarantool.
↪IterGt, cursor)
                if err != nil {
```

(continues on next page)

(продолжение с предыдущей страницы)

```
                        log.Fatalf("Failed to select: %s", err)
                }

                if resp.Code != tarantool.OkCode {
                        log.Fatalf("Select failed: %s", resp.Error)
                }

                if len(resp.Data) == 0 {
                        break
                }

                fmt.Println("Iteration")

                tuples := resp.Tuples()
                for _, tuple := range tuples {
                        fmt.Printf("\t%v\n", tuple)
                }

                lastTuple := tuples[len(tuples)-1]
                cursor = []interface{}{lastTuple[idFn], lastTuple[bandNameFn]}
        }
}
```

4.5 Администрирование

Tarantool устроен таким образом, что возможно запустить несколько экземпляров программы на одном компьютере.

Здесь мы показываем, как администрировать экземпляры Tarantool'а с помощью любой из следующих утилит:

- встроенные утилиты `systemd` или
- *tarantoolctl*, утилита, поставляемая и устанавливаемая вместе с дистрибутивом Tarantool'а.

Примечание:

- В отличие от остальной части руководства, в этой главе мы используем общесистемные пути.
- Здесь мы приводим примеры консольного вывода для Fedora.

Эта глава включает в себя следующие разделы:

4.5.1 Настройка экземпляров Tarantool'а

Для каждого экземпляра Tarantool'а понадобится два файла:

- [Необязательный] *Файл приложения*, содержащий логику данного экземпляра. Поместите его в папку `/usr/share/tarantool/`.

  Например, `/usr/share/tarantool/my_app.lua` (здесь мы реализуем его как *Lua-модуль*, который запускает базу данных и экспортирует функцию `start()` для API -вызовов):

```
local function start()
    box.schema.space.create("somedata")
    box.space.somedata:create_index("primary")
    <...>
end

return {
  start = start;
}
```

- *Файл экземпляра*, содержащий логику и параметры инициализации данного экземпляра. Поместите этот файл или символьную ссылку на него в **директорию экземпляра** (см. параметр *instance_dir* в конфигурационном файле `tarantoolctl`).

  Например, `/etc/tarantool/instances.enabled/my_app.lua` (здесь мы загружаем модуль `my_app.lua` и вызываем из него функцию `start()`):

```
#!/usr/bin/env tarantool

box.cfg {
    listen = 3301;
}

-- загрузить модуль my_app и вызвать функцию start()
-- некоторые опции приложения под контролем сисадминов
local m = require('my_app').start({...})
```

Файл экземпляра

После столь краткого предисловия может возникнуть вопрос: что из себя представляет файл экземпляра, для чего он нужен и как `tarantoolctl` использует его? Если Tarantool – это сервер приложений, так почему бы не запускать хранящееся в `/usr/share/tarantool` приложение напрямую?

Типичное приложение для Tarantool – это не скрипт, а демон, запущенный в фоновом режиме и обрабатывающий запросы, которые, как правило, посылаются через TCP/IP-сокет. Необходимо запускать этот демон со стартом операционной системы и управлять им с помощью стандартных средств операционной системы для управления сервисами – таких как `systemd` или `init.d`. С этой целью и были созданы **файлы экземпляра**.

Файлов экземпляра может быть больше одного. Например, одно и то же приложение в `/usr/share/tarantool` может быть запущено на нескольких экземплярах Tarantool'а, у каждого из которых есть свой файл экземпляра. Или в `/usr/share/tarantool` может быть несколько приложений, и на каждое из них будет опять же приходиться свой файл экземпляра.

Обычно файл экземпляра создает системный администратор, а файл приложения предоставляет разработчик в Lua-модуле или rpm/deb-пакете.

По своему устройству файл экземпляра ничем не отличается от Lua-приложения. Однако с его помощью должна настраиваться база данных, поэтому в нем должен содержаться вызов *box.cfg{}*, потому что это единственный способ превратить Tarantool-скрипт в фоновый процесс, а `tarantoolctl` – это инструмент для управления фоновыми процессами. За исключением этого вызова, файл экземпляра может содержать произвольный код на Lua и, теоретически, даже всю бизнес-логику приложения. Однако мы не рекомендуем хранить весь код в файле экземпляра, потому что это приводит как к замусориванию самого файла, так и к ненужному копированию кода при необходимости запустить несколько экземпляров приложения.

Конфигурационный файл *tarantoolctl*

Файлы экземпляра содержат конфигурацию экземпляра, тогда как конфигурационный файл `tarantoolctl` содержит конфигурацию, которую `tarantoolctl` использует, чтобы переопределять конфигурацию экземпляров. Другими словами, он содержит общесистемную конфигурацию по умолчанию. Если `tarantoolctl` не сможет обнаружить этот файл, используя метод, описанный в разделе *Запуск/остановка экземпляра*, будут использованы настройки по умолчанию.

Большинство параметров схожи с теми, которые используются в *box.cfg{}*. Ниже даны настройки по умолчанию (могут быть установлены в `/etc/default/tarantool` или `/etc/sysconfig/tarantool` как часть дистрибутива Tarantool'а – см. пути по умолчанию для разных ОС в *Замечаниях по поводу некоторых операционных систем*):

```
default_cfg = {
    pid_file  = "/var/run/tarantool",
    wal_dir   = "/var/lib/tarantool",
    memtx_dir = "/var/lib/tarantool",
    vinyl_dir = "/var/lib/tarantool",
    log       = "/var/log/tarantool",
    username  = "tarantool",
    language  = "Lua",
}
instance_dir = "/etc/tarantool/instances.enabled"
```

где:

- `pid_file`
  Директория, где хранятся pid-файл и socket-файл; `tarantoolctl` добавляет "/имя_экземпляра" к имени директории.
- `wal_dir`
  Директория, где хранятся .xlog-файлы; `tarantoolctl` добавляет "/имя_экземпляра" к имени директории.
- `memtx_dir`
  Директория, где хранятся .snap-файлы; `tarantoolctl` добавляет "/имя_экземпляра" к имени директории.
- `vinyl_dir`
  Директория, где хранятся vinyl-файлы; `tarantoolctl` добавляет "/имя_экземпляра" к имени директории.
- `log`
  Директория, где хранятся файлы журнала с сообщениями от Tarantool-приложения; `tarantoolctl` добавляет "/имя_экземпляра" к имени директории.
- `username`
  Пользователь, запускающий экземпляр Tarantool'а. Это пользователь операционной системы, а не Tarantool-клиента. Став демоном, Tarantool сменит своего пользователя на указанного.
- `language`
  The *interactive console* language. Can be either `Lua` or `SQL`.
- `instance_dir`
  Директория, где хранятся все файлы экземпляра для данного компьютера. Поместите сюда файлы экземпляра или создайте символьные ссылки на них.

  Директория с экземплярами, которая используется по умолчанию, зависит от параметра `WITH_SYSVINIT` сборки Tarantool'а: когда его значение «ON», то `/etc/tarantool/instances.`

`enabled`, в противном случае («OFF» или значение не установлено), то `/etc/tarantool/instances.available`. Последний случай характерен для сборок Tarantool'а для дистрибутивов Linux с `systemd`.

Для проверки параметров сборки выполните команду `tarantool --version`.

As a full-featured example, you can take example.lua script that ships with Tarantool and defines all configuration options.

4.5.2 Запуск/остановка экземпляра

Lua-приложение выполняется Tarantool'ом, тогда как файл экземпляра выполняется Tarantool-скриптом `tarantoolctl`.

Вот что делает `tarantoolctl` при вводе следующей команды:

```
$ tarantoolctl start <имя_экземпляра>
```

1. Считывает и разбирает аргументы командной строки. В нашем случае последний аргумент содержит имя экземпляра.
2. Считывает и разбирает собственный конфигурационный файл. Этот файл содержит параметры `tarantoolctl` по умолчанию – такие как путь до директории, в которой располагаются экземпляры.

   Когда `tarantoolctl` вызывается с root-правами, он ищет конфигурационный файл в `/etc/default/tarantool`. Если вызов `tarantool` производит локальный пользователь, сначала он ищет конфигурационный файл в текущей директории (`$PWD/.tarantoolctl`), а затем в домашней директории текущего пользователя (`$HOME/.config/tarantool/tarantool`). Если конфигурационный файл не найден, `tarantoolctl` принимает *встроенные параметры по умолчанию*.
3. Ищет файл экземпляра в директории, где располагаются экземпляры, например, в `/etc/tarantool/instances.enabled`. `tarantoolctl` строит путь до файла экземпляра следующим образом: «путь до директории с экземплярами» + «имя экземпляра» + «.lua».
4. Переопределяет функцию *box.cfg{}*, чтобы предобработать ее параметры и сделать так, чтобы пути к экземплярам указывали на пути, прописанные в конфигурационном файле `tarantoolctl`. Например, если в конфигурационном файле указано, что рабочей директорией экземпляра является `/var/tarantool`, то новая реализация `box.cfg{}` сделает так, чтобы параметр *work_dir* в `box.cfg{}` имел значение `/var/tarantool/<имя_экземпляра>`, независимо от того, какой путь указан в самом файле экземпляра.
5. Создает так называемый «файл для управления экземпляром». Это Unix-сокет с прикрепленной к нему Lua-консолью. В дальнейшем `tarantoolctl` использует этот файл для получения состояния экземпляра, отправки команд и т.д.
6. Задает значение переменной окружения TARANTOOLCTL = „true“. Это позволит пользователю понять, что экземпляр был запущен `tarantoolctl`.
7. Наконец, использует Lua-команду `dofile` для выполнения файла экземпляра.

При запуске экземпляра с помощью инструментария `systemd` указанным ниже способом (имя экземпляра – `my_app`):

```
$ systemctl start tarantool@my_app
$ ps axuf|grep my_app
taranto+  5350  1.3  0.3 1448872 7736 ?        Ssl  20:05   0:28 tarantool my_app.lua <running>
```

. . . на самом деле вызывается `tarantoolctl` – так же, как и в случае `tarantoolctl start my_app`.

Для проверки файла экземпляра на наличие синтаксических ошибок перед запуском экземпляра `my_app` используйте команду:

```
$ tarantoolctl check my_app
```

Для включения автоматической загрузки экземпляра `my_app` при запуске всей системы используйте команду:

```
$ systemctl enable tarantool@my_app
```

Для остановки работающего экземпляра `my_app` используйте команду:

```
$ tarantoolctl stop my_app
$ # - ИЛИ -
$ systemctl stop tarantool@my_app
```

Для перезапуска (т.е. остановки и запуска) работающего экземпляра `my_app` используйте команду:

```
$ tarantoolctl restart my_app
$ # - ИЛИ -
$ systemctl restart tarantool@my_app
```

Локальный запуск Tarantool'а

Иногда бывает необходимо запустить Tarantool локально – например, для тестирования. Давайте настроим локальный экземпляр, запустим его и будем мониторить с помощью `tarantoolctl`.

Сперва создадим директорию-песочницу по следующему пути:

```
$ mkdir ~/tarantool_test
```

. . . и поместим конфигурационный файл с параметрами `tarantoolctl` по умолчанию в `$HOME/.config/tarantool/tarantool`. Содержимое файла будет таким:

```
default_cfg = {
    pid_file  = "/home/user/tarantool_test/my_app.pid",
    wal_dir   = "/home/user/tarantool_test",
    snap_dir  = "/home/user/tarantool_test",
    vinyl_dir = "/home/user/tarantool_test",
    log       = "/home/user/tarantool_test/log",
}
instance_dir = "/home/user/tarantool_test"
```

Примечание:

- Указывайте полный путь к домашней директории пользователя вместо «~/».
- Опустите параметр `username`. Обычно, когда запуск производит локальный пользователь, у `tarantoolctl` нет разрешения на смену текущего пользователя. Экземпляр будет работать с пользователем „admin“.

Далее создадим файл экземпляра `~/tarantool_test/my_app.lua`. Содержимое файла будет таким:

```
box.cfg{listen = 3301}
box.schema.user.passwd('Gx5!')
box.schema.user.grant('guest','read,write,execute','universe')
fiber = require('fiber')
box.schema.space.create('tester')
box.space.tester:create_index('primary',{})
i = 0
while 0 == 0 do
    fiber.sleep(5)
    i = i + 1
    print('insert ' .. i)
    box.space.tester:insert{i, 'my_app tuple'}
end
```

Проверим наш файл экземпляра, сперва запустив его без `tarantoolctl`:

```
$ cd ~/tarantool_test
$ tarantool my_app.lua
2017-04-06 10:42:15.762 [54085] main/101/my_app.lua C> version 1.7.3-489-gd86e36d5b
2017-04-06 10:42:15.763 [54085] main/101/my_app.lua C> log level 5
2017-04-06 10:42:15.764 [54085] main/101/my_app.lua I> mapping 268435456 bytes for tuple arena...
2017-04-06 10:42:15.774 [54085] iproto/101/main I> binary: bound to [::]:3301
2017-04-06 10:42:15.774 [54085] main/101/my_app.lua I> initializing an empty data directory
2017-04-06 10:42:15.789 [54085] snapshot/101/main I> saving snapshot `./00000000000000000000.snap.
↪inprogress'
2017-04-06 10:42:15.790 [54085] snapshot/101/main I> done
2017-04-06 10:42:15.791 [54085] main/101/my_app.lua I> vinyl checkpoint done
2017-04-06 10:42:15.791 [54085] main/101/my_app.lua I> ready to accept requests
insert 1
insert 2
insert 3
<...>
```

Запустим экземпляр Tarantool'а с помощью `tarantoolctl`:

```
$ tarantoolctl start my_app
```

В консоли должны появиться сообщения о том, что экземпляр запущен. Затем выполним следующую команду:

```
$ ls -l ~/tarantool_test/my_app
```

В консоли должны появиться .snap-файл и .xlog-файл. Затем выполним следующую команду:

```
$ less ~/tarantool_test/log/my_app.log
```

В консоли должно отобразиться содержимое файла журнала для приложения `my_app`, в том числе сообщения об ошибках, если они были. Затем выполним серию команд:

```
$ tarantoolctl enter my_app
tarantool> box.cfg{}
tarantool> console = require('console')
tarantool> console.connect('localhost:3301')
tarantool> box.space.tester:select({0}, {iterator = 'GE'})
```

В консоли должны появиться несколько кортежей, которые создало приложение `my_app`.

Теперь остановим приложение `my_app`. Корректный способ остановки – это использовать“tarantoolctl“:

```
$ tarantoolctl stop my_app
```

Последний шаг – удаление тестовых данных.

```
$ rm -R tarantool_test
```

4.5.3 Журналирование

Все важные события Tarantool записывает в файл журнала – например, в `/var/log/tarantool/my_app.log`. `tarantoolctl` строит путь до файла журнала следующим образом: «путь до директории с экземплярами» + «имя экземпляра» + «.lua».

Запишем что-нибудь в файл журнала:

```
$ tarantoolctl enter my_app
/bin/tarantoolctl: connected to unix/:/var/run/tarantool/my_app.control
unix/:/var/run/tarantool/my_app.control> require('log').info("Hello for the manual readers")
---
...
```

Затем проверим содержимое журнала:

```
$ tail /var/log/tarantool/my_app.log
2017-04-04 15:54:04.977 [29255] main/101/tarantoolctl C> version 1.7.3-382-g68ef3f6a9
2017-04-04 15:54:04.977 [29255] main/101/tarantoolctl C> log level 5
2017-04-04 15:54:04.978 [29255] main/101/tarantoolctl I> mapping 134217728 bytes for tuple arena...
2017-04-04 15:54:04.985 [29255] iproto/101/main I> binary: bound to [::1]:3301
2017-04-04 15:54:04.986 [29255] main/101/tarantoolctl I> recovery start
2017-04-04 15:54:04.986 [29255] main/101/tarantoolctl I> recovering from `/var/lib/tarantool/my_
↪app/00000000000000000000.snap'
2017-04-04 15:54:04.988 [29255] main/101/tarantoolctl I> ready to accept requests
2017-04-04 15:54:04.988 [29255] main/101/tarantoolctl I> set 'checkpoint_interval' configuration␣
↪option to 3600
2017-04-04 15:54:04.988 [29255] main/101/my_app I> Run console at unix/:/var/run/tarantool/my_app.
↪control
2017-04-04 15:54:04.989 [29255] main/106/console/unix/:/var/ I> started
2017-04-04 15:54:04.989 [29255] main C> entering the event loop
2017-04-04 15:54:47.147 [29255] main/107/console/unix/: I> Hello for the manual readers
```

При включенном журналировании системный администратор должен обеспечивать своевременную ротацию журналов, чтобы избежать переполнения дискового пространства. Ротация журналов в `tarantoolctl` производится с помощью программы `logrotate`, которую необходимо установить заранее.

Файл `/etc/logrotate.d/tarantool` поставляется со стандартным дистрибутивом Tarantool. Его можно редактировать для изменения поведения по умолчанию. Содержимое файла обычно выглядит так:

```
/var/log/tarantool/*.log {
    daily
    size 512k
    missingok
    rotate 10
    compress
    delaycompress
    create 0640 tarantool adm
    postrotate
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        /usr/bin/tarantoolctl logrotate `basename ${1%%.*}`
    endscript
}
```

Если вы используете другую программу для ротации журналов, можно вызвать команду `tarantoolctl logrotate`, чтобы экземпляры переоткрыли свои файлы журнала после того, как выбранная вами программа переместила их.

Tarantool может писать события в файл журнала, `syslog` или программу, указанную в конфигурационном файле (см. параметр *log*).

По умолчанию запись производится в файл журнала, как указано в исходных настройках `tarantoolctl`. Скрипт `tarantoolctl` автоматически определяет, когда экземпляр использует для журналирования `syslog` или внешнюю программу, и не изменяет то, куда ведется запись. В таких случаях ротацию журналов обычно выполняет та же программа, которая используется для журналирования. Именно поэтому команда `tarantoolctl logrotate` сработает только в том случае, если в файле экземпляра включена возможность вести запись в файл.

4.5.4 Безопасность

Tarantool разрешает два типа подключений:

- Используя функцию *console.listen()* из модуля `console`, можно настроить порт для подключения к серверной административной консоли. Этот вариант для администраторов, которым необходимо подключиться к работающему экземпляру и послать некоторые запросы. `tarantoolctl` вызывает `console.listen()`, чтобы создать управляющий сокет для каждого запущенного экземпляра.
- Используя параметр *box.cfg{listen=...}* из модуля `box`, можно настроить бинарный порт для соединений, которые читают и пишут в базу данных или вызывают хранимые процедуры.

Если вы подключены к административной консоли:

- Клиент-серверный протокол – это простой текст.
- Пароль не требуется.
- Пользователь автоматически получает права администратора.
- Каждая команда напрямую обрабатывается встроенным интерпретатором Lua.

Поэтому порты для административной консоли следует настраивать очень осторожно. Если это TCP-порт, он должен быть открыть только для определенного IP-адреса. В идеале вместо TCP-порта лучше настроить доменный Unix-сокет, который требует наличие прав доступа к серверной машине. Тогда типичная настройка порта для административной консоли будет выглядеть следующим образом:

```
console.listen('/var/lib/tarantool/socket_name.sock')
```

а типичный *URI* для соединения будет таким:

```
/var/lib/tarantool/socket_name.sock
```

если у приемника событий есть права на запись в `/var/lib/tarantool` и у коннектора есть права на чтение из `/var/lib/tarantool`. Еще один способ подключиться к административной консоли экземпляра, запущенного с помощью `tarantoolctl`, – использовать *tarantoolctl enter*.

Выяснить, является ли некоторый TCP-порт портом для административной консоли, можно с помощью `telnet`. Например:

```
$ telnet 0 3303
Trying 0.0.0.0...
Connected to 0.
Escape character is '^]'.
Tarantool 2.1.0 (Lua console)
type 'help' for interactive help
```

В этом примере в ответе от сервера нет слова «binary» и есть слова «Lua console». Это значит, что мы успешно подключились к порту для административной консоли и можем вводить администраторские запросы на этом терминале.

Если вы подключены к бинарному порту:

- Клиент-серверный протокол – *бинарный*.
- Автоматически выбирается пользователь „*guest*“.
- Для смены пользователя необходимо пройти аутентификацию.

Для удобства использования команда `tarantoolctl connect` автоматически определяет тип подключения при установке соединения и использует команду бинарного протокола *EVAL* для выполнения Lua-команд по бинарному подключению. Чтобы выполнить команду EVAL, аутентифицированный пользователь должен иметь глобальные «EXECUTE»-права.

Поэтому при невозможности подключиться к машине по `ssh` системный администратор может получить **удаленный** доступ к экземпляру, создав пользователя Tarantool с глобальными «EXECUTE»-правами и непустым паролем.

4.5.5 Просмотр состояния сервера

Использование Tarantool'а в качестве клиента

Tarantool входит в интерактивный режим, если:

- вы запускаете его без *файла экземпляра*, либо
- в файле экземпляра содержится команда *console.start()*.

Tarantool выводит приглашение командной строки (например, «tarantool>») – и вы можете посылать запросы. Если использовать Tarantool таким образом, он может выступать клиентом для удаленного сервера, см. простые примеры в *Руководстве для начинающих*.

Скрипт `tarantoolctl` использует интерактивный режим для реализации команд «enter» и «connect».

Выполнение кода на экземпляре Tarantool'а

Можно подключиться к *административной консоли* экземпляра и выполнить некий Lua-код с помощью утилиты `tarantoolctl`:

```
$ # для локальных экземпляров:
$ tarantoolctl enter my_app
/bin/tarantoolctl: Found my_app.lua in /etc/tarantool/instances.available
/bin/tarantoolctl: Connecting to /var/run/tarantool/my_app.control
/bin/tarantoolctl: connected to unix/:/var/run/tarantool/my_app.control
unix/:/var/run/tarantool/my_app.control> 1 + 1
---
- 2
```

(continues on next page)

(продолжение с предыдущей страницы)

```
...
unix/:/var/run/tarantool/my_app.control>

$ # для локальных и удаленных экземпляров:
$ tarantoolctl connect username:password@127.0.0.1:3306
```

Можно также использовать `tarantoolctl` для выполнения Lua-кода на запущенном экземпляре Tarantool-сервера, не подключаясь к его административной консоли. Например:

```
$ # выполнение команд напрямую из командной строки
$ <command> | tarantoolctl eval my_app
<...>

$ # - ИЛИ -

$ # выполнение команд из скрипта
$ tarantoolctl eval my_app script.lua
<...>
```

Примечание: Еще можно использовать модули *console* и *net.box* из Tarantool-сервера. Также вы можете писать свои клиентские программы с использованием любого из доступных *коннекторов*. Однако большинство примеров в данном документе использует или `tarantoolctl connect`, или *Tarantool-сервер как клиент*.

Проверка состояния экземпляра

Чтобы проверить статус экземпляра Tarantool-сервера, выполните команду:

```
$ tarantoolctl status my_app
my_app is running (pid: /var/run/tarantool/my_app.pid)

$ # - ИЛИ -

$ systemctl status tarantool@my_app
tarantool@my_app.service - Tarantool Database Server
Loaded: loaded (/etc/systemd/system/tarantool@.service; disabled; vendor preset: disabled)
Active: active (running)
Docs: man:tarantool(1)
Process: 5346 ExecStart=/usr/bin/tarantoolctl start %I (code=exited, status=0/SUCCESS)
Main PID: 5350 (tarantool)
Tasks: 11 (limit: 512)
CGroup: /system.slice/system-tarantool.slice/tarantool@my_app.service
+ 5350 tarantool my_app.lua <running>
```

Если вы используете систему, на которой доступна утилита `systemd`, выполните следующую команду для проверки содержимого журнала загрузки:

```
$ journalctl -u tarantool@my_app -n 5
-- Logs begin at Fri 2016-01-08 12:21:53 MSK, end at Thu 2016-01-21 21:17:47 MSK. --
Jan 21 21:17:47 localhost.localdomain systemd[1]: Stopped Tarantool Database Server.
Jan 21 21:17:47 localhost.localdomain systemd[1]: Starting Tarantool Database Server...
Jan 21 21:17:47 localhost.localdomain tarantoolctl[5969]: /usr/bin/tarantoolctl: Found my_app.lua␣
↪in /etc/tarantool/instances.available
```

(continues on next page)

(продолжение с предыдущей страницы)

```
Jan 21 21:17:47 localhost.localdomain tarantoolctl[5969]: /usr/bin/tarantoolctl: Starting instance.
↪..
Jan 21 21:17:47 localhost.localdomain systemd[1]: Started Tarantool Database Server
```

Более подробная информация содержится в отчетах, которые можно получить с помощью функций из следующих подмодулей:

- *box.cfg* – проверка и указание всех конфигурационных параметров Tarantool-сервера,
- *box.slab* – мониторинг использования и фрагментированности памяти, выделенной для хранения данных в Tarantool'е,
- *box.info* – просмотр переменных Tarantool-сервера – в первую очередь тех, что относятся к репликации,
- *box.stat* – просмотр статистики Tarantool'а по запросам и использованию сети,

You can also try prometheus, a plugin that makes it easy to collect metrics (e.g. memory usage or number of requests) from Tarantool applications and databases and expose them via the Prometheus protocol.

Пример

Очень часто администраторам приходится вызывать функцию *box.slab.info()*, которая показывает подробную статистику по использованию памяти для конкретного экземпляра Tarantool'а.

```
tarantool> box.slab.info()
---
- items_size: 228128
  items_used_ratio: 1.8%
  quota_size: 1073741824
  quota_used_ratio: 0.8%
  arena_used_ratio: 43.2%
  items_used: 4208
  quota_used: 8388608
  arena_size: 2325176
  arena_used: 1003632
...
```

Tarantool занимает память операционной системы, например, когда пользователь вставляет много данных. Можно проверить, сколько памяти занято, выполнив команду (в Linux):

```
ps -eo args,%mem | grep "tarantool"
```

Tarantool почти никогда не освобождает эту память, даже если пользователь удалит все, что было вставлено, или уменьшит фрагментацию, вызвав сборщик мусора в Lua с помощью функции collectgarbage.

Как правило, это не влияет на производительность. Однако, чтобы заставить Tarantool высвободить память, можно вызвать *box.snapshot*, остановить экземпляр и перезапустить его.

Профилирование производительности

Иногда Tarantool может работать медленнее, чем обычно. Причин такого поведения может быть несколько: проблемы с диском, Lua-скрипты, активно использующие процессор, или неправильная настройка. В таких случаях в журнале Tarantool'а могут отсутствовать необходимые подробности, поэтому единственным признаком неправильного поведения является наличие в журнале записей вида

`W> too long DELETE: 8.546 sec`. Ниже приведены инструменты и приемы, которые облегчают снятие профиля производительности Tarantool'a. Эта процедура может помочь при решении проблем с замедлением.

Примечание: Большинство инструментов, за исключением `fiber.info()`, предназначено для дистрибутивов GNU/Linux, но не для FreeBSD или Mac OS.

---

fiber.info()

Самый простой способ профилирования – это использование встроенных функций Tarantool'a. *fiber.info()* возвращает информацию обо всех работающих файберах с соответствующей трассировкой стека для языка C. Эти данные показывают, сколько файберов запущенно на данный момент и какие функции, написанные на C, вызываются чаще остальных.

Сначала войдите в интерактивную административную консоль вашего экземпляра Tarantool'a:

```
$ tarantoolctl enter NAME
```

После этого загрузите модуль `fiber`:

```
tarantool> fiber = require('fiber')
```

Теперь можно получить необходимую информацию с помощью *fiber.info()*.

На этом шаге в вашей консоли должно выводиться следующее:

```
tarantool> fiber = require('fiber')
---
...
tarantool> fiber.info()
---
- 360:
    csw: 2098165
    backtrace:
    - '#0 0x4d1b77 in wal_write(journal*, journal_entry*)+487'
    - '#1 0x4bbf68 in txn_commit(txn*)+152'
    - '#2 0x4bd5d8 in process_rw(request*, space*, tuple**)+136'
    - '#3 0x4bed48 in box_process1+104'
    - '#4 0x4d72f8 in lbox_replace+120'
    - '#5 0x50f317 in lj_BC_FUNCC+52'
    fid: 360
    memory:
      total: 61744
      used: 480
    name: main
  129:
    csw: 113
    backtrace: []
    fid: 129
    memory:
      total: 57648
      used: 0
    name: 'console/unix/:'
...
```

Мы рекомендуем присваивать создаваемым файберам понятные имена, чтобы их можно было легко найти в списке, выводимом `fiber.info()`. В примере ниже создается файбер с именем `myworker`:

```
tarantool> fiber = require('fiber')
---
...
tarantool> f = fiber.create(function() while true do fiber.sleep(0.5) end end)
---
...
tarantool> f:name('myworker') <!-- присваивание имени файберу
---
...
tarantool> fiber.info()
---
- 102:
    csw: 14
    backtrace:
    - '#0 0x501a1a in fiber_yield_timeout+90'
    - '#1 0x4f2008 in lbox_fiber_sleep+72'
    - '#2 0x5112a7 in lj_BC_FUNCC+52'
    fid: 102
    memory:
      total: 57656
      used: 0
    name: myworker <!-- новый созданный фоновый файбер
  101:
    csw: 284
    backtrace: []
    fid: 101
    memory:
      total: 57656
      used: 0
    name: interactive
...
```

Для принудительного завершения файбера используется команда *fiber.kill(fid)*:

```
tarantool> fiber.kill(102)
---
...
tarantool> fiber.info()
---
- 101:
    csw: 324
    backtrace: []
    fid: 101
    memory:
      total: 57656
      used: 0
    name: interactive
...
```

Если вам необходимо динамически получать информацию с помощью `fiber.info()`, вам может пригодиться приведенный ниже скрипт. Он каждые полсекунды подключается к экземпляру Tarantool'а, указанному в переменной `NAME`, выполняет команду `fiber.info()` и записывает ее выход в файл `fiber-info.txt`:

```
$ rm -f fiber.info.txt
$ watch -n 0.5 "echo 'require(\"fiber\").info()' | tarantoolctl enter NAME | tee -a fiber-info.txt"
```

Если вы не можете самостоятельно разобраться, какой именно файбер вызывает проблемы с производительностью, запустите данный скрипт на 10-15 секунд и пришлите получившийся файл команде Tarantool'а на адрес support@tarantool.org.

Простейшие профилировщики

pstack <pid>

Чтобы использовать этот инструмент, его необходимо установить с помощью пакетного менеджера, поставляемого с вашим дистрибутивом Linux. Данная команда выводит трассировку стека выполнения для работающего процесса с соответствующим PID. При необходимости команду можно запустить несколько раз, чтобы выявить узкое место, которое вызывает падение производительности.

После установки воспользуйтесь следующей командой:

```
$ pstack $(pidof tarantool INSTANCENAME.lua)
```

Затем выполните:

```
$ echo $(pidof tarantool INSTANCENAME.lua)
```

чтобы вывести на экран PID экземпляра Tarantool'а, использующего файл `INSTANCENAME.lua`.

В вашей консоли должно отображаться приблизительно следующее:

```
Thread 19 (Thread 0x7f09d1bff700 (LWP 24173)):
#0 0x00007f0a1a5423f2 in ?? () from /lib64/libgomp.so.1
#1 0x00007f0a1a53fdc0 in ?? () from /lib64/libgomp.so.1
#2 0x00007f0a1ad5adc5 in start_thread () from /lib64/libpthread.so.0
#3 0x00007f0a1a050ced in clone () from /lib64/libc.so.6
Thread 18 (Thread 0x7f09d13fe700 (LWP 24174)):
#0 0x00007f0a1a5423f2 in ?? () from /lib64/libgomp.so.1
#1 0x00007f0a1a53fdc0 in ?? () from /lib64/libgomp.so.1
#2 0x00007f0a1ad5adc5 in start_thread () from /lib64/libpthread.so.0
#3 0x00007f0a1a050ced in clone () from /lib64/libc.so.6
<...>
Thread 2 (Thread 0x7f09c8bfe700 (LWP 24191)):
#0 0x00007f0a1ad5e6d5 in pthread_cond_wait@@GLIBC_2.3.2 () from /lib64/libpthread.so.0
#1 0x0000000000045d901 in wal_writer_pop(wal_writer*) ()
#2 0x0000000000045db01 in wal_writer_f(__va_list_tag*) ()
#3 0x0000000000429abc in fiber_cxx_invoke(int (*)(__va_list_tag*), __va_list_tag*) ()
#4 0x00000000004b52a0 in fiber_loop ()
#5 0x00000000006099cf in coro_init ()
Thread 1 (Thread 0x7f0a1c47fd80 (LWP 24172)):
#0 0x00007f0a1a0512c3 in epoll_wait () from /lib64/libc.so.6
#1 0x00000000006051c8 in epoll_poll ()
#2 0x0000000000607533 in ev_run ()
#3 0x0000000000428e13 in main ()
```

gdb -ex «bt» -p <pid>

Как и в случае с `pstack`, перед использованием GNU-отладчик (также известный как `gdb`) необходимо сначала установить через пакетный менеджер, встроенный в ваш дистрибутив Linux.

После установки воспользуйтесь следующей командой:

```
$ gdb -ex "set pagination 0" -ex "thread apply all bt" --batch -p $(pidof tarantool INSTANCENAME.
↪lua)
```

Затем выполните:

```
$ echo $(pidof tarantool INSTANCENAME.lua)
```

чтобы вывести на экран PID экземпляра Tarantool'а, использующего файл `INSTANCENAME.lua`.

После использования отладчика в консоль должна выводиться следующая информация:

```
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

[CUT]

Thread 1 (Thread 0x7f72289ba940 (LWP 20535)):
#0 _int_malloc (av=av@entry=0x7f7226e0eb20 <main_arena>, bytes=bytes@entry=504) at malloc.c:3697
#1 0x00007f7226acf21a in __libc_calloc (n=<optimized out>, elem_size=<optimized out>) at malloc.
↪c:3234
#2 0x00000000004631f8 in vy_merge_iterator_reserve (capacity=3, itr=0x7f72264af9e0) at /usr/src/
↪tarantool/src/box/vinyl.c:7629
#3 vy_merge_iterator_add (itr=itr@entry=0x7f72264af9e0, is_mutable=is_mutable@entry=true, belong_
↪range=belong_range@entry=false) at /usr/src/tarantool/src/box/vinyl.c:7660
#4 0x00000000004703df in vy_read_iterator_add_mem (itr=0x7f72264af990) at /usr/src/tarantool/src/
↪box/vinyl.c:8387
#5 vy_read_iterator_use_range (itr=0x7f72264af990) at /usr/src/tarantool/src/box/vinyl.c:8453
#6 0x000000000047657d in vy_read_iterator_start (itr=<optimized out>) at /usr/src/tarantool/src/
↪box/vinyl.c:8501
#7 0x00000000004766b5 in vy_read_iterator_next (itr=itr@entry=0x7f72264af990,␣
↪result=result@entry=0x7f72264afad8) at /usr/src/tarantool/src/box/vinyl.c:8592
#8 0x000000000047689d in vy_index_get (tx=tx@entry=0x7f7226468158, index=index@entry=0x2563860,␣
↪key=<optimized out>, part_count=<optimized out>, result=result@entry=0x7f72264afad8) at /usr/src/
↪tarantool/src/box/vinyl.c:5705
#9 0x0000000000477601 in vy_replace_impl (request=<optimized out>, request=<optimized out>,␣
↪stmt=0x7f72265a7150, space=0x2567ea0, tx=0x7f7226468158) at /usr/src/tarantool/src/box/vinyl.
↪c:5920
#10 vy_replace (tx=0x7f7226468158, stmt=stmt@entry=0x7f72265a7150, space=0x2567ea0, request=
↪<optimized out>) at /usr/src/tarantool/src/box/vinyl.c:6608
#11 0x00000000004615a9 in VinylSpace::executeReplace (this=<optimized out>, txn=<optimized out>,␣
↪space=<optimized out>, request=<optimized out>) at /usr/src/tarantool/src/box/vinyl_space.cc:108
#12 0x00000000004bd723 in process_rw (request=request@entry=0x7f72265a70f8,␣
↪space=space@entry=0x2567ea0, result=result@entry=0x7f72264afbc8) at /usr/src/tarantool/src/box/
↪box.cc:182
#13 0x00000000004bed48 in box_process1 (request=0x7f72265a70f8,␣
↪result=result@entry=0x7f72264afbc8) at /usr/src/tarantool/src/box/box.cc:700
#14 0x00000000004bf389 in box_replace (space_id=space_id@entry=513, tuple=<optimized out>, tuple_
↪end=<optimized out>, result=result@entry=0x7f72264afbc8) at /usr/src/tarantool/src/box/box.cc:754
#15 0x00000000004d72f8 in lbox_replace (L=0x413c5780) at /usr/src/tarantool/src/box/lua/index.c:72
#16 0x000000000050f317 in lj_BC_FUNCC ()
#17 0x00000000004d37c7 in execute_lua_call (L=0x413c5780) at /usr/src/tarantool/src/box/lua/call.
↪c:282
#18 0x000000000050f317 in lj_BC_FUNCC ()
#19 0x0000000000529c7b in lua_cpcall ()
#20 0x00000000004f6aa3 in luaT_cpcall (L=L@entry=0x413c5780, func=func@entry=0x4d36d0 <execute_lua_
↪call>, ud=ud@entry=0x7f72264afde0) at /usr/src/tarantool/src/lua/utils.c:962
#21 0x00000000004d3fe7 in box_process_lua (handler=0x4d36d0 <execute_lua_call>,␣
↪out=out@entry=0x7f7213020600, request=request@entry=0x413c5780) at /usr/src/tarantool/src/box/
↪lua/call.c:382
```

(continues on next page)

(продолжение с предыдущей страницы)

```
#22 box_lua_call (request=request@entry=0x7f7213040 1d8, out=out@entry=0x7f7213020600) at /usr/src/
↪tarantool/src/box/lua/call.c:405
#23 0x00000000004c0f27 in box_process_call (request=request@entry=0x7f7213040 1d8,␣
↪out=out@entry=0x7f7213020600) at /usr/src/tarantool/src/box/box.cc:1074
#24 0x000000000041326c in tx_process_misc (m=0x7f7213040 0170) at /usr/src/tarantool/src/box/iproto.
↪cc:942
#25 0x0000000000504554 in cmsg_deliver (msg=0x7f7213040 0170) at /usr/src/tarantool/src/cbus.c:302
#26 0x0000000000504c2e in fiber_pool_f (ap=<error reading variable: value has been optimized out>)␣
↪at /usr/src/tarantool/src/fiber_pool.c:64
#27 0x000000000041122c in fiber_cxx_invoke(fiber_func, typedef __va_list_tag __va_list_tag *) (f=
↪<optimized out>, ap=<optimized out>) at /usr/src/tarantool/src/fiber.h:645
#28 0x00000000005011a0 in fiber_loop (data=<optimized out>) at /usr/src/tarantool/src/fiber.c:641
#29 0x0000000000688fbf in coro_init () at /usr/src/tarantool/third_party/coro/coro.c:110
```

Запустите отладчик в цикле, чтобы собрать достаточно информации, которая поможет установить причину спада производительности Tarantool'a. Можно воспользоваться следующим скриптом:

```
$ rm -f stack-trace.txt
$ watch -n 0.5 "gdb -ex 'set pagination 0' -ex 'thread apply all bt' --batch -p $(pidof tarantool␣
↪INSTANCENAME.lua) | tee -a stack-trace.txt"
```

С точки зрения структуры и функциональности, этот скрипт идентичен тому, что используется выше с `fiber.info()`.

Если вам не удается отыскать причину пониженной производительности, запустите данный скрипт на 10-15 секунд и пришлите получившийся файл `stack-trace.txt` команде Tarantool'a на адрес support@tarantool.org.

Предупреждение: Следует использовать `pstack` и `gdb` с осторожностью: каждый раз, подключаясь с работающему процессу, они приостанавливают выполнение этого процесса приблизительно на одну секунду, что может иметь серьезные последствия для высоконагруженных сервисов.

gperftools

Чтобы использовать профилировщик процессора из набора Google Performance Tools с Tarantool'ом, необходимо сначала установить зависимости:

- Если вы используете Debian/Ubuntu, запустите эту команду:

```
$ apt-get install libgoogle-perftools4
```

- Если вы используете RHEL/CentOS/Fedora, запустите эту команду:

```
$ yum install gperftools-libs
```

После этого установите привязки для Lua:

```
$ tarantoolctl rocks install gperftools
```

После окончания установки войдите в интерактивную административную консоль вашего экземпляра Tarantool'a:

```
$ tarantoolctl enter NAME
```

Для запуска профилировщика выполните следующий код:

```
tarantool> cpuprof = require('gperftools.cpu')
tarantool> cpuprof.start('/home/<имя_пользователя>/tarantool-on-production.prof')
```

На сбор метрик производительности у профилировщика уходит по крайней мере пара минут. По истечении этого времени можно сохранять информацию на диск (неограниченное количество раз):

```
tarantool> cpuprof.flush()
```

Для остановки профилировщика выполните следующую команду:

```
tarantool> cpuprof.stop()
```

Теперь можно проанализировать собранные данные с помощью утилиты `pprof`, которая входит в пакет `gperftools`:

```
$ pprof --text /usr/bin/tarantool /home/<имя_пользователя>/tarantool-on-production.prof
```

Примечание: В дистрибутивах Debian/Ubuntu утилита `pprof` называется `google-pprof`.

В консоль должно выводиться приблизительно следующее:

```
Total: 598 samples
      83 13.9% 13.9% 83 13.9% epoll_wait
      54 9.0% 22.9% 102 17.1%
vy_mem_tree_insert.constprop.35
      32 5.4% 28.3% 34 5.7% __write_nocancel
      28 4.7% 32.9% 42 7.0% vy_mem_iterator_start_from
      26 4.3% 37.3% 26 4.3% _IO_str_seekoff
      21 3.5% 40.8% 21 3.5% tuple_compare_field
      19 3.2% 44.0% 19 3.2%
::TupleCompareWithKey::compare
      19 3.2% 47.2% 38 6.4% tuple_compare_slowpath
      12 2.0% 49.2% 23 3.8% __libc_calloc
       9 1.5% 50.7% 9 1.5%
::TupleCompare::compare@42efc0
       9 1.5% 52.2% 9 1.5% vy_cache_on_write
       9 1.5% 53.7% 57 9.5% vy_merge_iterator_next_key
       8 1.3% 55.0% 8 1.3% __nss_passwd_lookup
       6 1.0% 56.0% 25 4.2% gc_onestep
       6 1.0% 57.0% 6 1.0% lj_tab_next
       5 0.8% 57.9% 5 0.8% lj_alloc_malloc
       5 0.8% 58.7% 131 21.9% vy_prepare
```

perf

Этот инструмент для мониторинга и анализа производительности устанавливается отдельно с помощью пакетного менеджера. Попробуйте ввести в окне консоли команду `perf` и следуйте подсказкам, чтобы установить необходимые пакеты.

Примечание: По умолчанию некоторые команды из пакета `perf` можно выполнять только с **root**-правами, поэтому необходимо либо зайти в систему из-под пользователя **root**, либо добавлять перед

каждой командой sudo.

Чтобы начать сбор показателей производительности, выполните следующую команду:

```
$ perf record -g -p $(pidof tarantool INSTANCENAME.lua)
```

Эта команда сохраняет собранные данные в файл perf.data, который находится в текущей рабочей папке. Для остановки процесса (обычно через 10-15 секунд) нажмите **ctrl**+**C**. В консоли должно появиться следующее:

```
^C[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.225 MB perf.data (1573 samples) ]
```

Затем выполните эту команду:

```
$ perf report -n -g --stdio | tee perf-report.txt
```

Она превращает содержащиеся в perf.data статистические данные в отчет о производительности, который сохраняется в файл perf-report.txt.

Получившийся отчет выглядит следующим образом:

```
# Samples: 14K of event 'cycles'
# Event count (approx.): 9927346847
#
# Children Self Samples Command Shared Object Symbol
# ........ ........ ............ ......... ................... ....................................
↪..
#
    35.50% 0.55% 79 tarantool tarantool [.] lj_gc_step
            |
             --34.95%--lj_gc_step
                       |
                       |--29.26%--gc_onestep
                       | |
                       | |--13.85%--gc_sweep
                       | | |
                       | | |--5.59%--lj_alloc_free
                       | | |
                       | | |--1.33%--lj_tab_free
                       | | | |
                       | | |  --1.01%--lj_alloc_free
                       | | |
                       | |  --1.17%--lj_cdata_free
                       | |
                       | |--5.41%--gc_finalize
                       | | |
                       | | |--1.06%--lj_obj_equal
                       | | |
                       | |  --0.95%--lj_tab_set
                       | |
                       | |--4.97%--rehashtab
                       | | |
                       | |  --3.65%--lj_tab_resize
                       | | |
                       | | |--0.74%--lj_tab_set
                       | | |
```

(continues on next page)

(продолжение с предыдущей страницы)

```
                        | | --0.72%--lj_tab_newkey
                        | |
                        | |--0.91%--propagatemark
                        | |
                        | --0.67%--lj_cdata_free
                        |
                         --5.43%--propagatemark
                                   |
                                    --0.73%--gc_mark
```

Инструменты `gperftools` и `perf` отличаются от `pstack` и `gdb` низкой затратой ресурсов (пренебрежимо малой по сравнению с `pstack` и `gdb`): они подключаются к работающим процессам без больших задержек, а потому могут использоваться без серьезных последствий.

jit.p

Профилировщик «jit.p» входит в комплект сервера приложений Tarantool'а. Чтобы загрузить его, выполните команду `require('jit.p')` или `require('jit.profile')`. Есть много параметров для настройки выборки и вывода, они описаны в документации по Профилировщику LuaJIT.

Пример

Создайте функцию для вызова функции под названием f1, которая осуществляет 500 000 вставок и удалений в спейсе Tarantool'а. Запустите профилировщик, выполните функцию, завершите работу профилировщика. Получите результат выборки профилировщика.

```
box.space.t:drop()
box.schema.space.create('t')
box.space.t:create_index('i')
function f1() for i = 1,500000 do
  box.space.t:insert{i}
  box.space.t:delete{i}
  end
return 1
end
function f3() f1() end
jit_p = require("jit.profile")
sampletable = {}
jit_p.start("f", function(thread, samples, vmstate)
  local dump=jit_p.dumpstack(thread, "f", 1)
  sampletable[dump] = (sampletable[dump] or 0) + samples
end)
f3()
jit_p.stop()
for d,v in pairs(sampletable) do print(v, d) end
```

Как правило, результат покажет, что выборка многократно осуществлялась в рамках f1(), а также в рамках внутренних функций Tarantool'а, имена которых могут изменяться с каждой новой версией.

4.5.6 Контроль за фоновыми программами

Сигналы от сервера

Во время событийного цикла в потоке обработки транзакций Tarantool обрабатывает следующие сигналы:

Сигнал	Эффект
SIGHUP	Может привести к ротации журналов, см. *пример* в справочнике по параметрам журналирования Tarantool'а.
SIGUSR1	Может привести к созданию снимка состояния базы данных, см. описание функции *box.snapshot*.
SIGTERM	Может привести к корректному завершению работы (с предварительным сохранением всех данных).
SIGINT (или «прерывание от клавиатуры»)	Может привести к корректному завершению работы.
SIGKILL	Приводит к аварийному завершению работы.

Остальные сигналы приводят к заданному операционной системой поведению. Все сигналы, за исключением SIGKILL, можно игнорировать, особенно если Tarantool выполняет длительную процедуру и не может вернуться в событийный цикл в потоке обработки транзакций.

Автоматическая перезагрузка экземпляра

На платформах, где доступна утилита `systemd`, `systemd` автоматически перезагружает все экземпляры Tarantool'а при сбое. Чтобы продемонстрировать это, отключим один из экземпляров:

```
$ systemctl status tarantool@my_app|grep PID
Main PID: 5885 (tarantool)
$ tarantoolctl enter my_app
/bin/tarantoolctl: Found my_app.lua in /etc/tarantool/instances.available
/bin/tarantoolctl: Connecting to /var/run/tarantool/my_app.control
/bin/tarantoolctl: connected to unix/:/var/run/tarantool/my_app.control
unix/:/var/run/tarantool/my_app.control> os.exit(-1)
/bin/tarantoolctl: unix/:/var/run/tarantool/my_app.control: Remote host closed connection
```

А теперь убедимся, что `systemd` перезапустила его:

```
$ systemctl status tarantool@my_app|grep PID
Main PID: 5914 (tarantool)
```

И под конец проверим содержимое журнала загрузки:

```
$ journalctl -u tarantool@my_app -n 8
-- Записи начинаются в пятницу 08.01.2016 12:21:53 MSK, заканчиваются в четверг 21.01.2016 2016-01-
↪21 21:09:45 MSK. --
Jan 21 21:09:45 localhost.localdomain systemd[1]: tarantool@my_app.service: Unit entered failed␣
↪state.
Jan 21 21:09:45 localhost.localdomain systemd[1]: tarantool@my_app.service: Failed with result
↪'exit-code'.
Jan 21 21:09:45 localhost.localdomain systemd[1]: tarantool@my_app.service: Service hold-off time␣
↪over, scheduling restart.
Jan 21 21:09:45 localhost.localdomain systemd[1]: Stopped Tarantool Database Server.
Jan 21 21:09:45 localhost.localdomain systemd[1]: Starting Tarantool Database Server...
Jan 21 21:09:45 localhost.localdomain tarantoolctl[5910]: /usr/bin/tarantoolctl: Found my_app.lua␣
↪in /etc/tarantool/instances.available
```

(continues on next page)

(продолжение с предыдущей страницы)

```
Jan 21 21:09:45 localhost.localdomain tarantoolctl[5910]: /usr/bin/tarantoolctl: Starting instance.
↪..
Jan 21 21:09:45 localhost.localdomain systemd[1]: Started Tarantool Database Server.
```

Создание дампов памяти

Tarantool создает дамп памяти при получении одного из следующих сигналов: SIGSEGV, SIGFPE, SIGABRT или SIGQUIT. При сбое Tarantool'а дамп создается автоматически.

На платформах, где доступна утилита `systemd`, `coredumpctl` автоматически сохраняет дампы памяти и трассировку стека при аварийном завершении Tarantool-сервера. Вот как включить создание дампов памяти в Unix-системе:

1. Убедитесь, что лимиты для сессии установлены таким образом, чтобы можно было создавать дампы памяти, – выполните команду `ulimit -c unlimited`. Также проверьте «man 5 core» на другие причины, по которым дамп памяти может не создаваться.
2. Создайте директорию для записи дампов памяти и убедитесь, что в эту директорию действительно можно производить запись. На Linux путь до директории задается в параметре ядра, который настраивается через `/proc/sys/kernel/core_pattern`.
3. Убедитесь, что дампы памяти включают трассировку стека. При использовании бинарного дистрибутива Tarantool'а эта информация включается автоматически. При сборке Tarantool'а из исходников, если передать CMake флаг `-DCMAKE_BUILD_TYPE=Release`, вы не получите подробной информации.

Для симуляции сбоя можно попытаться выполнить нелегальную команду на работающем экземпляре Tarantool'а:

```
$ # !!! пожалуйста, никогда не делайте этого на боевом сервере !!!
$ tarantoolctl enter my_app
unix/:/var/run/tarantool/my_app.control> require('ffi').cast('char *', 0)[0] = 48
/bin/tarantoolctl: unix/:/var/run/tarantool/my_app.control: Remote host closed connection
```

Есть другой способ: если вы знаете PID экземпляра ($PID в нашем примере), можно остановить этот экземпляр, запустив отладчик `gdb`:

```
$ gdb -batch -ex "generate-core-file" -p $PID
```

или послав вручную сигнал SIGABRT:

```
$ kill -SIGABRT $PID
```

Примечание: Чтобы узнать PID экземпляра, можно:

- посмотреть его с помощью *box.info.pid*,
- использовать команду `ps -A | grep tarantool`, или
- выполнить `systemctl status tarantool@my_app|grep PID`.

Чтобы посмотреть на последние сбои Tarantool-демона на платформах, где доступна утилита `systemd`, выполните команду:

```
$ coredumpctl list /usr/bin/tarantool
MTIME                            PID   UID   GID SIG PRESENT EXE
Sat 2016-01-23 15:21:24 MSK   20681  1000  1000   6   /usr/bin/tarantool
Sat 2016-01-23 15:51:56 MSK   21035   995   992   6   /usr/bin/tarantool
```

Чтобы сохранить дамп памяти в файл, выполните команду:

```
$ coredumpctl -o filename.core info <pid>
```

Трассировка стека

Так как Tarantool хранит кортежи в памяти, файлы с дампами памяти могут быть довольно большими. Чтобы найти проблему, обычно целый файл не нужен – достаточно только «трассировки стека» или «обратной трассировки».

Чтобы сохранить трассировку стека в файл, выполните команду:

```
$ gdb -se "tarantool" -ex "bt full" -ex "thread apply all bt" --batch -c core> /tmp/tarantool_
↪trace.txt
```

где:

- «tarantool» – это путь до исполняемого файла Tarantool'а,
- «core» – это путь до файла с дампом памяти, и
- «/tmp/tarantool_trace.txt» – это пример пути до файла, в который сохраняется трассировка стека.

Примечание: Иногда может оказаться, что файл с трассировкой стека не содержит отладочных символов – в таких строках вместо имени будет стоять "??". Если это произошло, ознакомьтесь с инструкциями на этих двух wiki-страницах Tarantool'а: How to debug core dump of stripped tarantool и How to debug core from different OS.

Чтобы получить трассировку стека и прочую полезную информацию в консоли, выполните команду:

```
$ coredumpctl info 21035
          PID: 21035 (tarantool)
          UID: 995 (tarantool)
          GID: 992 (tarantool)
       Signal: 6 (ABRT)
    Timestamp: Sat 2016-01-23 15:51:42 MSK (4h 36min ago)
 Command Line: tarantool my_app.lua <running>
   Executable: /usr/bin/tarantool
Control Group: /system.slice/system-tarantool.slice/tarantool@my_app.service
         Unit: tarantool@my_app.service
        Slice: system-tarantool.slice
      Boot ID: 7c686e2ef4dc4e3ea59122757e3067e2
   Machine ID: a4a878729c654c7093dc6693f6a8e5ee
     Hostname: localhost.localdomain
      Message: Process 21035 (tarantool) of user 995 dumped core.

               Stack trace of thread 21035:
               #0  0x00007f84993aa618 raise (libc.so.6)
               #1  0x00007f84993ac21a abort (libc.so.6)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
#2  0x0000560d0a9e9233 _ZL12sig_fatal_cbi (tarantool)
#3  0x00007f849a211220 __restore_rt (libpthread.so.0)
#4  0x0000560d0aaa5d9d lj_cconv_ct_ct (tarantool)
#5  0x0000560d0aaa687f lj_cconv_ct_tv (tarantool)
#6  0x0000560d0aaabe33 lj_cf_ffi_meta___newindex (tarantool)
#7  0x0000560d0aaae2f7 lj_BC_FUNCC (tarantool)
#8  0x0000560d0aa9aabd lua_pcall (tarantool)
#9  0x0000560d0aa71400 lbox_call (tarantool)
#10 0x0000560d0aa6ce36 lua_fiber_run_f (tarantool)
#11 0x0000560d0a9e8d0c _ZL16fiber_cxx_invokePFiP13__va_list_tagES0_ (tarantool)
#12 0x0000560d0aa7b255 fiber_loop (tarantool)
#13 0x0000560d0ab38ed1 coro_init (tarantool)
...
```

Отладчик

Для запуска отладчика gdb, выполните команду:

```
$ coredumpctl gdb <pid>
```

Мы очень рекомендуем установить пакет `tarantool-debuginfo`, чтобы сделать отладку средствами gdb более эффективной. Например:

```
$ dnf debuginfo-install tarantool
```

С помощью gdb можно узнать, какие еще `debuginfo`-пакеты нужно установить:

```
$ gdb -p <pid>
...
Missing separate debuginfos, use: dnf debuginfo-install
glibc-2.22.90-26.fc24.x86_64 krb5-libs-1.14-12.fc24.x86_64
libgcc-5.3.1-3.fc24.x86_64 libgomp-5.3.1-3.fc24.x86_64
libselinux-2.4-6.fc24.x86_64 libstdc++-5.3.1-3.fc24.x86_64
libyaml-0.1.6-7.fc23.x86_64 ncurses-libs-6.0-1.20150810.fc24.x86_64
openssl-libs-1.0.2e-3.fc24.x86_64
```

В трассировке стека присутствуют символические имена, даже если у вас не установлен пакет `tarantool-debuginfo`.

4.5.7 Аварийное восстановление

Минимальная отказоустойчивая конфигурация Tarantool'а – это *репликационный кластер*, содержащий мастер и реплику или два мастера.

Основная рекомендация – настраивать все экземпляры Tarantool'а в кластере таким образом, чтобы они регулярно создавали *файлы-снимки*.

Ниже дано несколько инструкций для типовых аварийных сценариев.

Мастер-реплика

Конфигурация: один мастер и одна реплика.

Проблема: мастер вышел из строя.

План действий:

1. Убедитесь, что мастер полностью остановлен. Например, подключитесь к мастеру и используйте команду `systemctl stop tarantool@<имя_экземпляра>`.
2. Переключите реплику в режим мастера, установив параметру *box.cfg.read_only* значение *false*. Теперь вся нагрузка пойдет только на реплику (по сути ставшую мастером).
3. Настройте на свободной машине замену вышедшему из строя мастеру, установив параметру *replication* в качестве значения URI реплики (которая в данный момент выполняет роль мастера), чтобы новая реплика начала синхронизироваться с текущим мастером. Значение параметра *box.cfg.read_only* в новом экземпляре должно быть установлено на *true*.

Все немногочисленные транзакции в *WAL-файле* мастера, которые он не успел передать реплике до выхода из строя, будут потеряны. Однако если удастся получить .xlog-файл мастера, их можно будет восстановить. Для этого:

1. Узнайте позицию вышедшего из строя мастера – эта информация доступна из нового мастера.
   a. Посмотрите UUID экземпляра в *xlog-файле* вышедшего из строя мастера:

   ```
   $ head -5 *.xlog | grep Instance
   Instance: ed607cad-8b6d-48d8-ba0b-dae371b79155
   ```

   b. Используйте этот UUID на новом мастере для поиска позиции:

   ```
   tarantool> box.info.vclock[box.space._cluster.index.uuid:select{'ed607cad-8b6d-48d8-ba0b-
   ↪dae371b79155'}[1][1]]
   ---
   - 23425
   <...>
   ```

2. Запишите транзакции из .xlog-файла вышедшего из строя мастера в новый мастер, начиная с позиции нового мастера:
   a. Локально выполните эту команду на новом мастере, чтобы узнать его ID экземпляра:

   ```
   tarantool> box.space._cluster:select{}
   ---
   - - [1, '88580b5c-4474-43ab-bd2b-2409a9af80d2']
   ...
   ```

   b. Запишите транзакции в новый мастер:

   ```
   $ tarantoolctl <uri_нового_мастера> <xlog_файл> play --from 23425 --replica 1
   ```

Мастер-мастер

Конфигурация: два мастера.

Проблема: мастер #1 вышел из строя.

План действий:

1. Пусть вся нагрузка идет только на мастер #2 (действующий мастер).

2. Follow the same steps as in the *master-replica* recovery scenario to create a new master and salvage lost data.

Потеря данных

Конфигурация: мастер-мастер или мастер-реплика.

Проблема: данные были удалены на одном мастере, а затем эти изменения реплицировались на другом узле (мастере или реплике).

Эта инструкция применима только для данных, хранящихся на движке memtx. План действий:

1. Перевести все узлы в режим *read-only* и не разрешать функции *box.backup.start()* удалять старые контрольные точки. Это не даст сборщику мусора в Tarantool удалять файлы, созданные во время предыдущих контрольных точек, до тех пор пока не будет вызвана функция *box.backup.stop()*.
2. Возьмите последний корректный *.snap-файл* и, используя команду `tarantoolctl cat`, выясните, на каком именно lsn произошла потеря данных.
3. Запустите новый экземпляр (экземпляр #1) и с помощью команды `tarantoolctl play` скопируйте в него содержимое .snap/.xlog-файлов вплоть до вычисленного lsn.
4. Настройте новую реплику с помощью восстановленного мастера (экземпляра #1).

4.5.8 Резервное копирование

Архитектура Tarantool-хранилища позволяет производить обновление только путем присоединения новых записей: сами файлы никогда не перезаписываются. *Сборщик мусора Tarantool'а* удаляет старые файлы после определенной контрольной точки. В настройках *демона создания контрольных точек* можно отложить или запретить работу сборщика мусора. Резервное копирование может проводиться в любое время с минимальной затратой ресурсов.

Для резервного копирования в определенных ситуациях используются две функции:

- *box.backup.start()* сообщает серверу, что следует отложить некоторые действия, связанные с удалением устаревших резервных копий, и возвращает таблицу с именами файлов снимков и файлов vinyl'а, которые необходимо будет скопировать.
- *box.backup.stop()* затем уведомляет сервер, что работа может быть продолжена в обычном режиме.

Горячее резервирование (memtx)

Это особый случай, когда все таблицы хранятся в памяти.

Последний созданный Tarantool'ом *файл-снимок* является резервной копией всей базы данных; а *WAL-файлы*, созданные следом за последним файлом-снимком, являются инкрементными копиями. Поэтому процедура резервирования сводится к копированию последнего файла-снимка и следующих за ним WAL-файлов.

1. С помощью `tar` создайте (зачастую сжатую) копию последнего .snap-файла и следующих за ним .xlog-файлов из директорий *memtx_dir* и *wal_dir*.
2. Если того требуют правила безопасности, зашифруйте получившийся .tar-файл.
3. Скопируйте .tar-файл в надежное место.

В дальнейшем базу данных можно восстановить, разархивировав содержимое .tar-файла в директории `memtx_dir` и `wal_dir`.

Горячее резервирование (vinyl/memtx)

Vinyl хранит свои файлы в *vinyl_dir* и создает для каждого спейса в базе данных отдельную поддиректорию. Создание дампов и слияние – это процессы, которые могут лишь добавлять записи, поэтому в результате создаются новые файлы. Сборщик мусора Tarantool'а может удалять старые файлы после каждой контрольной точки.

Для создания смешанной резервной копии:

1. Выполните команду *box.backup.start()* в *административной консоли*. Эта команда покажет список файлов для резервирования и приостановит сборку мусора до следующего вызова `box.backup.stop()`.
2. Скопируйте файлы из списка в надежное место. Это касается файлов-снимков memtx, выполняемых vinyl-файлов и индексных файлов, соответствующих последней контрольной точке.
3. Выполните команду `box.backup.stop()`, чтобы сборщик мусора мог продолжить работу.

Непрерывное удаленное резервирование

Репликация используется не только для резервирования, но и для выравнивания нагрузки.

Поэтому процесс создания резервной копии сводится к обновлению (при необходимости) одной из реплик с последующим холодным резервированием. Так как все остальные реплики продолжают функционировать, с точки зрения конечного пользователя, этот процесс не является холодным резервированием. Такое резервирование можно выполнять регулярно с помощью планировщика `cron` или файбера Tarantool'а.

Непрерывное резервирование

По ходу работы системы необходимо сохранять записи об изменениях, внесенных со времени последнего холодного резервирования.

Для этого нужна специальная утилита для копирования файлов (например, rsync), которая позволит удаленно и на постоянной основе копировать только изменившиеся части WAL-файла, а не весь файл целиком.

Можно взять и обычную утилиту для копирования целых файлов, но тогда придется создавать файлы-снимки и WAL-файлы на каждое изменение, чтобы нужно было копировать только новые файлы.

4.5.9 Обновление

Обновление базы данных Tarantool

Если вы создали базу данных в старой версии Tarantool'а, а потом обновили Tarantool до более свежей версии, вызовите команду `box.schema.upgrade()`. Она обновляет системные спейсы Tarantool'а так, чтобы они совпадали с текущей установленной версией Tarantool'а.

Например, вот что происходит, если выполнить команду `box.schema.upgrade()` для базы данных, созданной в Tarantool версии 1.6.4 (показана лишь малая часть выводимых сообщений):

```
tarantool> box.schema.upgrade()
alter index primary on _space set options to {"unique":true}, parts to [[0,"unsigned"]]
alter space _schema set options to {}
create view _vindex...
```

(continues on next page)

(продолжение с предыдущей страницы)

```
grant read access to 'public' role for _vindex view
set schema version to 1.7.0
---
...
```

Обновление экземпляра Tarantool'a

Tarantool поддерживает обратную совместимость между двумя последовательными версиями. Например, обновление Tarantool 1.6 до 1.7 или Tarantool 1.7 до 2.x не должно вызвать затруднений, тогда как миграции с Tarantool 1.6 сразу на 2.x могут препятствовать несовместимые изменения.

How to upgrade from Tarantool 1.7 to 2.x

1. Остановите Tarantool-сервер.
2. Создайте копию всех данных (см. подразделы про горячее резервное копирование в разделе *Резервное копирование*) и пакета, из которого была установлена текущая (старая) версия (на случай отката).
3. Обновите Tarantool-сервер. Инструкции по установке доступны на странице загрузок Tarantool'a.
4. Запустите обновленный Tarantool-сервер с помощью `tarantoolctl` или `systemctl`.

Как обновить Tarantool 1.6 до 2.x

The procedure is fully analogous to *upgrading from 1.7 to 2.x*.

Как обновить Tarantool 1.6 до 1.7

Этот процесс предназначен для обновления индивидуальных экземпляров Tarantool'a с 1.6.x до 1.7.x на боевом сервере. Обратите внимание, что это **всегда приводит к некоторому простою**. Для обновления **без простоев** необходимо, чтобы несколько работающих Tarantool-серверов были объединены в репликационный кластер (см. *ниже*).

Tarantool 1.7 работает с несовместимыми форматами файлов – *.snap* и *.xlog*. Файлы Tarantool'a 1.6 поддерживаются при обновлении, но после непродолжительного использования Tarantool'a 1.7 вернуться к 1.6 уже нельзя. Также были переименованы некоторые конфигурационные параметры, но старые параметры еще поддерживаются. Список критических изменений доступен в Примечаниях к версиям Tarantool'a 1.7.

1. Уточните у разработчиков, необходимо ли обновлять файлы приложения из-за наличия несовместимых изменений (см. Примечания к версии Tarantool'a 1.7). Если да, то создайте резервные копии старых файлов приложения.
2. Остановите Tarantool-сервер.
3. Создайте копию всех данных (см. подразделы про горячее резервное копирование в разделе *Резервное копирование*) и пакета, из которого была установлена текущая (старая) версия (на случай отката).
4. Обновите Tarantool-сервер. Инструкции по установке доступны на странице загрузок Tarantool'a.

5. Обновите базу данных Tarantool. Выполните команду `box.schema.upgrade()`, поместив ее внутрь функции *box.once()* в *файле инициализации* Tarantool'a. В результате на этапе запуска Tarantool создаст новые системные спейсы, обновит названия типов данных (например, num -> unsigned, str -> string) и список доступных типов данных в системных спейсах.
6. При необходимости обновите файлы приложения.
7. Запустите обновленный Tarantool-сервер с помощью `tarantoolctl` или `systemctl`.

Обновление Tarantool'a в репликационном кластере

Tarantool 1.7 может служить *репликой* для Tarantool'a 1.6 – и наоборот. При установке соединения происходит обсуждение возможностей, и новые для 1.7 репликационные функции не используются при работе с репликами версии 1.6. Такой подход позволяет обновлять кластерные конфигурации.

Этот процесс позволяет осуществить последовательное обновление **без простоев** и подходит для любой конфигурации кластера: master-master или мастер-реплика.

1. Обновите Tarantool на всех репликах (или на любом мастере в кластере мастер-мастер). Подробные инструкции доступны в подразделе *Обновление экземпляра Tarantool'a*.
2. Проверьте работу реплик:
   a. Запустите Tarantool.
   b. Присоединитесь к мастеру и начните работать, как раньше.

   На мастере установлена старая версия Tarantool'a, которая всегда совместима со следующей мажорной версией.
3. Обновите мастер. Процесс такой же, как и при обновлении реплики.
4. Проверьте работу мастера:
   a. Запустите Tarantool в режиме реплики для получения последней версии данных.
   b. Переключитесь в режим мастера.
5. Обновите базу данных на любом мастере в кластере. Выполните команду `box.schema.upgrade()`. Это обновит системные спейсы Tarantool'a так, чтобы они совпадали с текущей установленной версией Tarantool'a. Изменения распространятся на другие узлы кластера через обычный механизм репликации.

4.5.10 Замечания по поводу некоторых операционных систем

Mac OS

Администрирование экземпляров Tarantool'a на Mac OS возможно только с помощью `tarantoolctl`. Встроенные системные инструменты не поддерживаются.

FreeBSD

Чтобы `tarantoolctl` и утилиты `init.d` работали на FreeBSD, используйте пути, отличные от предложенных в разделе *Настройка экземпляров Tarantool'a*. Используйте `/usr/local/etc/tarantool/` вместо `/usr/share/tarantool/` и создайте следующие поддиректории:

- `default` для хранения настроек `tarantoolctl` по умолчанию (см. пример ниже),
- `instances.available` для хранения всех доступных файлов экземпляра, и

- `instances.enabled` для хранения файлов экземпляра, которые необходимо запускать автоматически с помощью sysvinit.

Так выглядят настройки `tarantoolctl` по умолчанию на FreeBSD:

```
default_cfg = {
    pid_file   = "/var/run/tarantool", -- /var/run/tarantool/${INSTANCE}.pid
    wal_dir    = "/var/db/tarantool", -- /var/db/tarantool/${INSTANCE}/
    snap_dir   = "/var/db/tarantool", -- /var/db/tarantool/${INSTANCE}
    vinyl_dir = "/var/db/tarantool", -- /var/db/tarantool/${INSTANCE}
    logger     = "/var/log/tarantool", -- /var/log/tarantool/${INSTANCE}.log
    username   = "tarantool",
}

-- instances.available - все доступные экземпляры
-- instances.enabled - экземпляры для автоматического запуска через sysvinit
instance_dir = "/usr/local/etc/tarantool/instances.available"
```

Gentoo Linux

В разделе ниже описывается пакет «dev-db/tarantool», установленный из официального оверлея layman (под названием `tarantool`).

По умолчанию с экземплярами используется директория `/etc/tarantool/instances.available`, ее можно переопределить в `/etc/default/tarantool`.

Управление экземплярами Tarantool'а (запуск/остановка/перезагрузка/проверка статуса и т.д.) можно осуществлять с помощью OpenRC. Рассмотрим пример, как создать экземпляр с управлением OpenRC:

```
$ cd /etc/init.d
$ ln -s tarantool your_service_name
$ ln -s /usr/share/tarantool/your_service_name.lua /etc/tarantool/instances.available/your_service_
↪name.lua
```

Проверяем, что работает:

```
$ /etc/init.d/your_service_name start
$ tail -f -n 100 /var/log/tarantool/your_service_name.log
```

4.5.11 Сообщения об ошибках

Если вы нашли ошибку в Tarantool, вы окажете нам услугу, сообщив о ней.

Пожалуйста, откройте тикет в репозитории Tarantool на GitHub. Рекомендуем включить следующую информацию:

- Шаги для воспроизведения ошибки с объяснением того, как ошибочное поведение отличается от описанного в документации ожидаемого поведения. Пожалуйста, указывайте как можно более конкретную информацию. Например, вместо «Я не могу получить определенную информацию» лучше написать «box.space.x:delete() не указывает, что именно было удалено».
- Название и версию вашей операционной системы, название и версию Tarantool и любую информацию об особенностях вашей машины и ее конфигурации.
- Сопутствующие файлы – такие как *трассировка стека* или *файл журнала* Tarantool'а.

Если это запрос новой функции или это затрагивает определенную группу пользователей, не забудьте это указать.

Обычно член команды Tarantool отвечает в течение одного-двух рабочих дней, чтобы подтвердить, что тикет взят в работу, задать уточняющие вопросы или предложить альтернативное решение описанной проблемы.

4.5.12 Руководство по разрешению проблем

Проблема: при выполнении INSERT/UPDATE-запросов возникает ошибка ER_MEMORY_ISSUE

Возможные причины

- Нехватка памяти (значения параметров `arena_used_ratio` и `quota_used_ratio` из *box.slab.info()* приближаются к 100%).

  Чтобы проверить значения данных параметров, выполните соответствующие команды:

```
$ # подключаемся к админ-консоли нужного экземпляра
$ tarantoolctl enter <instance_name>
$ # -- ИЛИ --
$ tarantoolctl connect <URI>
```

```
-- запрашиваем значение arena_used_ratio
tarantool> box.slab.info().arena_used_ratio

-- запрашиваем значение quota_used_ratio
tarantool> box.slab.info().quota_used_ratio
```

Решение

У вас есть несколько вариантов действий:

- Зайти в *конфигурационный файл* Tarantool и увеличить значение параметра *box.cfg{memtx_memory}* (при наличии свободных ресурсов).

  В версиях Tarantool'а до 1.10 для изменения данного параметра требуется перезагрузить сервер. При обычной перезагрузке сервер будет недоступен на время старта Tarantool из .xlog-файлов. При перезагрузке в режиме горячего резервирования *hot standby* гарантирована практически 100%-ная доступность.

- Провести очистку базы данных.

- Проверьте, нет ли проблем с фрагментацией памяти:

```
-- запрашиваем значение quota_used_ratio
tarantool> box.slab.info().quota_used_ratio

-- запрашиваем значение items_used_ratio
tarantool> box.slab.info().items_used_ratio
```

  При высокой степени фрагментации памяти (значение параметра `quota_used_ratio` приближается к 100%, `items_used_ratio` около 50%) рекомендуется перезапустить Tarantool в режиме горячего резервирования *hot standby*.

Проблема: Tarantool создает большую нагрузку на CPU

Возможные причины

Поток обработки транзакций нагружает ЦП более чем на 60%.

Решение

Подключиться к Tarantool с помощью утилиты *tarantoolctl*, внимательно изучить статистику запросов с помощью *box.stat()* и выявить источник потребления. Для этой цели могут оказаться полезными следующие команды:

```
$ # подключаемся к админ-консоли нужного экземпляра
$ tarantoolctl enter <instance_name>
$ # -- ИЛИ --
$ tarantoolctl connect <URI>
```

```
-- запрашиваем RPS для вызовов хранимых процедур
tarantool> box.stat().CALL.rps
```

Критическое значение RPS – 75 000, в случае большого Lua-приложения (модульного приложения, содержащего более 200 строк кода) – 10 000 - 20 000.

```
-- запрашиваем RPS для запросов указанного типа
tarantool> box.stat().<query_type>.rps
```

Критическое значение RPS для запросов типа SELECT/INSERT/UPDATE/DELETE – 100 000.

Если основная нагрузка генерируется SELECT-запросами, следует добавить *slave-сервер* и часть запросов обрабатывать на нем.

Если же нагрузка по большей части приходится на INSERT/UPDATE/DELETE-запросы, рекомендуется провести шардинг базы данных.

Проблема: обработка запросов прекращается по таймауту

Возможные причины

Примечание: Все описанные ниже ситуации можно распознать по записям в журнале Tarantool, начинающимся со слов `'Too long...'`.

---

1. Быстрые и медленные запросы обрабатываются в одном подключении, что приводит к забиванию readahead-буфера медленными запросами.

   Решение

   У вас есть несколько вариантов действий:

   - Увеличить размер readahead-буфера (*box.cfg{readahead}*).

     Перезапускать Tarantool при этом не требуется. Для обновления конфигурации необходимо подключиться к Tarantool с помощью утилиты *tarantoolctl* и передать в `box.cfg{}` новое значение параметра `readahead`:

     ```
     $ # подключаемся к админ-консоли нужного экземпляра
     $ tarantoolctl enter <instance_name>
     $ # -- ИЛИ --
     $ tarantoolctl connect <URI>
     ```

     ```
     -- задаем новое значение readahead
     tarantool> box.cfg{readahead = 10 * 1024 * 1024}
     ```

Пример расчета: при 1000 RPS, размере одного запроса в 1 Кбайт и максимальном времени обработки одного запроса в 10 секунд минимальный размер readahead-буфера должен равняться 10 Мбайт.

- Обрабатывать быстрые и медленные запросы в отдельных подключениях (решается на уровне бизнес-логики).

2. Медленная работа дисков.

   Решение

   Проверить занятость дисков (с помощью утилиты iostat, iotop или strace посмотреть на параметр `iowait`) и попробовать разнести .xlog-файлы и снимки состояния базы данных по разным дискам (т.е. указать разные значения для параметров *wal_dir* и *memtx_dir*).

Проблема: параметры репликации lag и idle принимают отрицательные значения

Речь идет о параметрах `box.info.replication.(upstream.)lag` и `box.info.replication.(upstream.)idle` из сводной таблицы *box.info.replication*.

Возможные причины

Не синхронизированы часы на машинах или неправильно работает NTP-сервер.

Решение

Проверить настройки NTP-сервера.

Если проблем с NTP-сервером не обнаружено, то не следует ничего предпринимать, потому что при вычислении лага репликации используются показания системных часов на двух разных машинах, и в случае рассинхронизации может случиться так, что часы удаленного мастер-сервера всегда будут отставать от часов локального экземпляра Tarantool.

Проблема: значение параметра idle растет, но журнал не содержит связанных с этим сообщений

Речь идет о параметре `box.info.replication.(upstream.)idle` из сводной таблицы *box.info.replication*.

Возможные причины

Одному серверу были назначены различные IP-адреса или один и тот же сервер был указан в `box.cfg{}` дважды, что привело к установлению дублирующего подключения.

Решение

Обновить Tarantool 1.6 до 1.7, где эта ошибка была исправлена: в описанной ситуации репликация будет остановлена, а в журнал будет записана следующая ошибка: `'Incorrect value for option ''replication_source'': duplicate connection with the same replica UUID'`.

Проблема: общие параметры репликации не совпадают на репликах в рамках одного кластера

Речь идет о кластере, состоящем из одного мастера и нескольких реплик. В таком случае значения общих параметров из сводной таблицы *box.info.replication*, например `box.info.replication.lsn`, должны приходить с мастера и должны быть одинаковыми на всех репликах. Если такие параметры не совпадают, это свидетельствует о наличии проблем.

Возможные причины

Сбой репликации.

Решение

Перезапустить репликацию.

Проблема: репликация мастер-мастер остановлена

Речь идет о том, что параметр *box.info.replication(.upstream).status* имеет значение stopped.

Возможные причины

В репликационном кластере, состоящем из двух мастер-серверов, один из серверов попытался выполнить действие, уже выполненное другим сервером, – например, повторно вставить кортеж с таким же уникальным ключом (распознается по ошибке вида `'Duplicate key exists in unique index 'primary' in space <space_name>'`).

Решение

Возобновить репликацию с помощью следующих команд (должны быть выполнены на всех мастер-серверах):

```
$ # подключаемся к админ-консоли нужного экземпляра
$ tarantoolctl enter <instance_name>
$ # -- ИЛИ --
$ tarantoolctl connect <URI>
```

```
-- перезапускаем репликацию
tarantool> original_value = box.cfg.replication
tarantool> box.cfg{replication={}}
tarantool> box.cfg{replication=original_value}
```

Также рекомендуется перейти на текстовые первичные ключи или настроить *репликацию мастер-реплика*.

Проблема: Tarantool работает заметно медленнее, чем раньше

Возможные причины

Неэффективное использование памяти (память занята большим количеством неиспользуемых объектов).

Решение

Запустить сборщик мусора в Lua с помощью функции collectgarbage(count) и измерить время ее выполнения с помощью *clock.bench()* или *clock.proc()*.

Пример кода для подсчета потребляемой памяти:

```
$ # подключаемся к админ-консоли нужного экземпляра
$ tarantoolctl enter <instance_name>
$ # -- ИЛИ --
$ tarantoolctl connect <URI>
```

```
-- загрузка модуля clock для работы со временем
tarantool> local clock = require 'clock'
-- запускаем таймер
tarantool> local b = clock.proc()
-- запускаем сборку мусора
```

(continues on next page)

(продолжение с предыдущей страницы)

```
tarantool> local c = collectgarbage('count')
-- останавливаем таймер по завершении сборки мусора
tarantool> return c, clock.proc() - b
```

Если возвращаемое `clock.proc()` значение больше 0.001, это может являться признаком неэффективного использования памяти (активного вмешательства не требуется, но рекомендуется оптимизация кода). Если значение превышает 0.01, необходимо провести подробный анализ кода и оптимизировать потребление памяти.

Если значение больше 0,01, код приложения однозначно необходимо проанализировать на предмет оптимизации использования памяти.

Problem: Fiber switch is forbidden in `__gc` metamethod

Problem description

Fiber switch is forbidden in `__gc` metamethod since this change to avoid unexpected Lua OOM. However, one may need to use a yielding function to finalize resources, for example, to close a socket.

Below are examples of proper implementing such a procedure.

Solution

First, there come two simple examples illustrating the logic of the solution:

- *Example 1*
- *Example 2*.

Next comes the *Example 3* illustrating the usage of the `sched.lua` module that is the recommended method.

All the explanations are given in the comments in the code listing. `-- >` indicates the output in console.

Example 1

Implementing a valid finalizer for a particular FFI type (`custom_t`).

```
local ffi = require('ffi')
local fiber = require('fiber')

ffi.cdef('struct custom { int a; };')

local function __custom_gc(self)
  print(("Entered custom GC finalizer for %s... (before yield)"):format(self.a))
  fiber.yield()
  print(("Leaving custom GC finalizer for %s... (after yield)"):format(self.a))
end

local custom_t = ffi.metatype('struct custom', {
  __gc = function(self)
    -- XXX: Do not invoke yielding functions in __gc metamethod.
    -- Create a new fiber to run after the execution leaves
    -- this routine.
    fiber.new(__custom_gc, self)
    print(("Finalization is scheduled for %s..."):format(self.a))
  end
```

(continues on next page)

(продолжение с предыдущей страницы)

```
})

-- Create a cdata object of <custom_t> type.
local c = custom_t(42)

-- Remove a single reference to that object to make it subject
-- for GC.
c = nil

-- Run full GC cycle to purge the unreferenced object.
collectgarbage('collect')
-- > Finalization is scheduled for 42...

-- XXX: There is no finalization made until the running fiber
-- yields its execution. Let's do it now.
fiber.yield()
-- > Entered custom GC finalizer for 42... (before yield)
-- > Leaving custom GC finalizer for 42... (after yield)
```

Example 2

Implementing a valid finalizer for a particular user type (`struct custom`).

`custom.c`

```
#include <lauxlib.h>
#include <lua.h>
#include <module.h>
#include <stdio.h>

struct custom {
  int a;
};

const char *CUSTOM_MTNAME = "CUSTOM_MTNAME";

/*
 * XXX: Do not invoke yielding functions in __gc metamethod.
 * Create a new fiber to be run after the execution leaves
 * this routine. Unfortunately we can't pass the parameters to the
 * routine to be executed by the created fiber via <fiber_new_ex>.
 * So there is a workaround to load the Lua code below to create
 * __gc metamethod passing the object for finalization via Lua
 * stack to the spawned fiber.
 */
const char *gc_wrapper_constructor = " local fiber = require('fiber')         "
             " print('constructor is initialized')    "
             " return function(__custom_gc)           "
             "   print('constructor is called')       "
             "   return function(self)                "
             "     print('__gc is called')            "
             "     fiber.new(__custom_gc, self)       "
             "     print('Finalization is scheduled') "
             "   end                                  "
             " end                                    "
       ;
```

(continues on next page)

(продолжение с предыдущей страницы)

```
int custom_gc(lua_State *L) {
  struct custom *self = luaL_checkudata(L, 1, CUSTOM_MTNAME);
  printf("Entered custom_gc for %d... (before yield)\n", self->a);
  fiber_sleep(0);
  printf("Leaving custom_gc for %d... (after yield)\n", self->a);
  return 0;
}

int custom_new(lua_State *L) {
  struct custom *self = lua_newuserdata(L, sizeof(struct custom));
  luaL_getmetatable(L, CUSTOM_MTNAME);
  lua_setmetatable(L, -2);
  self->a = lua_tonumber(L, 1);
  return 1;
}

static const struct luaL_Reg libcustom_methods [] = {
  { "new", custom_new },
  { NULL, NULL }
};

int luaopen_custom(lua_State *L) {
  int rc;

  /* Create metatable for struct custom type */
  luaL_newmetatable(L, CUSTOM_MTNAME);
  /*
   * Run the constructor initializer for GC finalizer:
   * - load fiber module as an upvalue for GC finalizer
   *   constructor
   * - return GC finalizer constructor on the top of the
   *   Lua stack
   */
  rc = luaL_dostring(L, gc_wrapper_constructor);
  /*
   * Check whether constructor is initialized (i.e. neither
   * syntax nor runtime error is raised).
   */
  if (rc != LUA_OK)
    luaL_error(L, "test module loading failed: constructor init");
  /*
   * Create GC object for <custom_gc> function to be called
   * in scope of the GC finalizer and push it on top of the
   * constructor returned before.
   */
  lua_pushcfunction(L, custom_gc);
  /*
   * Run the constructor with <custom_gc> GCfunc object as
   * a single argument. As a result GC finalizer is returned
   * on the top of the Lua stack.
   */
  rc = lua_pcall(L, 1, 1, 0);
  /*
   * Check whether GC finalizer is created (i.e. neither
   * syntax nor runtime error is raised).
   */
```

(continues on next page)

(продолжение с предыдущей страницы)

```
  if (rc != LUA_OK)
    luaL_error(L, "test module loading failed: __gc init");
  /*
   * Assign the returned function as a __gc metamethod to
   * custom type metatable.
   */
  lua_setfield(L, -2, "__gc");

  /*
   * Initialize Lua table for custom module and fill it
   * with the custom methods.
   */
  lua_newtable(L);
  luaL_register(L, NULL, libcustom_methods);
  return 1;
}
```

custom_c.lua

```
-- Load custom Lua C extension.
local custom = require('custom')
-- > constructor is initialized
-- > constructor is called

-- Create a userdata object of <struct custom> type.
local c = custom.new(9)

-- Remove a single reference to that object to make it subject
-- for GC.
c = nil

-- Run full GC cycle to purge the unreferenced object.
collectgarbage('collect')
-- > __gc is called
-- > Finalization is scheduled

-- XXX: There is no finalization made until the running fiber
-- yields its execution. Let's do it now.
require('fiber').yield()
-- > Entered custom_gc for 9... (before yield)

-- XXX: Finalizer yields the execution, so now we are here.
print('We are here')
-- > We are here

-- XXX: This fiber finishes its execution, so yield to the
-- remaining fiber to finish the postponed finalization.
-- > Leaving custom_gc for 9... (after yield)
```

Example 3

It is important to note that the finalizer implementations in the examples above increase pressure on the platform performance by creating a new fiber on each `__gc` call. To prevent such an excessive fibers spawning, it's better to start a single «scheduler» fiber and provide the interface to postpone the required asynchronous action.

For this purpose, the module called `sched.lua` is implemented (see the listing below). It is a part of Tarantool and should be made required in your custom code. The usage example is given in the `init.lua` file below.

sched.lua

```
local fiber = require('fiber')

local worker_next_task = nil
local worker_last_task
local worker_fiber
local worker_cv = fiber.cond()

-- XXX: the module is not ready for reloading, so worker_fiber is
-- respawned when sched.lua is purged from package.loaded.

--
-- Worker is a singleton fiber for not urgent delayed execution of
-- functions. Main purpose - schedule execution of a function,
-- which is going to yield, from a context, where a yield is not
-- allowed. Such as an FFI object's GC callback.
--
local function worker_f()
  while true do
    local task
    while true do
      task = worker_next_task
      if task then break end
      -- XXX: Make the fiber wait until the task is added.
      worker_cv:wait()
    end
    worker_next_task = task.next
    task.f(task.arg)
    fiber.yield()
  end
end

local function worker_safe_f()
  pcall(worker_f)
  -- The function <worker_f> never returns. If the execution is
  -- here, this fiber is probably canceled and now is not able to
  -- sleep. Create a new one.
  worker_fiber = fiber.new(worker_safe_f)
end

worker_fiber = fiber.new(worker_safe_f)

local function worker_schedule_task(f, arg)
  local task = { f = f, arg = arg }
  if not worker_next_task then
    worker_next_task = task
  else
    worker_last_task.next = task
  end
  worker_last_task = task
  worker_cv:signal()
end

return {
  postpone = worker_schedule_task
}
```

init.lua

```
local ffi = require('ffi')
local fiber = require('fiber')
local sched = require('sched')

local function __custom_gc(self)
  print(("Entered custom GC finalizer for %s... (before yield)"):format(self.a))
  fiber.yield()
  print(("Leaving custom GC finalizer for %s... (after yield)"):format(self.a))
end

ffi.cdef('struct custom { int a; };')
local custom_t = ffi.metatype('struct custom', {
  __gc = function(self)
    -- XXX: Do not invoke yielding functions in __gc metamethod.
    -- Schedule __custom_gc call via sched.postpone to be run
    -- after the execution leaves this routine.
    sched.postpone(__custom_gc, self)
    print(("Finalization is scheduled for %s..."):format(self.a))
  end
})

-- Create several <custom_t> objects to be finalized later.
local t = { }
for i = 1, 10 do t[i] = custom_t(i) end

-- Run full GC cycle to collect the existing garbage. Nothing is
-- going to be printed, since the table <t> is still "alive".
collectgarbage('collect')

-- Remove the reference to the table and, ergo, all references to
-- the objects.
t = nil

-- Run full GC cycle to collect the table and objects inside it.
-- As a result all <custom_t> objects are scheduled for further
-- finalization, but the finalizer itself (i.e. __custom_gc
-- functions) is not called.
collectgarbage('collect')
-- > Finalization is scheduled for 10...
-- > Finalization is scheduled for 9...
-- > ...
-- > Finalization is scheduled for 2...
-- > Finalization is scheduled for 1...

-- XXX: There is no finalization made until the running fiber
-- yields its execution. Let's do it now.
fiber.yield()
-- > Entered custom GC finalizer for 10... (before yield)

-- XXX: Oops, we are here now, since the scheduler fiber yielded
-- the execution to this one. Check this out.
print("We're here now. Let's continue the scheduled finalization.")
-- > We're here now. Let's continue the finalization

-- OK, wait a second to allow the scheduler to cleanup the
-- remaining garbage.
fiber.sleep(1)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
-- > Leaving custom GC finalizer for 10... (after yield)
-- > Entered custom GC finalizer for 9... (before yield)
-- > Leaving custom GC finalizer for 9... (after yield)
-- > ...
-- > Entered custom GC finalizer for 1... (before yield)
-- > Leaving custom GC finalizer for 1... (after yield)

print("Did we finish? I guess so.")
-- > Did we finish? I guess so.

-- Stop the instance.
os.exit(0)
```

4.5.13 Monitoring

Monitoring is the process of measuring and tracking Tarantool performance according to key metrics influencing it. These metrics are typically monitored in real time, allowing you to identify or predict issues.

This chapter includes the following sections:

Monitoring: getting started

Tarantool

First, you need to install the `metrics` package:

```
$ cd ${PROJECT_ROOT}
$ tarantoolctl rocks install metrics
```

Next, require it in your code:

```
local metrics = require('metrics')
```

Set a global label for your metrics:

```
metrics.set_global_labels({alias = 'alias'})
```

Enable default Tarantool metrics such as network, memory, operations, etc:

```
metrics.enable_default_metrics()
```

If you use Cartridge, enable Cartridge metrics:

```
metrics.enable_cartridge_metrics()
```

Initialize the Prometheus Exporter, or export metrics in any other format:

```
local httpd = require('http.server')
local http_handler = require('metrics.plugins.prometheus').collect_http


httpd.new('0.0.0.0', 8088)
    :route({path = '/metrics'}, function(...)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        return http_handler(...)
end)
    :start()

box.cfg{
    listen = 3302
}
```

Now you can use the HTTP API endpoint `/metrics` to collect your metrics in the Prometheus format. If you need your custom metrics, see the *API reference*.

Instance health check

In production environments Tarantool Cluster usually has a large number of so called «routers», Tarantool instances that handle input load and it is required to evenly distribute the load. Various load-balancers are used for this, but any load-balancer have to know which «routers» are ready to accept the load at that very moment. Metrics library has a special plugin that creates an http handler that can be used by the load-balancer to check the current state of any Tarantool instance. If the instance is ready to accept the load, it will return a response with a 200 status code, if not, with a 500 status code.

Cartridge role

`cartridge.roles.metrics` is a role for Tarantool Cartridge. It allows using default metrics in a Cartridge application and manage them via configuration.

Usage

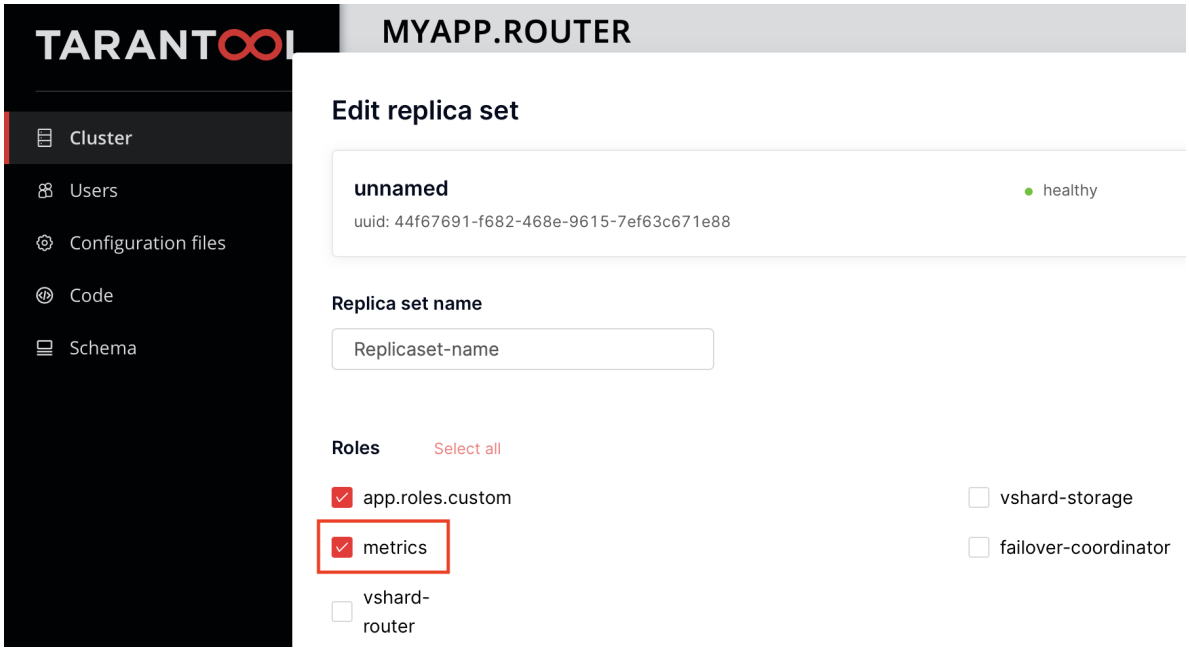
1. Add `metrics` package to dependencies in the `.rockspec` file. Make sure that you are using version **0.3.0** or higher.

   ```
   dependencies = {
       ...
       'metrics >= 0.3.0-1',
       ...
   }
   ```

2. Make sure that you have `cartridge.roles.metrics` in the roles list in `cartridge.cfg` in your entry-point file (e.g. `init.lua`).

   ```
   local ok, err = cartridge.cfg({
       ...
       roles = {
           ...
           'cartridge.roles.metrics',
           ...
       },
   })
   ```

3. Enable role in the interface:



Since version **0.6.0** metrics role is permanent and enabled on instances by default.

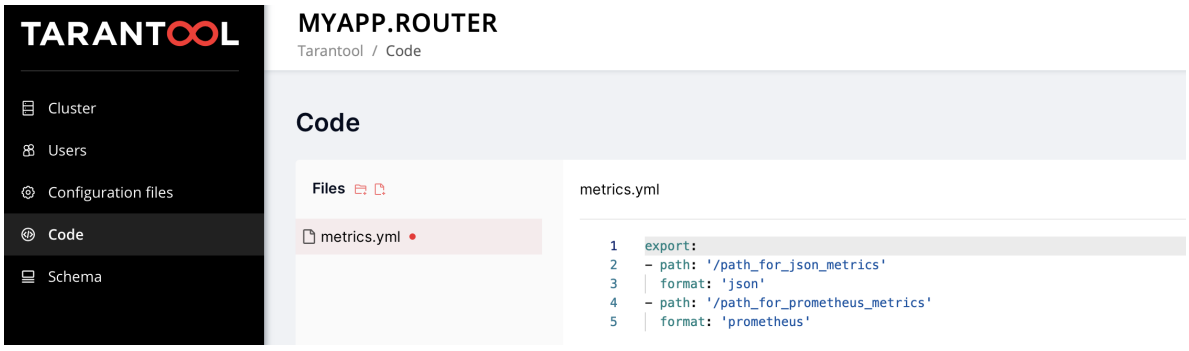
4. After role initialization, default metrics will be enabled and the global label `'alias'` will be set. **Note** that `'alias'` label value is set by instance configuration option `alias` or `instance_name` (since **0.6.1**).

   If you need to use the functionality of any metrics package, you may get it as a Cartridge service and use it like a regular package after `require`:

```
local cartridge = require('cartridge')
local metrics = cartridge.service_get('metrics')
```

5. To view metrics via API endpoints, use the following configuration (to learn more about Cartridge configuration, see this):

```
metrics:
  export:
  - path: '/path_for_json_metrics'
    format: 'json'
  - path: '/path_for_prometheus_metrics'
    format: 'prometheus'
  - path: '/health'
    format: 'health'
```



OR

Use `set_export`:

NOTE that `set_export` has lower priority than clusterwide config and won't work if metrics config is present.

```
metrics.set_export({
    {
        path = '/path_for_json_metrics',
        format = 'json'
    },
    {
        path = '/path_for_prometheus_metrics',
        format = 'prometheus'
    },
    {
        path = '/health',
        format = 'health'
    }
})
```

The metrics will be available on the path specified in `path` in the format specified in `format`.

You can add several entry points of the same format by different paths, like this:

```
metrics:
  export:
    - path: '/path_for_json_metrics'
      format: 'json'
    - path: '/another_path_for_json_metrics'
      format: 'json'
```

Metrics reference

This page provides detailed description of metrics from module `metrics`.

General metrics

General instance information.

- `tnt_cfg_current_time` - instance system time in Unix timestamp format
- `tnt_info_uptime` – time since instance was started, in seconds

Memory general

These metrics provide a picture of memory usage by Tarantool process.

- `tnt_info_info_memory_cache` - number of bytes in the cache for the tuples stored for the vinyl storage engine.
- `tnt_info_info_memory_data` - number of bytes used for storing user data (the tuples) with the memtx engine and with level 0 of the vinyl engine, without taking memory fragmentation into account.
- `tnt_info_info_memory_index` - number of bytes used for indexing user data, including memtx and vinyl memory tree extents, the vinyl page index, and the vinyl bloom filters.

- `tnt_info_info_memory_lua` - number of bytes used for Lua runtime. Lua memory is bounded by 2 GB per instance. Monitoring of this metric can prevent memory overflow.
- `tnt_info_info_memory_net` - number of bytes used for network input/output buffers.
- `tnt_info_info_memory_tx` - number of bytes in use by active transactions. For the vinyl storage engine, this is the total size of all allocated objects (struct txv, struct vy_tx, struct vy_read_interval) and tuples pinned for those objects.

Memory allocation

Provides memory usage report for the slab allocator. The slab allocator is the main allocator used to store tuples. This can be used to monitor the total memory usage and memory fragmentation. To learn more about use cases, see this

Available memory, bytes:

- `tnt_slab_quota_size` - the amount of memory available to store tuples and indexes, equals memtx_memory
- `tnt_slab_arena_size` - the total memory used for tuples and indexes together (including allocated, but currently free slabs)
- `tnt_slab_items_size` - the total amount of memory (including allocated, but currently free slabs) used only for tuples, no indexes

Memory usage, bytes:

- `tnt_slab_quota_used` - the amount of memory that is already reserved by the slab allocator
- `tnt_slab_arena_used` - the efficient memory used for storing tuples and indexes together (omitting allocated, but currently free slabs)
- `tnt_slab_items_used` - the efficient amount of memory (omitting allocated, but currently free slabs) used only for tuples, no indexes

Memory utilization, %:

- `tnt_slab_quota_used_ratio` - tnt_slab_quota_used / tnt_slab_quota_size
- `tnt_slab_arena_used_ratio` - tnt_slab_arena_used / tnt_slab_arena_used
- `tnt_slab_items_used_ratio` - tnt_slab_items_used / tnt_slab_items_size

Spaces

Those metrics provide specific information about each individual space in Tarantool instance.

- `tnt_space_len` - number of records in space. This metric always has 2 labels - `{name="test", engine="memtx"}`. `name` - the name of the space, `engine` - is the engine of the space.
- `tnt_space_bsize` - the total number of bytes in all tuples. This metric always has 2 labels - `{name="test", engine="memtx"}`. `name` - the name of the space, `engine` - is the engine of the space.
- `tnt_space_index_bsize` - the total number of bytes taken by the index. This metric always has 2 labels - `{name="test", index_name="pk"}`. `name` - the name of the space, `index_name` - is the name of the index.
- `tnt_space_total_bsize` - the total size of tuples and all indexes in space. This metric always has 2 labels - `{name="test", engine="memtx"}`. `name` - the name of the space, `engine` - is the engine of the space.

- `tnt_space_count` - the total tuples count for vinyl. This metric always has labels - `{name="test", engine="vinyl"}`. `name` - the name of the space. `engine` - is the engine of the space.

Network

Network activity stats. This can be used to monitor network load, usage peaks and traffic drops.

Sent bytes:

- `tnt_net_sent_total` - bytes sent from this instance over network since instance start

Received bytes:

- `tnt_net_received_total` - bytes this instance has received since instance start

Connections:

- `tnt_net_connections_total` - number of incoming network connections since instance start
- `tnt_net_connections_current` - number of active network connections

Requests:

- `tnt_net_requests_total` - number of network requests this instance has handled since instance start
- `tnt_net_requests_current` - amount of pending network requests

Fibers

Provides statistics of *fibers*. If your app creates a lot of fibers, it can be used for monitoring fibers count and memory usage.

- `tnt_fiber_count` - number of fibers
- `tnt_fiber_csw` - averall amount of fibers context switches
- `tnt_fiber_memalloc` - the amount of memory that is reserved for fibers
- `tnt_fiber_memused` - the amount of memory that is used by fibers

Operations

Number of iproto requests this instance has processed, aggregated by request type. It can be used to find out which type of operation clients make more often.

- `tnt_stats_op_total` - total number of calls since server start

That metric have `operation` label to be able to distinguish different request types, e.g.: `{operation="select"}`

Request type could be one of:

- `delete` - delete calls
- `error` - requests resulted in an error
- `update` - update calls
- `call` - requests to execute stored procedures
- `auth` - authentication requests

- `eval` - calls to evaluate lua code
- `replace` - replace call
- `execute` - execute SQL calls
- `select` - select calls
- `upsert` - upsert calls
- `prepare` - SQL prepare calls
- `insert` - insert calls

Replication

Provides information of current replication status. To learn more about replication mechanism in Tarantool, see *this*

- `tnt_info_lsn` – LSN of instance
- `tnt_info_vclock` – LSN number in vclock. This metric always has label - `{id="id"}`, where `id` is instance number in replicaset
- `tnt_replication_replica_<id>_lsn` / `tnt_replication_master_<id>_lsn` - LSN of master/replica, `id` is instance number in replicaset
- `tnt_replication_<id>_lag` - replication lag value in seconds, `id` is instance number in replicaset

Runtime

- `tnt_runtime_lua` – Lua garbage collector size in bytes
- `tnt_runtime_used` - number of bytes used for Lua runtime

Cartridge

- `cartridge_issues` - Number of *issues <https://www.tarantool.io/en/doc/latest/book/cartridge/cartridge_api/modules/* across cluster instances. This metric always has labels - `{level="critical"}`. `level` - the level of the issue. `critical` level is associated with critical cluster problems, e.g. memory used ratio > 90%, `warning` is associated with other cluster problems, e.g. replication issues on cluster.

LuaJIT metrics

LuaJIT metrics help to understand Lua GC state. Only in Tarantool 2.6+.

General JIT metrics:

- `lj_jit_snap_restore` - overall number of snap restores
- `lj_jit_trace_num` - number of JIT traces
- `lj_jit_trace_abort` - overall number of abort traces
- `lj_jit_mcode_size` - total size of all allocated machine code areas

JIT strings:

- `lj_strhash_hit` - number of strings being interned

- `lj_strhash_miss` - total number of string allocations

GC steps:

- `lj_gc_steps_atomic` - count of incremental GC steps (atomic state)
- `lj_gc_steps_sweepstring` - count of incremental GC steps (sweepstring state)
- `lj_gc_steps_finalize` - count of incremental GC steps (finalize state)
- `lj_gc_steps_sweep` - count of incremental GC steps (sweep state)
- `lj_gc_steps_propagate` - count of incremental GC steps (propagate state)
- `lj_gc_steps_pause` - count of incremental GC steps (pause state)

Allocations:

- `lj_gc_strnum` - number of allocated `string` objects
- `lj_gc_tabnum` - number of allocated `table` objects
- `lj_gc_cdatanum` - number of allocated `cdata` objects
- `lj_gc_udatanum` - number of allocated `udata` objects
- `lj_gc_freed` - total amount of freed memory
- `lj_gc_total` - current allocated Lua memory
- `lj_gc_allocated` - total amount of allocated memory

CPU metrics

Those metrics provides CPU usage statistics. Only for Linux.

- `tnt_cpu_count` - total number of processors configured by the operating system
- `tnt_cpu_total` - host CPU time
- `tnt_cpu_thread` - Tarantool thread cpu time. This metric always has labels - `{kind="user", thread_name="tarantool", thread_pid="pid", file_name="init.lua"}`, where `kind` is `user` or `system`, `thread_name` is `tarantool`, `wal`, `iproto` or `coio`, `file_name` is entrypoint file name, e.g. `init.lua`.

API reference

Collectors

An application using the `metrics` module has 4 primitives (called «collectors») at its disposal:

- *Counter*
- *Gauge*
- *Histogram*
- *Summary*

A collector represents one or more observations that are changing over time.

Counter

metrics.counter(*name*[, *help*])
Registers a new counter.

Параметры

- name (string) – Collector name. Must be unique.
- help (string) – Help description.

Return Counter object

Rtype counter_obj

object counter_obj

counter_obj:inc(*num*, *label_pairs*)
Increments an observation under label_pairs. If label_pairs didn't exist before, this creates it.

Параметры

- num (*number*) – Increase value.
- label_pairs (table) – Table containing label names as keys, label values as values.

counter_obj:collect()

Return Array of observation objects for the given counter.

```
{
    label_pairs: table,          -- `label_pairs` key-value table
    timestamp: ctype<uint64_t>,  -- current system time (in microseconds)
    value: number,               -- current value
    metric_name: string,         -- collector
}
```

Rtype *table*

Gauge

metrics.gauge(*name*[, *help*])
Registers a new gauge. Returns a Gauge object.

Параметры

- name (string) – Collector name. Must be unique.
- help (string) – Help description.

Return Gauge object

Rtype gauge_obj

object gauge_obj

gauge_obj:inc(*num*, *label_pairs*)
Same as Counter inc().

`gauge_obj:inc(`*num*, *label_pairs*`)`
Same as `inc()`, but decreases the observation.

`gauge_obj:set(`*num*, *label_pairs*`)`
Same as `inc()`, but sets the observation.

`gauge_obj:collect()`
Returns an array of `observation` objects for the given gauge. For `observation` description, see *counter_obj:collect()*.

Histogram

`metrics.histogram(`*name*[, *help*, *buckets*]`)`
Registers a new histogram.

Параметры

- `name` (`string`) – Collector name. Must be unique.
- `help` (`string`) – Help description.
- `buckets` (`table`) – Histogram buckets (an array of sorted positive numbers). Infinity bucket (`INF`) is appended automatically. Default is `{.005, .01, .025, .05, .075, .1, .25, .5, .75, 1.0, 2.5, 5.0, 7.5, 10.0, INF}`.

Return Histogram object

Rtype histogram_obj

Примечание: The histogram is just a set of collectors:

- `name .. "_sum"` - A counter holding the sum of added observations. Contains only an empty label set.
- `name .. "_count"` - A counter holding the number of added observations. Contains only an empty label set.
- `name .. "_bucket"` - A counter holding all bucket sizes under the label `le` (low or equal). So to access a specific bucket `x` (`x` is a number), you should specify the value `x` for the label `le`.

`object histogram_obj`

`histogram_obj:observe(`*num*, *label_pairs*`)`
Records a new value in a histogram. This increments all buckets sizes under labels `le` $>=$ `num` and labels matching `label_pairs`.

Параметры

- `num` (*number*) – Value to put in the histogram.
- `label_pairs` (`table`) – Table containing label names as keys, label values as values (table). A new value is observed by all internal counters with these labels specified.

`histogram_obj:collect()`
Returns a concatenation of `counter_obj:collect()` across all internal counters of `histogram_obj`. For `observation` description, see *counter_obj:collect()*.

Summary

`metrics.summary(`*name*[, *help*, *objectives*]`)`
Registers a new summary. Quantile computation is based on the algorithm «Effective computation of biased quantiles over data streams»

Параметры

- `name` (`string`) – Collector name. Must be unique.
- `help` (`string`) – Help description.
- `objectives` (`table`) – Quantiles to observe in the form `{quantile = error, ...}`. For example: `{[0.5]=0.01, [0.9]=0.01, [0.99]=0.01}`

Return Summary object

Rtype summary_obj

Примечание: The summary is just a set of collectors:

- `name .. "_sum"` - A counter holding the sum of added observations.
- `name .. "_count"` - A counter holding the number of added observations.
- `name` - It's holding all quantiles under observation under the label `quantile` (low or equal). So to access a specific quantile `x` (`x` is a number), you should specify the value `x` for the label `quantile`.

`object summary_obj`

`summary_obj:observe(`*num*, *label_pairs*`)`
Records a new value in a summary.

Параметры

- `num` (*number*) – Value to put in the data stream.
- `label_pairs` (`table`) – Table containing label names as keys, label values as values (table). A new value is observed by all internal counters with these labels specified.

`summary_obj:collect()`
Returns a concatenation of `counter_obj:collect()` across all internal counters of `summary_obj`. For `observation` description, see *counter_obj:collect()*.

Labels

All collectors support providing `label_pairs` on data modification. Labels are basically a metainfo that you associate with a metric in the format of key-value pairs. See tags in Graphite and labels in Prometheus. Labels are used to differentiate the characteristics of a thing being measured. For example, in a metric associated with the total number of http requests, you can use methods and statuses label pairs:

```
http_requests_total_counter:inc(1, {method = 'POST', status = '200'})
```

You don't have to predefine labels in advance.

Using labels on your metrics allows you to later derive new time series (visualize their graphs) by specifying conditions on label values. In the example above, we could derive these time series:

1. The total number of requests over time with method = «POST» (and any status).

2. The total number of requests over time with status = 500 (and any method).

You can also set global labels by calling `metrics.set_global_labels({ label = value, ...})`.

Metrics functions

`metrics.enable_default_metrics()`
Enables Tarantool metrics collections. See *metrics reference* for details.

`metrics.enable_cartridge_metrics()`
Enables Cartridge metrics collections. See *metrics reference* for details.

`metrics.set_global_labels(`*label_pairs*`)`
Set global labels that will be added to every observation.

Параметры

- `label_pairs` (`table`) – Table containing label names as string keys, label values as values (table).

Global labels are applied only on metrics collection and have no effect on how observations are stored.

Global labels can be changed on the fly.

Observation `label_pairs` has priority over global labels: if you pass `label_pairs` to an observation method with the same key as some global label, the method argument value will be used.

`metrics.register_callback(`*callback*`)`
Registers a function `callback` which will be called right before metrics collection on plugin export.

Параметры

- `callback` (*`function`*) – Function which takes no parameters.

Most common usage is for gauge metrics updates.

Collecting HTTP requests latency statistics

`metrics` also provides a middleware for monitoring HTTP (set by the http module) latency statistics.

`metrics.http_middleware.configure_default_collector(`*type_name*, *name*, *help*`)`
Registers a collector for the middleware and sets it as default.

Параметры

- `type_name` (`string`) – Collector type: «histogram» or «summary». Default is «histogram».
- `name` (`string`) – Collector name. Default is «http_server_request_latency».
- `help` (`string`) – Help description. Default is «HTTP Server Request Latency».

If a collector with the same type and name already exists in the registry, throws an error.

`metrics.http_middleware.build_default_collector(`*type_name*, *name*[, *help*]`)`
Registers a collector for the middleware and returns it.

Параметры

- `type_name` (`string`) – Collector type: «histogram» or «summary». Default is «histogram».
- `name` (`string`) – Collector name. Default is «http_server_request_latency».

- `help` (`string`) – Help description. Default is «HTTP Server Request Latency».

If a collector with the same type and name already exists in the registry, throws an error.

`metrics.http_middleware.set_default_collector(`*collector*`)`
Sets the default collector.

Параметры

- `collector` – Middleware collector object.

`metrics.http_middleware.get_default_collector()`
Returns the default collector. If the default collector hasn't been set yet, registers it (with default `http_middleware.build_default_collector(...)` parameters) and sets it as default.

`metrics.http_middleware.v1(`*handler*, *collector*`)`
Latency measure wrap-up for HTTP ver. 1.x.x handler. Returns a wrapped handler.

Параметры

- `handler` (*`function`*) – Handler function.
- `collector` – Middleware collector object. If not set, uses the default collector (like in `http_middleware.get_default_collector()`).

Usage: `httpd:route(route, http_middleware.v1(request_handler, collector))`

For a more detailed example, see https://github.com/tarantool/metrics/blob/master/example/HTTP/latency_v1.lua

`metrics.http_middleware.v2(`*collector*`)`
Returns the latency measure middleware for HTTP ver. 2.x.x.

Параметры

- `collector` – Middleware collector object. If not set, uses the default collector (like in `http_middleware.get_default_collector()`).

Usage:

```
router = require('http.router').new()
router:route(route, request_handler)
router:use(http_middleware.v2(collector), {name = 'http_instrumentation'}) -- the second
↪argument is optional, see HTTP docs
```

For a more detailed example, see https://github.com/tarantool/metrics/blob/master/example/HTTP/latency_v2.lua

CPU usage metrics

CPU metrics work only on Linux. See *metrics reference* for details. To enable it you should register callback:

```
local metrics = require('metrics')

metrics.register_callback(function()
    local cpu_metrics = require('metrics.psutils.cpu')
    cpu_metrics.update()
end)
```

Collected metrics example

```
# HELP tnt_cpu_total Host CPU time
# TYPE tnt_cpu_total gauge
tnt_cpu_total 15006759
# HELP tnt_cpu_thread Tarantool thread cpu time
# TYPE tnt_cpu_thread gauge
tnt_cpu_thread{thread_name="coio",file_name="init.lua",thread_pid="699",kind="system"} 160
tnt_cpu_thread{thread_name="tarantool",file_name="init.lua",thread_pid="1",kind="user"} 949
tnt_cpu_thread{thread_name="tarantool",file_name="init.lua",thread_pid="1",kind="system"} 920
tnt_cpu_thread{thread_name="coio",file_name="init.lua",thread_pid="11",kind="user"} 79
tnt_cpu_thread{thread_name="coio",file_name="init.lua",thread_pid="699",kind="user"} 44
tnt_cpu_thread{thread_name="coio",file_name="init.lua",thread_pid="11",kind="system"} 294
```

Prometheus query aggregated by thread name

```
sum by (thread_name) (idelta(tnt_cpu_thread[$__interval]))
  / scalar(idelta(tnt_cpu_total[$__interval]) / tnt_cpu_count)
```

Examples

Below are examples of using metrics primitives.

Notice that this usage is independent of export-plugins such as Prometheus / Graphite / etc. For documentation on plugins usage, see their the *Metrics plugins* section.

Using counters:

```
local metrics = require('metrics')

-- create a counter
local http_requests_total_counter = metrics.counter('http_requests_total')

-- somewhere in the HTTP requests middleware:
http_requests_total_counter:inc(1, {method = 'GET'})
```

Using gauges:

```
local metrics = require('metrics')

-- create a gauge
local cpu_usage_gauge = metrics.gauge('cpu_usage', 'CPU usage')

-- register a lazy gauge value update
-- this will be called whenever the export is invoked in any plugins
metrics.register_callback(function()
    local current_cpu_usage = math.random()
    cpu_usage_gauge:set(current_cpu_usage, {app = 'tarantool'})
end)
```

Using histograms:

```
local metrics = require('metrics')

-- create a histogram
local http_requests_latency_hist = metrics.histogram(
    'http_requests_latency', 'HTTP requests total', {2, 4, 6})
```

(continues on next page)

(продолжение с предыдущей страницы)

```
-- somewhere in the HTTP requests middleware:
local latency = math.random(1, 10)
http_requests_latency_hist:observe(latency)
```

Using summaries:

```
local metrics = require('metrics')

-- create a summary
local http_requests_latency = metrics.summary(
    'http_requests_latency', 'HTTP requests total',
    {[0.5]=0.01, [0.9]=0.01, [0.99]=0.01}
)

-- somewhere in the HTTP requests middleware:
local latency = math.random(1, 10)
http_requests_latency:observe(latency)
```

Metrics plugins

Plugins allow using a unified interface to collect metrics without worrying about the way metrics export is performed. If you want to use another DB to store metrics data, you can use an appropriate export plugin just by changing one line of code.

Available plugins

Prometheus

Usage

Import the Prometheus plugin:

```
local prometheus = require('metrics.plugins.prometheus')
```

Further, use the `prometheus.collect_http()` function, which returns:

```
{
    status = 200,
    headers = <headers>,
    body = <body>,
}
```

See the Prometheus exposition format for details on `<body>` and `<headers>`.

Use in Tarantool http.server as follows:

- In Tarantool http.server v1 (currently used in Tarantool Cartridge):

  ```
  local httpd = require('http.server').new(...)
  ...
  httpd:route( { path = '/metrics' }, prometheus.collect_http)
  ```

- In Tarantool http.server v2 (the latest version):

```
local httpd = require('http.server').new(...)
local router = require('http.router').new(...)
httpd:set_router(router)
...
router:route( { path = '/metrics' }, prometheus.collect_http)
```

Sample settings

- For Tarantool `http.server` v1:

```
metrics = require('metrics')
metrics.enable_default_metrics()
prometheus = require('metrics.plugins.prometheus')
httpd = require('http.server').new('0.0.0.0', 8080)
httpd:route( { path = '/metrics' }, prometheus.collect_http)
httpd:start()
```

- For Tarantool Cartridge (with `http.server` v1):

```
cartridge = require('cartridge')
httpd = cartridge.service_get('httpd')
metrics = require('metrics')
metrics.enable_default_metrics()
prometheus = require('metrics.plugins.prometheus')
httpd:route( { path = '/metrics' }, prometheus.collect_http)
```

- For Tarantool `http.server` v2:

```
metrics = require('metrics')
metrics.enable_default_metrics()
prometheus = require('metrics.plugins.prometheus')
httpd = require('http.server').new('0.0.0.0', 8080)
router = require('http.router').new({charset = "utf8"})
httpd:set_router(router) router:route( { path = '/metrics' },
prometheus.collect_http)
httpd:start()
```

Graphite

Usage

Import the Graphite plugin:

```
local graphite = require('metrics.plugins.graphite')
```

To start automatically exporting the current values of all `metrics.{counter,gauge,histogram}`, just call:

`metrics.plugins.graphite.init(`*options*`)`

Параметры

- `options` (`table`) – Possible options:
  - `prefix` (string) - metrics prefix (default is `'tarantool'`);
  - `host` (string) - graphite server host (default is `'127.0.0.1'`);
  - `port` (number) - graphite server port (default is 2003);

– `send_interval` (number) - metrics collect interval in seconds (default is 2);

This creates a background fiber that periodically sends all metrics to a remote Graphite server.

Exported metric name is sent in the format `<prefix>.<metric_name>`.

JSON

Usage

Import the JSON plugin:

```
local json_metrics = require('metrics.plugins.json')
```

`metrics.plugins.json.export()`

Return the following structure

```
[
    {
        "name": "<name>",
        "label_pairs": {
            "<name>": "<value>",
            "...": "..."
            },
        "timestamp": "<number>",
        "value": "<value>"
    },
    "..."
]
```

Rtype *string*

Важно: Values can be `+-math.huge`, `math.huge * 0`. Then:

- `math.inf` is serialized to `"inf"`
- `-math.inf` is serialized to `"-inf"`
- `nan` is serialized to `"nan"`

Example

```
[
    {
        "label_pairs": {
            "type": "nan"
        },
        "timestamp": 1559211080514607,
        "metric_name": "test_nan",
        "value": "nan"
    },
    {
        "label_pairs": {
            "type": "-inf"
        },
        "timestamp": 1559211080514607,
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        "metric_name": "test_inf",
        "value": "-inf"
    },
    {
        "label_pairs": {
            "type": "inf"
        },
        "timestamp": 1559211080514607,
        "metric_name": "test_inf",
        "value": "inf"
    }
]
```

To be used in Tarantool `http.server` as follows:

```
local httpd = require('http.server').new(...)
...
httpd:route({
        method = 'GET',
        path = '/metrics',
        public = true,
    },
    function(req)
        return req:render({
            text = json_exporter.export()
        })
    end
)
```

Plugin-specific API

We encourage you to use the following methods **only when developing a new plugin**.

`metrics.invoke_callbacks()`
Invokes the function registered via `metrics.register_callback(<callback>)`. Used in exporters.

`metrics.collectors()`
Designed to be used in exporters in favor of `metrics.collect()`.

Return a list of created collectors

`object collector_object`

`collector_object:collect()`

Примечание: You'll probably want to use `metrics.collectors()` instead.

Equivalent to:

```
for _, c in pairs(metrics.collectors()) do
    for _, obs in ipairs(c:collect()) do
        ...  -- handle observation
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    end
end
```

Return

Concatenation of `observation` **objects across all** created collectors.

```
{
    label_pairs: table,          -- `label_pairs` key-value table
    timestamp: ctype<uint64_t>, -- current system time (in microseconds)
    value: number,               -- current value
    metric_name: string,         -- collector
}
```

Rtype *table*

Writing custom plugins

Inside your main export function:

```
-- Invoke all callbacks registered via `metrics.register_callback(<callback-function>)`.
metrics.invoke_callbacks()

-- Loop over collectors
for _, c in pairs(metrics.collectors()) do
    ...

    -- Loop over instant observations in the collector.
    for _, obs in pairs(c:collect()) do
        -- Export observation `obs`
        ...
    end

end
```

4.6 Репликация

Механизм репликации позволяет сразу многим экземплярам Tarantool'а работать с копиями одних и тех же баз данных. При этом все базы остаются в синхронизированном состоянии благодаря тому, что каждый экземпляр может сообщать другим экземплярам о совершенных им изменениях.

Эта глава включает в себя следующие разделы:

4.6.1 Архитектура механизма репликации

Механизм репликации

Набор реплик (replica set) – это совокупность экземпляров, которые работают на копиях одной базы данных. У каждого экземпляра в наборе реплик есть роль: **мастер** или **реплика**.

Реплика получает все обновления от мастера, постоянно запрашивая и применяя данные *журнала упреждающей записи (WAL)*. Каждая запись в WAL представляет собой отдельный запрос на изменение данных в Tarantool'е, например, *INSERT*, *UPDATE* или *DELETE*. Такой записи присваивается монотонно возрастающее число, представляющее регистрационный номер в журнале (**LSN**). По сути, репликация в Tarantool'е является **построчной**: каждая команда на изменение данных полностью детерминирована и относится к отдельному *кортежу*. Однако в отличие от типичного построчного журнала, который содержит копии измененных строк полностью, WAL в Tarantool'е включает в себя копии запросов. Например, для запросов типа UPDATE (обновление) Tarantool сохранит только первичный ключ строки и операции обновления для экономии места.

Вызовы **хранимых процедур** не регистрируются в журнале упреждающей записи. Между тем, события по запросам **изменения фактических данных, которые выполняют Lua-скрипты**, регистрируются в журнале. Таким образом, возможное недетерминированное выполнение Lua гарантированно не приведет к рассинхронизации.

Операции по определению данных во **временных спейсах**, такие как создание/удаление, добавление индексов, усечение и т.д., регистрируются в журнале, поскольку информация о временных спейсах хранится в постоянных системных спейсах, например *box.space._space*. Операции по изменению данных во временных спейсах не регистрируются в журнале и не реплицируются.

Операции по изменению данных в спейсах с **локальной репликацией** (спейсах, *созданных* с параметром `is_local = true`) не регистрируются в журнале и не реплицируются.

Чтобы создать подходящее начальное состояние, к которому можно применить изменения из WAL-файла, для каждого экземпляра из набора реплик должен быть исходный набор *файлов контрольной точки* – .snap-файлы для memtx и .run-файлы для vinyl. Когда реплика включается в существующий набор реплик, она выбирает существующего мастера и автоматически загружает с него начальное состояние. Это называется **начальным включением**.

При первой настройке целого набора реплик нет мастера, который предоставил бы начальную контрольную точку. В таком случае реплики подключаются друг к другу и выбирают мастера, который затем создает начальный набор файлов контрольной точки и отправляет его всем репликам. Это называется **самонастройкой** набора реплик.

Когда реплика впервые подключается к мастеру (может быть много мастеров), она становится частью набора реплик. В последующих случаях она всегда должна подключаться к мастеру в этом наборе реплик. После подключения к мастеру реплика запрашивает все изменения, произошедшие с момента последнего локального LSN (может быть много LSN – у каждого мастера свой LSN).

Каждый набор реплик можно определить по глобально-уникальному идентификатору, который называется **UUID набора реплик**. Идентификатор создается мастером во время создания самой первой контрольной точки и является частью файла контрольной точки. Он хранится в системном спейсе *box.space._schema*. Пример:

```
tarantool> box.space._schema:select{'cluster'}
---
- - ['cluster', '6308acb9-9788-42fa-8101-2e0cb9d3c9a0']
...
```

Кроме того, каждому экземпляру в наборе реплик присваивается свой UUID, когда он включается в набор реплик. Такой глобально-уникальный идентификатор называется **UUID экземпляра**. UUID экземпляра проверяется, чтобы экземпляры не подключались к различным наборам реплик, например, из-за ошибки конфигурации. Уникальный идентификатор экземпляра также необходим для однократного применения строк от разных мастеров, то есть для многомастерной репликации. Вот почему каждая строка в журнале упреждающей записи, помимо номер записи в журнале, хранит идентификатор экземпляра, где запись была создана. Но использование UUID в качестве такого идентификатора заняло бы слишком много места в журнале упреждающей записи, поэтому экземпляру присваивается

целое число при включении в набор реплик. Это число, которое называется **ID экземпляра**, затем используется для ссылок на экземпляр в журнале упреждающей записи. Все идентификаторы хранятся в системном спейсе *box.space._cluster*. Например:

```
tarantool> box.space._cluster:select{}
---
- - [1, '88580b5c-4474-43ab-bd2b-2409a9af80d2']
...
```

Здесь ID экземпляра – 1 (уникальный номер в рамках набора реплик), а UUID экземпляра – 88580b5c-4474-43ab-bd2b-2409a9af80d2 (глобально уникальный).

Использование идентификаторов экземпляра также полезно для отслеживания состояния всего набора реплик. Например, *box.info.vclock* описывает состояние репликации в отношении каждого подключенного узла.

```
tarantool> box.info.vclock
---
- {1: 827, 2: 584}
...
```

Здесь `vclock` содержит номера записей в журнале (827 и 584) для экземпляров с идентификаторами экземпляра 1 и 2.

Начиная с Tarantool 1.7.7, появилась возможность для администраторов назначать UUID экземпляра и UUID набора реплик вместо сгенерированных системой значений – см. описание конфигурационного параметра *replicaset_uuid*.

Настройка репликации

Чтобы включить репликацию, необходимо указать два параметра в запросе `box.cfg{}`:

- *replication*, который определяет источники репликации, и
- *read_only* со значением `true` для реплики и `false` для мастера.

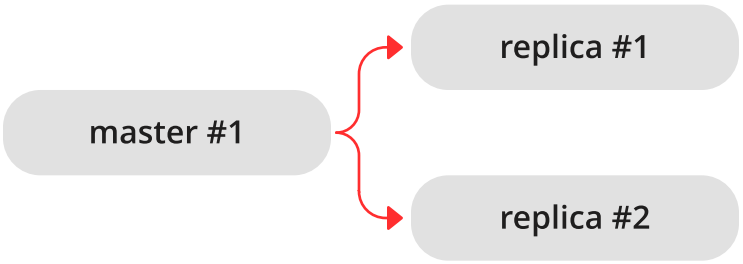
«Динамические» параметры репликации можно менять на лету, что позволяет назначать реплику на роль мастера и наоборот. Для этого используется запрос `box.cfg{}`.

Далее подробно рассмотрим пример *настройки набора реплик*.

Роли в репликации: мастер и реплика

Конфигурационный параметр *read_only* определяет роль в репликации (мастер или реплика). Рекомендованная роль для всех экземпляров в наборе реплик, кроме одного – «read-only» (реплика).

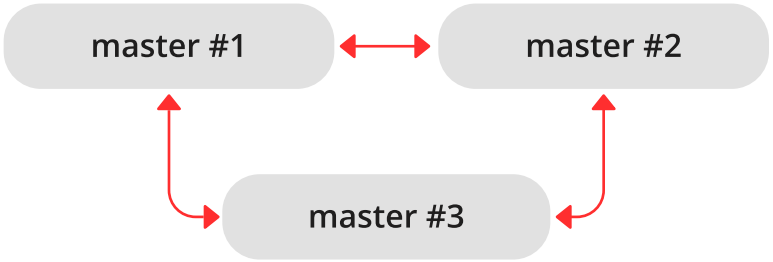
В конфигурации мастер-реплика каждое изменение, сделанное на мастере, будет отображаться на репликах, но не наоборот.



Простой набор реплик с двумя экземплярами, один из которых является мастером и расположен на одной машине, а другой – реплика – расположен на другой машине, дает два преимущества:

- **восстановление после отказа**, поскольку в случае отказа мастера реплика может взять работу на себя, и
- **балансировка нагрузки**, потому что клиенты во время запросов чтения могут подключаться к мастеру или к реплике.

В конфигурации **мастер-мастер** (которая также называется «многомастерной») каждое изменение на любом экземпляре будет также отображаться на другом.



Восстановление после отказа в таком случае также будет преимуществом, а балансировка нагрузки улучшится, поскольку любой экземпляр может обрабатывать запросы и на чтение, и на запись. В то же время, при многомастерной конфигурации необходимо понимать **гарантии репликации**, которые обеспечивает асинхронный протокол, внедренный в Tarantool.

Многомастерная репликация Tarantool'а гарантирует, что каждое изменение на каждом мастере передается на все экземпляры и применяется только один раз. Изменения с одного экземпляра применяются в том же порядке, что и на исходном экземпляре. Однако изменения с разных экземпляров могут сме-

шиваться и применяться в различном порядке на разных экземплярах. В определенных случаях это может привести к рассинхронизации.

Например, принимая, что проводятся только операции добавления данных в базу (т.е. она содержит только вставки), многомастерная конфигурация сработает хорошо. Если данные также удаляются, но порядок операций удаления на разных репликах не играет важной роли (например, DELETE используется для отсечения устаревших данных), то конфигурация мастер-мастер также безопасна.

Однако операции обновления UPDATE могут с легкостью привести к рассинхронизации. Например, операции присваивания и увеличения не обладают коммутативностью и могут привести к различным результатам, если применять их в различном порядке на разных экземплярах.

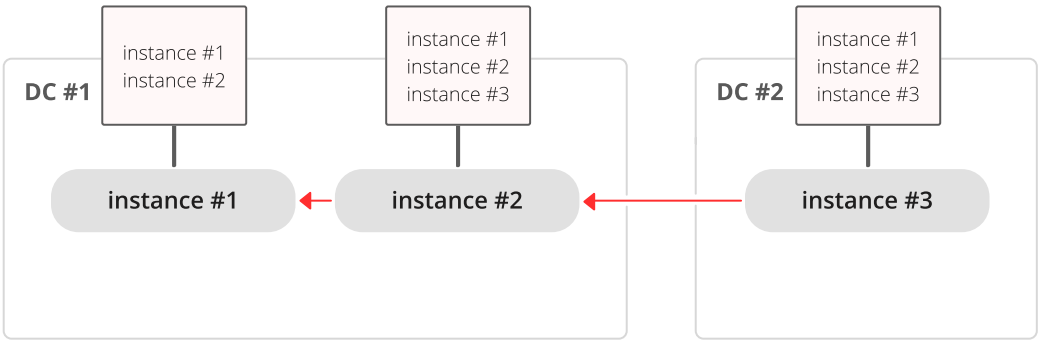
В общем смысле, безопасно использовать репликацию мастер-мастер в Tarantool'е, если все изменения в базе данных являются **коммутативными**: конечный результат не зависит от порядка, в котором применяются изменения. Дополнительную информацию о бесконфликтных типах реплицируемых данных можно получить здесь.

Топологии репликации: каскадная, кольцевая и полная ячеистая

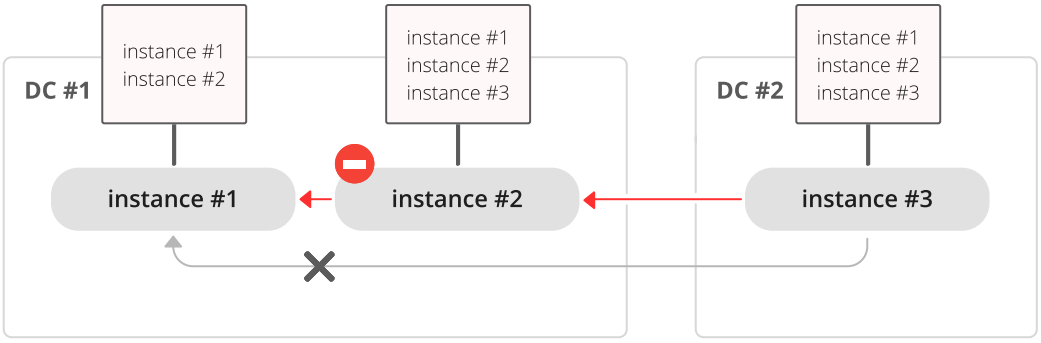
Топология репликации определяется в конфигурационном параметре *replication*. Рекомендована **полная ячеистая** конфигурация, поскольку она облегчает возможное восстановление после сбоя.

Некоторые СУБД предлагают топологии **каскадной репликации**: создание реплики на реплике. Tarantool не рекомендует такие настройки.

Недостаток каскадного набора реплик заключается в том, что некоторые экземпляры не подключаются к другим экземплярам, поэтому не могут получать от них изменения. Одно важное изменение, которое следует передавать на все экземпляры в наборе реплик – запись в системный спейс `box.space._cluster` с UUID набора реплик. Не зная UUID набора реплик, мастер отклоняет подключения от таких экземпляров при изменении топологии репликации. Вот как это может произойти:

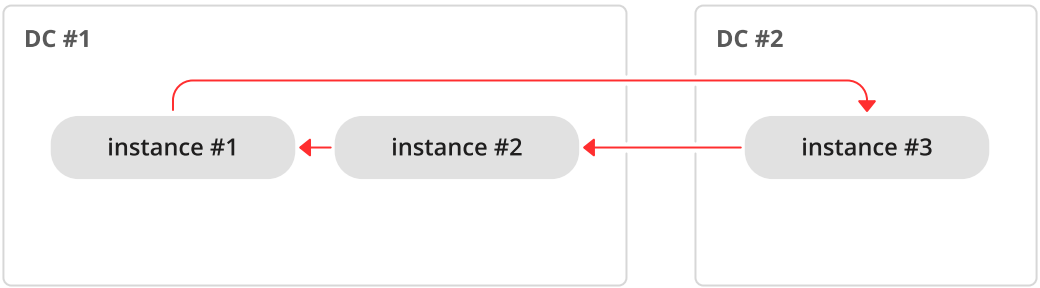


У нас есть цепочка из трех экземпляров. Экземпляр №1 содержит записи для экземпляров №1 и №2 в спейсе `_cluster`. Экземпляры №2 и №3 содержат записи для экземпляров №1, №2 и №3 в своих спейсах `_cluster`.



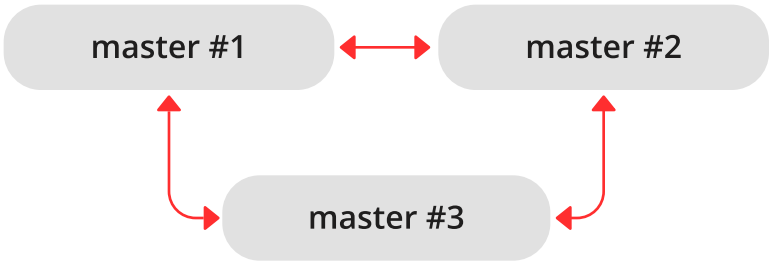
Теперь экземпляр №2 неисправен. Экземпляр №3 пытается подключиться к экземпляру №1, как к новому мастеру, но мастер отклоняет подключение, поскольку не содержит запись для экземпляра №3.

Тем не менее, **кольцевая топология** поддерживается:



Поэтому если необходима каскадная топология, можно первоначально создать кольцо, чтобы все экземпляры знали UUID друг друга, а затем разъединить цепочку в необходимом месте.

Как бы то ни было, для репликации мастер-мастер рекомендуется **полная ячеистая** топология:



В таком случае можно решить, где расположить экземпляры ячейки – в том же центре обработки данных или разместить в нескольких центрах. Tarantool будет автоматически следить за тем, что каждая строка применяется однократно на каждом экземпляре. Чтобы удалить экземпляр из ячейки после отказа, просто измените конфигурационный параметр `replication`.

Таким образом можно обеспечить доступность всего кластера в случае локального отказа, например отказа одного экземпляра в одном центре обработки данных, а также в случае отказа всего центра обработки данных.

Максимальное количество реплик в ячейке – 32.

4.6.2 Настройка набора реплик

Настройка репликации мастер-реплика

Сначала настроим простой набор **мастер-реплика** с двумя экземплярами, каждый из которых находится на отдельном сервере. Для удобства администрирования сделаем *файлы экземпляров* практически одинаковыми.

Ниже пример файла экземпляра для мастера:

```
-- файл экземпляра для мастера
box.cfg{
  listen = 3301,
  replication = {'replicator:password@192.168.0.101:3301',  -- URI мастера
                 'replicator:password@192.168.0.102:3301'}, -- URI реплики
  read_only = false
}
box.once("schema", function()
  box.schema.user.create('replicator', {password = 'password'})
  box.schema.user.grant('replicator', 'replication') -- настроить роль для репликации
  box.schema.space.create("test")
```

(continues on next page)

(продолжение с предыдущей страницы)

```
  box.space.test:create_index("primary")
  print('box.once executed on master')
end)
```

где:

- параметр *listen* в `box.cfg{}` определяет URI (порт 3301 в нашем примере), на котором мастер может принимать подключения от реплик.
- параметр *replication* в `box.cfg{}` определяет URI, на которых все экземпляры в наборе реплик могут принимать подключения. Он включает в себя также URI реплики, хотя реплики в данном случае не является источником репликации. Этот параметр является обязательным только при настройке кластеров с конфигурацией master-master или full-mesh.

  Примечание: Для целей безопасности рекомендуем администраторам не допускать репликацию из неавторизованных источников с помощью установки пароля для каждого пользователя, у которого есть *роль* для репликации. Таким образом, *URI* для параметра `replication` должен иметь развернутый вид `username:password@host:port`.

- параметр *read_only = false* разрешает операции по изменению данных на экземпляре и заставляет данный экземпляр работать в качестве мастера, а не реплики. *Это единственное значение параметра, которое отличается в наших файлах экземпляров.*
- функция *box.once()* содержит логику инициализации базы данных, которая должна выполняться однократно в течение срока работы набора реплик.

В данном примере создаем спейс с первичным индексом и пользователя для целей репликации. Также выполним команду `print('box.once executed on master')`, чтобы позднее увидеть в консоли, была ли выполнена функция `box.once()`.

Примечание: Репликация требует настройки прав. Права на доступ к спейсам можно задать напрямую для пользователя, под чьим именем запущен экземпляр. Но обычно права на доступ к спейсам задаются с помощью *роли*, которая затем присваивается пользователю, под чьим именем запущена реплика.

Здесь мы используем предварительно определенную роль Tarantool'а под названием «replication», которая по умолчанию предоставляет права на чтение всех объектов в базе данных («universe»), а также сможем настроить необходимые права для этой роли.

В файле экземпляра для реплики устанавливаем значение «true» для параметра `read_only` и выполняем команду `print('box.once executed on replica')`, чтобы позднее убедиться, что `box.once()` выполняется только однократно. В других отношениях файл экземпляра для реплики совпадает с файлом экземпляра для мастера.

```
-- файл экземпляра для реплики
box.cfg{
  listen = 3301,
  replication = {'replicator:password@192.168.0.101:3301',  -- URI мастера
                 'replicator:password@192.168.0.102:3301'}, -- URI реплики
  read_only = true
}
box.once("schema", function()
  box.schema.user.create('replicator', {password = 'password'})
```

(continues on next page)

(продолжение с предыдущей страницы)

```
  box.schema.user.grant('replicator', 'replication') -- настроить роль для репликации
  box.schema.space.create("test")
  box.space.test:create_index("primary")
  print('box.once executed on replica')
end)
```

Примечание: Реплика не берет конфигурационные параметры с мастера, например настройки запуска *фоновой программы для работы с контрольными точками* на мастере. Чтобы получить те же настройки на реплике, необходимо задать их явным образом.

Теперь можно запустить два экземпляра. Мастер...

```
$ # запуск мастера
$ tarantool master.lua
2017-06-14 14:12:03.847 [18933] main/101/master.lua C> version 1.7.4-52-g980d30092
2017-06-14 14:12:03.848 [18933] main/101/master.lua C> log level 5
2017-06-14 14:12:03.849 [18933] main/101/master.lua I> mapping 268435456 bytes for tuple arena...
2017-06-14 14:12:03.859 [18933] iproto/101/main I> binary: bound to [::]:3301
2017-06-14 14:12:03.861 [18933] main/105/applier/replicator@192.168.0. I> can't connect to master
2017-06-14 14:12:03.861 [18933] main/105/applier/replicator@192.168.0. coio.cc:105 !> SystemError␣
↪connect, called on fd 14, aka 192.168.0.102:56736: Connection refused
2017-06-14 14:12:03.861 [18933] main/105/applier/replicator@192.168.0. I> will retry every 1 second
2017-06-14 14:12:03.861 [18933] main/104/applier/replicator@192.168.0. I> remote master is 1.7.4␣
↪at 192.168.0.101:3301
2017-06-14 14:12:19.878 [18933] main/105/applier/replicator@192.168.0. I> remote master is 1.7.4␣
↪at 192.168.0.102:3301
2017-06-14 14:12:19.879 [18933] main/101/master.lua I> initializing an empty data directory
2017-06-14 14:12:19.908 [18933] snapshot/101/main I> saving snapshot `/var/lib/tarantool/master/
↪00000000000000000000.snap.inprogress'
2017-06-14 14:12:19.914 [18933] snapshot/101/main I> done
2017-06-14 14:12:19.914 [18933] main/101/master.lua I> vinyl checkpoint done
2017-06-14 14:12:19.917 [18933] main/101/master.lua I> ready to accept requests
2017-06-14 14:12:19.918 [18933] main/105/applier/replicator@192.168.0. I> failed to authenticate
2017-06-14 14:12:19.918 [18933] main/105/applier/replicator@192.168.0. xrow.cc:431 E> ER_LOADING:␣
↪Instance bootstrap hasn't finished yet
box.once executed on master
2017-06-14 14:12:19.920 [18933] main C> entering the event loop
```

... (выведенный результат подтверждает, что функция `box.once()` была выполнена на мастере) – и реплику:

```
$ # запуск реплики
$ tarantool replica.lua
2017-06-14 14:12:19.486 [18934] main/101/replica.lua C> version 1.7.4-52-g980d30092
2017-06-14 14:12:19.486 [18934] main/101/replica.lua C> log level 5
2017-06-14 14:12:19.487 [18934] main/101/replica.lua I> mapping 268435456 bytes for tuple arena...
2017-06-14 14:12:19.494 [18934] iproto/101/main I> binary: bound to [::]:3311
2017-06-14 14:12:19.495 [18934] main/104/applier/replicator@192.168.0. I> remote master is 1.7.4␣
↪at 192.168.0.101:3301
2017-06-14 14:12:19.495 [18934] main/105/applier/replicator@192.168.0. I> remote master is 1.7.4␣
↪at 192.168.0.102:3302
2017-06-14 14:12:19.496 [18934] main/104/applier/replicator@192.168.0. I> failed to authenticate
2017-06-14 14:12:19.496 [18934] main/104/applier/replicator@192.168.0. xrow.cc:431 E> ER_LOADING:␣
↪Instance bootstrap hasn't finished yet
```

В обоих журналах есть сообщения о том, что реплика получила настройки от мастера:

```
$ # настройка реплики (из журнала мастера)
<...>
2017-06-14 14:12:20.503 [18933] main/106/main I> initial data sent.
2017-06-14 14:12:20.505 [18933] relay/[::ffff:192.168.0.101]:/101/main I> recover from `/var/lib/
↪tarantool/master/00000000000000000000.xlog'
2017-06-14 14:12:20.505 [18933] main/106/main I> final data sent.
2017-06-14 14:12:20.522 [18933] relay/[::ffff:192.168.0.101]:/101/main I> recover from `/Users/e.
↪shebunyaeva/work/tarantool-test-repl/master_dir/00000000000000000000.xlog'
2017-06-14 14:12:20.922 [18933] main/105/applier/replicator@192.168.0. I> authenticated
```

```
$ # настройка реплики (из журнала реплики)
<...>
2017-06-14 14:12:20.498 [18934] main/104/applier/replicator@192.168.0. I> authenticated
2017-06-14 14:12:20.498 [18934] main/101/replica.lua I> bootstrapping replica from 192.168.0.
↪101:3301
2017-06-14 14:12:20.512 [18934] main/104/applier/replicator@192.168.0. I> initial data received
2017-06-14 14:12:20.512 [18934] main/104/applier/replicator@192.168.0. I> final data received
2017-06-14 14:12:20.517 [18934] snapshot/101/main I> saving snapshot `/var/lib/tarantool/replica/
↪00000000000000000005.snap.inprogress'
2017-06-14 14:12:20.518 [18934] snapshot/101/main I> done
2017-06-14 14:12:20.519 [18934] main/101/replica.lua I> vinyl checkpoint done
2017-06-14 14:12:20.520 [18934] main/101/replica.lua I> ready to accept requests
2017-06-14 14:12:20.520 [18934] main/101/replica.lua I> set 'read_only' configuration option to␣
↪true
2017-06-14 14:12:20.520 [18934] main C> entering the event loop
```

Обратите внимание, что функция `box.once()` была выполнена только на мастере, хотя мы добавили `box.once()` в оба файла экземпляра.

Также можно было сначала запустить реплику:

```
$ # запуск реплики
$ tarantool replica.lua
2017-06-14 14:35:36.763 [18952] main/101/replica.lua C> version 1.7.4-52-g980d30092
2017-06-14 14:35:36.765 [18952] main/101/replica.lua C> log level 5
2017-06-14 14:35:36.765 [18952] main/101/replica.lua I> mapping 268435456 bytes for tuple arena...
2017-06-14 14:35:36.772 [18952] iproto/101/main I> binary: bound to [::]:3301
2017-06-14 14:35:36.772 [18952] main/104/applier/replicator@192.168.0. I> can't connect to master
2017-06-14 14:35:36.772 [18952] main/104/applier/replicator@192.168.0. coio.cc:105 !> SystemError␣
↪connect, called on fd 13, aka 192.168.0.101:56820: Connection refused
2017-06-14 14:35:36.772 [18952] main/104/applier/replicator@192.168.0. I> will retry every 1 second
2017-06-14 14:35:36.772 [18952] main/105/applier/replicator@192.168.0. I> remote master is 1.7.4␣
↪at 192.168.0.102:3301
```

... а затем уже мастера:

```
$ # запуск мастера
$ tarantool master.lua
2017-06-14 14:35:43.701 [18953] main/101/master.lua C> version 1.7.4-52-g980d30092
2017-06-14 14:35:43.702 [18953] main/101/master.lua C> log level 5
2017-06-14 14:35:43.702 [18953] main/101/master.lua I> mapping 268435456 bytes for tuple arena...
2017-06-14 14:35:43.709 [18953] iproto/101/main I> binary: bound to [::]:3301
2017-06-14 14:35:43.709 [18953] main/105/applier/replicator@192.168.0. I> remote master is 1.7.4␣
↪at 192.168.0.102:3301
2017-06-14 14:35:43.709 [18953] main/104/applier/replicator@192.168.0. I> remote master is 1.7.4␣
↪at 192.168.0.101:3301
```

(continues on next page)

(продолжение с предыдущей страницы)

```
2017-06-14 14:35:43.709 [18953] main/101/master.lua I> initializing an empty data directory
2017-06-14 14:35:43.721 [18953] snapshot/101/main I> saving snapshot `/var/lib/tarantool/master/
↪00000000000000000000.snap.inprogress'
2017-06-14 14:35:43.722 [18953] snapshot/101/main I> done
2017-06-14 14:35:43.723 [18953] main/101/master.lua I> vinyl checkpoint done
2017-06-14 14:35:43.723 [18953] main/101/master.lua I> ready to accept requests
2017-06-14 14:35:43.724 [18953] main/105/applier/replicator@192.168.0. I> failed to authenticate
2017-06-14 14:35:43.724 [18953] main/105/applier/replicator@192.168.0. xrow.cc:431 E> ER_LOADING:␣
↪Instance bootstrap hasn't finished yet
box.once executed on master
2017-06-14 14:35:43.726 [18953] main C> entering the event loop
2017-06-14 14:35:43.779 [18953] main/103/main I> initial data sent.
2017-06-14 14:35:43.780 [18953] relay/[::ffff:192.168.0.101]:/101/main I> recover from `/var/lib/
↪tarantool/master/00000000000000000000.xlog'
2017-06-14 14:35:43.780 [18953] main/103/main I> final data sent.
2017-06-14 14:35:43.796 [18953] relay/[::ffff:192.168.0.102]:/101/main I> recover from `/var/lib/
↪tarantool/master/00000000000000000000.xlog'
2017-06-14 14:35:44.726 [18953] main/105/applier/replicator@192.168.0. I> authenticated
```

В данном случае реплика ожидает доступности мастера, поэтому порядок запуска не имеет значения. Наша функция `box.once()` также будет выполняться однократно, только на мастере.

```
$ # реплика в итоге подключена к мастеру
$ # и получила настройки (из журнала реплики)
2017-06-14 14:35:43.777 [18952] main/104/applier/replicator@192.168.0. I> remote master is 1.7.4␣
↪at 192.168.0.101:3301
2017-06-14 14:35:43.777 [18952] main/104/applier/replicator@192.168.0. I> authenticated
2017-06-14 14:35:43.777 [18952] main/101/replica.lua I> bootstrapping replica from 192.168.0.
↪199:3310
2017-06-14 14:35:43.788 [18952] main/104/applier/replicator@192.168.0. I> initial data received
2017-06-14 14:35:43.789 [18952] main/104/applier/replicator@192.168.0. I> final data received
2017-06-14 14:35:43.793 [18952] snapshot/101/main I> saving snapshot `/var/lib/tarantool/replica/
↪00000000000000000005.snap.inprogress'
2017-06-14 14:35:43.793 [18952] snapshot/101/main I> done
2017-06-14 14:35:43.795 [18952] main/101/replica.lua I> vinyl checkpoint done
2017-06-14 14:35:43.795 [18952] main/101/replica.lua I> ready to accept requests
2017-06-14 14:35:43.795 [18952] main/101/replica.lua I> set 'read_only' configuration option to␣
↪true
2017-06-14 14:35:43.795 [18952] main C> entering the event loop
```

Контролируемое восстановление после сбоя

Чтобы провести **контролируемое восстановление после сбоя**, то есть поменять роли мастера и реплики, нужно лишь настроить параметры `read_only=true` на мастере и `read_only=false` на реплике. Порядок действий в данном случае имеет значение. Если система принята в эксплуатацию, нам не нужна параллельная запись на реплике и на мастере. Нежелательно также, чтобы новая реплика принимала запись, пока не получит все реплицируемые данные со старого мастера. Чтобы сопоставить состояние реплики и мастера, можно использовать *box.info.signature*.

1. Настройте `read_only=true` на мастере.

   ```
   # на мастере
   tarantool> box.cfg{read_only=true}
   ```

2. Зарегистрируйте текущее состояние мастера с помощью `box.info.signature`, которое содержит общее количество всех LSN в векторных часах мастера.

```
# на мастере
tarantool> box.info.signature
```

3. Подождите, пока сигнатура реплики не совпадет с сигнатурой мастера.

```
# на реплике
tarantool> box.info.signature
```

4. Настройте `read_only=false` на реплике, чтобы запустить операции записи данных.

```
# на реплике
tarantool> box.cfg{read_only=false}
```

Эти шаги нужны для того, чтобы реплика гарантированно не принимала новые записи, пока не получит данные от мастера.

Настройка репликации мастер-мастер

Теперь настроим набор с двумя экземплярами **мастер-мастер**. Для удобства управления сделаем файлы экземпляра для мастера №1 и мастера №2 практически одинаковыми.

Переиспользуем файл экземпляра для мастера из вышеописанного *примера мастер-реплика*.

```
-- файл экземпляра для любого из двух мастеров
  box.cfg{
  listen      = 3301,
  replication = {'replicator:password@192.168.0.101:3301',  -- URI мастера 1
                 'replicator:password@192.168.0.102:3301'}, -- URI мастера 2
  read_only   = false
}
box.once("schema", function()
  box.schema.user.create('replicator', {password = 'password'})
  box.schema.user.grant('replicator', 'replication') -- настроить роль для репликации
  box.schema.space.create("test")
  box.space.test:create_index("primary")
  print('box.once executed on master #1')
end)
```

В параметре *replication* определим URI обоих мастеров в наборе реплик и выполним команду `print('box.once executed on master #1')`, чтобы увидеть, когда и где будет выполнена логика функции `box.once()`.

Теперь можно запустить оба мастера. Повторимся, что порядок запуска не имеет значения. Логика `box.once()` также будет выполняться лишь однократно на мастере, который будет выбран лидером (*leader*) в наборе реплик при настройке.

```
$ # запуск мастера №1
$ tarantool master1.lua
2017-06-14 15:39:03.062 [47021] main/101/master1.lua C> version 1.7.4-52-g980d30092
2017-06-14 15:39:03.062 [47021] main/101/master1.lua C> log level 5
2017-06-14 15:39:03.063 [47021] main/101/master1.lua I> mapping 268435456 bytes for tuple arena...
2017-06-14 15:39:03.065 [47021] iproto/101/main I> binary: bound to [::]:3301
```

(continues on next page)

(продолжение с предыдущей страницы)

```
2017-06-14 15:39:03.065 [47021] main/105/applier/replicator@192.168.0.10 I> can't connect to master
2017-06-14 15:39:03.065 [47021] main/105/applier/replicator@192.168.0.10 coio.cc:107 !>␣
↪SystemError connect, called on fd 14, aka 192.168.0.102:57110: Connection refused
2017-06-14 15:39:03.065 [47021] main/105/applier/replicator@192.168.0.10 I> will retry every 1␣
↪second
2017-06-14 15:39:03.065 [47021] main/104/applier/replicator@192.168.0.10 I> remote master is 1.7.4␣
↪at 192.168.0.101:3301
2017-06-14 15:39:08.070 [47021] main/105/applier/replicator@192.168.0.10 I> remote master is 1.7.4␣
↪at 192.168.0.102:3301
2017-06-14 15:39:08.071 [47021] main/105/applier/replicator@192.168.0.10 I> authenticated
2017-06-14 15:39:08.071 [47021] main/101/master1.lua I> bootstrapping replica from 192.168.0.
↪102:3301
2017-06-14 15:39:08.073 [47021] main/105/applier/replicator@192.168.0.10 I> initial data received
2017-06-14 15:39:08.074 [47021] main/105/applier/replicator@192.168.0.10 I> final data received
2017-06-14 15:39:08.074 [47021] snapshot/101/main I> saving snapshot `/Users/e.shebunyaeva/work/
↪tarantool-test-repl/master1_dir/00000000000000000008.snap.inprogress'
2017-06-14 15:39:08.074 [47021] snapshot/101/main I> done
2017-06-14 15:39:08.076 [47021] main/101/master1.lua I> vinyl checkpoint done
2017-06-14 15:39:08.076 [47021] main/101/master1.lua I> ready to accept requests
box.once executed on master #1
2017-06-14 15:39:08.077 [47021] main C> entering the event loop
```

```
$ # запуск мастера №2
$ tarantool master2.lua
2017-06-14 15:39:07.452 [47022] main/101/master2.lua C> version 1.7.4-52-g980d30092
2017-06-14 15:39:07.453 [47022] main/101/master2.lua C> log level 5
2017-06-14 15:39:07.453 [47022] main/101/master2.lua I> mapping 268435456 bytes for tuple arena...
2017-06-14 15:39:07.455 [47022] iproto/101/main I> binary: bound to [::]:3301
2017-06-14 15:39:07.455 [47022] main/104/applier/replicator@192.168.0.19 I> remote master is 1.7.4␣
↪at 192.168.0.101:3301
2017-06-14 15:39:07.455 [47022] main/105/applier/replicator@192.168.0.10 I> remote master is 1.7.4␣
↪at 192.168.0.102:3301
2017-06-14 15:39:07.455 [47022] main/101/master2.lua I> initializing an empty data directory
2017-06-14 15:39:07.457 [47022] snapshot/101/main I> saving snapshot `/Users/e.shebunyaeva/work/
↪tarantool-test-repl/master2_dir/00000000000000000000.snap.inprogress'
2017-06-14 15:39:07.457 [47022] snapshot/101/main I> done
2017-06-14 15:39:07.458 [47022] main/101/master2.lua I> vinyl checkpoint done
2017-06-14 15:39:07.459 [47022] main/101/master2.lua I> ready to accept requests
2017-06-14 15:39:07.460 [47022] main C> entering the event loop
2017-06-14 15:39:08.072 [47022] main/103/main I> initial data sent.
2017-06-14 15:39:08.073 [47022] relay/[::ffff:192.168.0.102]:/101/main I> recover from `/Users/e.
↪shebunyaeva/work/tarantool-test-repl/master2_dir/00000000000000000000.xlog'
2017-06-14 15:39:08.073 [47022] main/103/main I> final data sent.
2017-06-14 15:39:08.077 [47022] relay/[::ffff:192.168.0.102]:/101/main I> recover from `/Users/e.
↪shebunyaeva/work/tarantool-test-repl/master2_dir/00000000000000000000.xlog'
2017-06-14 15:39:08.461 [47022] main/104/applier/replicator@192.168.0.10 I> authenticated
```

4.6.3 Добавление экземпляров

Добавление реплики

Чтобы добавить вторую **реплику** в набор реплик с конфигурацией **мастер-реплика** из нашего *примера настройки*, необходим аналог файла экземпляра, который мы создали для первой реплики в этом наборе:

```
-- файл экземпляра для реплики №2
box.cfg{
  listen = 3301,
  replication = {'replicator:password@192.168.0.101:3301',  -- URI мастера
                 'replicator:password@192.168.0.102:3301',  -- URI реплики №1
                 'replicator:password@192.168.0.103:3301'}, -- URI реплики №2
  read_only = true
}
box.once("schema", function()
   box.schema.user.create('replicator', {password = 'password'})
   box.schema.user.grant('replicator', 'replication') -- предоставить роль для репликации
   box.schema.space.create("test")
   box.space.test:create_index("primary")
   print('box.once executed on replica #2')
end)
```

Здесь мы добавляем URI реплики №2 в параметр *replication*, так что теперь он содержит три URI.

После запуска новая реплика подключается к мастер-серверу и получает от него журнал упреждающей записи и файлы снимков:

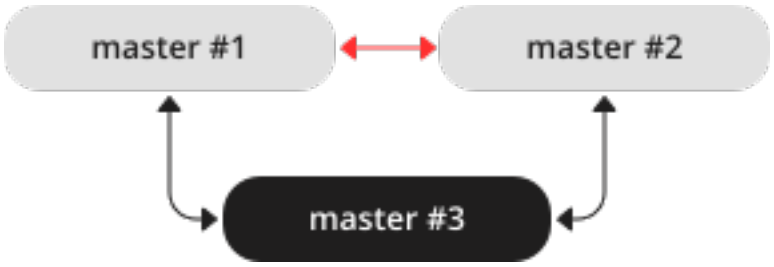
```
$ # запуск реплики №2
$ tarantool replica2.lua
2017-06-14 14:54:33.927 [46945] main/101/replica2.lua C> version 1.7.4-52-g980d30092
2017-06-14 14:54:33.927 [46945] main/101/replica2.lua C> log level 5
2017-06-14 14:54:33.928 [46945] main/101/replica2.lua I> mapping 268435456 bytes for tuple arena...
2017-06-14 14:54:33.930 [46945] main/104/applier/replicator@192.168.0.10 I> remote master is 1.7.4␣
↪at 192.168.0.101:3301
2017-06-14 14:54:33.930 [46945] main/104/applier/replicator@192.168.0.10 I> authenticated
2017-06-14 14:54:33.930 [46945] main/101/replica2.lua I> bootstrapping replica from 192.168.0.
↪101:3301
2017-06-14 14:54:33.933 [46945] main/104/applier/replicator@192.168.0.10 I> initial data received
2017-06-14 14:54:33.933 [46945] main/104/applier/replicator@192.168.0.10 I> final data received
2017-06-14 14:54:33.934 [46945] snapshot/101/main I> saving snapshot `/var/lib/tarantool/replica2/
↪00000000000000000010.snap.inprogress'
2017-06-14 14:54:33.934 [46945] snapshot/101/main I> done
2017-06-14 14:54:33.935 [46945] main/101/replica2.lua I> vinyl checkpoint done
2017-06-14 14:54:33.935 [46945] main/101/replica2.lua I> ready to accept requests
2017-06-14 14:54:33.935 [46945] main/101/replica2.lua I> set 'read_only' configuration option to␣
↪true
2017-06-14 14:54:33.936 [46945] main C> entering the event loop
```

Поскольку мы добавляем экземпляр только для чтения (read-only), нет необходимости в динамическом

обновлении параметра `replication` на других работающих экземплярах. Такое обновление необходимо, если бы мы *добавляли мастера*.

Тем не менее, рекомендуем указать URI реплики №3 во всех файлах экземпляра в наборе реплик. Это поможет сохранить единообразие файлов и согласовать их с текущей топологией репликации, а также не допустить ошибок конфигурации в случае последующего обновления конфигурации и перезапуска набора реплик.

Добавление мастера



Чтобы добавить третьего мастера в набор реплик с конфигурацией **мастер-мастер** из нашего *примера настройки*, необходим аналог файлов экземпляров, которые мы создали для настройки других мастеров в этом наборе:

```
-- файл экземпляра для мастера №3
box.cfg{
  listen      = 3301,
  replication = {'replicator:password@192.168.0.101:3301',  -- URI мастера №1
                 'replicator:password@192.168.0.102:3301',  -- URI мастера №2
                 'replicator:password@192.168.0.103:3301'}, -- URI мастера №3
  read_only   = true, -- временно только для чтения
}
box.once("schema", function()
   box.schema.user.create('replicator', {password = 'password'})
   box.schema.user.grant('replicator', 'replication') -- выдача роли для репликации
   box.schema.space.create("test")
   box.space.test:create_index("primary")
end)
```

Здесь мы вносим следующие изменения:

- Добавить URI мастера №3 в параметр *replication*.
- Временно укажите *read_only=true*, чтобы отключить операции по изменению данных на этом экземпляре. После запуска мастер №3 будет работать в качестве реплики, пока не получит все данные от других мастеров в наборе реплик.

После запуска мастер №3 подключается к другим мастер-экземплярам и получает от них файлы журнала упреждающей записи и файлы снимков:

```
$ # запуск мастера №3
$ tarantool master3.lua
2017-06-14 17:10:00.556 [47121] main/101/master3.lua C> version 1.7.4-52-g980d30092
2017-06-14 17:10:00.557 [47121] main/101/master3.lua C> log level 5
2017-06-14 17:10:00.557 [47121] main/101/master3.lua I> mapping 268435456 bytes for tuple arena...
2017-06-14 17:10:00.559 [47121] iproto/101/main I> binary: bound to [::]:3301
2017-06-14 17:10:00.559 [47121] main/104/applier/replicator@192.168.0.10 I> remote master is 1.7.4␣
↪at 192.168.0.101:3301
```

(continues on next page)

(продолжение с предыдущей страницы)

```
2017-06-14 17:10:00.559 [47121] main/105/applier/replicator@192.168.0.10 I> remote master is 1.7.4␣
↪at 192.168.0.102:3301
2017-06-14 17:10:00.559 [47121] main/106/applier/replicator@192.168.0.10 I> remote master is 1.7.4␣
↪at 192.168.0.103:3301
2017-06-14 17:10:00.559 [47121] main/105/applier/replicator@192.168.0.10 I> authenticated
2017-06-14 17:10:00.559 [47121] main/101/master3.lua I> bootstrapping replica from 192.168.0.
↪102:3301
2017-06-14 17:10:00.562 [47121] main/105/applier/replicator@192.168.0.10 I> initial data received
2017-06-14 17:10:00.562 [47121] main/105/applier/replicator@192.168.0.10 I> final data received
2017-06-14 17:10:00.562 [47121] snapshot/101/main I> saving snapshot `/Users/e.shebunyaeva/work/
↪tarantool-test-repl/master3_dir/00000000000000000009.snap.inprogress'
2017-06-14 17:10:00.562 [47121] snapshot/101/main I> done
2017-06-14 17:10:00.564 [47121] main/101/master3.lua I> vinyl checkpoint done
2017-06-14 17:10:00.564 [47121] main/101/master3.lua I> ready to accept requests
2017-06-14 17:10:00.565 [47121] main/101/master3.lua I> set 'read_only' configuration option to␣
↪true
2017-06-14 17:10:00.565 [47121] main C> entering the event loop
2017-06-14 17:10:00.565 [47121] main/104/applier/replicator@192.168.0.10 I> authenticated
```

Затем добавляем URI мастера №3 в параметр `replication` на существующих мастерах. В конфигурации репликации используются динамические параметры, поэтому необходимо только выполнить запрос `box.cfg{}` на каждом работающем экземпляре:

```
# добавление URI мастера №3 в источники репликации
tarantool> box.cfg{replication =
         > {'replicator:password@192.168.0.101:3301',
         > 'replicator:password@192.168.0.102:3301',
         > 'replicator:password@192.168.0.103:3301'}}
---
...
```

Когда мастер №3 получает все необходимые изменения от других мастеров, можно отключить режим только для чтения:

```
# назначение мастера №3 настоящим мастером
tarantool> box.cfg{read_only=false}
---
...
```

Также рекомендуется указать URI мастера №3 во всех файлах экземпляра, чтобы сохранить единообразие файлов и согласовать их с текущей топологией репликации.

Статус orphan (одиночный)

Начиная с версии Tarantool'а 1.9, процедура подключения реплики к набору реплик изменяется. Во время `box.cfg()` экземпляр попытается подключиться ко всем мастерам, указанным в *box.cfg.replication*. Если не было успешно выполнено подключение к количеству мастеров, указанному в *replication_connect_quorum*, экземпляр переходит в **статус orphan** (одиночный). Когда экземпляр находится в статусе orphan, он доступен только для чтения.

Чтобы «подключиться» к мастеру, реплика должна «установить соединение» с узлом мастера, а затем «выполнить синхронизацию».

«Установка соединения» означает контакт с мастером по физической сети и получение подтверждения. Если нет подтверждения соединения через *box.replication_connect_timeout* секунд (обычно 4 секунды), и повторные попытки подключения не сработали, то соединение не установлено.

«Синхронизация» означает получение обновлений от мастера для создания локальной копии базы данных. Синхронизация завершена, когда реплика получила все обновления или хотя бы получила достаточное количество обновлений, чтобы отставание реплики (см. *replication.upstream.lag* в `box.info()`) было меньше или равно количеству секунд, указанному в *box.cfg.replication_sync_lag*. Если значение `replication_sync_lag` не задано (nil) или указано как «TIMEOUT_INFINITY», то реплика пропускает шаг «синхронизация» и сразу же переходит на «отслеживание».

Чтобы вывести узел из одиночного статуса, нужно синхронизировать его с достаточным (т.е. равным *replication_connect_quorum*) количеством других узлов. Этого можно добиться, выполнив любое из следующих действий:

- Уменьшить значение *replication_connect_quorum*.
- Убрать из списка `box.cfg.replication` недоступные и прочие узлы, с которыми нельзя синхронизироваться.
- Вообще задать `""` (пустую строку) в качестве значения `box.cfg.replication`.

Возможны следующие ситуации.

Ситуация 1: настройка

Здесь впервые происходит вызов `box.cfg{}`. Реплика подключается, но набора реплик пока нет.

1. Установка статуса „orphan“ (одиночный).
2. Попытка установить соединение со всеми узлами из `box.cfg.replication` или с количеством узлов, указанным в параметре *replication_connect_quorum*. Допускаются три повторные попытки за 30 секунд, поскольку идет стадия настройки, параметр *replication_connect_timeout* не учитывается.
3. Прекращение работы и выдача ошибки в случае отсутствия соединения со всеми узлами в `box.cfg.replication` или *replication_connect_quorum*.
4. Экземпляр может быть выбран в качестве лидера „leader“ в наборе реплик. Критерии выбора лидера включают в себя значение vclock (чем больше, тем лучше), а также доступность только для чтения или для чтения и записи (лучше всего для чтения и записи, кроме случаев, когда других вариантов нет). Лидер является мастером, к которому должны подключиться другие экземпляры. Лидер является мастером, который выполняет функции *box_once()*.
5. Если данный экземпляр выбран лидером набора реплик, выполняется «самонастройка»:

   a. Установка статуса „running“ (запущен).

   b. Возврат из `box.cfg{}`.

   В противном случае, данный экземпляр будет репликой, которая подключается к существующему набору реплик, поэтому:

   a. Настройка от лидера. См. примеры в разделе *Настройка набора реплик*.

   b. Синхронизация со всеми остальными узлами в наборе реплик в фоновом режиме.

Ситуация 2: восстановление

Здесь вызов `box.cfg{}` происходит не впервые, а повторно для осуществления восстановления.

1. Проведение *восстановления* из последнего локального снимка и WAL-файлов.
2. Установить соединение с количеством узлов не меньшим, чем *replication_connect_quorum*. Если не получается – установить статус „orphan“. (Попытки синхронизации будут повторяться в фоновом режиме, и когда/если они окажутся успешными, статус „orphan“ сменится на „connected“.)
3. Если соединение установлено - осуществлять синхронизацию со всеми подключенными узлами до тех пор, пока отличия не будут более *replication_sync_lag* секунд.

Ситуация 3: обновление конфигурации

Здесь вызов `box.cfg{}` происходит не впервые, а повторно, поскольку изменились некоторые параметры репликации или что-то в наборе реплик.

1. Попытка установить соединение со всеми узлами из `box.cfg.replication` или с количеством узлов, указанным в параметре *replication_connect_quorum* в течение периода времени, указанного в *replication_connect_timeout*.
2. Попытка синхронизации со всеми подключенными узлами в течение периода времени, указанного в *replication_sync_timeout*.
3. Если предыдущие шаги не выполнены, статус изменяется на „orphan“ (одиночный). (Попытки синхронизации будут продолжаться в фоновом режиме, и когда/если они будут успешны, статус „orphan“ отключится.)
4. Если предыдущие шаги выполнены, статус изменяется на „running“ (мастер) или „follow“ (реплика).

Ситуация 4: повторная настройка

Здесь не происходит вызов `box.cfg{}`. В определенный момент в прошлом реплика успешно установила соединение и в настоящий момент ожидает обновления от мастера. Однако мастер не может передать обновления, что может произойти случайно, или же если реплика работает слишком медленно (большое значение *lag*), а WAL-файлы (.xlog) с обновлениями были удалены. Такая ситуация не является критической – реплика может сбросить ранее полученные данные, а затем запросить содержание последнего файла снимка (.snap) мастера. Поскольку фактически в таком случае повторно проводится процесс настройки, это называется «повторная настройка». Тем не менее, есть отличие от обычной настройки – *идентификатор реплики* останется прежним. Если он изменится, то мастер посчитает, что в кластер добавляется новая реплика, и сохранит идентификатор экземпляра реплики, которой уже не существует. Полностью автоматизированный процесс повторной настройки появился в версии Tarantool'а 1.10.2.

Запуск сервера с репликацией

Помимо процесса восстановления, описанного в разделе *Процесс восстановления*, сервер должен предпринять дополнительные шаги и меры предосторожности, если включена *репликация*.

И снова процедура запуска начинается с запроса `box.cfg{}`. Одним из параметров запроса `box.cfg` может быть *replication*, в котором указываются источники репликации. Реплику, которая запускается сейчас с помощью `box.cfg`, мы будем называть локальной, чтобы отличать ее от других реплик в наборе реплик, которые мы будем называть удаленными.

Если нет файла снимка .snap и не указано значение параметра `replication`: то локальная реплика предполагает, что является нереплицируемым обособленным экземпляром или же первой репликой в новом наборе реплик. Она сгенерирует новые UUID для себя и для набора реплик. UUID реплики хранится в спейсе `_cluster`; UUID набора реплик хранится в спейсе `_schema`. Поскольку снимок содержит все данные во всех спейсах, это означает, что снимок локальной реплики будет содержать UUID реплики и UUID набора реплик. Таким образом, когда локальная реплика будет позднее перезапускаться, она сможет восстановить эти UUID после прочтения файла снимка .snap.

Если нет файла снимка .snap, указано значение параметра `replication`, *а в спейсе* `_cluster` *отсутствуют UUID других реплик*: то локальная реплика предполагает, что не является обособленным экземпляром, но еще не входит в набор реплик. Сейчас она должна быть подключиться в набор реплик. Она отправит свой UUID реплики первой удаленной реплике, указанной в параметре `replication`, которая будет выступать в качестве мастера. Это называется «запрос на подключение». Когда удаленная реплика получает запрос на подключение, она отправляет в ответ:

(1) UUID набора реплик, в который входит удаленная реплика

(2) содержимое файла снимка .snap удаленной реплики. Когда локальная реплика получает эту информацию, она размещает UUID набора реплики в своем спейсе `_schema`, UUID удаленной реплики и информацию о подключении в своем спейсе `_cluster`, а затем создает снимок, который содержит все данные, отправленные удаленной репликой. Затем, если в WAL-файлах .xlog локальной реплики содержатся данные, они отправляются на удаленную реплику. Удаленная реплика получается данные и обновляет свою копию данных, а затем добавляет UUID локальной реплики в свой спейс `_cluster`.

Если нет файла снимка .snap, указано значение параметра `replication`, *а в спейсе* `_cluster` *есть UUID других реплик*: то локальная реплика предполагает, что не является обособленным экземпляром, и уже входит в набор реплик. Она отправит свой UUID реплики и UUID набора реплик всем удаленным репликам, указанным в параметре `replication`. Это называется «подтверждение связи при подключении». Когда удаленная реплика получает подтверждение связи при подключении:

(1) удаленная реплика сопоставляет свою версию UUID набора реплик с UUID, переданным в ходе подтверждения связи при подключении. Если они не совпадают, связь не устанавливается, и локальная реплика отобразит ошибку.

(2) удаленная реплика ищет запись о подключающемся экземпляре в своем спейсе `_cluster`. Если такой записи нет, связь не устанавливается. Если есть, связь подтверждается. Удаленная реплика выполняет чтение любой новой информации из своих файлов .snap и .xlog и отправляет новые запросы на локальную реплику.

Наконец, локальная реплика понимает, к какому набору реплик относится, удаленная реплика понимает, что локальная реплика входит в набор реплик, и у двух реплик одинаковое содержимое базы данных.

Если есть файл снимка и указан источник репликации: сначала локальная реплика проходит процесс восстановления, описанный в предыдущем разделе, используя свои собственные файлы .snap и .xlog. Затем она отправляет запрос подписки всем репликам в наборе реплик. Запрос подписки содержит векторные часы сервера. Векторные часы включают набор пар „идентификатор сервера, LSN“ для каждой реплики в системном спейсе `_cluster`. Каждая удаленная реплика, получив запрос подписки, выполняет чтение запросов из файла .xlog и отправляет их на локальную реплику, если LSN из запроса файла .xlog больше, чем LSN векторных часов из запроса подписки. После того, как все реплики из набора реплик отправили ответ на запрос подписки локальной реплики, запуск реплики завершен.

Следующие временные ограничения применимы к версиям Tarantool'а ниже 1.7.7:

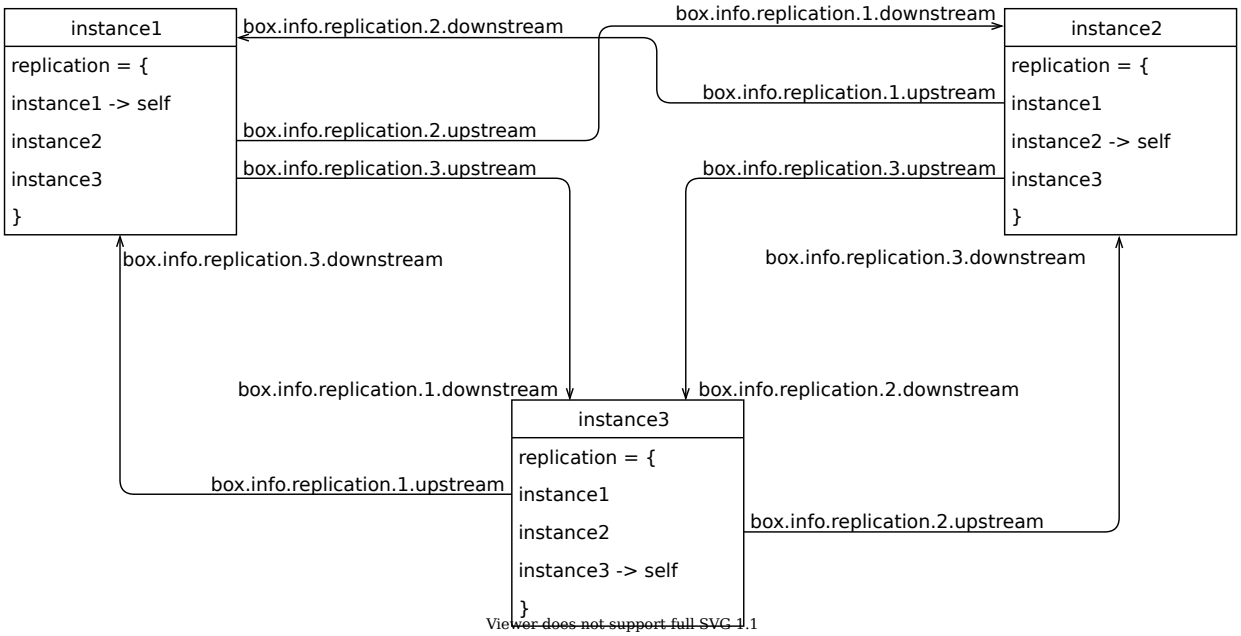
- URI в параметре `replication` должны быть указаны в одинаковом порядке на всех репликах. Это необязательно, но помогает соблюдать консистентность.
- Реплики в наборе реплик должны запускаться не одновременно. Это необязательно, но помогает избежать ситуации, когда все реплики ждут готовности друг друга.

Следующее ограничение всё еще применимо к текущей версии Tarantool'а:

- Максимальное количество записей в спейсе `_cluster` – *32*. Кортежи для устаревших реплик не переиспользуются автоматически, поэтому по достижении предела в 32 реплики, может понадобиться реорганизация спейса `_cluster` вручную.

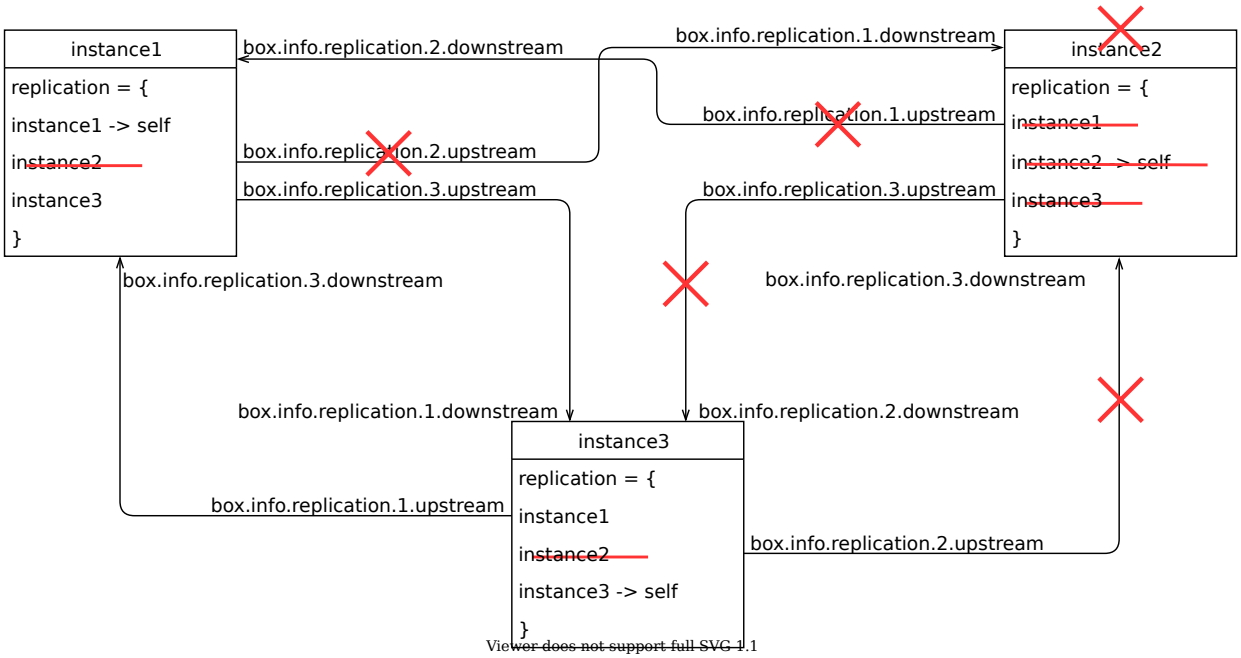
4.6.4 Удаление экземпляров

Предположим, что у нас настроен следующий набор реплик с 3 экземплярами (*instance1*, *instance2* и *instance3*), и мы хотим удалить *instance2*.



Чтобы правильно удалить экземпляр из набора реплик, выполните следующие шаги:

1. Отключите *instance2* от кластера.
2. Отключите кластер от *instance2*.
3. Исключите *instance2* из системного спейса `_cluster`.



Шаг 1: отключение экземпляра от кластера

На отключаемом *instance2* выполните `box.cfg{}` с пустым источником репликации:

```
tarantool> box.cfg{replication=''}
```

Теперь проверьте, что экземпляр был отсоединен. Взгляните на `box.info.replication` на *instance2* (заметьте, что строки для `replication.{1,3}.upstream` отсутствуют):

```
tarantool> box.info.replication
---
- 1:
   id: 1
   uuid: db89978f-7115-4537-8417-9982bb5a256f
   lsn: 9
-- upstream is absent
   downstream:
      status: follow
      idle: 0.93983899999876
      vclock: {1: 9}
2:
   id: 2
   uuid: 0a756d14-e437-4296-85b0-d27a0621613e
   lsn: 0
3:
   id: 3
   uuid: bb362584-c265-4e53-aeb6-450ae818bf59
   lsn: 0
-- upstream is absent
   downstream:
      status: follow
      idle: 0.26624799999991
      vclock: {1: 9}
...
```

Проверьте также *instance1* и *instance3* (заметьте, что статус `replication.2.downstream` поменялся на `stopped`):

```
-- instance1
tarantool> box.info.replication
---
- 1:
   id: 1
   uuid: db89978f-7115-4537-8417-9982bb5a256f
   lsn: 9
2:
   id: 2
   uuid: 0a756d14-e437-4296-85b0-d27a0621613e
   lsn: 0
   upstream:
      status: follow
      idle: 0.3533439999992
      peer: replicator@localhost:3302
      lag: 0.0001220703125
   downstream:
      status: stopped     -- status has changed:
      message: unexpected EOF when reading from socket, called on fd 13, aka [::1]:3301,
      peer of [::1]:53776
      system_message: Broken pipe
3:
   id: 3
   uuid: bb362584-c265-4e53-aeb6-450ae818bf59
   lsn: 0
   upstream:
```

(continues on next page)

(продолжение с предыдущей страницы)

```
      status: follow
      idle: 0.35327999999936
      peer: replicator@localhost:3303
      lag: 0.00018095970153809
   downstream:
      status: follow
      idle: 0.68685100000221
      vclock: {1: 9}
...
```

Шаг 2: отключение кластера от удаляемого экземпляра

На всех остальных экземплярах в кластере уберите *instance2* из списка `box.cfg{ replication }` и вызовите актуальный список `box.cfg{ replication = {instance1, instance3} }`:

```
tarantool> box.cfg{ replication = { 'instance1-uri', 'instance3-uri' } }
```

Взгляните на `box.info.replication` на *instance2*, чтобы убедиться, что `instance1` и `instance3` были отсоединены (заметьте, что статус `replication.2.downstream` поменялся на `stopped`):

```
tarantool> box.info.replication
---
- 1:
   id: 1
   uuid: db89978f-7115-4537-8417-9982bb5a256f
   lsn: 9
   downstream:
      status: stopped    -- status has changed
      message: unexpected EOF when reading from socket, called on fd 16, aka [::1]:3302,
      peer of [::1]:53832
      system_message: Broken pipe
2:
   id: 2
   uuid: 0a756d14-e437-4296-85b0-d27a0621613e
   lsn: 0
3:
   id: 3
   uuid: bb362584-c265-4e53-aeb6-450ae818bf59
   lsn: 0
   downstream:
      status: stopped    -- status has changed
      message: unexpected EOF when reading from socket, called on fd 18, aka [::1]:3302,
      peer of [::1]:53825
      system_message: Broken pipe
...
```

Проверьте также *instance1* и *instance3* (заметьте, что статус `replication.2.upstream` поменялся на `stopped`):

```
-- instance1
tarantool> box.info.replication
---
- 1:
   id: 1
   uuid: db89978f-7115-4537-8417-9982bb5a256f
   lsn: 9
```

(continues on next page)

(продолжение с предыдущей страницы)

```
2:
   id: 2
   uuid: 0a756d14-e437-4296-85b0-d27a0621613e
   lsn: 0
   downstream:
      status: stopped   -- status has changed
      message: unexpected EOF when reading from socket, called on fd 13, aka [::1]:3301,
      peer of [::1]:53776
      system_message: Broken pipe
3:
   id: 3
   uuid: bb362584-c265-4e53-aeb6-450ae818bf59
   lsn: 0
   upstream:
      status: follow
      idle: 0.50240100000156
      peer: replicator@localhost:3303
      lag: 0.00015711784362793
   downstream:
      status: follow
      idle: 0.14237199999843
      vclock: {1: 9}
...
```

Шаг 3: окончательное удаление

Если выбывший экземпляр снова вернется в кластер, то он получит информацию обо всех изменениях, которые произошли на остальных экземплярах за время его отсутствия.

Если экземпляр нужно вывести из эксплуатации навсегда, то нужно очистить `cluster` спейс. Для этого сначала узнайте `id` и `uuid` удаляемого экземпляра. На *instance2* вызовите `return box.info.id, box.info.uuid`:

```
tarantool> return box.info.id, box.info.uuid
---
- 2
- '0a756d14-e437-4296-85b0-d27a0621613e'
...
```

Запомните `id` and `uuid`.

Теперь выберите любой мастер из оставшегося кластера и выполните на нем следующие действия (предположим, это будет *instance1*):

1. Запросите все записи из спейса `_cluster`:

```
tarantool> box.space._cluster:select{}
---
- - [1, 'db89978f-7115-4537-8417-9982bb5a256f']
- [2, '0a756d14-e437-4296-85b0-d27a0621613e']
- [3, 'bb362584-c265-4e53-aeb6-450ae818bf59']
...
```

2.Проверьте корректность `id` и `uuid` *instance2* и удалите их из кластера:

```
tarantool> box.space._cluster:delete(2)
---
```

(continues on next page)

(продолжение с предыдущей страницы)

```
- [2, '0a756d14-e437-4296-85b0-d27a0621613e']
...
```

Финальная проверка

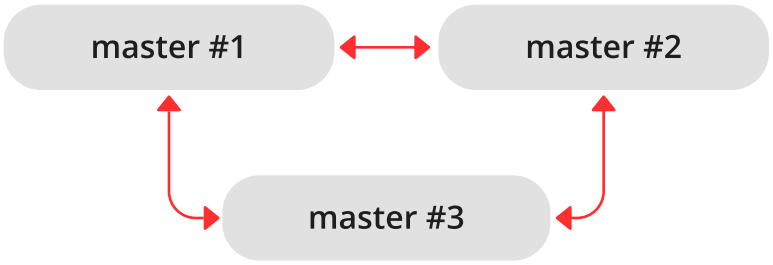
После всех модификаций выполните `box.info.replication`, чтобы проверить, что состояние кластера теперь корректное.

4.6.5 Мониторинг набора реплик

Чтобы узнать, какие экземпляры входят в набор реплик и получить статистику по всем этим экземплярам, передайте запрос *box.info.replication*:

```
tarantool> box.info.replication
---
  replication:
    1:
      id: 1
      uuid: b8a7db60-745f-41b3-bf68-5fcce7a1e019
      lsn: 88
    2:
      id: 2
      uuid: cd3c7da2-a638-4c5d-ae63-e7767c3a6896
      lsn: 31
      upstream:
        status: follow
        idle: 43.187747001648
        peer: replicator@192.168.0.102:3301
        lag: 0
      downstream:
        vclock: {1: 31}
    3:
      id: 3
      uuid: e38ef895-5804-43b9-81ac-9f2cd872b9c4
      lsn: 54
      upstream:
        status: follow
        idle: 43.187621831894
        peer: replicator@192.168.0.103:3301
        lag: 2
      downstream:
        vclock: {1: 54}
...
```

Данный отчет сгенерирован для набора реплик из трех экземпляров с конфигурацией мастер-мастер, у каждого из которых есть свой собственный ID экземпляра, UUID и номер записи в журнале.



Запрос был выполнен с мастера №1, и ответ включает в себя статистику по двум другим мастерам относительно мастера №1.

Основные индикаторы работоспособности репликации:

- *бездействие*, время (в секундах) с момента получения последнего события от мастера.

  A master sends heartbeat messages to a replica every second, and the master is programmed to disconnect if it does not see acknowledgments of the heartbeat messages within *replication_timeout* * 4 seconds.

  Таким образом, в работоспособном состоянии значение `idle` никогда не должно превышать значение `replication_timeout`: в противном случае, либо репликация сильно отстает, поскольку мастер опережает реплику, либо отсутствует сетевое подключение между экземплярами.

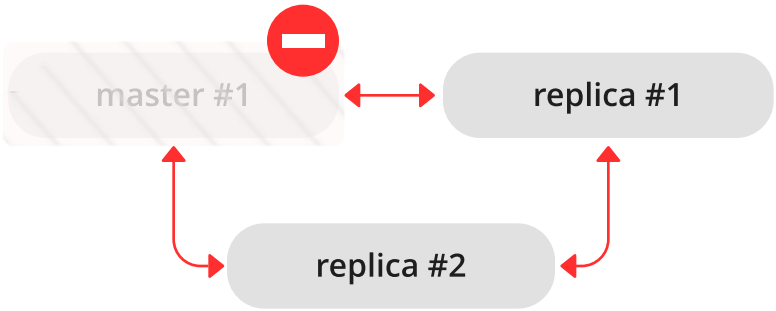
- *отставание*, разница во времени между локальным временем на экземпляре, зарегистрированным при получении события, и локальное время на другом мастере, зарегистрированное при записи события в *журнал упреждающей записи* на этом мастере.

  Поскольку при расчете `отставания` используются часы операционной системы с двух разных машин, не удивляйтесь, получив отрицательное число: смещение во времени может привести к постоянному запаздыванию времени на удаленном мастере относительно часов на локальном экземпляре.

  Для многомастерной конфигурации это максимально возможное отставание.

4.6.6 Восстановление после сбоя

«Сбой» – это ситуация, когда мастер становится недоступен вследствие проблем с оборудованием, сетевых неполадок или программной ошибки.



В конфигурации мастер-реплика, если мастер пропадает, на репликах выводятся сообщения об ошибке с указанием потери соединения:

```
$ # сообщения из журнала реплики
2017-06-14 16:23:10.993 [19153] main/105/applier/replicator@192.168.0. I> can't read row
2017-06-14 16:23:10.993 [19153] main/105/applier/replicator@192.168.0. coio.cc:349 !> SystemError
unexpected EOF when reading from socket, called on fd 17, aka 192.168.0.101:57815,
peer of 192.168.0.101:3301: Broken pipe
2017-06-14 16:23:10.993 [19153] main/105/applier/replicator@192.168.0. I> will retry every 1 second
2017-06-14 16:23:10.993 [19153] relay/[::ffff:192.168.0.101]:/101/main I> the replica has closed␣
↪its socket, exiting
2017-06-14 16:23:10.993 [19153] relay/[::ffff:192.168.0.101]:/101/main C> exiting the relay loop
```

... а статус мастера выводится как «отключенный» (disconnected):

```
# отчет от реплики № 1
tarantool> box.info.replication
---
- 1:
    id: 1
    uuid: 70e8e9dc-e38d-4046-99e5-d25419267229
    lsn: 542
    upstream:
      peer: replicator@192.168.0.101:3301
      lag: 0.00026607513427734
      status: disconnected
      idle: 182.36929893494
      message: connect, called on fd 13, aka 192.168.0.101:58244
  2:
    id: 2
    uuid: fb252ac7-5c34-4459-84d0-54d248b8c87e
    lsn: 0
  3:
    id: 3
    uuid: fd7681d8-255f-4237-b8bb-c4fb9d99024d
    lsn: 0
    downstream:
      vclock: {1: 542}
...
```

```
# отчет от реплики № 2
tarantool> box.info.replication
---
- 1:
    id: 1
    uuid: 70e8e9dc-e38d-4046-99e5-d25419267229
    lsn: 542
    upstream:
      peer: replicator@192.168.0.101:3301
      lag: 0.00027203559875488
      status: disconnected
      idle: 186.76988101006
      message: connect, called on fd 13, aka 192.168.0.101:58253
  2:
    id: 2
    uuid: fb252ac7-5c34-4459-84d0-54d248b8c87e
    lsn: 0
    upstream:
      status: follow
      idle: 186.76960110664
      peer: replicator@192.168.0.102:3301
      lag: 0.00020599365234375
  3:
    id: 3
    uuid: fd7681d8-255f-4237-b8bb-c4fb9d99024d
    lsn: 0
...
```

Чтобы объявить, что одна из реплик должна стать новым мастером:

1. Убедитесь, что старый мастер окончательно недоступен:
   - измените правила маршрутизации в сети, чтобы больше не отправлять пакеты на мастер, или
   - отключите мастер-экземпляр, если у вас есть доступ к машине, или
   - отключите питание контейнера или машины.
2. Выполните `box.cfg{read_only=false, listen=URI}` на реплике и `box.cfg{replication=URI}` на других репликах в наборе.

Примечание: Если на старом мастере есть обновления, не переданные до выхода старого мастера из строя, *примените их вручную* на новом мастере с помощью команд `tarantoolctl cat` и `tarantoolctl play`.

Реплика не может автоматически определить, что мастер не будет доступен в будущем, поскольку причины отказа и среды репликации могут существенно отличаться друг от друга. Поэтому обнаруживать сбой должен человек.

4.6.7 Перезагрузка реплики

Если один из файлов формата .xlog/.snap/.run на реплике поврежден или удален, можно «перезагрузить» реплику данными:

1. Остановите реплику и удалите все локальные файлы базы данных (с расширениями .xlog/.snap/.run/.inprogress).

2. Удалите запись о реплике из следующих мест:

   a. параметр `replication` на всех работающих экземплярах в наборе реплик.

   b. кортеж `box.space._cluster` на мастер-экземпляре.

   Для получения подробной информации см. Раздел *Удаление экземпляров*.

3. Перезапустите реплику с тем же файлом экземпляра для повторного подключения к мастеру. Реплика синхронизируется с мастером после получения всех кортежей.

Примечание: Следует отметить, что эта процедура сработает только в том случае, если на мастере есть WAL-файлы.

4.6.8 Решение конфликтов репликации

Устранение конфликтов репликации мастер-мастер

Tarantool гарантирует, что все обновления применяются однократно на каждой реплике. Однако, поскольку репликация носит асинхронный характер, порядок обновлений не гарантируется. Сейчас мы проанализируем данную проблему более подробно с примерами рассинхронизации репликации и предложим соответствующие решения.

Замена по одному и тому же первичному ключу

Кейс 1: у вас есть два экземпляра Тарантула. Например, вы пытаетесь сделать операцию замены одного и того же первичного ключа на обоих экземплярах одновременно. Случится конфликт из-за того, какой кортеж сохранить, а какой отбросить.

Триггер-функции Тарантула могут помочь в реализации правил разрешения конфликтов при определенных условиях. Например, если у вас есть метка времени, то можно указать, что сохранять нужно кортеж с большей меткой.

Во-первых, вам нужно повесить триггер *before_replace()* на спейс, в котором могут быть конфликты. В этом триггере вы можете сравнить старую и новую записи реплики и выбрать, какую из них использовать (или полностью пропустить обновление, или объединить две записи вместе).

Затем вам нужно установить триггер в нужное время, прежде чем спейс начнет получать обновления. Триггер `before_replace` нужно устанавливать в тот момент, когда спейс создается, поэтому еще нужен триггер, чтобы установить другой триггер на системном спейсе `_space`, чтобы поймать момент, когда ваш спейс создается, и установить триггер там. Для этого подходит триггер *on_replace()*.

Разница между `before_replace` и `on_replace` заключается в том, что `on_replace` вызывается после вставки строки в спейс, а `before_replace` вызывается перед ней.

Устанавливать триггер `_space:on_replace()` также нужно в определенный момент. Лучшее время для его использования – это когда только что создан `_space`, что является триггером на `box.ctl.on_schema_init()`.

Вам также нужно использовать `box.on_commit`, чтобы получить доступ к создаваемому спейсу. В результате код будет выглядеть следующим образом:

```
local my_space_name = 'my_space'
local my_trigger = function(old, new) ... end -- ваша функция, устраняющая конфликт
box.ctl.on_schema_init(function()
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    box.space._space:on_replace(function(old_space, new_space)
        if not old_space and new_space and new_space.name == my_space_name then
            box.on_commit(function()
                box.space[my_space_name]:before_replace(my_trigger)
            end
        end
    end)
end)
```

Предотвращение дублирующей вставки

Кейс 2: Предположим, что в наборе реплик с двумя мастерами мастер №1 пытается вставить кортеж с одинаковым уникальным ключом:

```
tarantool> box.space.tester:insert{1, 'data'}
```

Это вызовет сообщение об ошибке дубликата ключа (`Duplicate key exists in unique index 'primary' in space 'tester'`), и репликация остановится. Такое поведение системы обеспечивается использованием рекомендуемого значения `false` (по умолчанию) для конфигурационного параметра *replication_skip_conflict*.

```
$ # сообщения об ошибках от мастера №1
2017-06-26 21:17:03.233 [30444] main/104/applier/rep_user@100.96.166.1 I> can't read row
2017-06-26 21:17:03.233 [30444] main/104/applier/rep_user@100.96.166.1 memtx_hash.cc:226 E> ER_
↪TUPLE_FOUND:
Duplicate key exists in unique index 'primary' in space 'tester'
2017-06-26 21:17:03.233 [30444] relay/[::ffff:100.96.166.178]/101/main I> the replica has closed␣
↪its socket, exiting
2017-06-26 21:17:03.233 [30444] relay/[::ffff:100.96.166.178]/101/main C> exiting the relay loop

$ # сообщения об ошибках от мастера №2
2017-06-26 21:17:03.233 [30445] main/104/applier/rep_user@100.96.166.1 I> can't read row
2017-06-26 21:17:03.233 [30445] main/104/applier/rep_user@100.96.166.1 memtx_hash.cc:226 E> ER_
↪TUPLE_FOUND:
Duplicate key exists in unique index 'primary' in space 'tester'
2017-06-26 21:17:03.234 [30445] relay/[::ffff:100.96.166.178]/101/main I> the replica has closed␣
↪its socket, exiting
2017-06-26 21:17:03.234 [30445] relay/[::ffff:100.96.166.178]/101/main C> exiting the relay loop
```

Если мы проверим статус репликации с помощью `box.info`, то увидим, что репликация на мастере №1 остановлена (`1.upstream.status = stopped`). Кроме того, данные с этого мастера не реплицируются (группа `1.downstream` отсутствует в отчете), поскольку встречается та же ошибка:

```
# статусы репликации (отчет от мастера №3)
tarantool> box.info
---
- version: 1.7.4-52-g980d30092
  id: 3
  ro: false
  vclock: {1: 9, 2: 1000000, 3: 3}
  uptime: 557
  lsn: 3
  vinyl: []
  cluster:
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    uuid: 34d13b1a-f851-45bb-8f57-57489d3b3c8b
  pid: 30445
  status: running
  signature: 1000012
  replication:
    1:
      id: 1
      uuid: 7ab6dee7-dc0f-4477-af2b-0e63452573cf
      lsn: 9
      upstream:
        peer: replicator@192.168.0.101:3301
        lag: 0.00050592422485352
        status: stopped
        idle: 445.8626639843
        message: Duplicate key exists in unique index 'primary' in space 'tester'
    2:
      id: 2
      uuid: 9afbe2d9-db84-4d05-9a7b-e0cbbf861e28
      lsn: 1000000
      upstream:
        status: follow
        idle: 201.99915885925
        peer: replicator@192.168.0.102:3301
        lag: 0.0015020370483398
      downstream:
        vclock: {1: 8, 2: 1000000, 3: 3}
    3:
      id: 3
      uuid: e826a667-eed7-48d5-a290-64299b159571
      lsn: 3
  uuid: e826a667-eed7-48d5-a290-64299b159571
...
```

Когда позднее репликация возобновлена вручную:

```
# возобновление остановленной репликации (на всех мастерах)
tarantool> original_value = box.cfg.replication
tarantool> box.cfg{replication={}}
tarantool> box.cfg{replication=original_value}
```

. . . запись с ошибкой в журнале упреждающей записи пропущена.

Решение #1: рассинхронизация репликации

Предположим, что мы выполняем следующую операцию в кластере из двух экземпляров с конфигурацией мастер-мастер:

```
tarantool> box.space.tester:upsert({1}, {{'=', 2, box.info.uuid}})
```

Когда эта операция применяется на обоих экземплярах в наборе реплик:

```
# на мастере #1
tarantool> box.space.tester:upsert({1}, {{'=', 2, box.info.uuid}})
# на мастере #2
tarantool> box.space.tester:upsert({1}, {{'=', 2, box.info.uuid}})
```

. . . можно получить следующие результаты в зависимости от порядка выполнения:

- каждая строка мастера содержит UUID из мастера №1,
- каждая строка мастера содержит UUID из мастера №2,
- у мастера №1 UUID мастера №2, и наоборот.

Решение #2: коммутативные изменения

Случаи, описанные в предыдущих абзацах, представляют собой примеры **некоммутативных** операций, т.е. операций, результат которых зависит от порядка их выполнения. Для **коммутативных операций** порядок выполнения значения не имеет.

Рассмотрим, например, следующую команду:

```
tarantool> box.space.tester:upsert{{1, 0}, {{'+', 2, 1)}
```

Эта операция коммутативна: получаем одинаковый результат, независимо от порядка, в котором обновление применяется на других мастерах.

Решение #3: использование триггера

Логика и установка триггера будет такой же, как в кейсе 1. Но сама триггер-функция будет отличаться:

```
local my_space_name = 'test'
local my_trigger = function(old, new, sp, op)
    -- op:  'INSERT', 'DELETE', 'UPDATE', or 'REPLACE'
    if new == nil then
        print("No new during "..op, old)
        return -- удаление допустимо
    end
    if old == nil then
        print("Insert new, no old", new)
        return new  -- вставка без старого значения допустима
    end
    print(op.." duplicate", old, new)
    if op == 'INSERT' then
        if new[2] > old[2] then
            -- Создание нового кортежа сменит оператор на REPLACE
            return box.tuple.new(new)
        end
        return old
    end
    if new[2] > old[2] then
        return new
    else
        return old
    end
    return
end

box.ctl.on_schema_init(function()
    box.space._space:on_replace(function(old_space, new_space)
        if not old_space and new_space and new_space.name == my_space_name then
            box.on_commit(function()
                box.space[my_space_name]:before_replace(my_trigger)
            end)
        end
    end)
end)
```

4.7 Коннекторы

В этой главе описаны API для различных языков программирования.

4.7.1 Протокол

Бинарный протокол для передачи данных в Tarantool'е был разработан с учетом потребностей асинхронного ввода-вывода для облегчения интеграции с прокси-серверами. Каждый клиентский запрос начинается с бинарного заголовка переменной длины. В заголовке указывается идентификатор и тип запроса, идентификатор экземпляра, номер записи в журнале и т.д.

Также в заголовке обязательно указывается длина запроса, что облегчает обработку данных. Ответ на запрос посылается по мере готовности. В заголовке ответа указывается тот же идентификатор и тип запроса, что и в изначальном запросе. По идентификатору можно легко соотнести запрос с ответом, даже если ответ был получен не в порядке отсылки запросов.

Вдаваться в тонкости реализации Tarantool-протокола нужно только при разработке нового коннектора для Tarantool'а – см. *полное описание бинарного протокола в Tarantool'е* в виде аннотированных BNF-диаграмм (Backus-Naur Form). В остальных случаях достаточно взять уже существующий коннектор для нужного вам языка программирования. Такие коннекторы позволяют легко хранить структуры данных из разных языков в формате Tarantool'а.

4.7.2 Пример пакета данных

С помощью API Tarantool'а клиентские программы могут отправлять пакеты с запросами в адрес экземпляра и получать на них ответы. Вот пример для запроса `box.space[513]:insert{'A', 'BB'}`. Описания компонентов запроса (в виде BNF-диаграмм) вы найдете на странице о *бинарном протоколе в Tarantool'е*.

Компонент	Байт #0	Байт #1	Байт #2	Байт #3
код для вставки	02			
остаток заголовка	...	...	...	...
число из 2 цифр: ID спейса	cd	02	01	
код для кортежа	21			
число из 1 цифры: количество полей = 2	92			
строка из 1 символа: поле[1]	a1	41		
строка из 2 символов: поле[2]	a2	42	42	

Теперь получившийся пакет можно послать в адрес экземпляра Tarantool'а и затем расшифровать ответ (описания формата пакета ответов и вопросов вы найдете на той же странице о *бинарном протоколе в Tarantool'е*). Но более простым и верным способом будет вызвать процедуру, которая сформирует готовый пакет с заданными параметрами. Что-то вроде `response = tarantool_routine("insert", 513, "A", "B");`. Для этого и существуют API для драйверов для Perl, Python, PHP и т.д.

4.7.3 Настройка окружения для примеров работы с коннекторами

В этой главе приводятся примеры того, как можно установить соединение с Tarantool-сервером с помощью коннекторов для языков Perl, PHP, Python, node.js и C. Обратите внимание, что в примерах указаны фиксированные значения, поэтому для корректной работы всех примеров нужно соблюсти следующие условия:

- экземпляр (Tarantool) запущен на локальной машине (`localhost = 127.0.0.1`), а прослушивание для него настроено на порту 3301 (`box.cfg.listen = '3301'`),
- в базе есть спейс“examples“ с идентификатором 999 (`box.space.examples.id = 999`), и у него есть первичный индекс, построенный по ключу числового типа (`box.space[999].index[0].parts[1].type = "unsigned"`),
- для пользователя „guest“ настроены права на чтение и запись.

Можно легко соблюсти все условия, запустив экземпляр и выполнив следующий скрипт:

```
box.cfg{listen=3301}
box.schema.space.create('examples',{id=999})
box.space.examples:create_index('primary', {type = 'hash', parts = {1, 'unsigned'}})
box.schema.user.grant('guest','read,write','space','examples')
box.schema.user.grant('guest','read','space','_space')
```

4.7.4 Java

См. http://github.com/tarantool/tarantool-java/.

4.7.5 Go

См. https://github.com/mialinx/go-tarantool.

4.7.6 R

См. https://github.com/thekvs/tarantoolr.

4.7.7 Erlang

См. Erlang-драйвер для Tarantool'a.

4.7.8 Perl

Самый используемый драйвер для Perl – tarantool-perl. Он не входит в репозиторий Tarantool'a, его необходимо устанавливать отдельно. Проще всего установить его путем клонирования с GitHub.

Во избежание незначительных предупреждений, которые может выдать система после первой установки `tarantool-perl`, начните установку с некоторых других модулей, которые использует `tarantool-perl`, с CPAN, the Comprehensive Perl Archive Network (Всеобъемлющая сеть архивов Perl):

```
$ sudo cpan install AnyEvent
$ sudo cpan install Devel::GlobalDestruction
```

Затем для установки самого `tarantool-perl`, выполните:

```
$ git clone https://github.com/tarantool/tarantool-perl.git tarantool-perl
$ cd tarantool-perl
$ git submodule init
$ git submodule update --recursive
$ perl Makefile.PL
```

(continues on next page)

(продолжение с предыдущей страницы)

```
$ make
$ sudo make install
```

Далее приводится пример полноценной программы на языке Perl, которая осуществляет вставку кортежа `[99999,'BB']` в спейс `space[999]` с помощью API для языка Perl. Перед запуском проверьте, что у экземпляра задан порт для прослушивания на `localhost:3301`, и в базе создан спейс `examples`, как *описано выше*. Чтобы запустить программу, сохраните код в файл с именем `example.pl` и выполните команду `perl example.pl`. Программа установит соединение, используя определение спейса для этой цели, откроет сокет для соединения с экземпляром по `localhost:3301`, пошлет запрос *space_object:INSERT*, а затем – если всё хорошо – закончит работу без каких-либо сообщений. Если Tarantool не запущен на `localhost` на *прослушивание* по порту = 3301, то программа выдаст сообщение об ошибке «Connection refused».

```
#!/usr/bin/perl
#!/usr/bin/perl
use DR::Tarantool ':constant', 'tarantool';
use DR::Tarantool ':all';
use DR::Tarantool::MsgPack::SyncClient;

  host    => '127.0.0.1',                    # поиск Tarantool-сервера по адресу localhost
  port    => 3301,                           # на порту 3301
  user    => 'guest',                        # имя пользователя; здесь же можно добавить
↪'password=>...'

  spaces  => {
    999 => {                                 # определение спейса  space[999] ...
      name => 'examples',                    # имя спейса space[999] = 'examples'
      default_type => 'STR',                 # если тип поля в space[999] не задан, то = 'STR'
      fields => [ {                          # определение полей в спейсе space[999] ...
          name => 'field1', type => 'NUM' } ], # имя поля space[999].field[1]='field1', тип ='NUM'
      indexes => {                           # определение индексов спейса space[999] ...
        0 => {
          name => 'primary', fields => [ 'field1' ] } } } } );

$tnt->insert('examples' => [ 99999, 'BB' ]);
```

Из-за временных ограничений в языке Perl, вместо полей типа *„string“ и „unsigned“* в тестовой программе указаны поля типа „STR“ и „NUM“.

В этой программе мы привели пример использования лишь одного запроса. Для полноценной работы с Tarantool'ом обратитесь к документации из репозитория tarantool-perl.

4.7.9 PHP

tarantool-php – это официальный PHP-коннектор для Tarantool'а. Он не входит в репозиторий Tarantool'а, его необходимо устанавливать отдельно (инструкции по установке см. в файле коннектора `README`).

Далее приводится пример полноценной программы на языке PHP, которая осуществляет вставку кортежа `[99999,'BB']` в спейс `examples` с помощью API для языка PHP.

Перед запуском проверьте, что у экземпляра задан порт для *прослушивания* на `localhost:3301`, и в базе создан спейс `examples`, как *описано выше*.

Чтобы запустить программу, сохраните код в файл с именем `example.php` и выполните:

```
$ php -d extension=~/tarantool-php/modules/tarantool.so example.php
```

Программа откроет сокет для соединения с экземпляром по `localhost:3301`, отправит *INSERT-запрос*, а затем – если всё хорошо – выдаст сообщение «Insert succeeded».

Если такой кортеж уже существует, то программа выдаст сообщение об ошибке “Duplicate key exists in unique index „primary“ in space „examples““.

```
<?php
$tarantool = new Tarantool('localhost', 3301);

try {
    $tarantool->insert('examples', [99999, 'BB']);
    echo "Insert succeeded\n";
} catch (Exception $e) {
    echo $e->getMessage(), "\n";
}
```

В этой программе мы привели пример использования лишь одного запроса. Для полноценной работы с Tarantool'ом обратитесь к документации из проекта tarantool-php на GitHub.

Кроме того, сообщество разработчиков поддерживает проект на GitHub, который включает в себя вариант коннектора, написанный на чистом PHP, модуль сопоставления объектов, администратор очередей и другие пакеты.

4.7.10 Python

tarantool-python – это официальный Python-коннектор для Tarantool'а. Он не входит в репозиторий Tarantool'а, его необходимо устанавливать отдельно (см. ниже подробную информацию).

Далее приводится пример полноценной программы на языке Python, которая осуществляет вставку `[99999,'Value','Value']` в спейс `examples` с помощью высокоуровневого API для языка Python.

```
#!/usr/bin/python
from tarantool import Connection

c = Connection("127.0.0.1", 3301)
result = c.insert("examples",(99999,'Value', 'Value'))
print result
```

Чтобы запустить тестовую программу, сохраните ее исходный код в файл с именем `example.py` и установите коннектор `tarantool-python`. Для установки коннектора воспользуйтесь либо командой `pip install tarantool>0.4` (для установки в директорию `/usr`; вам потребуются права уровня **root**), либо командой `pip install tarantool>0.4 --user` (для установки в директорию `~`, т.е. в используемую по умолчанию директорию текущего пользователя).

Перед запуском данной программы проверьте, что у Tarantool-сервера задан порт `localhost:3301` для *прослушивания* и в базе создан спейс `examples`), как *описано выше*. Чтобы запустить тестовую программу, выполните команду `python example.py`. Программа установит соединение с Tarantool-сервером, пошлет запрос *INSERT* и не сгенерирует никакого исключения, если всё прошло хорошо. Если окажется, что такой кортеж уже существует, то программа сгенерирует исключение *tarantool.error.DatabaseError: (3, «Duplicate key exists in unique index „primary“ in space „examples“»)*.

В этой программе мы привели пример использования лишь одного запроса. Для полноценной работы с Tarantool'ом обратитесь к документации из проекта tarantool-python на GitHub. А на странице проекта queue-python на GitHub вы сможете найти примеры использования Python API для работы с очередями сообщений в Tarantool'е.

Кроме того, сообщество разработчиков поддерживает другие Python-коннекторы:

- asynctnt с поддержкой asyncio
- aiotarantool также с поддержкой asyncio
- gtarantool с поддержкой gevent **не обновляется**

4.7.11 Node.js

Самый используемый драйвер для node.js – Node Tarantool driver. Он не входит в репозиторий Tarantool'a, его необходимо устанавливать отдельно. Проще всего установить его вместе с npm. Например, на Ubuntu, когда npm уже установлен, установка драйвера будет выглядеть следующим образом:

```
$ npm install tarantool-driver --global
```

Далее приводится пример полноценной программы на языке node.js, которая осуществляет вставку кортежа `[99999,'BB']` в спейс `space[999]` с помощью API для языка node.js. Перед запуском проверьте, что у экземпляра задан порт для *прослушивания* на `localhost:3301`, и в базе создан спейс `examples`, как *описано выше*. Чтобы запустить программу, сохраните код в файл с именем `example.rs` и выполните команду `node example.rs`. Программа установит соединение, используя определение спейса для этой цели, откроет сокет для соединения с экземпляром по `localhost:3301`, отправит *INSERT-запрос*, а затем – если всё хорошо – выдаст сообщение «Insert succeeded». Если Tarantool не запущен на `localhost` на прослушивание по порту = 3301, то программа выдаст сообщение об ошибке "Connect failed". Если у *пользователя „guest“* нет прав на соединение, программа выдаст сообщение об ошибке «Auth failed». Если запрос вставки по какой-либо причине не сработает, например поскольку такой кортеж уже существует, то программа выдаст сообщение об ошибке «Insert failed».

```
var TarantoolConnection = require('tarantool-driver');
var conn = new TarantoolConnection({port: 3301});
var insertTuple = [99999, "BB"];
conn.connect().then(function() {
    conn.auth("guest", "").then(function() {
        conn.insert(999, insertTuple).then(function() {
            console.log("Insert succeeded");
            process.exit(0);
    }, function(e) { console.log("Insert failed");  process.exit(1); });
    }, function(e) { console.log("Auth failed");    process.exit(1); });
    }, function(e) { console.log("Connect failed"); process.exit(1); });
```

В этой программе мы привели пример использования лишь одного запроса. Для полноценной работы с Tarantool'ом обратитесь к документации из репозитория драйвера для node.js.

4.7.12 C#

Самый используемый драйвер для C# – progaudi.tarantool, который раньше назывался `tarantool-csharp`. Он не входит в репозиторий Tarantool'a, его необходимо устанавливать отдельно. Создатели драйвера рекомендуют кроссплатформенную установку с помощью Nuget.

Чтобы придерживаться метода оформления других инструкций в данной главе, дадим описание способа установки драйвера напрямую на 16.04.

1. Установите среду .NET Core от Microsoft. Следуйте инструкциям по установке .NET Core.

Примечание:

- Mono не сработает, как не сработает и .Net от xbuild. Только .NET Core поддерживается на Linux и Mac.
- Сначала прочитайте Условия лицензионного соглашения с Microsoft, поскольку оно не похоже на обычные соглашения для ПО с открытым кодом, и во время установки система выдаст сообщение о том, что ПО может собирать информацию («This software may collect information about you and your use of the software, and send that to Microsoft.»). Несмотря на это, можно определить переменные окружения, чтобы отказаться от участия в сборе телеметрических данных.

2. Создайте новый консольный проект.

```
$ cd ~
$ mkdir progaudi.tarantool.test
$ cd progaudi.tarantool.test
$ dotnet new console
```

3. Добавьте ссылку на `progaudi.tarantool`.

```
$ dotnet add package progaudi.tarantool
```

4. Измените код в `Program.cs`.

```
$ cat <<EOT > Program.cs
using System;
using System.Threading.Tasks;
using ProGaudi.Tarantool.Client;

public class HelloWorld
{
  static public void Main ()
  {
    Test().GetAwaiter().GetResult();
  }
  static async Task Test()
  {
    var box = await Box.Connect("127.0.0.1:3301");
    var schema = box.GetSchema();
    var space = await schema.GetSpace("examples");
    await space.Insert((99999, "BB"));
  }
}
EOT
```

5. Соберите и запустите приложение.

   Перед запуском проверьте, что у экземпляра задан порт для прослушивания на“localhost:3301“, и в базе создан спейс `examples`, как *описано выше*.

```
$ dotnet restore
$ dotnet run
```

   Программа:

   - установит соединение, используя определение спейса для этой цели,
   - откроет сокет для соединения с экземпляром по `localhost:3301`,
   - отправит INSERT-запрос, а затем – если всё хорошо – закончит работу без каких-либо сообщений.

Если Tarantool не запущен на `localhost` на прослушивание по порту 3301, или у пользователя „guest“ нет прав на соединение, или запрос вставки по какой-либо причине не сработает, то программа выдаст сообщение об ошибке и другую информацию (трассировку стека и т.д.).

В этой программе мы привели пример использования лишь одного запроса. Для полноценной работы с Tarantool'ом с помощью PHP API, пожалуйста, обратитесь к документации из проекта tarantool-php на GitHub.

4.7.13 C

В этом разделе даны два примера использования высокоуровневого API для Tarantool'а и языка C.

Пример 1

Далее приводится пример полноценной программы на языке C, которая осуществляет вставку кортежа `[99999,'B']` в спейс `examples` с помощью высокоуровневого API для языка C.

```
#include <stdio.h>
#include <stdlib.h>

#include <tarantool/tarantool.h>
#include <tarantool/tnt_net.h>
#include <tarantool/tnt_opt.h>

void main() {
   struct tnt_stream *tnt = tnt_net(NULL);           /* См. ниже = НАСТРОЙКА */
   tnt_set(tnt, TNT_OPT_URI, "localhost:3301");
   if (tnt_connect(tnt) < 0) {                       /* См. ниже = СОЕДИНЕНИЕ */
       printf("Connection refused\n");
       exit(-1);
   }
   struct tnt_stream *tuple = tnt_object(NULL);      /* См. ниже = СОЗДАНИЕ ЗАПРОСА */
   tnt_object_format(tuple, "[%d%s]", 99999, "B");
   tnt_insert(tnt, 999, tuple);                      /* См. ниже = ОТПРАВКА ЗАПРОСА */
   tnt_flush(tnt);
   struct tnt_reply reply;  tnt_reply_init(&reply); /* См. ниже = ПОЛУЧЕНИЕ ОТВЕТА */
   tnt->read_reply(tnt, &reply);
   if (reply.code != 0) {
       printf("Insert failed %lu.\n", reply.code);
   }
   tnt_close(tnt);                                   /* См. ниже = ЗАВЕРШЕНИЕ */
   tnt_stream_free(tuple);
   tnt_stream_free(tnt);
}
```

Скопируйте исходный код программы в файл с именем `example.c` и установите коннектор `tarantool-c`. Вот один из способов установки `tarantool-c` (под Ubuntu):

```
$ git clone git://github.com/tarantool/tarantool-c.git ~/tarantool-c
$ cd ~/tarantool-c
$ git submodule init
$ git submodule update
$ cmake .
$ make
$ make install
```

Чтобы скомпилировать и слинковать тестовую программу, выполните следующую команду:

```
$ # иногда это необходимо:
$ export LD_LIBRARY_PATH=/usr/local/lib
$ gcc -o example example.c -ltarantool
```

Before trying to run, check that a server instance is listening at `localhost:3301` and that the space `examples` exists, as *described earlier*. To run the program, say `./example`. The program will connect to the Tarantool instance, and will send the request. If Tarantool is not running on localhost with listen address = 3301, the program will print "Connection refused". If the insert fails, the program will print «Insert failed» and an error number (see all error codes in the source file /src/box/errcode.h).

Далее следуют примечания, на которые мы ссылались в комментариях к исходному коду тестовой программы.

НАСТРОЙКА: Настройка начинается с создания потока (`tnt_stream`).

```
struct tnt_stream *tnt = tnt_net(NULL);
tnt_set(tnt, TNT_OPT_URI, "localhost:3301");
```

В нашей программе поток назван `tnt`. Перед установкой соединения с потоком `tnt` нужно задать ряд опций. Самая важная из них – TNT_OPT_URI. Для этой опции указан *URI* `localhost:3301`, т.е. адрес, по которому должно быть настроено прослушивание на стороне экземпляра Tarantool'а.

Описание функции:

```
struct tnt_stream *tnt_net(struct tnt_stream *s)
int tnt_set(struct tnt_stream *s, int option, variant option-value)
```

СОЕДИНЕНИЕ: Теперь когда мы создали поток с именем `tnt` и связали его с конкретным URI, наша программа может устанавливать соединение с экземпляром.

```
if (tnt_connect(tnt) < 0)
   { printf("Connection refused\n"); exit(-1); }
```

Описание функции:

```
int tnt_connect(struct tnt_stream *s)
```

Попытка соединения может и не удаться по разным причинам, например если Tarantool-сервер не запущен или в URI-строке указан неверный *пароль*. В случае неудачи функция вернет -1.

СОЗДАНИЕ ЗАПРОСА: В большинстве запросов требуется передавать структурированные данные, например содержимое кортежа.

```
struct tnt_stream *tuple = tnt_object(NULL);
tnt_object_format(tuple, "[%d%s]", 99999, "B");
```

В данной программе мы используем запрос *INSERT*, а кортеж содержит целое число и строку. Это простой набор значений без каких-либо вложенных структур или массивов. И передаваемые значения мы можем указать самым простым образом – аналогично тому, как это сделано в стандартной C-функции `printf()`: `%d` для обозначения целого числа, `%s` для обозначения строки, затем числовое значение, затем указатель на строковое значение.

Описание функции:

```
ssize_t tnt_object_format(struct tnt_stream *s, const char *fmt, ...)
```

ОТПРАВКА ЗАПРОСА: Отправка запросов на изменение данных в базе делается аналогично тому, как это делается в Tarantool-библиотеке `box`.

```
tnt_insert(tnt, 999, tuple);
tnt_flush(tnt);
```

В данной программе мы делаем INSERT-запрос. В этом запросе мы передаем поток `tnt`, который ранее использовали для установки соединения, и поток `tuple`, который также ранее настроили с помощью функции `tnt_object_format()`.

Описание функции:

```
ssize_t tnt_insert(struct tnt_stream *s, uint32_t space, struct tnt_stream *tuple)
ssize_t tnt_replace(struct tnt_stream *s, uint32_t space, struct tnt_stream *tuple)
ssize_t tnt_select(struct tnt_stream *s, uint32_t space, uint32_t index,
                   uint32_t limit, uint32_t offset, uint8_t iterator,
                   struct tnt_stream *key)
ssize_t tnt_update(struct tnt_stream *s, uint32_t space, uint32_t index,
                   struct tnt_stream *key, struct tnt_stream *ops)
```

ПОЛУЧЕНИЕ ОТВЕТА: На большинство запросов клиент получает ответ, который содержит информацию о том, был ли данный запрос успешно выполнен, а также содержит набор кортежей.

```
struct tnt_reply reply;  tnt_reply_init(&reply);
tnt->read_reply(tnt, &reply);
if (reply.code != 0)
   { printf("Insert failed %lu.\n", reply.code); }
```

Данная программа проверяет, был ли запрос выполнен успешно, но никак не интерпретирует оставшуюся часть ответа.

Описание функции:

```
struct tnt_reply *tnt_reply_init(struct tnt_reply *r)
tnt->read_reply(struct tnt_stream *s, struct tnt_reply *r)
void tnt_reply_free(struct tnt_reply *r)
```

ЗАВЕРШЕНИЕ: По окончании сессии нам нужно закрыть соединение, созданное с помощью функции `tnt_connect()`, и удалить объекты, созданные на этапе настройки.

```
tnt_close(tnt);
tnt_stream_free(tuple);
tnt_stream_free(tnt);
```

Описание функции:

```
void tnt_close(struct tnt_stream *s)
void tnt_stream_free(struct tnt_stream *s)
```

Пример 2

Here is a complete C program that selects, using index key `[99999]`, from space `examples` via the high-level C API. To display the results, the program uses functions in the MsgPuck library which allow decoding of MessagePack arrays.

```
#include <stdio.h>
#include <stdlib.h>
#include <tarantool/tarantool.h>
#include <tarantool/tnt_net.h>
#include <tarantool/tnt_opt.h>

#define MP_SOURCE 1
#include <msgpuck.h>

void main() {
    struct tnt_stream *tnt = tnt_net(NULL);
    tnt_set(tnt, TNT_OPT_URI, "localhost:3301");
    if (tnt_connect(tnt) < 0) {
        printf("Connection refused\n");
        exit(1);
    }
    struct tnt_stream *tuple = tnt_object(NULL);
    tnt_object_format(tuple, "[%d]", 99999); /* кортеж tuple = ключ для поиска */
    tnt_select(tnt, 999, 0, (2^32) - 1, 0, 0, tuple);
    tnt_flush(tnt);
    struct tnt_reply reply; tnt_reply_init(&reply);
    tnt->read_reply(tnt, &reply);
    if (reply.code != 0) {
        printf("Select failed.\n");
        exit(1);
    }
    char field_type;
    field_type = mp_typeof(*reply.data);
    if (field_type != MP_ARRAY) {
        printf("no tuple array\n");
        exit(1);
    }
    long unsigned int row_count;
    uint32_t tuple_count = mp_decode_array(&reply.data);
    printf("tuple count=%u\n", tuple_count);
    unsigned int i, j;
    for (i = 0; i < tuple_count; ++i) {
        field_type = mp_typeof(*reply.data);
        if (field_type != MP_ARRAY) {
            printf("no field array\n");
            exit(1);
        }
        uint32_t field_count = mp_decode_array(&reply.data);
        printf("  field count=%u\n", field_count);
        for (j = 0; j < field_count; ++j) {
            field_type = mp_typeof(*reply.data);
            if (field_type == MP_UINT) {
                uint64_t num_value = mp_decode_uint(&reply.data);
                printf("    value=%lu.\n", num_value);
            } else if (field_type == MP_STR) {
                const char *str_value;
                uint32_t str_value_length;
                str_value = mp_decode_str(&reply.data, &str_value_length);
                printf("    value=%.*s.\n", str_value_length, str_value);
            } else {
                printf("wrong field type\n");
                exit(1);
```

(continues on next page)

(продолжение с предыдущей страницы)

```
            }
        }
    }
    tnt_close(tnt);
    tnt_stream_free(tuple);
    tnt_stream_free(tnt);
}
```

Аналогично первому примеру, сохраните исходный код программы в файле с именем `example2.c`.

Чтобы скомпилировать и слинковать тестовую программу, выполните следующую команду:

```
$ gcc -o example2 example2.c -ltarantool
```

Для запуска программы выполните команду `./example2`.

В этих двух программах мы привели пример использования лишь двух запросов. Для полноценной работы с Tarantool'ом с помощью C API, пожалуйста, обратитесь к документации из проекта tarantool-c на GitHub.

4.7.14 Интерпретация возвращаемых значений

При работе с любым Tarantool-коннектором функции, вызванные с помощью Tarantool'а, возвращают значения в формате MsgPack. Если функция была вызвана через API коннектора, то формат возвращаемых значений будет следующим: скалярные значения возвращаются в виде кортежей (сначала идет идентификатор типа из формата MsgPack, а затем идет значение); все прочие (не скалярные) значения возвращаются в виде групп кортежей (сначала идет идентификатор массива в формате MsgPack, а затем идут скалярные значения). Но если функция была вызвана в рамках бинарного протокола (с помощью команды `eval`), а не через API коннектора, то подобных изменений формата возвращаемых значений не происходит.

Далее приводится пример создания Lua-функции. Поскольку эту функцию будет вызывать внешний пользователь *„guest“ user*, то нужно настроить права на исполнение с помощью *grant*. Эта функция возвращает пустой массив, строку-скаляр, два логических значения и короткое целое число. Значение будут теми же, что описаны в разделе про MsgPack в таблице *Стандартные типы в MsgPack-кодировке*.

```
tarantool> box.cfg{listen=3301}
2016-03-03 18:45:52.802 [27381] main/101/interactive I> ready to accept requests
---
...
tarantool> function f() return {},'a',false,true,127; end
---
...
tarantool> box.schema.func.create('f')
---
...
tarantool> box.schema.user.grant('guest','execute','function','f')
---
...
```

Далее идет пример программы на C, из который мы вызываем эту Lua-функцию. Хотя в примере использован код на C, результат будет одинаковым, на каком бы языке ни была написана вызываемая программа: Perl, PHP, Python, Go или Java.

```
#include <stdio.h>
#include <stdlib.h>
#include <tarantool/tarantool.h>
#include <tarantool/tnt_net.h>
#include <tarantool/tnt_opt.h>
void main() {
  struct tnt_stream *tnt = tnt_net(NULL);              /* НАСТРОЙКА */
  tnt_set(tnt, TNT_OPT_URI, "localhost:3301");
   if (tnt_connect(tnt) < 0) {                         /* СОЕДИНЕНИЕ */
       printf("Connection refused\n");
       exit(-1);
   }
   struct tnt_stream *arg; arg = tnt_object(NULL);     /* СОЗДАНИЕ ЗАПРОСА */
   tnt_object_add_array(arg, 0);
   struct tnt_request *req1 = tnt_request_call(NULL);  /* ВЫЗОВ функции f() */
   tnt_request_set_funcz(req1, "f");
   uint64_t sync1 = tnt_request_compile(tnt, req1);
   tnt_flush(tnt);                                     /* ОТПРАВКА ЗАПРОСА */
   struct tnt_reply reply;  tnt_reply_init(&reply);    /* ПОЛУЧЕНИЕ ОТВЕТА */
   tnt->read_reply(tnt, &reply);
   if (reply.code != 0) {
     printf("Call failed %lu.\n", reply.code);
     exit(-1);
   }
   const unsigned char *p= (unsigned char*)reply.data; /* ВЫВОД ОТВЕТА */
   while (p < (unsigned char *) reply.data_end)
   {
     printf("%x ", *p);
     ++p;
   }
   printf("\n");
   tnt_close(tnt);                                     /* ЗАВЕРШЕНИЕ */
   tnt_stream_free(arg);
   tnt_stream_free(tnt);
}
```

По завершении программа выведет на экран следующие значения:

```
dd 0 0 0 5 90 91 a1 61 91 c2 91 c3 91 7f
```

Первые пять байт – `dd 0 0 0 5` – это фрагмент данных в формате MsgPack, означающий «32-битный заголовок массива со значением 5» (см. спецификацию на формат MsgPack). Остальные значения описаны в таблице *Стандартные типы в MsgPack-кодировке*.

4.8 Вопросы и ответы

В В чем особенности Tarantool'а?

О Tarantool – представитель нового поколения семейства серверов для in-memory базы данных, разработанный для веб-приложений. Он создан в компании Mail.Ru на основе практического опыта, полученного методом проб и ошибок с начала разработки в 2008 году.

В Почему Lua?

О Lua – это легкий, быстрый и расширяемый язык, позволяющий использовать различ-

ные парадигмы программирования. Lua также легко встраивается в различные приложения. Сопрограммы (coroutines) в Lua близко соотносятся с файберами (fibers) в Tarantool'е, а вся Lua-архитектура гладко ложится на его внутреннюю реализацию. Lua – это первый язык, на котором можно писать хранимые процедуры для Tarantool'а. В будущем список поддерживаемых языков планируется расширить.

В В чем ключевое преимущество Tarantool'а?

О

Tarantool обеспечивает богатый набор функций базы данных (HASH-индексы, TREE-индексы, RTREE-индексы, BITSET-индексы, вторичные индексы, составные индексы, транзакции, триггеры. асинхронная репликация) в гибкой среде Lua-интерпретатора.

Благодаря этим характеристикам, он представляет собой быстрый и надежный in-memory сервер с легким доступом к базе данных, который обрабатывает нетривиальную проблемно-ориентированную логику. Преимущество по сравнению с традиционными SQL-серверами – в производительности: архитектура без блокировок с малой перегрузкой означает, что Tarantool может обслуживать на порядок больше запросов в секунду на аналогичном оборудовании. Преимущество NoSQL-аналогов – в гибкости: Lua допускает гибкую обработку данных, хранимых в компактном денормализированном формате.

В Кто разрабатывает Tarantool?

О Во-первых, этим занимается команда разработки в Mail.Ru – см. историю коммитов на github.com/tarantool. Вся разработка ведется открытым образом. Кроме того, активную роль играют члены сообщества разработчиков Tarantool'а. Их силами было создано большинство коннекторов и ведутся доработки под разные дистрибутивы.

В Возникают ли проблемы из-за того, что Tarantool является in-memory решением?

О Основной движок баз данных в Tarantool'е работает с оперативной памятью, но при этом он гарантирует сохранность данных благодаря механизму WAL (write ahead log), т.е. журналу упреждающей записи. Также в Tarantool'е используются технологии сжатия и распределения данных, которые позволяют использовать все виды памяти наиболее эффективно. Если Tarantool сталкивается с нехваткой оперативной памяти, то он приостанавливает прием запросов на изменение данных до тех пор, пока не появится свободная память, но при этом с успехом продолжает обработку запросов на чтение и удаление данных. А для больших баз, где объем данных значительно превосходит имеющийся объем оперативной памяти, у Tarantool'а есть второй движок, чьи возможности ограничены лишь размером жесткого диска.

В Можно ли хранить (большие) объекты BLOB в Tarantool'е?

О Начиная с Tarantool 1.7, нет «жесткого» ограничения на максимальный размер кортежа. Однако Tarantool предназначен для работы с множеством фрагментов на высокой скорости. Например, при изменении существующего кортежа Tarantool создает новую версию кортежа в памяти. Таким образом, оптимальный размер кортежа – несколько килобайтов.

В Я удаляю данные из vinyl'а, но использование диска не изменяется. В чем дело?

О Данные, записываемые в vinyl, сохраняются в исполняемых файлах, обновление которых происходит только путем присоединения новых записей. Такие файлы нельзя изменить, а для удаления маркер удаления (удаленная запись) записывается в новый исполняемый файл. Для уплотнения данных новый и старый исполняемые файлы объединяются, и создается новый исполняемый файл. Независимо от этого, менеджер контрольных

точек следит за всеми исполняемыми файлами в контрольной точке и удаляет устаревшие файлы, как только в них отпадает необходимость.

ГЛАВА 5

Справочники

5.1 SQL reference

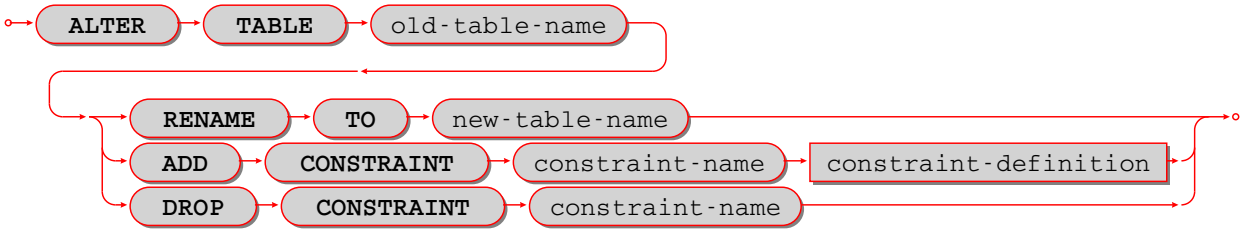
This reference covers all the SQL statements and clauses supported by Tarantool.

5.1.1 SQL statements and clauses

ALTER TABLE

Syntax:

- ALTER TABLE *table-name* RENAME TO *new-table-name*;
- ALTER TABLE *table-name* ADD CONSTRAINT *constraint-name* *constraint-definition*;
- ALTER TABLE *table-name* DROP CONSTRAINT *constraint-name*;



ALTER is used to change a table's name or to add new constraints or to drop old constraints.

Examples:

```
-- renaming a table:
ALTER TABLE t1 RENAME TO t2;
```

For ALTER ... RENAME, the *old-table* must exist, the *new-table* must not exist.

```
-- adding a foreign-key constraint definition:
ALTER TABLE t1 ADD CONSTRAINT c FOREIGN KEY (s1) REFERENCES t1;
```

For `ALTER ... ADD CONSTRAINT`, the table must exist, table must be empty, the constraint name must not already exist for the table.

It is not possible to say `CREATE TABLE table_a ... REFERENCES table_b ...` if table `b` does not exist yet. This is a situation where `ALTER TABLE` is handy – users can `CREATE TABLE table_a` without the foreign key, then `CREATE TABLE table_b`, then `ALTER TABLE table_a ... REFERENCES table_b ...`.

```
-- adding a primary-key constraint definition:
-- This is unusual because primary keys are created automatically
-- and it is illegal to have two primary keys for the same table.
-- However, it is possible to drop a primary-key index, and this
-- is a way to restore the primary key if that happens.
ALTER TABLE t1 ADD CONSTRAINT primary_key PRIMARY KEY (s1);

-- adding a unique-constraint definition:
-- Alternatively, you can say CREATE UNIQUE INDEX unique_key ON t1 (s1);
ALTER TABLE t1 ADD CONSTRAINT unique_key UNIQUE (s1);

-- Adding a check-constraint definition:
ALTER TABLE t1 ADD CONSTRAINT check_ CHECK (s1 > 0);
```

For `ALTER ... DROP CONSTRAINT`, it is only legal to drop a named constraint, and Tarantool only looks for names of foreign-key constraints. (Tarantool generates the constraint names automatically if the user does not provide them.)

To remove a unique constraint, use DROP INDEX, which will drop the constraint as well.

```
-- dropping a constraint:
ALTER TABLE t1 DROP CONSTRAINT c;
```

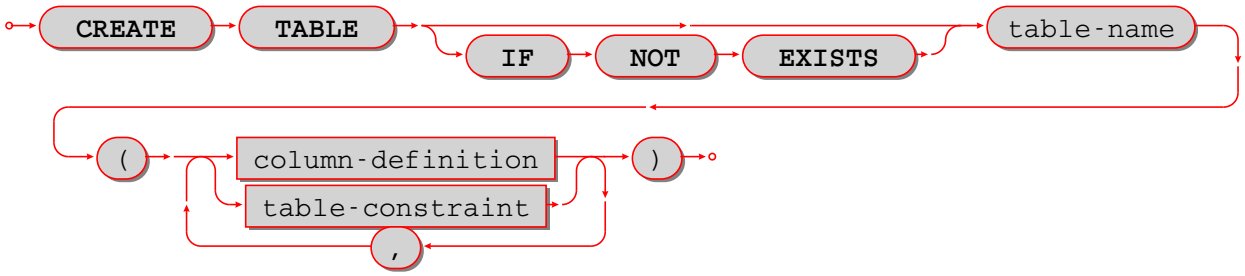
Limitations:

- It is not possible to add or drop a column.
- It is not possible to modify NOT NULL constraints or column properties DEFAULT and data type. However, it is possible to modify them with Tarantool/NOSQL, for example by calling *space_object:format()* with a different `is_nullable` value.

CREATE TABLE

Syntax:

`CREATE TABLE [IF NOT EXISTS]` *table-name* `((column-definition or table-constraint list);`



Create a new base table, usually called a «table».

Примечание: A table is a *base table* if it is created with CREATE TABLE and contains data in persistent storage.

A table is a *viewed table*, or just «view», if it is created with CREATE VIEW and gets its data from other views or from base tables.

The *table-name* must be an identifier which is valid according to the rules for identifiers, and must not be the name of an already existing base table or view.

The *column-definition* or *table-constraint* list is a comma-separated list of column definitions or table constraints.

A *table-element-list* must be a comma-separated list of table elements; each table element may be either a column definition or a table constraint definition.

Rules:

- A primary key is necessary; it can be specified with a table constraint `PRIMARY KEY`.
- There must be at least one column.
- When `IF NOT EXISTS` is specified, and there is already a table with the same name, the statement is ignored.

Actions:

1. Tarantool evaluates each column definition and *table-constraint*, and returns an error if any of the rules is violated.
2. Tarantool makes a new definition in the schema.
3. Tarantool makes new indexes for `PRIMARY KEY` or `UNIQUE` constraints. A unique index name is created automatically.
4. Tarantool effectively executes a `COMMIT` statement.

Examples:

```
-- the simplest form, with one column and one constraint:
CREATE TABLE t1 (s1 INTEGER, PRIMARY KEY (s1));

-- you can see the effect of the statement by querying
-- Tarantool system spaces:
SELECT * FROM "_space" WHERE "name" = 'T1';
SELECT * FROM "_index" JOIN "_space" ON "_index"."id" = "_space"."id"
         WHERE "_space"."name" = 'T1';

-- variation of the simplest form, with delimited identifiers
-- and an inline comment:
CREATE TABLE "T1" ("S1" INT /* synonym of INTEGER */, PRIMARY KEY ("S1"));

-- two columns, one named constraint
CREATE TABLE t1 (s1 INTEGER, s2 STRING, CONSTRAINT c1 PRIMARY KEY (s1, s2));
```

Limitations:

- The maximum number of columns is 2000.
- The maximum length of a row depends on the *memtx_max_tuple_size* or *vinyl_max_tuple_size* configuration option.

Column definition

Syntax:

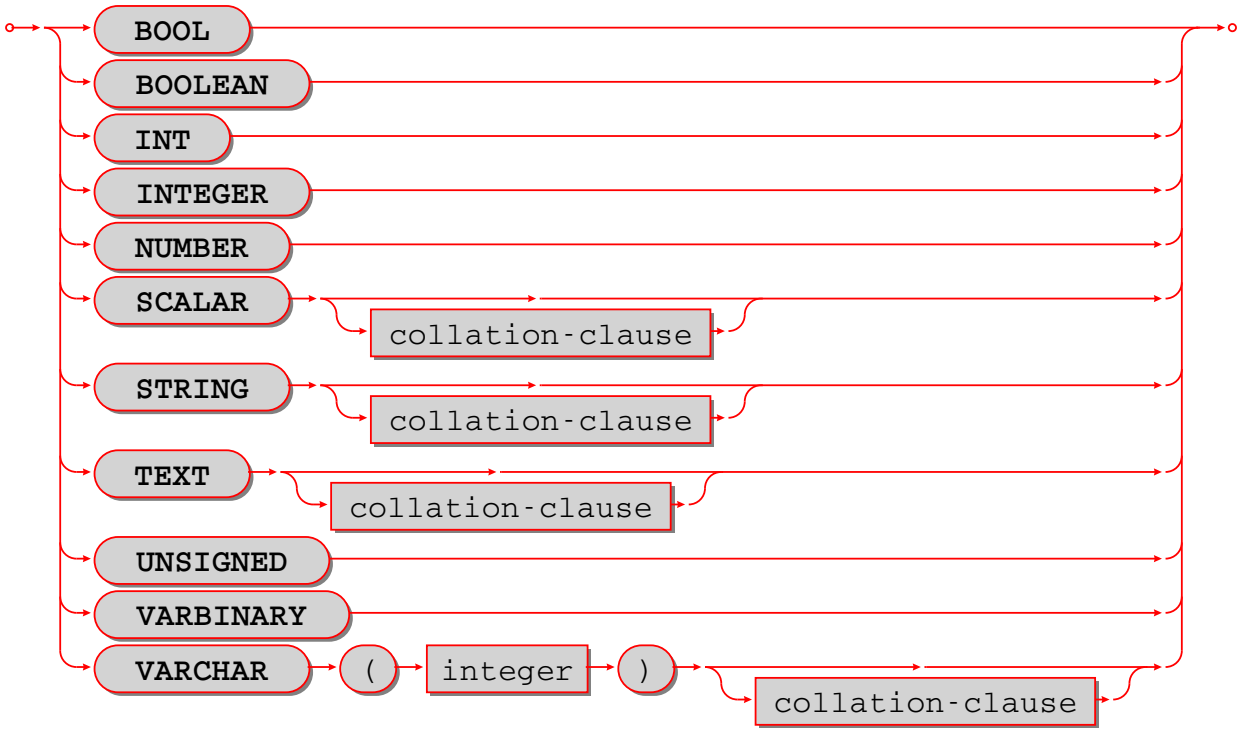
`column-name data-type [, column-constraint]`

Define a column, which is a table-element used in a CREATE TABLE statement.

The `column-name` must be an identifier which is valid according to the rules for identifiers.

Each `column-name` must be unique within a table.

Column definition – data type



Every operand has a data type.

For literals, the data type is usually determined by the format.

For identifiers, the data type is usually determined by the definition.

The usual determination may change because of context or because of explicit casting.

For some SQL data type names there are *aliases*. An alias may be used for data definition. For example VARCHAR(5) and TEXT are aliases of STRING and may appear in *CREATE TABLE table_name (column_name VARCHAR(5) PRIMARY KEY);* but Tarantool, if asked, will report that the data type of *column_name* is STRING.

For every SQL data type there is a corresponding NoSQL type, for example an SQL STRING is stored in a NoSQL space as type = „string“.

To avoid confusion in this manual, all references to SQL data type names are in upper case and all similar words which refer to NoSQL types or to other kinds of object are in lower case, for example:

- STRING is a data type name, but string is a general term;

- NUMBER is a data type name, but number is a general term.

Although it is common to say that a VARBINARY value is a «binary string», this manual will not use that term and will instead say «byte sequence».

Here are all the SQL data types, their corresponding NoSQL types, their aliases, and minimum / maximum literal examples.

Data types

SQL type	NoSQL type	Aliases	Minimum	Maximum
BOOLEAN	boolean (логический)	BOOL	FALSE	TRUE
INTEGER	целое число	INT	-9223372036854775808	18446744073709551615
UNSIGNED	unsigned	(none)	0	18446744073709551615
NUMBER	число	(none)	-1.79769e308	1.79769e308
STRING	строка	TEXT, VARCHAR(n)	„“	„many-characters“
VARBINARY	varbinary	(none)	X““	„X'many-hex-digits“
SCALAR	scalar	(none)	FALSE	X'many-hex-digits“

BOOLEAN values are FALSE, TRUE, and UNKNOWN (which is the same as NULL). FALSE is less than TRUE.

INTEGER values are numbers that do not contain decimal points and are not expressed with exponential notation. The range of possible values is between -2^63 and +2^64, or NULL.

UNSIGNED values are numbers that do not contain decimal points and are not expressed with exponential notation. The range of possible values is between 0 and +2^64, or NULL.

NUMBER values are numbers that do contain decimal points (for example 0.5) or are expressed with exponential notation (for example 5E-1). The range of possible values is the same as for the IEEE 754 floating-point standard, or NULL. Numbers outside the range of NUMBER literals may be displayed as -inf or inf.

STRING values are any sequence of zero or more characters encoded with UTF-8, or NULL. The possible character values are the same as for the Unicode standard. Byte sequences which are not valid UTF-8 characters are allowed but not recommended. STRING literal values are enclosed within single quotes, for example „literal“. If the VARCHAR alias is used for column definition, it must include a maximum length, for example column_1 VARCHAR(40). However, the maximum length is ignored. The data-type may be followed by `[COLLATE collation-name]`. .. // see section COLLATE clause.

VARBINARY values are any sequence of zero or more octets (bytes), or NULL. VARBINARY literal values are expressed as X followed by pairs of hexadecimal digits enclosed within single quotes, for example X'0044“. VARBINARYs NoSQL equivalent is „varbinary“ but not character string – the MessagePack storage is MP_BIN (MsgPack binary).

SCALAR can be used for column definitions but the individual column values have one of the preceding types – BOOLEAN, INTEGER, UNSIGNED, NUMBER, STRING, or VARBINARY. See more about SCALAR in the next section. The data-type may be followed by `[COLLATE collation-name]`. .. // see section COLLATE clause.

Any value of any data type may be NULL. Ordinarily NULL will be cast to the data type of any operand it is being compared to or to the data type of the column it is in. If the data type of NULL cannot be determined from context, it is BOOLEAN.

Column definition – the rules for the SCALAR data type

SCALAR is a «complex» data type, unlike all the other data types which are «primitive». Two column values in a SCALAR column can have two different primitive data types.

1. Any item defined as SCALAR has an underlying primitive type. For example, here:

```
CREATE TABLE t (s1 SCALAR PRIMARY KEY);
INSERT INTO t VALUES (55),('41');
```

   the underlying primitive type of the item in the first row is INTEGER because literal 55 has data type INTEGER, and the underlying primitive type in the second row is STRING (the data type of a literal is always clear from its format).

   An item's primitive type is far more important than its defined type. Incidentally Tarantool might find the primitive type by looking at the way MsgPack stores it, but that is an implementation detail.

2. A SCALAR definition may not include a maximum length, as there is no suggested restriction.
3. A SCALAR definition may include a COLLATE clause, which affects any items whose primitive data type is STRING. The default collation is «binary».
4. Some assignments are illegal when data types differ, but legal when the target is a SCALAR item. For example `UPDATE ... SET column1 = 'a'` is illegal if `column1` is defined as INTEGER, but is legal if `column1` is defined as SCALAR – values which happen to be INTEGER will be changed so their data type is STRING.
5. There is no literal syntax which implies data type SCALAR.
6. TYPEOF(x) is never SCALAR, it is always the underlying data type. This is true even if `x` is null (in that case the data type is BOOLEAN). In fact there is no function that is guaranteed to return the defined data type. For example, `TYPEOF(CAST(1 AS SCALAR));` returns INTEGER, not SCALAR.
7. For any operation that requires implicit casting from an item defined as SCALAR, the syntax is legal but the operation may fail at runtime. At runtime, Tarantool detects the underlying primitive data type and applies the rules for that. For example, if a definition is:

```
CREATE TABLE t (s1 SCALAR PRIMARY KEY, s2 INTEGER);
```

   and within any row `s1 = 'a'`, that is, its underlying primitive type is STRING to indicate character strings, then `UPDATE t SET s2 = s1;` is illegal. Tarantool usually does not know that in advance.

8. For any dyadic operation that requires implicit casting for comparison, the syntax is legal and the operation will not fail at runtime. Take this situation: comparison with a primitive type VARBINARY and a primitive type STRING.

```
CREATE TABLE t (s1 SCALAR PRIMARY KEY);
INSERT INTO t VALUES (X'41');
SELECT * FROM t WHERE s1 > 'a';
```

   The comparison is valid, because Tarantool knows the ordering of X'41" and „a" in Tarantool/NoSQL „scalar". This would be true even if `s1` was not defined as SCALAR.

9. The result data type of min/max operation on a column defined as SCALAR is the data type of the minimum/maximum operand, unless the result value is NULL. For example:

```
CREATE TABLE t (s1 INT, s2 SCALAR PRIMARY KEY);
INSERT INTO t VALUES (1,X'44'),(2,11),(3,1E4),(4,'a');
SELECT MIN(s2), HEX(MAX(s2)) FROM t;
```

The result is: `- - [11, '44',]`

That is only possible with Tarantool/NoSQL scalar rules, but `SELECT SUM(s2)` would not be legal because addition would in this case require implicit casting from VARBINARY to integer, which is not sensible.

10. The result data type of a primitive combination is never SCALAR because we in effect use TYPEOF(item) not the defined data type. (Here we use the word «combination» in the way that the standard document uses it for section «Result of data type combinations».) Therefore for `MAX(1E308, 'a', 0, X'00')` the result is X'00".

Column definition – relation to NoSQL

All the SQL data types correspond to *Tarantool/NoSQL types with the same name*. For example an SQL STRING is stored in a NoSQL space as type = „string".

Therefore specifying an SQL data type X determines that the storage will be in a space with a format column saying that the NoSQL type is „x".

The rules for that NoSQL type are applicable to the SQL data type.

If two items have SQL data types that have the same underlying type, then they are compatible for all assignment or comparison purposes.

If two items have SQL data types that have different underlying types, then the rules for explicit casts, or implicit (assignment) casts, or implicit (comparison) casts, apply.

There is one floating-point value which is not handled by SQL: -nan is seen as NULL.

There are also some Tarantool/NoSQL data types which have no corresponding SQL data types. For example, `SELECT "flags" FROM "_space";` will return a column whose data type is „map". Such columns can only be manipulated in SQL by invoking Lua functions.

Column definition – column-constraint or default clause

The column-constraint or default clause may be as follows:

Data types

Type	Comment
NOT NULL	means «it is illegal to assign a NULL to this column»
PRIMARY KEY	explained in the later section «Constraint definition»
UNIQUE	explained in the later section «Constraint definition»
CHECK (expression)	explained in the later section «Constraint definition»
DEFAULT expression	means «if INSERT does not assign to this column then assign expression result to this column» – if there is no DEFAULT clause then DEFAULT NULL is assumed.

If column-constraint is PRIMARY KEY, this is a shorthand for a separate table-constraint definition: «PRIMARY KEY (column-name)».

If column-constraint is UNIQUE, this is a shorthand for a separate table-constraint definition: «UNIQUE (column-name)».

Columns defined with PRIMARY KEY are automatically NOT NULL.

To enforce some restrictions that Tarantool does not enforce automatically, add CHECK clauses, like these:

```
CREATE TABLE t ("smallint" INTEGER PRIMARY KEY, CHECK ("smallint" <= 32767 AND "smallint" >= -
↪32768));
CREATE TABLE t ("shorttext" CHAR(10) PRIMARY KEY, CHECK (length("shorttext") <= 10));
```

but this may cause inserts or updates to be slow.

Column definition – examples

These are shown within CREATE TABLE statements. Data types may also appear in CAST functions.

```
-- the simple form with column-name and data-type
CREATE TABLE t (column1 INTEGER ...);
-- with column-name and data-type and column-constraint
CREATE TABLE t (column1 STRING PRIMARY KEY ...);
-- with column-name and data-type and collate-clause and two column-constraints
CREATE TABLE t (column1 SCALAR COLLATE "unicode" ...);
```

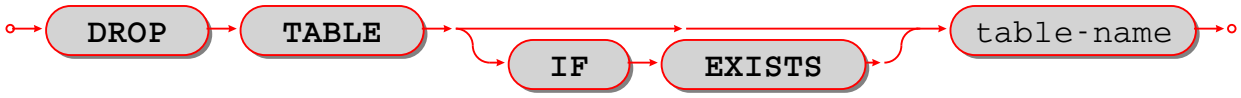
```
-- with all possible data types and aliases
CREATE TABLE t
(column1 BOOLEAN, column2 BOOL,
 column3 INT PRIMARY KEY, column4 INTEGER,
 column4 NUMBER,
 column7 STRING, column8 STRING COLLATE "unicode",
 column9 TEXT, columna TEXT COLLATE "unicode_sv_s1",
 columnb VARCHAR(0), columnc VARCHAR(100000) COLLATE "binary",
 columnd VARBINARY,
 columne SCALAR, columnf SCALAR COLLATE "unicode_uk_s2");
```

```
-- with all possible column constraints and a default clause
CREATE TABLE t
(column1 INT PRIMARY KEY,
 column2 INT UNIQUE,
 column3 INT CHECK (column3 > column2),
 column4 INT REFERENCES t,
 column6 INT DEFAULT NULL);
```

DROP TABLE

Syntax:

DROP TABLE [IF EXISTS] *table-name* ;



Drop a table.

The *table-name* must identify a table that was created earlier with the *CREATE TABLE statement*.

Rules:

- If there is a view that references the table, the drop will fail. Please drop the referencing view with DROP VIEW first.

- If there is a foreign key that references the table, the drop will fail. Please drop the referencing constraint with *ALTER TABLE ... DROP* first.

Actions:

1. Tarantool returns an error if the table does not exist.
2. The table and all its data are dropped.
3. All indexes for the table are dropped.
4. All triggers for the table are dropped.
5. Tarantool effectively executes a COMMIT statement.

Examples:

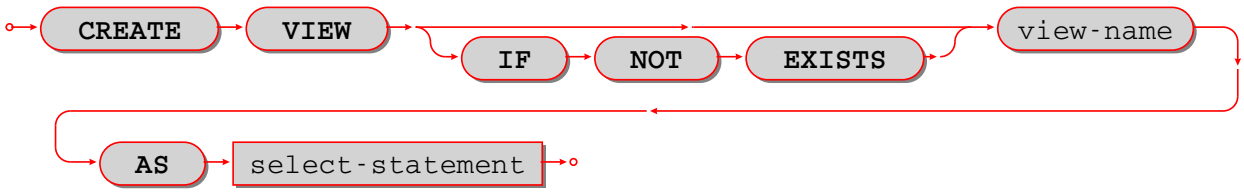
```
-- the simple case:
DROP TABLE t31;
-- with an IF EXISTS clause:
DROP TABLE IF EXISTS t31;
```

See also: *DROP VIEW*.

CREATE VIEW

Syntax:

```
CREATE VIEW [IF NOT EXISTS] view-name [(column-list)] AS subquery;
```



Create a new viewed table, usually called a «view».

The *view-name* must be valid according to the rules for identifiers.

The optional *column-list* must be a comma-separated list of names of columns in the view.

The syntax of the subquery must be the same as the syntax of a SELECT statement, or of a VALUES clause.

Rules:

- There must not already be a base table or view with the same name as *view-name*.
- If *column-list* is specified, the number of columns in *column-list* must be the same as the number of columns in the *select-list* of the subquery.

Actions:

1. Tarantool will throw an error if a rule is violated.
2. Tarantool will create a new persistent object with *column-names* equal to the names in the *column-list* or the names in the subquery's *select-list*.
3. Tarantool effectively executes a COMMIT statement.

Examples:

```
-- the simple case:
CREATE VIEW v AS SELECT column1, column2 FROM t;
-- with a column-list:
CREATE VIEW v (a,b) AS SELECT column1, column2 FROM t;
```

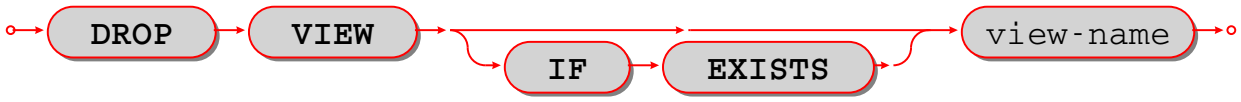
Limitations:

- It is not possible to insert or update or delete from a view, although sometimes a possible substitution is to create an INSTEAD OF trigger.

DROP VIEW

Syntax:

DROP VIEW [IF EXISTS] *view-name* ;



Drop a view.

The *view-name* must identify a view that was created earlier with the *CREATE VIEW statement*.

Rules: none

Actions:

1. Tarantool returns an error if the view does not exist.
2. The view is dropped.
3. All triggers for the view are dropped.
4. Tarantool effectively executes a COMMIT statement.

Examples:

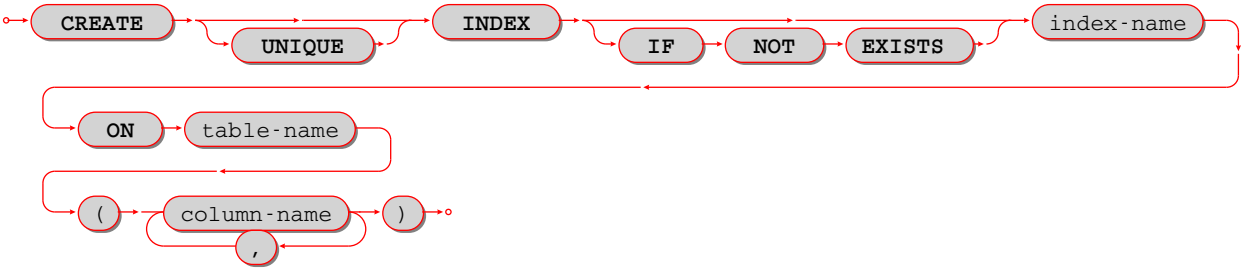
```
-- the simple case:
DROP VIEW v31;
-- with an IF EXISTS clause:
DROP VIEW IF EXISTS v31;
```

See also: *DROP TABLE*.

CREATE INDEX

Syntax:

CREATE [UNIQUE] INDEX [IF NOT EXISTS] *index-name* ON *table-name* (column-list);



Create an index.

The *index-name* must be valid according to the rules for identifiers.

The *table-name* must refer to an existing table.

The *column-list* must be a comma-separated list of names of columns in the table.

Rules:

- There must not already be, for the same table, an index with the same name as *index-name*.
- An index name is local to the table the index is defined on.
- The maximum number of indexes per table is 128.

Actions:

1. Tarantool will throw an error if a rule is violated.
2. If the new index is UNIQUE, Tarantool will throw an error if any row exists with columns that have duplicate values.
3. Tarantool will create a new index.
4. Tarantool effectively executes a COMMIT statement.

Automatic indexes:

Indexes may be created automatically for columns mentioned in the PRIMARY KEY or UNIQUE clauses of a CREATE TABLE statement. If an index was created automatically, then the *index-name* is based on four items:

1. `pk` if this is for a PRIMARY KEY clause, `unique` if this is for a UNIQUE clause;
2. `_unnamed_`;
3. the name of the table;
4. `_` and an ordinal number; the first index is 1, the second index is 2, and so on.

For example, after `CREATE TABLE t (s1 INT PRIMARY KEY, s2 INT, UNIQUE (s2));` there are two indexes named `pk_unnamed_T_1` and `unique_unnamed_T_2`. You can confirm this by saying `SELECT * FROM "_index";` which will list all indexes on all tables. There is no need to say `CREATE INDEX` for columns that already have automatic indexes.

Examples:

```
-- the simple case
CREATE INDEX i ON t (column1);
-- with IF NOT EXISTS clause
CREATE INDEX IF NOT EXISTS i ON t (column1);
-- with UNIQUE specifier and more than one column
CREATE UNIQUE INDEX i ON t (column1, column2);
```

Dropping an automatic index created for a unique constraint will drop the unique constraint as well.

DROP INDEX

Syntax:

```
DROP INDEX [IF EXISTS] index-name ON table-name ;
```

The *index-name* must be the name of an existing index, which was created with CREATE INDEX. Or, the *index-name* must be the name of an index that was created automatically due to a PRIMARY KEY or UNIQUE clause in the CREATE TABLE statement. To see what a table's indexes are, use `PRAGMA index_list (table-name)`.

Rules: none

Actions:

1. Tarantool throws an error if the index does not exist, or is an automatically created index.
2. Tarantool will drop the index.
3. Tarantool effectively executes a COMMIT statement.

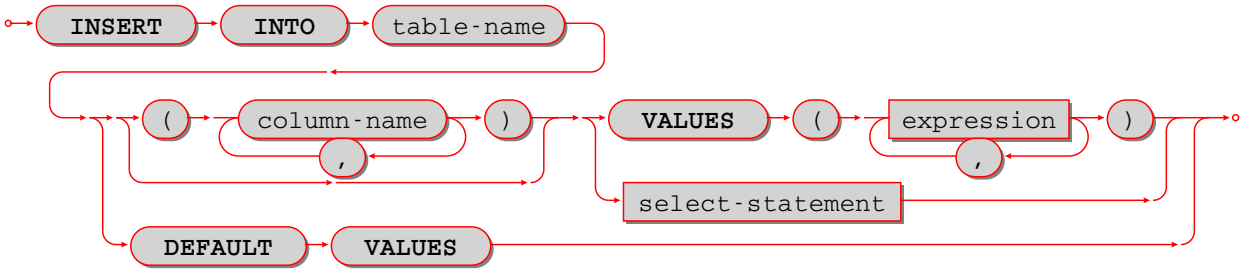
Пример:

```
-- the simplest form:
DROP INDEX i ON t;
```

INSERT

Syntax:

- `INSERT INTO` *`table-name`* `[(column-list)] VALUES (expression-list) [, (expression-list)];`
- `INSERT INTO` *`table-name`* `[(column-list)] select-statement;`
- `INSERT INTO` *`table-name`* `DEFAULT VALUES;`



Insert one or more new rows into a table.

The *table-name* must be a name of a table defined earlier with CREATE TABLE.

The optional *column-list* must be a comma-separated list of names of columns in the table.

The *expression-list* must be a comma-separated list of expressions; each expression may contain literals and operators and subqueries and function invocations.

Rules:

- The values in the *expression-list* are evaluated from left to right.
- The order of the values in the *expression-list* must correspond to the order of the columns in the table, or (if a *column-list* is specified) to the order of the columns in the *column-list*.
- The data type of the value should correspond to the data type of the column, that is, the data type that was specified with CREATE TABLE.

- If a *column-list* is not specified, then the number of expressions must be the same as the number of columns in the table.
- If a *column-list* is specified, then some columns may be omitted; omitted columns will get default values.
- The parenthesized *expression-list* may be repeated – `(expression-list),(expression-list),...` – for multiple rows.

Actions:

1. Tarantool evaluates each expression in *expression-list*, and returns an error if any of the rules is violated.
2. Tarantool creates zero or more new rows containing values based on the values in the VALUES list or based on the results of the *select-expression* or based on the default values.
3. Tarantool executes constraint checks and trigger actions and the actual insertion.
4. Tarantool inserts values into the table.

Examples:

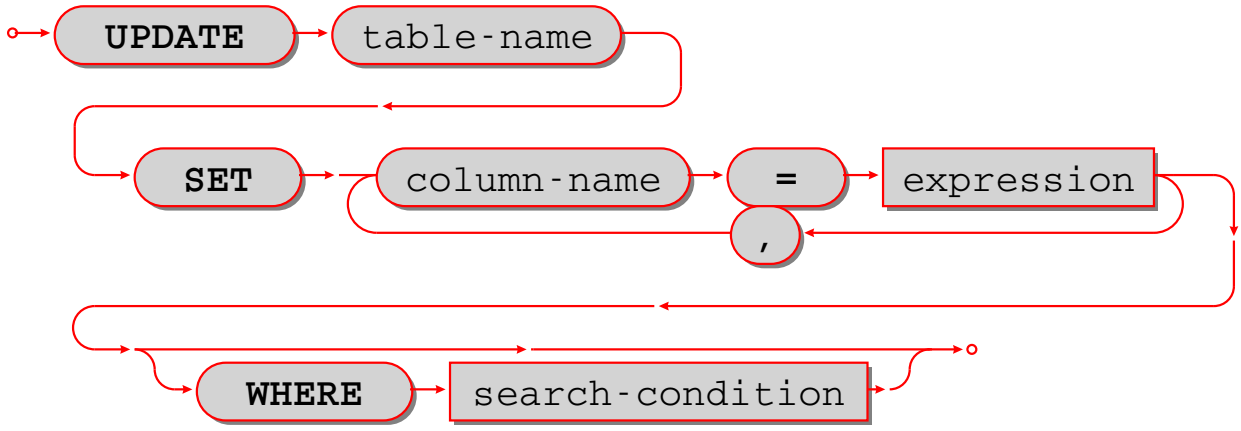
```
-- the simplest form:
INSERT INTO table1 VALUES (1, 'A');
-- with a column list:
INSERT INTO table1 (column1, column2) VALUES (2, 'B');
-- with an arithmetic operator in the first expression:
INSERT INTO table1 VALUES (2 + 1, 'C');
-- put two rows in the table:
INSERT INTO table1 VALUES (4, 'D'), (5, 'E');
```

See also: *REPLACE statement*.

UPDATE

Syntax:

`UPDATE` *`table-name`* `SET column-name = expression [, column-name = expression ...] [WHERE search-condition];`



Update zero or more existing rows in a table.

The *table-name* must be a name of a table defined earlier with CREATE TABLE or CREATE VIEW.

The *column-name* must be an updatable column in the table.

The *expression* may contain literals and operators and subqueries and function invocations and column names.

Rules:

- The values in the SET clause are evaluated from left to right.
- The data type of the value should correspond to the data type of the column, that is, the data type that was specified with CREATE TABLE.
- If a *search-condition* is not specified, then all rows in the table will be updated; otherwise only those rows which match the *search-condition* will be updated.

Actions:

1. Tarantool evaluates each expression in the SET clause, and returns an error if any of the rules is violated. For each row that is found by the WHERE clause, a temporary new row is formed based on the original contents and the modifications caused by the SET clause.
2. Tarantool executes constraint checks and trigger actions and the actual update.

Examples:

```
-- the simplest form:
UPDATE t SET column1 = 1;
-- with more than one assignment in the SET clause:
UPDATE t SET column1 = 1, column2 = 2;
-- with a WHERE clause:
UPDATE t SET column1 = 5 WHERE column2 = 6;
```

Special cases:

It is legal to say SET (list of columns) = (list of values). For example:

```
UPDATE t SET (column1, column2, column3) = (1,2,3);
```

It is not legal to assign to a column more than once. For example:

```
INSERT INTO t (column1) VALUES (0);
UPDATE t SET column1 = column1 + 1, column1 = column1 + 1;
```

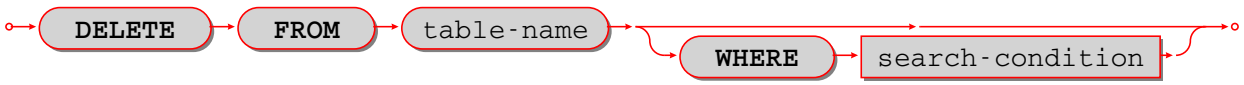
The result is an error: «duplicate column name».

It is not legal to assign to a primary-key column.

DELETE

Syntax:

DELETE FROM *table-name* [WHERE search-condition];



Delete zero or more existing rows in a table.

The *table-name* must be a name of a table defined earlier with CREATE TABLE or CREATE VIEW.

The *search-condition* may contain literals and operators and subqueries and function invocations and column names.

Rules:

- If a search-condition is not specified, then all rows in the table will be deleted; otherwise only those rows which match the *search-condition* will be deleted.

Actions:

1. Tarantool evaluates each expression in the *search-condition*, and returns an error if any of the rules is violated.
2. Tarantool finds the set of rows that are to be deleted.
3. Tarantool executes constraint checks and trigger actions and the actual deletion.
4. Tarantool deletes the set of matching rows from the table.

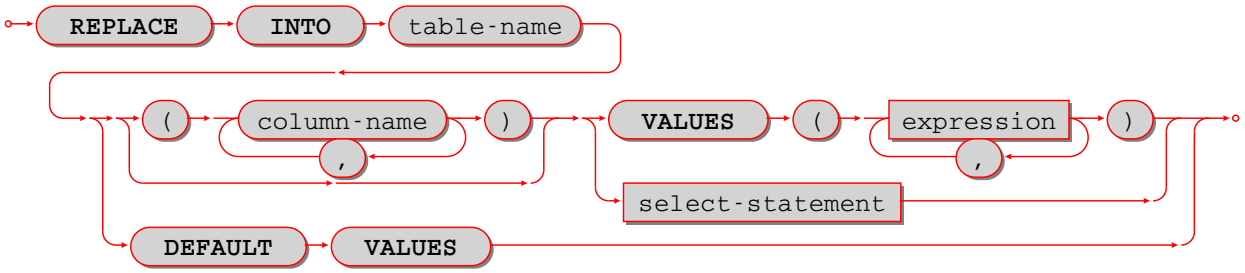
Examples:

```
-- the simplest form:
DELETE FROM t;
-- with a WHERE clause:
DELETE FROM t WHERE column2 = 6;
```

REPLACE

Syntax:

- `REPLACE INTO` *`table-name`* `[(column-list)] VALUES (expression-list) [, (expression-list)];`
- `REPLACE INTO` *`table-name`* `[(column-list)] select-statement;`
- `REPLACE INTO` *`table-name`* `DEFAULT VALUES;`



Insert one or more new rows into a table, or update existing rows.

If a row already exists (as determined by the primary key or any unique key), then the action is delete + insert, and the rules are the same as for a DELETE statement followed by an INSERT statement. Otherwise the action is insert, and the rules are the same as for the INSERT statement.

Examples:

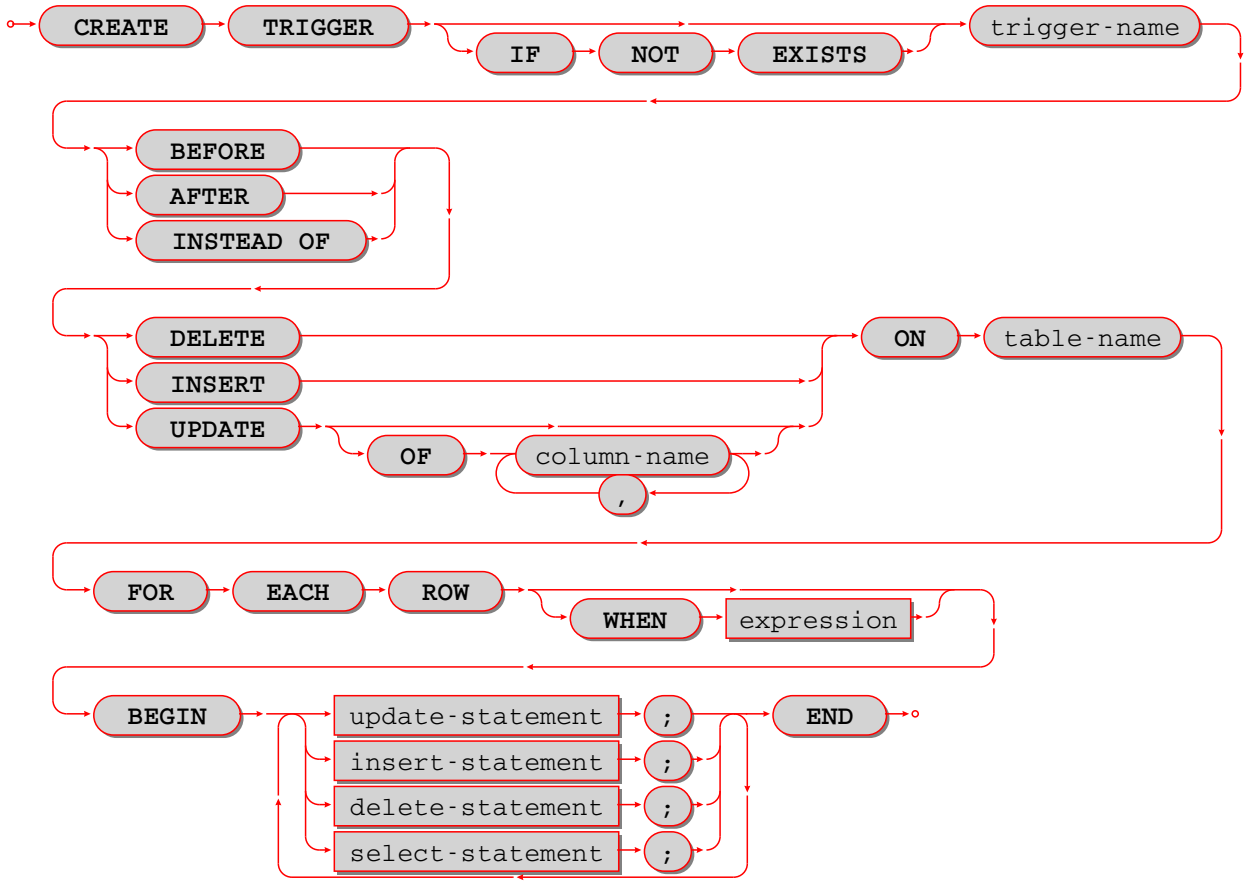
```
-- the simplest form:
REPLACE INTO table1 VALUES (1, 'A');
-- with a column list:
REPLACE INTO table1 (column1, column2) VALUES (2, 'B');
-- with an arithmetic operator in the first expression:
REPLACE INTO table1 VALUES (2 + 1, 'C');
-- put two rows in the table:
REPLACE INTO table1 VALUES (4, 'D'), (5, 'E');
```

See also: *INSERT Statement*, *UPDATE Statement*.

CREATE TRIGGER

Syntax:

```
CREATE TRIGGER [IF NOT EXISTS] trigger-name BEFORE|AFTER|INSTEAD OF INSERT|UPDATE|DELETE
ON table-name FOR EACH ROW [WHEN (search-condition)] BEGIN update-statement
| insert-statement | delete-statement | select-statement; [update-statement |
insert-statement | delete-statement | select-statement; ...] END;
```



The *trigger-name* must be valid according to the rules for identifiers.

If the trigger action time is BEFORE or AFTER, then the *table-name* must refer to an existing base table.

If the trigger action time is INSTEAD OF, then the *table-name* must refer to an existing view.

Rules:

- There must not already be a trigger with the same name as *trigger-name*.
- Triggers on different tables or views share the same namespace.
- The statements between BEGIN and END should not refer to the *table-name* mentioned in the ON clause.
- The statements between BEGIN and END should not contain an INDEXED BY clause.

SQL triggers are not fired upon Tarantool/NoSQL requests. This will change in version 2.2.

On a replica, effects of trigger execution are applied, and the SQL triggers themselves are not fired upon replication events.

NoSQL triggers are fired both on replica and master, thus if you have a NoSQL trigger on replica, it is fired when applying effects of an SQL trigger.

Actions:

1. Tarantool will throw an error if a rule is violated.
2. Tarantool will create a new trigger.
3. Tarantool effectively executes a COMMIT statement.

Examples:

```
-- the simple case:
CREATE TRIGGER delete_if_insert BEFORE INSERT ON stores FOR EACH ROW
  BEGIN DELETE FROM warehouses; END;
-- with IF NOT EXISTS clause:
CREATE TRIGGER IF NOT EXISTS delete_if_insert BEFORE INSERT ON stores FOR EACH ROW
  BEGIN DELETE FROM warehouses; END;
-- with FOR EACH ROW and WHEN clauses:
CREATE TRIGGER delete_if_insert BEFORE INSERT ON stores FOR EACH ROW WHEN a=5
  BEGIN DELETE FROM warehouses; END;
-- with multiple statements between BEGIN and END:
CREATE TRIGGER delete_if_insert BEFORE INSERT ON stores FOR EACH ROW
  BEGIN DELETE FROM warehouses; INSERT INTO inventories VALUES (1); END;
```

Trigger extra clauses

- `UPDATE OF column-list`

  After BEFORE|AFTER UPDATE it is optional to add `OF column-list`. If any of the columns in *column-list* is affected at the time the row is processed, then the trigger will be activated for that row. For example:

  ```
  CREATE TRIGGER trigger_on_table1
   BEFORE UPDATE  OF column1, column2 ON table1
   FOR EACH ROW
   BEGIN UPDATE table2 SET column1 = column1 + 1; END;
  UPDATE table1 SET column3 = column3 + 1; -- Trigger will not be activated
  UPDATE table1 SET column2 = column2 + 0; -- Trigger will be activated
  ```

- `WHEN`

  After *table-name* FOR EACH ROW it is optional to add [`WHEN expression`]. If the expression is true at the time the row is processed, only then the trigger will be activated for that row. For example:

  ```
  CREATE TRIGGER trigger_on_table1 BEFORE UPDATE ON table1 FOR EACH ROW
   WHEN (SELECT COUNT(*) FROM table1) > 1
   BEGIN UPDATE table2 SET column1 = column1 + 1; END;
  ```

  This trigger will not be activated unless there is more than one row in `table1`.

- `OLD and NEW`

  The keywords OLD and NEW have special meaning in the context of trigger action:

  – OLD.column-name refers to the value of *column-name* before the change.

  – NEW.column-name refers to the value of *column-name* after the change.

  Пример:

```
CREATE TABLE table1 (column1 VARCHAR(15), column2 INT PRIMARY KEY);
CREATE TABLE table2 (column1 VARCHAR(15), column2 VARCHAR(15), column3 INT PRIMARY KEY);
INSERT INTO table1 VALUES ('old value', 1);
INSERT INTO table2 VALUES ('', '', 1);
CREATE TRIGGER trigger_on_table1 BEFORE UPDATE ON table1 FOR EACH ROW
 BEGIN UPDATE table2 SET column1 = old.column1, column2 = new.column1; END;
UPDATE table1 SET column1 = 'new value';
SELECT * FROM table2;
```

At the beginning of the UPDATE for the single row of `table1`, the value in `column1` is „old value“ – so that is what is seen as `old.column1`.

At the end of the UPDATE for the single row of `table1`, the value in `column1` is „new value“ – so that is what is seen as `new.column1`. (OLD and NEW are qualifiers for `table1`, not `table2`.)

Therefore, `SELECT * FROM table2;` returns `['old value', 'new value']`.

`OLD.column-name` does not exist for an INSERT trigger.

`NEW.column-name` does not exist for a DELETE trigger.

OLD and NEW are read-only; you cannot change their values.

- Deprecated or illegal statements:

  It is legal for the trigger action to include a SELECT statement or a REPLACE statement, but not recommended.

  It is illegal for the trigger action to include a qualified column reference other than `OLD.column-name` or `NEW.column-name`. For example, `CREATE TRIGGER ... BEGIN UPDATE table1 SET table1.column1=5; END;` is illegal.

  It is illegal for the trigger action to include statements that include a WITH clause, a DEFAULT VALUES clause, or an INDEXED BY clause.

  It is usually not a good idea to have a trigger on `table1` which causes a change on `table2`, and at the same time have a trigger on `table2` which causes a change on `table1`. For example:

```
CREATE TRIGGER trigger_on_table1
 BEFORE UPDATE ON table1
 FOR EACH ROW
 BEGIN UPDATE table2 SET column1 = column1 + 1; END;
CREATE TRIGGER trigger_on_table2
 BEFORE UPDATE ON table2
 FOR EACH ROW
 BEGIN UPDATE table1 SET column1 = column1 + 1; END;
```

  Luckily `UPDATE table1 ...` will not cause an infinite loop, because Tarantool recognizes when it has already updated so it will stop. However, not every DBMS acts this way.

Trigger activation

These are remarks concerning trigger activation.

Standard terminology:

- «trigger action time» = BEFORE or AFTER or INSTEAD OF
- «trigger event» = INSERT or DELETE or UPDATE
- «triggered statement» = BEGIN ... INSERT|DELETE|UPDATE ... END

- «triggered when clause» = WHEN (search condition)
- «activate» = execute a triggered statement
- some vendors use the word «fire» instead of «activate»

If there is more than one trigger for the same trigger event, Tarantool may execute the triggers in any order.

It is possible for a triggered statement to cause activation of another triggered statement. For example, this is legal:

```
CREATE TRIGGER on_t1 BEFORE DELETE ON t1 BEGIN DELETE FROM t2; END;
CREATE TRIGGER on_t2 BEFORE DELETE ON t2 BEGIN DELETE FROM t3; END;
```

Activation occurs FOR EACH ROW, not FOR EACH STATEMENT. Therefore, if no rows are candidates for insert or update or delete, then no triggers are activated.

The BEFORE trigger is activated even if the trigger event fails.

If an UPDATE trigger event does not make a change, the trigger is activated anyway. For example, if row 1 `column1` contains „a“, and the trigger event is `UPDATE ... SET column1 = 'a';`, the trigger is activated.

The triggered statement may refer to a function: `RAISE(FAIL, error-message)`. If a triggered statement invokes a `RAISE(FAIL, error-message)` function, or if a triggered statement causes an error, then statement execution stops immediately.

The triggered statement may refer to column values within the rows being changed. in this case:

- The row «as of before» the change is called the «old» row (which makes sense only for UPDATE and DELETE statements).
- The row «as of after» the change is called the «new» row (which makes sense only for UPDATE and INSERT statements).

This example shows how an INSERT can be done to a view by referring to the «new» row:

```
CREATE TABLE t (s1 INT PRIMARY KEY, s2 INT);
CREATE VIEW v AS SELECT s1, s2 FROM t;
CREATE TRIGGER tv INSTEAD OF INSERT ON v
  FOR EACH ROW
  BEGIN INSERT INTO t VALUES (new.s1, new.s2); END;
INSERT INTO v VALUES (1,2);
```

Ordinarily saying `INSERT INTO view_name ...` is illegal in Tarantool, so this is a workaround.

It is possible to generalize this so that all data-change statements on views will change the base tables, provided that the view contains all the columns of the base table, and provided that the triggers refer to those columns when necessary, as in this example:

```
CREATE TABLE base_table (primary_key_column INT PRIMARY KEY, value_column INT);
CREATE VIEW viewed_table AS SELECT primary_key_column, value_column FROM base_table;
CREATE TRIGGER viewed_insert INSTEAD OF INSERT ON viewed_table FOR EACH ROW
  BEGIN
    INSERT INTO base_table VALUES (new.primary_key_column, new.value_column);
  END;
CREATE TRIGGER viewed_update INSTEAD OF UPDATE ON viewed_table FOR EACH ROW
  BEGIN
    UPDATE base_table
    SET primary_key_column = new.primary_key_column, value_column = new.value_column
    WHERE primary_key_column = old.primary_key_column;
  END;
```

(continues on next page)

(продолжение с предыдущей страницы)

```
CREATE TRIGGER viewed_delete INSTEAD OF DELETE ON viewed_table FOR EACH ROW
  BEGIN
    DELETE FROM base_table WHERE primary_key_column = old.primary_key_column;
  END;
```

When INSERT or UPDATE or DELETE occurs for table `X`, Tarantool usually operates in this order (a basic scheme):

```
For each row
  Perform constraint checks
  For each BEFORE trigger that refers to table X
    Check that the trigger's WHEN condition is true.
    Execute what is in the trigger's BEGIN|END block.
  Insert or update or delete the row in table X.
  Perform more constraint checks
  For each AFTER trigger that refers to table X
    Check that the trigger's WHEN condition is true.
    Execute what is in the trigger's BEGIN|END block.
```

However, Tarantool does not guarantee execution order when there are multiple constraints, or multiple triggers for the same event (including NoSQL `on_replace` triggers or SQL INSTEAD OF triggers that affect a view of table `X`).

The maximum number of trigger activations per statement is 32.

INSTEAD OF triggers

A trigger which is created with the clause `INSTEAD OF` *`INSERT|UPDATE|DELETE`* `ON` *`view-name`* is an INSTEAD OF trigger. For each affected row, the trigger action is performed «instead of» the INSERT or UPDATE or DELETE statement that causes trigger activation.

For example, ordinarily it is illegal to INSERT rows in a view, but it is legal to create a trigger which intercepts attempts to INSERT, and puts rows in the underlying base table:

```
CREATE TABLE t1 (column1 INT PRIMARY KEY, column2 INT);
CREATE VIEW v1 AS SELECT column1, column2 FROM t1;
CREATE TRIGGER t1 INSTEAD OF INSERT ON v1 FOR EACH ROW BEGIN
 INSERT INTO t1 VALUES (NEW.column1, NEW.column2); END;
INSERT INTO v1 VALUES (1, 1);
-- ... The result will be: table t1 will contain a new row.
```

INSTEAD OF triggers are only legal for views, while BEFORE or AFTER triggers are not legal for views.

It is legal to create INSTEAD OF triggers with triggered WHEN clauses.

Limitations:

- It is legal to create INSTEAD OF triggers with UPDATE OF *column-list* clauses, but they are not standard SQL.

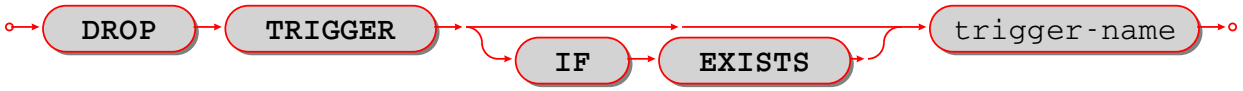
  Пример:

```
CREATE TRIGGER et1
  INSTEAD OF UPDATE OF column2,column1 ON ev1
  FOR EACH ROW BEGIN
  INSERT INTO et2 VALUES (NEW.column1, NEW.column2); END;
```

DROP TRIGGER

Syntax:

`DROP TRIGGER [IF EXISTS]` *trigger-name* `;`



Drop a trigger.

The *trigger-name* must identify a trigger that was created earlier with the CREATE TRIGGER statement.

Rules: none

Actions:

1. Tarantool returns an error if the trigger does not exist.
2. The trigger is dropped.
3. Tarantool effectively executes a COMMIT statement.

Examples:

```
-- the simple case:
DROP TRIGGER tr;
-- with an IF EXISTS clause:
DROP TRIGGER IF EXISTS tr;
```

TRUNCATE

Syntax:

`TRUNCATE TABLE` *table-name* `;`

Remove all rows in the table.

TRUNCATE is considered to be a schema-change rather than a data-change statement, so it does not work within transactions (it cannot be rolled back).

Rules:

- It is illegal to truncate a table which is referenced by a foreign key.
- It is illegal to truncate a table which is also a system space, such as `_space`.
- The table must be a base table rather than a view.

Actions:

1. All rows in the table are removed. Usually this is faster than `DELETE FROM table-name;`.
2. If the table has an autoincrement primary key, its sequence is reset to zero.
3. There is no effect for any triggers associated with the table.
4. There is no effect on the counts for the `row_count()` function.

5. Only one action is written to the write-ahead log (with `DELETE FROM table-name;` there would be one action for each deleted row).

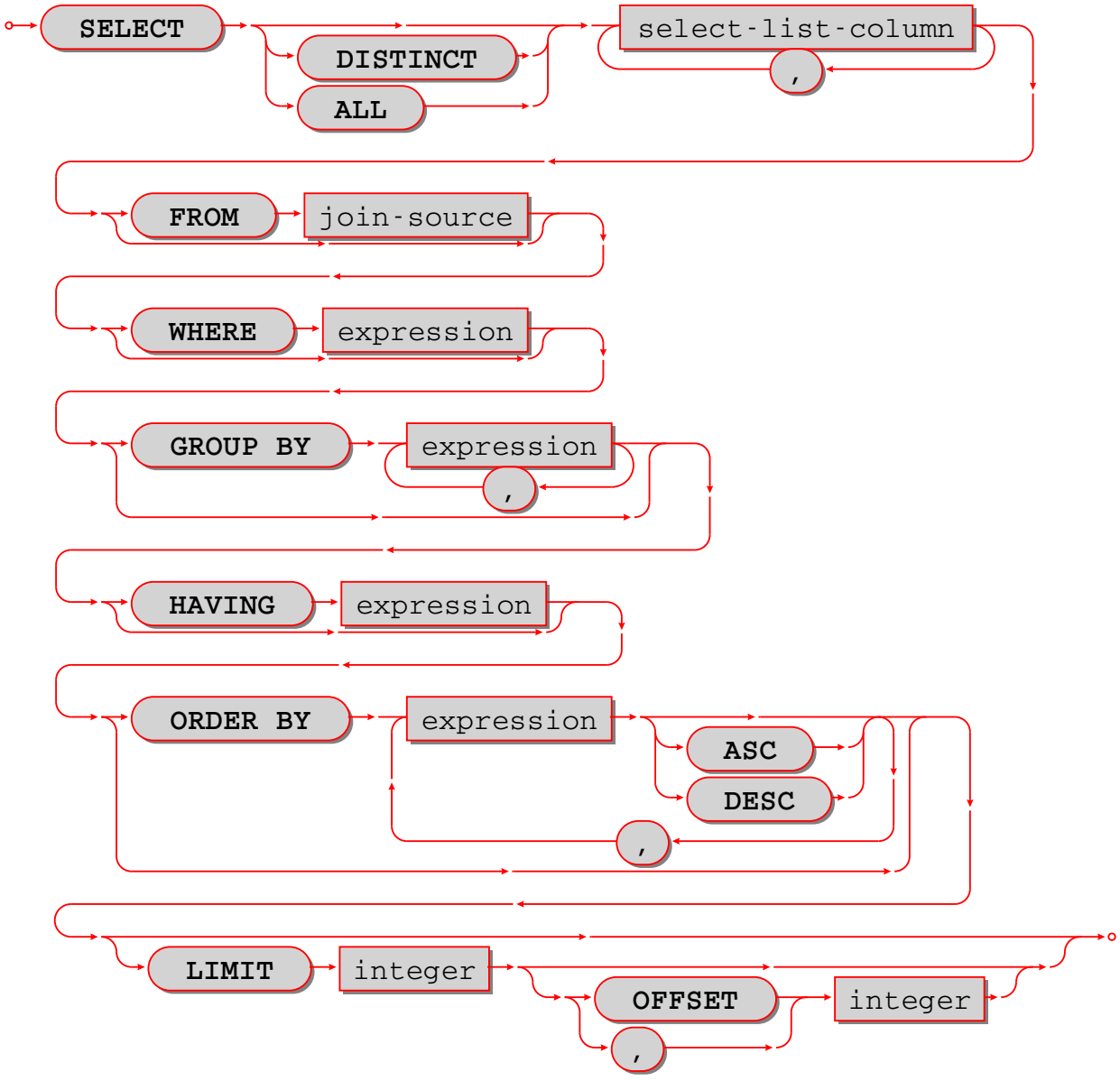
Пример:

```
TRUNCATE TABLE t;
```

SELECT

Syntax:

```
SELECT [ALL|DISTINCT] select-list [from clause] [where clause] [group-by clause] [having
clause] [order-by clause];
```



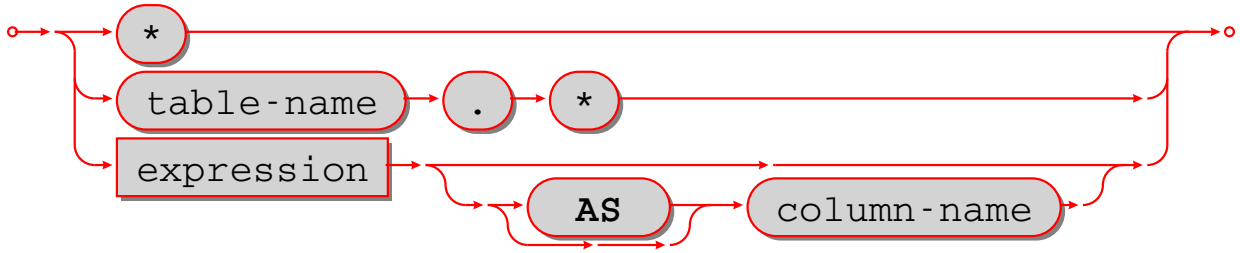
Select zero or more rows.

The clauses of the SELECT statement are discussed in the following five sections.

Select-list

Syntax:

```
select-list-column [, select-list-column ...] select-list-column:
```



Define what will be in a result set; this is a clause in a SELECT statement.

The *select-list* is a comma-delimited list of expressions, or `*` (asterisk). An expression can have an alias provided with `[AS [column-name]]` clause.

The `*` «asterisk» shorthand is valid if and only if the SELECT statement also contains a FROM clause which specifies the table or tables (details about the FROM clause are in the next section). The simple form is `*` which means «all columns» – for example, if the select is done for a table which contains three columns `s1` `s2` `s3`, then `SELECT * ...` is equivalent to `SELECT s1, s2, s3 ...`. The qualified form is `table-name.*` which means «all columns in the specified table», which again must be a result of the FROM clause – for example, if the table is named `table1`, then `table1.*` is equivalent to a list of the columns of `table1`.

The `[AS [column-name]]` clause determines the column name. The column name is useful for two reasons:

- in a tabular display, the column names are the headings
- if the results of the SELECT are used in `CREATE TABLE new-table-name ... AS SELECT select-list ...`, then the column names in the new table will be the column names in the *select-list*.

If `[AS [column-name]]` is missing, Tarantool makes a name equal to the expression, for example `SELECT 5*88` will cause the column name to be `5*88`, but such names may be ambiguous or illegal in other contexts, so it is better to say, for example, `SELECT 5 * 88 AS column1`.

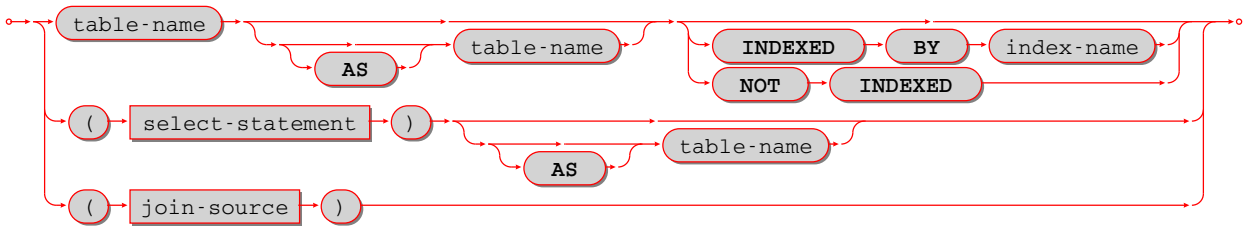
Examples:

```
-- the simple form:
SELECT 5;
-- with multiple expressions including operators:
SELECT 1, 2 * 2, 'Three' || 'Four';
-- with [[AS] column-name] clause:
SELECT 5 AS column1;
-- * which must be eventually followed by a FROM clause:
SELECT * FROM table1;
-- as a list:
SELECT 1 AS a, 2 AS b, table1.* FROM table1;
```

FROM clause

Syntax:

```
FROM table-reference [, table-reference ...]
```



Specify the table or tables for the source of a SELECT statement.

The *table-reference* must be a name of an existing table, or a subquery, or a joined table.

A joined table looks like this:

`table-reference-or-joined-table join-operator table-reference-or-joined-table [join-specification]`

A *join-operator* must be any of the standard types:

- [NATURAL] LEFT [OUTER] JOIN,
- [NATURAL] INNER JOIN, or
- CROSS JOIN

A *join-specification* must be any of:

- ON expression, or
- USING (column-name [, column-name ...])

Parentheses are allowed, and `[[AS] correlation-name]` is allowed.

The maximum number of joins in a FROM clause is 64.

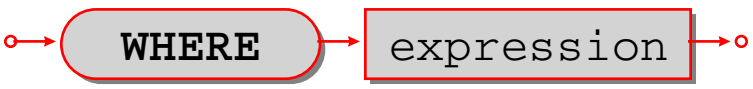
Examples:

```
-- the simplest form:
SELECT * FROM t;
-- with two tables, making a Cartesian join:
SELECT * FROM t1, t2;
-- with one table joined to itself, requiring correlation names:
SELECT a.*, b.* FROM t1 AS a, t1 AS b;
-- with a left outer join:
SELECT * FROM t1 LEFT JOIN t2;
```

WHERE clause

Syntax:

`WHERE condition;`



Specify the condition for filtering rows from a table; this is a clause in a SELECT or UPDATE or DELETE statement.

The condition may contain any expression that returns a BOOLEAN (TRUE or FALSE or UNKNOWN) value.

For each row in the table:

- if the condition is true, then the row is kept;
- if the condition is false or unknown, then the row is ignored.

In effect, WHERE condition takes a table with n rows and returns a table with n or fewer rows.

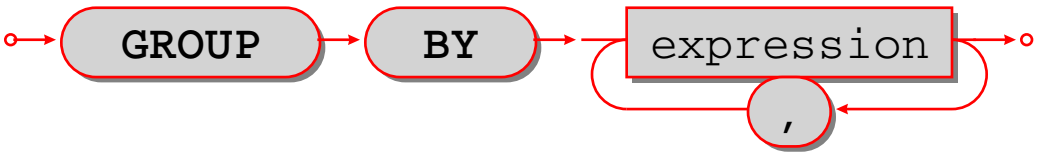
Examples:

```
-- with a simple condition:
SELECT 1 FROM t WHERE column1 = 5;
-- with a condition that contains AND and OR and parentheses:
SELECT 1 FROM t WHERE column1 = 5 AND (x > 1 OR y < 1);
```

GROUP BY clause

Syntax:

```
GROUP BY expression [, expression ...]
```



Make a grouped table; this is a clause in a SELECT statement.

The expressions should be column names in the table, and each column should be specified only once.

In effect, GROUP BY clause takes a table with rows that may have matching values, combines rows that have matching values into single rows, and returns a table which, because it is the result of GROUP BY, is called a grouped table.

Thus, if the input is a table:

```
a    b      c
-    -      -
1    'a'    'b
1    'b'    'b'
2    'a'    'b'
3    'a'    'b'
1    'b'    'b'
```

then `GROUP BY a, b` will produce a grouped table:

```
a    b      c
-    -      -
1    'a'    'b'
1    'b'    'b'
2    'a'    'b'
3    'a'    'b'
```

The rows where column `a` and column `b` have the same value have been merged; column `c` has been preserved but its value should not be depended on – if the rows were not all „b“, Tarantool could pick any value.

It is useful to envisage a grouped table as having hidden extra columns for the aggregation of the values, for example:

```
a    b      c    COUNT(a) SUM(a) MIN(c)
-    -      -    -------- ------ ------
1    'a'  'b'         2      2    'b'
1    'b'  'b'         1      1    'b'
2    'a'  'b'         1      2    'b'
     'a'  'b'         1      3    'b'
```

These extra columns are what *aggregate functions* are for.

Examples:

```
-- with a single column:
SELECT 1 FROM t GROUP BY column1;
-- with two columns:
SELECT 1 FROM t GROUP BY column1, column2;
```

Limitations:

- `SELECT s1,s2 FROM t GROUP BY s1;` is legal.
- `SELECT s1 AS q FROM t GROUP BY q;` is legal.
- `SELECT s1 FROM t GROUP by 1;` is legal.

Aggregate functions

Syntax:

`function-name (one or more expressions)`

Apply a built-in aggregate function to one or more expressions and return a scalar value.

Aggregate functions are only legal in certain clauses of SELECT for grouped tables. (A table is a grouped table if a GROUP BY clause is present.) Also, if an aggregate function is used in a select-list and GROUP BY clause is omitted, then Tarantool assumes `SELECT ... GROUP BY [all columns];`.

NULLs are ignored for all aggregate functions except COUNT(*).

`AVG([DISTINCT] expression)` Return the average value of expression.

Example: `AVG(column1)`

`COUNT([DISTINCT] expression)` Return the number of occurrences of expression.

Example: `COUNT(column1)`

`COUNT(*)` Return the number of occurrences of a row.

Example: `COUNT(*)`

`GROUP_CONCAT(expression-1 [, expression-2])` Return a list of *expression-1* values, separated by commas if *expression-2* is omitted, or separated by the *expression-2* value if *expression-2* is not omitted.

Example: `GROUP_CONCAT(column1)`

`MAX([DISTINCT] expression)` Return the maximum value of expression.

Example: `MAX(column1)`

`MIN([DISTINCT] expression)` Return the minimum value of expression.

Example: `MIN(column1)`

`SUM([DISTINCT] expression)` Return the sum of values of expression.

Example: `SUM(column1)`

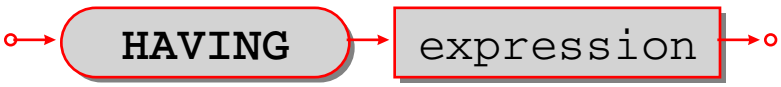
`TOTAL([DISTINCT] expression)` Return the sum of values of expression.

Example: `TOTAL(column1)`

HAVING clause

Syntax:

```
HAVING condition;
```



Specify the condition for filtering rows from a grouped table; this is a clause in a SELECT statement.

The clause preceding the HAVING clause may be a GROUP BY clause. HAVING operates on the table that the GROUP BY produces, which may contain grouped columns and aggregates.

If the preceding clause is not a GROUP BY clause, then there is only one group and the HAVING clause may only contain aggregate functions or literals.

For each row in the table:

- if the condition is true, then the row is kept;
- if the condition is false or unknown, then the row is ignored.

In effect, HAVING condition takes a table with n rows and returns a table with n or fewer rows.

Examples:

```
-- with a simple condition:
SELECT 1 FROM t GROUP BY column1 HAVING column2 > 5;
-- with a more complicated condition:
SELECT 1 FROM t GROUP BY column1 HAVING column2 > 5 OR column2 < 5;
-- with an aggregate:
SELECT x, SUM(y) FROM t GROUP BY x HAVING SUM(y) > 0;
-- with no GROUP BY and an aggregate:
SELECT SUM(y) FROM t GROUP BY x HAVING MIN(y) < MAX(y);
```

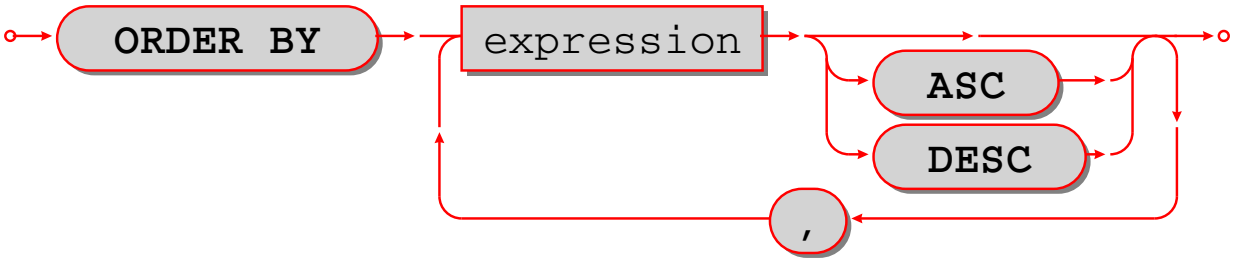
Limitations:

- HAVING without GROUP BY is not supported for multiple tables.

ORDER BY clause

Syntax:

```
ORDER BY expression [ASC|DESC] [, expression [ASC|DESC] ...]
```



Put rows in order; this is a clause in a SELECT statement.

An ORDER BY expression has one of three types which are checked in order:

1. Expression is a positive integer, representing the ordinal position of the column in the select list. For example, in the statement `SELECT x, y, z FROM t ORDER BY 2; ORDER BY 2` means «order by the second column in the select list», which is `y`.
2. Expression is a name of a column in the select list, which is determined by an AS clause. For example, in the statement `SELECT x, y AS x, z FROM t ORDER BY x; ORDER BY x` means «order by the column explicitly named `x` in the select list», which is the second column.
3. Expression contains a name of a column in a table of the FROM clause. For example, in the statement `SELECT x, y FROM t1 JOIN t2 ORDER BY z; ORDER BY z` means «order by a column named `z` which is expected to be in table `t1` or table `t2`».

If both tables contain a column named `z`, then Tarantool will choose the first column that it finds.

The expression may also contain operators and function names and literals. For example, in the statement `SELECT x, y FROM t ORDER BY UPPER(z); ORDER BY UPPER(z)` means «order by the uppercase form of column `t.z`», which may be similar to doing ordering in a case-insensitive manner.

Type 3 is illegal if the SELECT statement contains UNION or EXCEPT or INTERSECT.

If an ORDER BY clause contains multiple expressions, then expressions on the left are processed first and expressions on the right are processed only if necessary for tie-breaking. For example, in the statement `SELECT x, y FROM t ORDER BY x, y;` if there are two rows which both have the same values for column `x`, then an additional check is made to see which row has a greater value for column `y`.

In effect, ORDER BY clause takes a table with rows that may be out of order, and returns a table with rows in order.

Sorting order:

- The default order is ASC (ascending), the optional order is DESC (descending).
- NULLs come first, then numbers (INTEGER or NUMBER), then STRINGs, then VARBINARYs.
- Within STRINGs, ordering is according to collation.
- Collation may be specified within the ORDER BY column-list, or may be default.

Examples:

```
-- with a single column:
SELECT 1 FROM t ORDER BY column1;
-- with two columns:
SELECT 1 FROM t ORDER BY column1, column2;
-- with a variety of data:
CREATE TABLE h (s1 INT PRIMARY KEY, s2 INT);
INSERT INTO h VALUES (7,'A'),(4,'A '),(-4,'AZ'),(17,17),(23,NULL);
INSERT INTO h VALUES (17.5,'Д'),(1e+300,'a'),(0,''),(-1,'');
```

(continues on next page)

(продолжение с предыдущей страницы)

```
SELECT * FROM h ORDER BY s2, s1;
-- The result of the above SELECT will be:
- - [23, null]
  - [17, 17]
  - [-1, '']
  - [0, '']
  - [7, 'A']
  - [4, 'A ']
  - [-4, 'AZ']
  - [1e+300, 'a']
  - [17.5, 'Д']
...
```

Limitations:

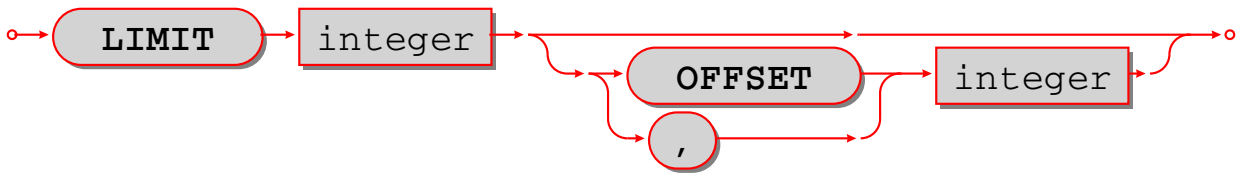
- ORDER BY 1 is legal. This is common but is not standard SQL nowadays.

LIMIT clause

Syntax:

- `LIMIT limit-expression [OFFSET offset-expression]`
- `LIMIT offset-expression, limit-expression`

Примечание: The above is not a typo: *offset-expression* and *limit-expression* are in reverse order if a comma is used.



Specify a maximum number of rows and a start row; this is a clause in a SELECT statement.

Expressions may contain integers and arithmetic operators or functions, for example `ABS(-3/1)`. However, the result must be an integer value greater than or equal to zero.

Usually the LIMIT clause follows an ORDER BY clause, because otherwise Tarantool does not guarantee that rows are in order.

Examples:

```
-- simple case:
SELECT * FROM t LIMIT 3;
-- both limit and order:
SELECT * FROM t LIMIT 3 OFFSET 1;
-- applied to a UNIONed result (LIMIT clause must be the final clause):
SELECT column1 FROM table1 UNION SELECT column1 FROM table2 ORDER BY 1 LIMIT 1;
```

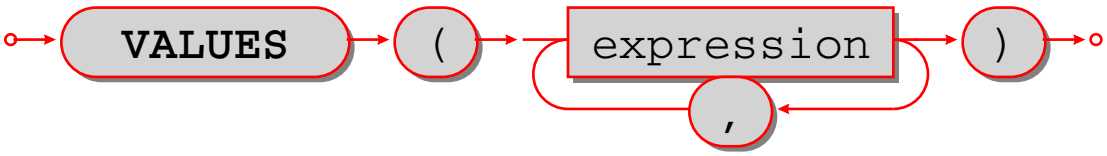
Limitations:

- If ORDER BY ... LIMIT is used, then all order-by columns must be ASC or all must be DESC.

VALUES

Syntax:

```
VALUES (expression [, expression ...]) [, (expression [, expression ...])
```



Select one or more rows.

VALUES has the same effect as SELECT, that is, it returns a result set, but VALUES statements may not have FROM or GROUP or ORDER BY or LIMIT clauses.

VALUES may be used wherever SELECT may be used, for example in subqueries.

Examples:

```
-- simple case:
VALUES (1);
-- equivalent to SELECT 1, 2, 3:
VALUES (1, 2, 3);
-- two rows:
VALUES (1, 2, 3), (4, 5, 6);
```

Subquery

Syntax:

- *SELECT-statement* syntax
- *VALUES-statement* syntax

A subquery has the same syntax as a SELECT statement or VALUES statement embedded inside a main statement.

Примечание: The SELECT and VALUES statements are called «queries» because they return answers, in the form of result sets.

Subqueries may be the second part of INSERT statements. For example:

```
INSERT INTO t2 SELECT a,b,c FROM t1;
```

Subqueries may be in the FROM clause of SELECT statements.

Subqueries may be expressions, or be inside expressions. In this case they must be parenthesized, and usually the number of rows must be 1. For example:

```
SELECT 1, (SELECT 5), 3 FROM t WHERE c1 * (SELECT COUNT(*) FROM t2) > 5;
```

Subqueries may be expressions on the right side of certain comparison operators, and in this unusual case the number of rows may be greater than 1. The comparison operators are: [NOT] EXISTS and [NOT] IN. For example:

```
DELETE FROM t WHERE s1 NOT IN (SELECT s2 FROM t);
```

Subqueries may refer to values in the outer query. In this case, the subquery is called a «correlated subquery».

Subqueries may refer to rows which are being updated or deleted by the main query. In that case, the subquery finds the matching rows first, before starting to update or delete. For example, after:

```
CREATE TABLE t (s1 INT PRIMARY KEY, s2 INT);
INSERT INTO t VALUES (1,3),(2,1);
DELETE FROM t WHERE s2 NOT IN (SELECT s1 FROM t);
```

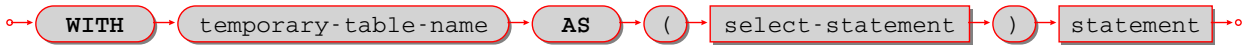
only one of the rows is deleted, not both rows.

WITH clause

WITH clause (common table expression)

Syntax:

WITH *temporary-table-name* AS (subquery) [, *temporary-table-name* AS (subquery)] SELECT statement | INSERT statement | DELETE statement | UPDATE statement | REPLACE statement;




```
WITH v AS (SELECT * FROM t) SELECT * FROM v;
```

is equivalent to creating a view and selecting from it:

```
CREATE VIEW v AS SELECT * FROM t;
SELECT * FROM v;
```

The difference is that a WITH-clause «view» is temporary and only useful within the same statement. No CREATE privilege is required.

The WITH-clause can also be thought of as a subquery that has a name. This is useful when the same subquery is being repeated. For example:

```
SELECT * FROM t WHERE a < (SELECT s1 FROM x) AND b < (SELECT s1 FROM x);
```

can be replaced with:

```
WITH S AS (SELECT s1 FROM x) SELECT * FROM t,S WHERE a < S.s1 AND b < S.s1;
```

This «factoring out» of a repeated expression is regarded as good practice.

Examples:

```
WITH cte AS (VALUES (7,'') INSERT INTO j SELECT * FROM cte;
WITH cte AS (SELECT s1 AS x FROM k) SELECT * FROM cte;
WITH cte AS (SELECT COUNT(*) FROM k WHERE s2 < 'x' GROUP BY s3)
  UPDATE j SET s2 = 5
  WHERE s1 = (SELECT s1 FROM cte) OR s3 = (SELECT s1 FROM cte);
```

WITH can only be used at the beginning of a statement, therefore it cannot be used at the beginning of a subquery or after a set operator or inside a CREATE statement.

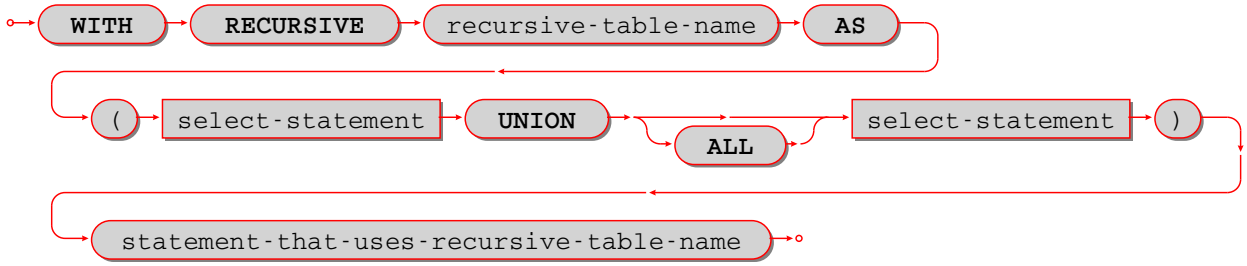
A WITH-clause «view» is read-only because Tarantool does not support updatable views.

WITH RECURSIVE

WITH RECURSIVE clause (iterative common table expression)

The real power of WITH lies in the WITH RECURSIVE clause, which is useful when it is combined with UNION or UNION ALL:

```
WITH RECURSIVE recursive-table-name AS (SELECT ... FROM non-recursive-table-name ... UNION
[ALL] SELECT ... FROM recursive-table-name ...) statement-that-uses-recursive-table-name;
```



In non-SQL this can be read as: starting with a seed value from a non-recursive table, produce a recursive viewed table, UNION that with itself, UNION that with itself, UNION that with itself ... forever, or until a condition in the WHERE clause says «stop».

Пример:

```
CREATE TABLE ts (s1 INT PRIMARY KEY);
INSERT INTO ts VALUES (1);
WITH RECURSIVE w AS (
  SELECT s1 FROM ts
  UNION ALL
  SELECT s1+1 FROM w WHERE s1 < 4)
SELECT * FROM w;
```

First, table `w` is seeded from `t1`, so it has one row: [1].

Then, `UNION ALL (SELECT s1+1 FROM w)` takes the row from `w` – which contains [1] – adds 1 because the select list says «s1+1», and so it has one row: [2].

Then, `UNION ALL (SELECT s1+1 FROM w)` takes the row from `w` – which contains [2] – adds 1 because the select list says «s1+1», and so it has one row: [3].

Then, `UNION ALL (SELECT s1+1 FROM w)` takes the row from `w` – which contains [3] – adds 1 because the select list says «s1+1», and so it has one row: [4].

Then, `UNION ALL (SELECT s1+1 FROM w)` takes the row from `w` – which contains [4] – and now the importance of the WHERE clause becomes evident, because «s1 < 4» is false for this row, and therefore we have reached the «stop» condition.

So, before the «stop», table `w` got 4 rows – [1], [2], [3], [4] – and the result of the statement looks like:

```
tarantool> WITH RECURSIVE w AS (
         >   SELECT s1 FROM ts
         >   UNION ALL
         >   SELECT s1+1 FROM w WHERE s1 < 4)
         > SELECT * FROM w;
---
- - [1]
  - [2]
  - [3]
```

(continues on next page)

(продолжение с предыдущей страницы)

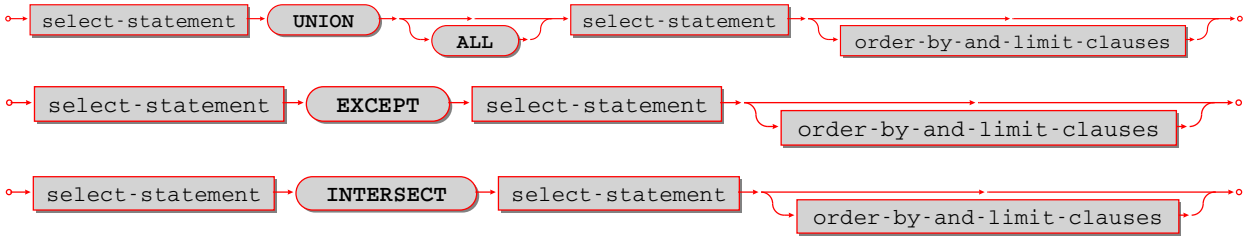
```
  - [4]
...
```

In other words, this `WITH RECURSIVE ... SELECT` produces a table of auto-incrementing values.

UNION, EXCEPT, and INTERSECT clauses

Syntax:

- `select-statement UNION [ALL] select-statement [ORDER BY clause] [LIMIT clause];`
- `select-statement EXCEPT select-statement [ORDER BY clause] [LIMIT clause];`
- `select-statement INTERSECT select-statement [ORDER BY clause] [LIMIT clause];`



UNION, EXCEPT, and INTERSECT are collectively called «set operators» or «table operators». In particular:

- `a UNION b` means «take rows which occur in a OR b».
- `a EXCEPT b` means «take rows which occur in a AND NOT b».
- `a INTERSECT b` means «take rows which occur in a AND b».

Duplicate rows are eliminated unless ALL is specified.

The *select-statements* may be chained: `SELECT ... SELECT ... SELECT ...;`

Each *select-statement* must result in the same number of columns.

The *select-statements* may be replaced with VALUES statements.

The maximum number of set operations is 50.

Пример:

```
CREATE TABLE t1 (s1 INT PRIMARY KEY, s2 VARCHAR(1));
CREATE TABLE t2 (s1 INT PRIMARY KEY, s2 VARCHAR(1));
INSERT INTO t1 VALUES (1,'A'),(2,'B'),(3,NULL);
INSERT INTO t2 VALUES (1,'A'),(2,'C'),(3,NULL);
SELECT s2 FROM t1 UNION SELECT s2 FROM t2;
SELECT s2 FROM t1 UNION ALL SELECT s2 FROM t2 ORDER BY s2;
SELECT s2 FROM t1 EXCEPT SELECT s2 FROM t2;
SELECT s2 FROM t1 INTERSECT SELECT s2 FROM t2;
```

В данном примере:

- The UNION query returns 4 rows: NULL, „A“, „B“, „C“.
- The UNION ALL query returns 6 rows: NULL, NULL, „A“, „A“, „B“, „C“.
- The EXCEPT query returns 1 row: „B“.

- The INTERSECT query returns 2 rows: NULL, „A“.

Limitations:

- Parentheses are not allowed.
- Evaluation is left to right, INTERSECT does not have precedence.

Пример:

```
CREATE TABLE t01 (s1 INT PRIMARY KEY, s2 VARCHAR(1));
CREATE TABLE t02 (s1 INT PRIMARY KEY, s2 VARCHAR(1));
CREATE TABLE t03 (s1 INT PRIMARY KEY, s2 VARCHAR(1));
INSERT INTO t01 VALUES (1,'A');
INSERT INTO t02 VALUES (1,'B');
INSERT INTO t03 VALUES (1,'A');
SELECT s2 FROM t01 INTERSECT SELECT s2 FROM t03 UNION SELECT s2 FROM t02;
SELECT s2 FROM t03 UNION SELECT s2 FROM t02 INTERSECT SELECT s2 FROM t03;
-- ... results are different.
```

INDEXED BY clause

Syntax:

INDEXED BY *index-name*

The INDEXED BY clause may be used in a SELECT, DELETE, or UPDATE statement, immediately after the *table-name*. For example:

```
DELETE FROM table7 INDEXED BY index7 WHERE column1 = 'a';
```

In this case the search for „a“ will take place within `index7`. For example:

```
SELECT * FROM table7 NOT INDEXED WHERE column1 = 'a';
```

In this case the search for „a“ will be done via a search of the whole table, what is sometimes called a «full table scan», even if there is an index for `column1`.

Ordinarily Tarantool chooses the appropriate index or lookup method depending on a complex set of «optimizer» rules; the INDEXED BY clause overrides the optimizer choice.

Пример:

Suppose a table has two columns:

- The first column is the primary key and therefore it has an automatic index named `pk_unnamed_T_1`.
- The second column has an index created by the user.

The user selects with `INDEXED BY the-index-on-column1`, then selects with `INDEXED BY the-index-on-column-2`.

```
CREATE TABLE t (column1 INT PRIMARY KEY, column2 INT);
CREATE INDEX i ON t (column2);
INSERT INTO t VALUES (1,2),(2,1);
SELECT * FROM t INDEXED BY "pk_unnamed_T_1";
```

(continues on next page)

(продолжение с предыдущей страницы)

```
SELECT * FROM t INDEXED BY i;
-- Result for the first select: (1,2),(2,1)
-- Result for the second select: (2,1),(1,2).
```

Transactions

START TRANSACTION

Syntax:

```
START TRANSACTION;
```

Start a transaction. After `START TRANSACTION;`, a transaction is «active». If a transaction is already active, then `START TRANSACTION;` is illegal.

Transactions should be active for fairly short periods of time, to avoid concurrency issues. To end a transaction, say `COMMIT;` or `ROLLBACK;`.

Just like in NoSQL, transaction control statements are subject to limitations set by the storage engine involved:

- For memtx storage engine, if a yield happens within an active transaction, the transaction is rolled back.
- For vinyl engine, yields are allowed.

However,transaction control statements still may not work as you expect when run over a network connection: a transaction is associated with a fiber, not a network connection, and different transaction control statements sent via the same network connection may be executed by different fibers from the fiber pool.

In order to ensure that all statements are part of the intended transaction, put all of them between `START TRANSACTION;` and `COMMIT;` or `ROLLBACK;` then send as a single batch. For example:

- Enclose each separate SQL statement in a *box.execute()* function.
- Pass all the `box.execute()` functions to the server in a single message.

  If you are using a console, you can do this by writing everything on a single line.

  If you are using *net.box*, you can do this by putting all the function calls in a single string and calling *eval(string)*.

Пример:

```
START TRANSACTION;
```

Example of a whole transaction sent to a server on `localhost:3301` with `eval(string)`:

```
net_box = require('net.box')
conn = net_box.new('localhost', 3301)
s = 'box.execute([[START TRANSACTION; ]]) '
s = s .. 'box.execute([[INSERT INTO t VALUES (1); ]]) '
s = s .. 'box.execute([[ROLLBACK; ]]) '
conn:eval(s)
```

COMMIT

Syntax:

```
COMMIT;
```

Commit an active transaction, so all changes are made permanent and the transaction ends.

COMMIT is illegal unless a transaction is active. If a transaction is not active then SQL statements are committed automatically.

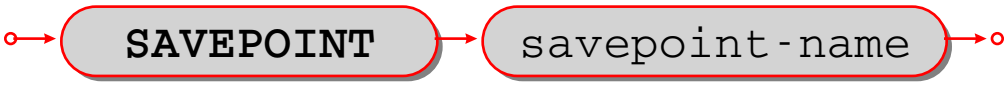
Пример:

```
COMMIT;
```

SAVEPOINT

Syntax:

```
SAVEPOINT savepoint-name;
```



Set a savepoint, so that ROLLBACK TO *savepoint-name* is possible.

SAVEPOINT is illegal unless a transaction is active.

If a savepoint with the same name already exists, it is released before the new savepoint is set.

Пример:

```
SAVEPOINT x;
```

RELEASE SAVEPOINT

Syntax:

```
RELEASE SAVEPOINT savepoint-name;
```

Release (destroy) a savepoint created by SAVEPOINT statement.

RELEASE is illegal unless a transaction is active.

Savepoints are released automatically when a transaction ends.

Пример:

```
RELEASE SAVEPOINT x;
```

ROLLBACK

Syntax:

```
ROLLBACK [TO [SAVEPOINT] savepoint-name];
```

If ROLLBACK does not specify a *savepoint-name*, rollback an active transaction, so all changes since START TRANSACTION are cancelled, and the transaction ends.

If ROLLBACK does specify a *savepoint-name*, rollback an active transaction, so all changes since *savepoint-name* are cancelled, and the transaction does not end.

ROLLBACK is illegal unless a transaction is active.

Examples:

```
-- the simple form:
ROLLBACK;
-- the form so changes before a savepoint are not cancelled:
ROLLBACK TO SAVEPOINT x;
```

```
-- An example of a Lua function that will do a transaction
-- containing savepoint and rollback to savepoint.
function f()
box.execute([[DROP TABLE IF EXISTS t;]]) -- commits automatically
box.execute([[CREATE TABLE t (s1 VARCHAR(20) PRIMARY KEY);]]) -- commits automatically
box.execute([[START TRANSACTION;]]) -- after this succeeds, a transaction is active
box.execute([[INSERT INTO t VALUES ('Data change #1');]])
box.execute([[SAVEPOINT "1";]])
box.execute([[INSERT INTO t VALUES ('Data change #2');]])
box.execute([[ROLLBACK TO SAVEPOINT "1";]]) -- rollback Data change #2
box.execute([[ROLLBACK TO SAVEPOINt "1";]]) -- this is legal but does nothing
box.execute([[COMMIT;]]) -- make Data change #1 permanent, end the transaction
end
```

Functions

Syntax:

```
function-name (one or more expressions)
```

Apply a built-in function to one or more expressions and return a scalar value.

Tarantool supports 32 built-in functions.

CHAR

Syntax:

```
CHAR([numeric-expression [,numeric-expression...])
```

Return the characters whose Unicode code point values are equal to the numeric expressions.

Short example:

The first 128 Unicode characters are the «ASCII» characters, so CHAR(65,66,67) is „ABC“.

Long example:

For the current list of Unicode characters, in order by code point, see www.unicode.org/Public/UCD/latest/ucd/UnicodeData.txt. In that list, there is a line for a Linear B ideogram

```
100CC;LINEAR B IDEOGRAM B240 WHEELED CHARIOT ...
```

Therefore, for a string with a chariot in the middle, use the concatenation operator `||` and the CHAR function

```
'start of string ' || CHAR(0X100CC) || ' end of string'.
```

COALESCE

Syntax:

```
COALESCE(expression, expression [, expression ...])
```

Return the value of the first non-NULL expression, or, if all expression values are NULL, return NULL.

Пример: `COALESCE(NULL, 17, 32)` is 17.

HEX

Syntax:

```
HEX(expression)
```

Return the hexadecimal code for each byte in **expression**, which may be either a string or a byte sequence. For ASCII characters, this is straightforward because the encoding is the same as the code point value. For non-ASCII characters, since character strings are usually encoded in UTF-8, each character will require two or more bytes.

Examples:

- `HEX('A')` will return `41`.
- `HEX('Д')` will return `D094`.

IFNULL

Syntax:

```
IFNULL(expression, expression)
```

Return the value of the first non-NULL expression, or, if both expression values are NULL, return NULL. Thus `IFNULL(expression, expression)` is the same as *COALESCE(expression, expression)*.

Пример: `IFNULL(NULL, 17)` is 17

LENGTH

Syntax:

```
LENGTH(expression)
```

Return the number of characters in the **expression**, or the number of bytes in the **expression**. It depends on the data type: strings with data type STRING are counted in characters, byte sequences with data type VARBINARY are counted in bytes and are not ended by the nul character. There are two aliases for `LENGTH(expression)` – `CHAR_LENGTH(expression)` and `CHARACTER_LENGTH(expression)` do the same thing.

Examples:

- `LENGTH('ДД')` is 2, the string has 2 characters.
- `LENGTH(CAST('ДД' AS VARBINARY))` is 4, the string has 4 bytes.
- `LENGTH(CHAR(0,65))` is 2, „0“ does not mean „end of string“.
- `LENGTH(X'410041')` is 3, X“...“ byte sequences have type VARBINARY.

LOWER

Syntax:

```
LOWER(string-expression)
```

Return the expression, with upper-case characters converted to lower case. This is the reverse of *UPPER(string-expression)*.

Example: `LOWER('-4ЩL')` is „-4щl“.

NULLIF

Syntax:

```
NULLIF(expression-1, expression-2)
```

Return *expression-1* if *expression-1* <> *expression-2*, otherwise return NULL.

Examples:

- `NULLIF('a','A')` is „a“.
- `NULLIF(1.00, 1)` is NULL.

PRINTF

Syntax:

```
PRINTF(string-expression [, expression ...])
```

Return a string formatted according to the rules of the C `sprintf()` function, where `%d%s` means the next two arguments are a number and a string, etc.

If an argument is missing or is NULL, it becomes:

- „0“ if the format requires an integer,
- „0.0“ if the format requires a decimal number,
- „“ if the format requires a string.

Example: `PRINTF('%da', 5)` is „5a“.

QUOTE

Syntax:

`QUOTE(string-literal)`

Return a string with enclosing quotes if necessary, and with quotes inside the enclosing quotes if necessary. This function is useful for creating strings which are part of SQL statements, because of SQL's rules that string literals are enclosed by single quotes, and single quotes inside such strings are shown as two single quotes in a row.

Example: `QUOTE('a')` is `'a'`.

SOUNDEX

Syntax:

`SOUNDEX(string-expression)`

Return a four-character string which represents the sound of `string-expression`. Often words and names which have different spellings will have the same Soundex representation if they are pronounced similarly, so it is possible to search by what they sound like. The algorithm works with characters in the Latin alphabet and works best with English words.

Example: `SOUNDEX('Crater')` and `SOUNDEX('Creature')` both return `C636`.

UNICODE

Syntax:

`UNICODE(string-expression)`

Return the Unicode code point value of the first character of **string-expression**. If *string-expression* is empty, the return is NULL. This is the reverse of *CHAR(integer)*.

Example: `UNICODE('Щ')` is 1065 (hexadecimal 0429).

UPPER

Syntax:

`UPPER(string-expression)`

Return the expression, with lower-case characters converted to upper case. This is the reverse of *LOWER(string-expression)*.

Example: `UPPER('-4щl')` is „-4ЩL“.

VERSION

Syntax:

`VERSION()`

Return the Tarantool version.

Example: for a March 2019 build VERSION() is `2.1.1-374-g27283debc`.

5.1.2 SQL features

In this section we will go through SQL:2016's «Feature taxonomy and definition for mandatory features».

For each feature in that list, we will come up with a simple example SQL statement. If Tarantool appears to handle the example, we will mark it «Okay», else we will mark it «Fail». Since this is rough and arbitrary, we believe that tests which are unfairly marked «Okay» will probably be balanced by tests which are unfairly marked «Fail».

Feature ID	Feature	Example	Test
E011	Numeric data types		
E011-01	INTEGER and SMALLINT	`create table t (s1 int primary key);`	Okay.
E011-02	REAL, DOUBLE PRECISION, and FLOAT data types	`create table tr (s1 float primary key);`	Okay. **Note:** Floating point SQL types are not planned to be compatible between 2.1 and 2.2 releases. The reason is that in 2.1 we set „number“ format for columns of these types, but will restrict it to „float32“ and „float64“ in 2.2. The format change requires data migration and cannot be done automatically, because in 2.1 we have no information to distinguish „number“ columns (created from Lua) from FLOAT/DOUBLE/REAL ones (created from SQL).
E011-03	DECIMAL and NUMERIC data types	`create table td (s1 numeric primary key);`	Fail, DECIMAL and NUMERIC data types are not supported and a number containing post-decimal digits will be treated as approximate numeric.
E011-04	Arithmetic operators	`select 10+1,9-2, 8*3,7/2 from t;`	Okay.
E011-05	Numeric comparisons	`select * from t where 1 < 2;`	Okay.
E011-06	Implicit casting among the numeric data types	`select * from t where s1 = 1.00;`	Okay, but only because Tarantool doesn't distinguish between numeric data types.
E021	Character string types		

Продолжается на следующей странице

Таблица 1 – продолжение с предыдущей страницы

Feature ID	Feature	Example	Test
E021-01	Character data type (including all its spellings)	`create table t44 (s1 char primary key);`	Fail, CHAR is not supported. This type of Fail will only be counted once.
E021-02	CHARACTER VARYING data type (including all its spellings)	`create table t45 (s1 varchar primary key);`	Fail, only the spelling VARCHAR is allowed. **Note:** VARCHAR(N) does not check the string length.
E021-03	Character literals	`insert into t45 values ('');`	Okay, and the bad practice of accepting ««„s for character literals is avoided.
E021-04	*CHARACTER_LENGTH* function	`select character_length(s1) from t;`	Fail. There is no such function. There is a function LENGTH(), which is okay.
E021-05	OCTET_LENGTH	`select octet_length(s1) from t;`	Fail. There is no such function.
E021-06	SUBSTRING function.	`select substring(s1 from 1 for 1) from t;`	Fail. There is no such function. There is a function SUBSTR(x,n,n) which is okay.
E021-07	Character concatenation	`select 'a' || 'b' from t;`	Okay.
E021-08	*UPPER* and *LOWER* functions	`select upper('a'), lower('B') from t;`	Okay. SUBSTR(x,n,n) which is okay.
E021-09	TRIM function	`select trim('a ') from t;`	Okay.
E021-10	Implicit casting among the fixed-length and variable-length character string types	`select * from tm where char_column > varchar_column;`	Fail, there is no fixed-length character string type.
E021-11	POSITION function	`select position(x in y) from z;`	Fail. Tarantool's function uses „,“ rather than „in“
E021-12	Character comparison	`select * from t where s1 > 'a';`	Okay. We should note here that comparisons use a binary collation by default, but it is easy to specify unicode or unicode_ci collations, or create new collations.
E031	Identifiers	`create table rank (ceil int primary key);`	Fail. Tarantool's list of reserved words differs from the standard's list of reserved words.

Продолжается на следующей странице

Таблица 1 – продолжение с предыдущей страницы

Feature ID	Feature	Example	Test
E031-01	Delimited Identifiers	`create table "t47" (s1 int primary key);`	Okay. And enclosing identifiers inside double quotes means they won't be converted to upper case or lower case, this is behavior that some other DBMSs sadly lack.
E031-02	Lower case identifiers	`create table t48 (s1 int primary key);`	Okay.
E031-03	Trailing underscore	`create table t49_ (s1 int primary key);`	Okay.
E051	Basic query specification		
E051-01	SELECT DISTINCT	`select distinct s1 from t;`	Okay.
E051-02	*GROUP BY* clause	`select distinct s1 from t group by s1;`	Okay.
E051-04	GROUP BY can contain columns not in select list	`select s1 from t group by lower(s1);`	Okay.
E051-05	Select list items can be renamed	`select s1 as K from t order by K;`	Okay.
E051-06	*HAVING* clause	`select count(*) from t having count(*) > 0;`	Okay. GROUP BY is not mandatory before HAVING.
E051-07	Qualified * in select list	`select t.* from t;`	Okay.
E051-08	Correlation names in the FROM clause	`select * from t as K;`	Okay.
E051-09	Rename columns in the FROM clause	`select * from t as x(q,c);`	Fail.
E061	Basic predicates and search conditions		
E061-01	Comparison predicate	`select * from t where 0 = 0;`	Okay.
E061-02	BETWEEN predicate	`select * from t where ' ' between '' and ' ';`	Okay.
E061-03	IN predicate with list of values	`select * from t where s1 in ('a', upper('a'));`	Okay.
E061-04	LIKE predicate	`select * from t where s1 like '_';`	Okay.
E061-05	LIKE predicate: ESCAPE clause	`VALUES ('abc_' LIKE 'abcX_' ESCAPE 'X');`	Okay.
E061-06	NULL predicate	`select * from t where s1 is not null;`	Okay.

Продолжается на следующей странице

Таблица 1 – продолжение с предыдущей страницы

Feature ID	Feature	Example	Test
E061-07	Quantified comparison predicate	`select * from t where s1 = any (select s1 from t);`	Fail. Syntax error.
E061-08	EXISTS predicate	`select * from t where not exists (select * from t);`	Okay.
E061-09	Subqueries in comparison predicate	`select * from t where s1 > (select s1 from t);`	Okay.
E061-11	Subqueries in IN predicate	`select * from t where s1 in (select s1 from t);`	Okay.
E061-12	Subqueries in quantified comparison predicate	`select * from t where s1 >= all (select s1 from t);`	Fail. Syntax error.
E061-13	Correlated subqueries	`select * from t where s1 = (select s1 from t2 where t2.s2 = t.s1);`	Okay.
E061-14	Search condition	`select * from t where 0 <> 0 or 'a' < 'b' and s1 is null;`	Okay.
E071	Basic query expressions		
E071-01	UNION DISTINCT table operator	`select * from t union distinct select * from t;`	Fail. However, «select * from t union select * from t;» is okay.
E071-02	UNION ALL table operator	`select * from t union all select * from t;`	Okay.
E071-03	EXCEPT DISTINCT table operator	`select * from t except distinct select * from t;`	Fail. However, `select * from t except select * from t;` is okay.
E071-05	Columns combined via table operators need not have exactly the same data type.	`select s1 from t union select 5 from t;`	Okay, but only because Tarantool doesn't distinguish data types very well.
E071-06	Table operators in subqueries	`select * from t where 'a' in (select * from t union select * from t);`	Okay.
E081	Basic privileges		
E081-01	Select privilege at the table level		Fail. Syntax error. (Tarantool doesn't support privileges.)

Продолжается на следующей странице

Таблица 1 – продолжение с предыдущей страницы

Feature ID	Feature	Example	Test
E081-02	DELETE privilege		Fail. (Tarantool doesn't support privileges.)
E081-03	INSERT privilege at the table level		Fail. (Tarantool doesn't support privileges.)
E081-04	UPDATE privilege at the table level		Fail. (Tarantool doesn't support privileges.)
E081-05	UPDATE privilege at column level		Fail. (Tarantool doesn't support privileges.)
E081-06	REFERENCES privilege at the table level		Fail. (Tarantool doesn't support privileges.)
E081-07	REFERENCES privilege at column level		Fail. (Tarantool doesn't support privileges.)
E081-08	WITH GRANT OPTION		Fail. (Tarantool doesn't support privileges.)
E081-09	USAGE privilege		Fail. (Tarantool doesn't support privileges.)
E081-10	EXECUTE privilege		Fail. (Tarantool doesn't support privileges.)
E091	Set functions		
E091-01	*AVG*	`select avg(s1) from t7;`	Fail. No warning that nulls were eliminated.
E091-02	*COUNT*	`select count(*) from t7 where s1 > 0;`	Okay.
E091-03	*MAX*	`select max(s1) from t7 where s1 > 0;`	Okay.
E091-04	*MIN*	`select min(s1) from t7 where s1 > 0;`	Okay.
E091-05	*SUM*	`select sum(1) from t7 where s1 > 0;`	Okay.
E091-06	ALL quantifier	`select sum(all s1) from t7 where s1 > 0;`	Okay.
E091-07	DISTINCT quantifier	`select sum(distinct s1) from t7 where s1 > 0;`	Okay.
E101	Basic data manipulation		
E101-01	INSERT statement	`insert into t (s1, s2) values (1,''), (2,null),(3,55);`	Okay.
E101-03	Searched UPDATE statement	`update t set s1 = null where s1 in (select s1 from t2);`	Okay.
E101-04	Searched DELETE statement	`delete from t where s1 in (select s1 from t);`	Okay.

Продолжается на следующей странице

Таблица 1 – продолжение с предыдущей страницы

Feature ID	Feature	Example	Test
E111	Single row SELECT statement	`select count(*) from t;`	Okay.
E121	Basic cursor support		
E121-01	DECLARE CURSOR		Fail. Tarantool doesn't support cursors.
E121-02	ORDER BY columns need not be in select list	`select s1 from t order by s2;`	Okay.
E121-03	Value expressions in select list	`select s1 from t7 order by -s1;`	Okay.
E121-04	OPEN statement		Fail. Tarantool doesn't support cursors.
E121-06	Positioned UPDATE statement		Fail. Tarantool doesn't support cursors.
E121-07	Positioned DELETE statement		Fail. Tarantool doesn't support cursors.
E121-08	CLOSE statement		Fail. Tarantool doesn't support cursors.
E121-10	FETCH statement implicit next		Fail. Tarantool doesn't support cursors.
E121-17	WITH HOLD cursors		Fail. Tarantool doesn't support cursors.
E131	Null value support (nulls in lieu of values)	`select s1 from t7 where s1 is null;`	Okay.
E141	Basic integrity constraints		
E141-01	NOT NULL constraints	`create table t8 (s1 int primary key, s2 int not null);`	Okay.
E141-02	UNIQUE constraints of NOT NULL columns	`create table t9 (s1 int primary key , s2 int not null unique);`	Okay.
E141-03	PRIMARY KEY constraints	`create table t10 (s1 int primary key);`	Okay, although Tarantool shouldn't always insist on having a primary key.
E141-04	Basic FOREIGN KEY constraint with the NO ACTION default for both referential delete action and referential update action.	`create table t11 (s0 int primary key, s1 int references t10);`	Okay.
E141-06	CHECK constraints	`create table t12 (s1 int primary key, s2 int, check (s1 = s2));`	Okay.
E141-07	Column defaults	`create table t13 (s1 int primary key, s2 int default -1);`	Okay.

Продолжается на следующей странице

Таблица 1 – продолжение с предыдущей страницы

Feature ID	Feature	Example	Test
E141-08	NOT NULL inferred on primary key	`create table t14 (s1 int primary key);`	Okay. We are unable to insert NULL although we don't explicitly say the column is NOT NULL.
E141-10	Names in a foreign key can be specified in any order	`create table t15 (s1 int, s2 int, primary key (s1, s2)); create table t16 (s1 int primary key, s2 int, foreign key (s2,s1) references t15 (s1,s2));`	Okay.
E151	Transaction support		
E151-01	COMMIT statement	`commit;`	Fail. We have to say START TRANSACTION first.
E151-02	ROLLBACK statement	`rollback;`	Okay.
E152	Basic SET TRANSACTION statement		
E152-01	SET TRANSACTION statement ISOLATION SERIALIZABLE clause	`set transaction isolation level serializable;`	Fail. Syntax error.
E152-02	SET TRANSACTION statement READ ONLY and READ WRITE clauses	`set transaction read only;`	Fail. Syntax error.
E153	Updatable queries with subqueries		
E161	SQL comments using leading double minus	`--comment;`	Okay.
E171	SQLSTATE support	`drop table no_such_table;`	Fail. At least, the error message doesn't hint that SQLSTATE exists.
E182	Host language binding		Okay. Any of the Tarantool connectors should be able to call `box.execute()`.
F031	Basic schema manipulation		
F031-01	CREATE TABLE statement to create persistent base tables	`create table t20 (t20_1 int not null);`	Fail. We always have to say PRIMARY KEY (we only count this flaw once).
F031-02	CREATE VIEW statement	`create view t21 as select * from t20;`	Okay.
F031-03	GRANT statement		Fail. Tarantool doesn't support privileges except via NoSQL.
F031-04	ALTER TABLE statement: add column	`alter table t7 add column t7_2 varchar default 'q';`	Fail. Table alterations work but not this clause.

Продолжается на следующей странице

Таблица 1 – продолжение с предыдущей страницы

Feature ID	Feature	Example	Test
F031-13	DROP TABLE statement: RESTRICT clause	`drop table t20 restrict;`	Fail. Syntax error, and RESTRICT is not assumed.
F031-16	DROP VIEW statement: RESTRICT clause	`drop view v2 restrict;`	Fail. Syntax error, and RESTRICT is not assumed.
F031-19	REVOKE statement: RESTRICT clause		Fail. Tarantool does not support privileges except via NoSQL.
F041	Basic joined table		
F041-01	Inner join but not necessarily the INNER keyword	`select a.s1 from t7 a join t7 b;`	Okay.
F041-02	INNER keyword	`select a.s1 from t7 a inner join t7 b;`	Okay.
F041-03	LEFT OUTER JOIN	`select t7.*,t22. * from t22 left outer join t7 on (t22_1=s1);`	Okay.
F041-04	RIGHT OUTER JOIN	`select t7.*,t22. * from t22 right outer join t7 on (t22_1=s1);`	Fail. Syntax error.
F041-05	Outer joins can be nested	`select t7.*,t22. * from t22 left outer join t7 on (t22_1=s1) left outer join t23;.`	Okay.
F041-07	The inner table in a left or right outer join can also be used in an inner join	`select t7.* from (t22 left outer join t7 on (t22_1=s1)) j inner join t22 on (j. t22_4=t7.s1);`	Okay.
F041-08	All comparison operators are supported	`select * from t where 0=1 or 0>1 or 0<1 or 0<>1;`	Okay.
F051 Basic date and time			
F051-01	DATE data type (including support of DATE literal)	`create table dates (s1 date);`	Fail. Tarantool does not support DATE data type.
F051-02	TIME data type (including support of TIME literal)	`create table times (s1 time default time '1:2:3');`	Fail. Syntax error.
F051-03	TIMESTAMP data type (including support of TIMESTAMP literal)	`create table timestamps (s1 timestamp);`	Fail. Syntax error.

Продолжается на следующей странице

Таблица 1 – продолжение с предыдущей страницы

Feature ID	Feature	Example	Test
F051-04	Comparison predicate on DATE, TIME and TIMESTAMP data types	`select * from dates where s1 = s1;`	Fail. The data types are not supported.
F051-05	Explicit CAST between date-time types and character string types	`select cast(s1 as varchar(10)) from dates;`	Fail. The data types are not supported.
F051-06	CURRENT_DATE	`select current_date from t;`	Fail. Syntax error.
F051-07	LOCALTIME	`select localtime from t;`	Fail. Syntax error.
F051-08	LOCALTIMESTAMP	`select localtimestamp from t;`	Fail. Syntax error.
F081	UNION and EXCEPT in views	`create view vv as select * from t7 except select * from t15;`	Okay.
F131	Grouped operations		
F131-01	WHERE, GROUP BY, and HAVING clauses supported in queries with grouped views	`create view vv2 as select * from vv group by s1;`	Okay.
F131-02	Multiple tables supported in queries with grouped views	`create view vv3 as select * from vv2,t30;`	Okay.
F131-03	Set functions supported in queries with grouped views	`create view vv4 as select count(*) from vv2;`	Okay.
F131-04	Subqueries with GROUP BY and HAVING clauses and grouped views	`create view vv5 as select count(*) from vv2 group by s1 having count(*) > 0;`	Okay.
F131-05	Single row SELECT with GROUP BY and HAVING clauses and grouped views	`select count(*) from vv2 group by s1 having count(*) > 0;`	Okay.
F181	Multiple module support		Fail. Tarantool doesn't have modules.
F201	CAST function	`select cast(s1 as int) from t;`	Okay.
F221	Explicit defaults	`update t set s1 = default;`	Fail. Syntax error.
F261	CASE expression		
F261-01	Simple CASE	`select case when 1 = 0 then 5 else 7 end from t;`	Okay.

Продолжается на следующей странице

Таблица 1 – продолжение с предыдущей страницы

Feature ID	Feature	Example	Test
F261-02	Searched CASE	`select case 1 when 0 then 5 else 7 end from t;`	Okay.
F261-03	NULLIF	`select nullif(s1,7) from t;`	Okay.
F261-04	COALESCE	`select coalesce(s1, 7) from t;`	Okay.
F311	Schema definition statement		
F311-01	CREATE SCHEMA		Fail. Tarantool doesn't have schemas or databases.
F311-02	CREATE TABLE for persistent base tables		Fail. Tarantool doesn't have CREATE TABLE inside CREATE SCHEMA.
F311-03	CREATE VIEW		Fail. Tarantool doesn't have CREATE VIEW inside CREATE SCHEMA.
F311-04	CREATE VIEW: WITH CHECK OPTION		Fail. Tarantool doesn't have CREATE VIEW inside CREATE SCHEMA.
F311-05	GRANT statement		Fail. Tarantool doesn't have GRANT inside CREATE SCHEMA.
F471	Scalar subquery values	`select s1 from t where s1 = (select count(*) from t);`	Okay.
F481	Expanded NULL Predicate	`select * from t where row(s1,s1) is not null;`	Fail. Syntax error.
F812	Basic flagging		Fail. Tarantool doesn't support any flagging.
S011	Distinct types	`create type x as float;`	Fail. Tarantool doesn't support distinct types.
T321	Basic SQL-invoked routines		
T321-01	User-defined functions with no overloading	`create function f () returns int return 5;`	Fail. Tarantool doesn't support user-defined SQL functions.
T321-02	User-defined procedures with no overloading	`create procedure p () begin end;`	Fail. Tarantool doesn't support user-defined procedures.
T321-03	Function invocation	`select f(1) from t;`	Okay. Tarantool can invoke Lua user-defined functions.
T321-04	CALL statement.	`call p();`	Fail. Tarantool doesn't support user-defined procedures.

Продолжается на следующей странице

Таблица 1 – продолжение с предыдущей страницы

Feature ID	Feature	Example	Test
T321-05	RETURN statement.	`create function f() returns int return 5;`	Fail. Tarantool doesn't support user-defined functions.
T631	IN predicate with one list element	`select * from t where 1 in (1);`	Okay.
F021	Basic information schema	`select * from information_schema.tables;`	Fail. There is no schema with that name (not counted in the final score).

Total number of items marked «Fail»: 68

Total number of items marked «Okay»: 78

5.2 Справочник по встроенным модулям

В данном справочнике рассматриваются встроенные Lua-модули Tarantool'a.

Примечание: Некоторые функции в данных модулях представляют собой аналоги функций из стандартных Lua-библиотек. Для достижения наилучшего результата мы рекомендуем использовать функции из встроенных модулей Tarantool'a.

5.2.1 Модуль *box*

Помимо выполнения фрагментов кода на Lua или определения собственных функций, с помощью модуля `box` и вложенных модулей можно использовать функции хранилища Tarantool'a.

Каждый вложенный модуль включает в себя одну или более Lua-функций. Несколько вложенных модулей включают в себя элементы класса, а также функции. Функции обеспечивают определение данных (create alter drop), управление данными (insert delete update upsert select replace) и просмотр состояния (просмотр содержимого спейсов, получение доступа к конфигурации сервера).

Чтобы найти ошибки, которые могут выдать вложенные модули `box`, используйте *pcall*.

Содержимое модуля `box` можно просмотреть во время исполнения кода с помощью команды `box` без аргументов. Модуль `box` включает в себя следующее:

Вложенный модуль *box.backup*

Модуль `box.backup` содержит две функции, которые помогают при работе с *резервированным копированием*.

`backup.start(`$[n]$`)`

Оповещает сервер о том, что следует приостановить все активности, связанные с удалением устаревших резервных копий.

Чтобы гарантировать возможность скопировать эти файлы, Tarantool не станет их удалять. При этом он не переходит в режим read-only, и создание контрольных точек делается по расписанию, как обычно.

Параметры

- `n` (*number*) – необязательный аргумент, начиная с Tarantool 1.10.1, который указывает нужную контрольную точку относительно самой свежей точки. Например, `n = 0` означает “резервная копия будет создана на основе самой свежей контрольной точки”, `n = 1` означает «резервная копия будет создана на основе контрольной точки, которая была создана перед самой свежей точкой», и т.д. По умолчанию n = 0.

Возвращает: таблицу с именами снапшотов и файлов vinyl, которые нужно скопировать

Пример:

```
tarantool> box.backup.start()
---
- - ./00000000000000000015.snap
  - ./00000000000000000000.vylog
  - ./513/0/00000000000000000002.index
  - ./513/0/00000000000000000002.run
...
```

`backup.stop()`
оповещает сервер о том, что можно вернуться к работе в обычном режиме.

Вложенный модуль *box.cfg*

Вложенный модуль `box.cfg` предназначен для того, чтобы задавать *параметры конфигурации сервера*.

Для просмотра текущей конфигурации введите команду `box.cfg` без фигурных скобок:

```
tarantool> box.cfg
 ---
 - checkpoint_count: 2
   too_long_threshold: 0.5
   slab_alloc_factor: 1.1
   memtx_max_tuple_size: 1048576
   background: false
   <...>
 ...
```

Чтобы задать определенные параметры, используйте следующий синтаксис: `box.cfg{ключ = значение [, ключ = значение ...]}` (для краткости далее по тексту `box.cfg{...}`). Например:

```
tarantool> box.cfg{listen = 3301}
```

Если для параметров явным образом не указаны значения при вызове `box.cfg{...}`, они получают *значения по умолчанию*.

Если ввести `box.cfg{}` без параметров, Tarantool применит следующие настройки ко всем параметрам по умолчанию:

```
tarantool> box.cfg{}
tarantool> box.cfg -- sorted in the alphabetic order
---
- background                  = false
  checkpoint_count            = 2
  checkpoint_interval         = 3600
  checkpoint_wal_threshold    = 1000000000000000000
```

(continues on next page)

(продолжение с предыдущей страницы)

```
coredump                    = false
custom_proc_title           = nil
feedback_enabled            = true
feedback_host               = 'https://feedback.tarantool.io'
feedback_interval           = 3600
force_recovery              = false
hot_standby                 = false
io_collect_interval         = nil
listen                      = nil
log                         = nil
log_format                  = plain
log_level                   = 5
log_nonblock                = true
memtx_dir                   = '.'
memtx_max_tuple_size        = 1024 * 1024
memtx_memory                = 256 * 1024 *1024
memtx_min_tuple_size        = 16
net_msg_max                 = 768
pid_file                    = nil
readahead                   = 16320
read_only                   = false
replication                 = nil
replication_connect_quorum  = nil
replication_connect_timeout = 30
replication_skip_conflict   = false
replication_sync_lag        = 10
replication_sync_timeout    = 300
replication_timeout         = 1
slab_alloc_factor           = 1.05
snap_io_rate_limit          = nil
strip_core                  = true
too_long_threshold          = 0.5
username                    = nil
vinyl_bloom_fpr             = 0.05
vinyl_cache                 = 128 * 1024 * 1024
vinyl_dir                   = '.'
vinyl_max_tuple_size        = 1024 * 1024* 1024 * 1024
vinyl_memory                = 128 * 1024 * 1024
vinyl_page_size             = 8 * 1024
vinyl_range_size            = nil
vinyl_read_threads          = 1
vinyl_run_count_per_level   = 2
vinyl_run_size_ratio        = 3.5
vinyl_timeout               = 60
vinyl_write_threads         = 4
wal_dir                     = '.'
wal_dir_rescan_delay        = 2
wal_max_size                = 256 * 1024 * 1024
wal_mode                    = 'write'
worker_pool_threads         = 4
work_dir                    = nil
```

The first call to `box.cfg{...}` (with or without parameters) initiates Tarantool's database module *box*. Before Tarantool 2.0, you needed to call `box.cfg{...}` prior to performing any database operations. Now you can start working with the database outright, without calling `box.cfg{...}`. In this case, Tarantool initiates the database module and applies default settings, as if you said `box.cfg{}` (without parameters).

Команда `box.cfg{...}` также перезагружает *файлы с данными длительного хранения* в оперативную память при перезапуске после получения данных.

Вложенный модуль *box.ctl*

Вложенный модуль `box.ctl` включает в себя две функции: `wait_ro` (дождаться режима только для чтения) и `wait_rw` (дождаться режима чтения и записи). Эти функции используются во время инициализации сервера.

Для *box_ once()* есть особое предназначение. Например, при инициализации реплика может вызвать функцию `box.once()`, пока сервер все еще находится в режиме только для чтения, и не сможет применить изменения однократно до окончательной инициализации реплики. Это может привести к конфликту между мастером и репликой, если мастер находится в режиме чтения и записи, а реплика доступна только для чтения. Ожидание условия «read only mode = false» (режим только для чтения отключен) решает эту проблему.

Чтобы проверить режим функции – только для чтения или чтение и запись, используйте *box.info.ro*.

`box.ctl.wait_ro`([*timeout*])

Дождаться, пока не будет выполнено `box.info.ro`.

Параметры

- `timeout` (*number*) – максимальное количество секунд ожидания

возвращается нулевое значение nil или ошибка, которая может возникнуть из-за превышения времени ожидания или прерывания работы файбера

Пример:

```
tarantool> box.info().ro
---
- false
...

tarantool> n = box.ctl.wait_ro(0.1)
---
- error: timed out
...
```

`box.ctl.wait_rw`([*timeout*])

Дождаться, пока не перестанет соблюдаться box.info.ro.

Параметры

- `timeout` (*number*) – максимальное количество секунд ожидания

возвращается нулевое значение nil или ошибка, которая может возникнуть из-за превышения времени ожидания или прерывания работы файбера

Пример:

```
tarantool> box.ctl.wait_rw(0.1)
---
...
```

Встроенный модуль `box.ctl` также содержит две функции для определения двух *серверных триггеров*: `on_shutdown` и `on_schema_init`. Пожалуйста, ознакомьтесь с механизмом триггерных функций перед их использованием.

box.ctl.on_shutdown(*trigger-function*[, *old-trigger-function*])
Создать «*триггер* выключения». Триггер-функция будет выполняться всякий раз, когда происходит *os.exit()*, или когда сервер выключается после получения сигнала SIGTERM или SIGINT или SIGHUP (но не после сигнала SIGSEGV или SIGABORT или любого другого сигнала, вызывающего немедленное завершение программы).

Параметры

- `trigger-function` (*function*) – функция, в которой будет триггер
- `old-trigger-function` (*function*) – существующая функция с триггером, которую заменит новая

возвращается nil или указатель функции

Если указаны параметры (nil, old-trigger-function), старый триггер будет удален.

Подробная информация о характеристиках триггера находится в разделе *Триггеры*.

box.ctl.on_schema_init(*trigger-function*[, *old-trigger-function*])
Создать *триггер* `schema_init`. Функция триггера будет выполнена, когда *box.cfg{}* произойдет в первый раз. То есть триггер `schema_init` вызывается до начала конфигурирования и восстановления сервера, и поэтому `box.ctl.on_schema_init` должен быть вызван до вызова `box.cfg`.

Параметры

- `trigger-function` (*function*) – функция, в которой будет триггер
- `old-trigger-function` (*function*) – существующая функция с триггером, которую заменит новая

возвращается nil или указатель функции

Если указаны параметры (nil, old-trigger-function), старый триггер будет удален.

Обычно используется следующее: сделать триггерную функцию `schema_init`, которая создает триггерную функцию `before_replace` на системном спейсе. Таким образом, поскольку системные спейсы создаются при старте сервера, триггеры `before_replace` будут активированы для каждого кортежа в каждом системном спейсе. Например, такой триггер может изменить механизм хранения заданного спейса, или сделать заданный спейс *локальной репликой* во время загрузки реплики. Выполнение такого изменения после `box.cfg` не является надежным, поскольку другие подключения могут использовать базу данных до внесения изменения.

Подробная информация о характеристиках триггера находится в разделе *Триггеры*.

Пример:

Предположим, что до того, как сервер будет полностью готов к подключениям, вы хотите убедиться, что движком `space_name` является vinyl. Поэтому вы хотите сделать триггер, который будет активирован при вставке кортежа в системный спейс `_space`. В этом случае может получиться мастер, который имеет space-name с `engine='memtx'` и реплику, которая имеет space_name с `engine='vinyl'`, с тем же самым содержимым.

```
function function_for_before_replace(old, new)
  if old == nil and new ~= nil and new[3] == 'space_name' and new[4] ~= 'vinyl' then
    return new:update{{'=', 4, 'vinyl'}}
  end
end

box.ctl.on_schema_init(function()
  box.space._space:before_replace(function_for_before_replace)
end)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
box.cfg{replication='master_uri', ...}
```

Вложенный модуль *box.error*

Общие сведения

The `box.error` function is for raising an error. The difference between this function and Lua's built-in error function is that when the error reaches the client, its error code is preserved. In contrast, a Lua error would always be presented to the client as `ER_PROC_LUA`.

Указатель

Ниже приведен перечень всех функций модуля `box.error`.

Имя	Назначение
box.error()	Вызов ошибки
box.error.last()	Получение описания последней ошибки
box.error.clear()	Очистка записи об ошибках
box.error.new()	Создание ошибки без выдачи

`box.error`(*reason* = *string*[, *code* = *number*])
: При вызове с аргументом из Lua-таблицы значения параметров code и reason будут любыми по желанию пользователя. Результатом будут эти значения.

  Параметры

  - `reason` (`string`) – (строка) описание ошибки, задается пользователем
  - `code` (*integer*) – (целое число) числовой код ошибки, задается пользователем

`box.error()`
: При вызове без аргументов `box.error()` повторно вызывает последнюю ошибку.

`box.error`(*code*, *errtext*[, *errtext* ...])
: Моделирование ошибки запроса с текстом на основе одной из ошибок Tarantool'a, заданных в файле errcode.h в исходном дереве. Lua-постоянные, которые соответствуют этим ошибкам в Tarantool'е, определяются как элементы `box.error`, например `box.error.NO_SUCH_USER == 45`.

  Параметры

  - `code` (*number*) – номер предварительно заданной ошибки
  - `errtext(s)` (`string`) – часть сообщения, которое сопровождает ошибку

  Пример:

  сообщение `NO_SUCH_USER` = «`User '%s' is not found`» (пользователь не найден) – оно включает в себя компонент «`%s`», который будет заменен значением параметра errtext. Таким образом, вызов `box.error(box.error.NO_SUCH_USER, 'joe')` или `box.error(45, 'joe')` приведет к ошибке с сообщением «`User 'joe' is not found`» (пользователь „joe“ не найден).

  Исключение: то, что указано в номере errcode.

  Пример:

```
tarantool> box.error{code = 555, reason = 'Arbitrary message'}
---
- error: Arbitrary message
...
tarantool> box.error()
---
- error: Arbitrary message
...
tarantool> box.error(box.error.FUNCTION_ACCESS_DENIED, 'A', 'B', 'C')
---
- error: A access denied for user 'B' to function 'C'
...
```

box.error.last()
: Возвращает описание последней ошибки в виде Lua-таблицы с 5 элементами: «line» (число) – номер строки в исходном файле Tarantool'a, «code» (число) – номер ошибки, «type» (строка) – C++ класс ошибки, «message» (строка) – сообщение об ошибке, «file» (строка) – исходный файл Tarantool'a. Кроме того, если ошибка является системной (например, по причине ошибки в сокете или файловом вводы-выводе), может быть дополнительный шестой элемент: «errno» (число) стандартный номер ошибки на языке C.

  Тип возвращаемого значения: таблица

box.error.clear()
: Очистка записи об ошибках, то есть функции *box.error()* или *box.error.last()* не сработают.

  Пример:

```
tarantool> box.error{code = 555, reason = 'Arbitrary message'}
---
- error: Arbitrary message
...
tarantool> box.schema.space.create('#')
---
- error: Invalid identifier '#' (expected letters, digits or an underscore)
...
tarantool> box.error.last()
---
- line: 278
  code: 70
  type: ClientError
  message: Invalid identifier '#' (expected letters, digits or an underscore)
  file: /tmp/buildd/tarantool-1.7.0.252.g1654e31~precise/src/box/key_def.cc
...
tarantool> box.error.clear()
---
...
tarantool> box.error.last()
---
- null
...
```

box.error.new(*code*, *errtext*[, *errtext* ...])
: Создание ошибки без выдачи. Используется, когда необходимо сохранить информацию об ошибке для последующей выборки. Используются такие же параметры, как в *box.error()*, см. описание по ссылке.

  Параметры

- code (*number*) – номер предварительно заданной ошибки
- errtext(s) (string) – часть сообщения, которое сопровождает ошибку

Пример:

```
tarantool> e = box.error.new{code = 555, reason = 'Arbitrary message'}
---
...
tarantool> e:unpack()
---
- type: ClientError
  code: 555
  message: Arbitrary message
  trace:
  - file: '[string "e = box.error.new{code = 555, reason = ''Arbit..."]'
    line: 1
...
```

Вложенный модуль *box.index*

Общие сведения

Вложенный модуль box.index обеспечивает доступ к схемам индекса и ключам индекса в режиме только для чтения. Индексы хранятся в массиве box.space.*имя-спейса*.index в каждом спейсе. Они предоставляют API для упорядоченной итерации по кортежам. Этот API представляет собой прямую привязку к соответствующим методам объектов типа box.index в движке базы данных.

Индекс

Ниже приведен перечень всех функций и элементов модуля box.index.

Имя	Использование
index_object.unique	Флаг, если индекс уникальный – true
index_object.type	Тип индекса
index_object.parts	Массив полей с ключами индекса
index_object:pairs()	Подготовка к итерации
index_object:select()	Выбор одного или более кортежей по индексу
index_object:get()	Выбор кортежа по индексу
index_object:min()	Поиск минимального значения в индексе
index_object:max()	Поиск максимального значения в индексе
index_object:random()	Поиск случайного значения в индексе
index_object:count()	Подсчет кортежей с совпадающим значением ключа
index_object:update()	Обновление кортежа
index_object:delete()	Удаление кортежа по ключу
index_object:alter()	Изменение индекса
index_object:drop()	Удаление индекса
index_object:rename()	Переименование индекса
index_object:bsize()	Подсчет байтов для индекса
index_object:stat()	Получение статистических данных по индексу
index_object:compact()	Удаление неиспользуемого пространства индекса
index_object:user_defined()	Любая функция / метод, которые хочет добавить любой пользователь

object index_object

index_object.unique
Если индекс уникальный – true, если индекс не уникален – false.

тип возвращаемого значения boolean (логический)

index_object.type
Тип индекса: „TREE“ или „HASH“ или „BITSET“ или „RTREE“.

index_object.parts
Массив, описывающий поля индекса. Чтобы узнать больше о типах полей индекса, обращайтесь к *этой таблице*.

тип возвращаемого значения таблица

Пример:

```
tarantool> box.space.tester.index.primary
---
- unique: true
  parts:
  - type: unsigned
    is_nullable: false
    fieldno: 1
  id: 0
  space_id: 513
  name: primary
  type: TREE
...
```

index_object:pairs(*[key[, {iterator = iterator-type}]]*)
Поиск кортежа или набора кортежей по заданному индексу и итерация по одному кортежу за раз.

Параметр ***key*** (ключ) задает, что именно должно совпадать в индексе.

Примечание: ***key*** используется в поиске только первого совпадения. Не стоит ожидать, что все подобранные кортежи будут содержать этот ключ.

Параметр ***iterator*** (итератор) задает правило для совпадений и упорядочивания. Различные типы индексов поддерживают различные итераторы. Например, TREE-индекс поддерживает строгий порядок ключей и может вернуть все кортежи в порядке по возрастанию или по убыванию, начиная с указанного ключа. Однако другие типы индексов не поддерживают упорядочивание.

Чтобы понять логику возврата кортежей с помощью итератора, важно знать принципы работы подсистемы обработки транзакций в Tarantool'е. В итераторе Tarantool'а нет собственного постоянного вида просмотра. Наоборот, каждая процедура получает эксклюзивный доступ ко всем кортежам и спейсам до тех пор, пока не «переключится контекст», что может произойти по причине *неявной передачи управления* или в результате явного вызова функции *fiber.yield*. Когда поток выполнения возвращается к процедуре, передавшей управление, набор данных может уже значительно измениться. Итерация возобновляется после стадии передачи управления и не сохраняет вид просмотра, а продолжает работу с новым содержимым базы данных. В практическом задании *«Индексированный поиск по шаблонам»* демонстрируется один из способов одновременного использования итераторов и передачи управления.

Для получения информации о внутренней структуре итераторов см. документацию по библиотеке для функционального программирования в Lua «Lua Functional library».

Параметры

- `index_object` (*index_object*) – *ссылка на объект*.
- `key` (*scalar/table*) – значение должно совпасть с индексным ключом, который может быть составным
- `iterator` – как определено в таблицах ниже. По умолчанию используется итератор „EQ“

возвращается итератор, который может использовать в цикле for/end или с функцией totable()

Возможные ошибки:

- спейс отсутствует; неправильный тип;
- выбранный тип итерации не поддерживается для данного типа индекса;
- ключ не поддерживается для данного типа итерации.

Факторы сложности Размер индекса, тип индекса; количество кортежей, к которым получен доступ.

Значение искомого ключа может представлять собой число (например, `1234`), строку (например, `'abcd'`) или таблицу из чисел и строк (например, `{1234, 'abcd'}`). Каждая часть ключа будет сопоставляться с каждой частью ключа в индексе.

Найденные кортежи будут упорядочены по значению ключа в индексе или по хешу значения ключа, если тип индекса – „hash“. Если индекс не уникален, то дубликаты будут упорядочены во вторую очередь по первичному значению ключа. Порядок будет обратным, если тип итератора – „LT“, „LE“ или „REQ“.

Типы итераторов для TREE-индексов

Тип итератора	Аргументы	Описание
box.index.EQ или „EQ“	искомое значение	Оператором сравнения будет „==“ (равно). Если ключ индекса равен искомому значению, получим совпадение. Найденные кортежи упорядочены по возрастанию по ключу индекса. Этот тип используется по умолчанию.
box.index.REQ или „REQ“	искомое значение	Совпадения находятся таким же образом, что и для `box.index.EQ`. Разница только в том, что найденные кортежи упорядочены по ключу индекса по убыванию, а не по возрастанию.
box.index.GT или „GT“	искомое значение	Оператором сравнения будет „>“ (больше чем). Если ключ индекса больше, чем искомое значение, получим совпадение. Найденные кортежи упорядочены по возрастанию по ключу индекса.
box.index.GE или „GE“	искомое значение	Оператором сравнения будет „>=“ (больше или равен). Если ключ индекса больше искомого значения или равен ему, получим совпадение. Найденные кортежи упорядочены по возрастанию по ключу индекса.
box.index.ALL или „ALL“	искомое значение	Как для box.index.GE.
box.index.LT или „LT“	искомое значение	Оператором сравнения будет „<“ (меньше чем). Если ключ индекса меньше искомого значения, получим совпадение. Найденные кортежи упорядочены по убыванию по ключу индекса.
box.index.LE или „LE“	искомое значение	Оператором сравнения будет „<=“ (меньше или равен). Если ключ индекса меньше искомого значения или равен ему, получим совпадение. Найденные кортежи упорядочены по убыванию по ключу индекса.

Неофициально можно сказать, что поиск с помощью TREE-индексов пользователи обычно считают интуитивно понятным при условии, что нет нулевых значений и отсутствующих частей. Формально же логика заключается в следующем. Ключ поиска состоит из нуля или более частей, например, {}, {1,2,3},{1,nil,3}. Ключ индекса состоит из одной или более частей, например, {1}, {1,2,3},{1,2,3}. Ключ поиска может содержать нулевое значение nil (но не msgpack.NULL, этот тип не будет правильным). Ключ индекса не может содержать nil или msgpack.NULL, хотя в последующих версиях правила работы Tarantool'а будут другие – поведение поиска с nil может измениться. Возможные итераторы: LT, LE, EQ, REQ, GE, GT. Считается, что ключ поиска соответствует ключу индекса, если следующие операторы, которые представляют собой псевдокод для операции сопоставления, возвращают TRUE.

```
If (number-of-search-key-parts > number-of-index-key-parts) return ERROR
If (number-of-search-key-parts == 0) return TRUE
for (i = 1; ; ++i)
{
  if (i > number-of-search-key-parts) OR (search-key-part[i] is nil)
  {
    if (iterator is LT or GT) return FALSE
    return TRUE
  }
  if (type of search-key-part[i] is not compatible with type of index-key-part[i])
  {
    return ERROR
```

(continues on next page)

(продолжение с предыдущей страницы)

```
  }
  if (search-key-part[i] == index-key-part[i])
  {
    continue
  }
  if (search-key-part[i] > index-key-part[i])
  {
    if (iterator is EQ or REQ or LE or LT) return FALSE
    return TRUE
  }
  if (search-key-part[i] < index-key-part[i])
  {
    if (iterator is EQ or REQ or GE or GT) return FALSE
    return TRUE
  }
}
```

Типы итераторов для HASH-индексов

Тип	Аргументы	Описание
box.index.ALL	нет	Все ключи индекса являются совпадениями. Найденные кортежи упорядочены по возрастанию по хешу ключа индекса, который будет выглядеть случайным.
box.index.EQ или „EQ“	искомое значение	Оператором сравнения будет „==“ (равный). Если ключ индекса равен искомому значению, получим совпадение. Количество найденных кортежей будет 0 или 1. Этот тип используется по умолчанию.
box.index.GT или „GT“	искомое значение	Оператором сравнения будет „>“ (больше чем). Если хеш ключа индекса больше, чем хеш искомого значения, получим совпадение. Найденные кортежи упорядочены по возрастанию по хешу ключа индекса, который будет выглядеть случайным. При условии, что спейс не обновляется, можно получить все кортежи в спейсе, N кортежей за раз, используя {iterator=“GT“, limit=N} в каждом поиске и последнее найденное значение из предыдущего результата поиска в качестве начального значения для следующего поиска.

Типы итераторов для BITSET-индексов

Тип	Аргументы	Описание
box.index.ALL или „ALL“	нет	Все ключи индекса являются совпадениями. Найденные кортежи упорядочены по положению в спейсе.
box.index.EQ или „EQ“	значение bitset (битовое множество)	Если ключ индекса равен искомому значению, получим совпадение. Найденные кортежи упорядочены по положению в спейсе. Этот тип используется по умолчанию.
box.index.BITS_ALL_SET	значение bitset (битовое множество)	Если все биты, которые равны 1 в битовом множестве, также равны 1 в ключе индекса, получим совпадение. Найденные кортежи упорядочены по положению в спейсе.
box.index.BITS_ANY_SET	значение bitset (битовое множество)	Если один из битов, которые равны 1 в битовом множестве, также равен 1 в ключе индекса, получим совпадение. Найденные кортежи упорядочены по положению в спейсе.
box.index.BITS_ALL_NOT_SET	значение bitset (битовое множество)	Если все биты, которые равны 1 в битовом множестве, равны 0 в ключе индекса, получим совпадение. Найденные кортежи упорядочены по положению в спейсе.

Типы итераторов для RTREE-индексов

Тип	Аргументы	Описание
box.index.ALL или „ALL“	нет	Все ключи являются совпадениями. Найденные кортежи упорядочены по положению в спейсе.
box.index.EQ или „EQ“	искомое значение	Если все точки прямоугольника-или-параллелепипеда, определенные искомым значением, совпадают с точками прямоугольника-или-параллелепипеда, определенного ключом индекса, получим совпадение. Найденные кортежи упорядочены по положению в спейсе. «Прямоугольник-или-параллелепипед» означает «прямоугольник-или-параллелепипед, как описано в разделе о *RTREE*». Этот тип используется по умолчанию.

Продолжается на следующей странице

Таблица 2 – продолжение с предыдущей страницы

Тип	Аргументы	Описание
box.index.GT или „GT“	искомое значение	Если все точки прямоугольника-или-параллелепипеда, определенные искомым значением, находятся в пределах прямоугольника-или-параллелепипеда, определенного ключом индекса, получим совпадение. Найденные кортежи упорядочены по положению в спейсе.
box.index.GE или „GE“	искомое значение	Если все точки прямоугольника-или-параллелепипеда, определенные искомым значением, находятся в пределах прямоугольника-или-параллелепипеда, определенного ключом индекса, или рядом с ним, получим совпадение. Найденные кортежи упорядочены по положению в спейсе.
box.index.LT или „LT“	искомое значение	Если все точки прямоугольника-или-параллелепипеда, определенные ключом индекса, находятся в пределах прямоугольника-или-параллелепипеда, определенного искомым значением, получим совпадение. Найденные кортежи упорядочены по положению в спейсе.
box.index.LE или „LE“	искомое значение	Если все точки прямоугольника-или-параллелепипеда, определенные ключом индекса, находятся в пределах прямоугольника-или-параллелепипеда, определенного искомым значением, или рядом с ним, получим совпадение. Найденные кортежи упорядочены по положению в спейсе.
box.index.OVERLAPS или „OVERLAPS“	искомое значение	Если некоторые точки прямоугольника-или-параллелепипеда, определенные искомым значением, находятся в пределах прямоугольника-или-параллелепипеда, определенного ключом индекса, получим совпадение. Найденные кортежи упорядочены по положению в спейсе.
box.index.NEIGHBOR или „NEIGHBOR“	искомое значение	Если некоторые точки прямоугольника-или-параллелепипеда, определенные ключом, находятся в пределах, определенных ключом индекса, или рядом с ним, получим совпадение. Найденные кортежи упорядочены следующим образом: сначала ближайший сосед.

Первый пример pairs():

„TREE“-индекс, используемый по умолчанию, и функция `pairs()`:

```
tarantool> s = box.schema.space.create('space17')
---
...
```

(continues on next page)

(продолжение с предыдущей страницы)

```
tarantool> s:create_index('primary', {
         >   parts = {1, 'string', 2, 'string'}
         > })
---
...
tarantool> s:insert{'C', 'C'}
---
- ['C', 'C']
...
tarantool> s:insert{'B', 'A'}
---
- ['B', 'A']
...
tarantool> s:insert{'C', '!'}
---
- ['C', '!']
...
tarantool> s:insert{'A', 'C'}
---
- ['A', 'C']
...
tarantool> function example()
         >   for _, tuple in
         >     s.index.primary:pairs(nil, {
         >         iterator = box.index.ALL}) do
         >       print(tuple)
         >   end
         > end
---
...
tarantool> example()
['A', 'C']
['B', 'A']
['C', '!']
['C', 'C']
---
...
tarantool> s:drop()
---
...
```

Второй пример pairs():

Данный код на Lua найдет все кортежи, значения первичного ключа в которых начинаются с „XY“. Рабочие предположения заключаются в следующем: есть однокомпонентный первичный TREE-индекс по первому полю, которое должно представлять собой строку. Цикл с итератором обеспечивает поиск кортежей, в которых первое значение больше или равно „XY“. Условный оператор в цикле служит для того, чтобы цикл останавливался, если первые две буквы не „XY“.

```
for _, tuple in
box.space.t.index.primary:pairs("XY",{iterator = "GE"}) do
  if (string.sub(tuple[1], 1, 2) ~= "XY") then break end
  print(tuple)
end
```

Третий пример pairs():

Данный код на Lua найдет все кортежи, значения первичного ключа которых равны или больше 1000 и меньше или равны 1999 (такой тип запроса иногда называют поиском по диапазону или поиском в заданных пределах). Рабочие предположения заключаются в следующем: есть однокомпонентный первичный TREE-индекс по первому полю, которое должно представлять собой *число*. Цикл с итератором обеспечивает поиск кортежей, в которых первое значение больше или равно 1000. Условный оператор в цикле служит для того, чтобы цикл останавливался, если первое значение больше 1999.

```
for _, tuple in
box.space.t2.index.primary:pairs(1000,{iterator = "GE"}) do
  if (tuple[1] > 1999) then break end
  print(tuple)
end
```

index_object:select(*search-key*, *options*)

Это может быть альтернативой для функции *box.space. . . select()*, которая проходит по определенному индексу и может использовать дополнительные параметры, которые определяют тип итератора и пределы (то есть максимальное количество возвращаемых кортежей) и смещение (то есть с какого кортежа в списке начинать).

Параметры

- index_object (*index_object*) – *ссылка на объект*.
- key (*scalar/table*) – значения для сопоставления с ключом индекса
- options (*table/nil*) – ни один, любой или все следующие параметры:
  - interator – тип итератора
  - limit – максимальное количество кортежей
  - offset – номер начального кортежа

возвращается кортеж или кортежи, которые совпадают со значениями поля.

тип возвращаемого значения массив кортежей

Пример:

```
-- Создать спейс под названием tester.
tarantool> sp = box.schema.space.create('tester')
-- Создать уникальный индекс 'primary'
-- который не будет нужен для данного примера..
tarantool> sp:create_index('primary', {parts = {1, 'unsigned' }})
-- Создать неуникальный индекс 'secondary'
-- по второму полю.
tarantool> sp:create_index('secondary', {
         >   type = 'tree',
         >   unique = false,
         >   parts = {2, 'string'}
         > })
-- Вставить три кортежа, значения в поле2 field[2]
-- равны 'X', 'Y' и 'Z'.
tarantool> sp:insert{1, 'X', 'Row with field[2]=X'}
tarantool> sp:insert{2, 'Y', 'Row with field[2]=Y'}
tarantool> sp:insert{3, 'Z', 'Row with field[2]=Z'}
-- Выбрать все кортежи, где вторичные ключи
-- больше, чем 'X'.
tarantool> sp.index.secondary:select({'X'}, {
         >   iterator = 'GT',
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        >   limit = 1000
        > })
```

Результатом будет следующая таблица кортежа:

```
---
- - [2, 'Y', 'Row with field[2]=Y']
  - [3, 'Z', 'Row with field[2]=Z']
...
```

Примечание: Аргументы необязательны. Если вы вызываете `box.space.`*`имя-спейса`*`:select{}`, то каждый ключ в индексе считается совпадающим, независимо от типа итератора. Таким образом, в приведённом выше примере, `box.space.tester:select{}` выберет каждый кортеж в спейсе `tester` по первому индексу (первичный ключ).

Примечание: Параметр `index.`*`имя-индекса`* необязателен. Если он пропущен, то подразумевается первый индекс (первичный ключ). Таким образом, для примера выше, `box.space.tester:select({1}, {iterator = 'GT'})` вернет две одинаковых строки по первичному индексу „primary“.

Примечание: Параметр типа итератора `iterator =` *`тип-итератора`* необязателен. Если он пропущен, то подразумевается, что `iterator = 'EQ'`.

Примечание: `box.space.`*`имя-спейса`*`.index.`*`имя-индекса`*`:select(...)[1]` можно заменить `box.space.`*`имя-спейса`*`.index.`*`имя-индекса`*`:get(...)`. А именно, `get` можно использовать в качестве удобного сокращения для получения первого кортежа в наборе кортежей, который был бы выведен по запросу `select`. Однако, если в наборе кортежей больше одного кортежа, `get` завершится с ошибкой.

Пример с индексом BITSET:

Следующий скрипт показывает создание BITSET-индекса и поиск по нему. Обратите внимание, что битовое множество BITSET не может быть уникальным, поэтому сначала создается первичный индекс. Обратите внимание, что битовые значения вводятся как шестнадцатеричные литералы для удобства чтения.

```
tarantool> s = box.schema.space.create('space_with_bitset')
tarantool> s:create_index('primary_index', {
         >   parts = {1, 'string'},
         >   unique = true,
         >   type = 'TREE'
         > })
tarantool> s:create_index('bitset_index', {
         >   parts = {2, 'unsigned'},
         >   unique = false,
         >   type = 'BITSET'
         > })
tarantool> s:insert{'Tuple with bit value = 01', 0x01}
```

(continues on next page)

(продолжение с предыдущей страницы)

```
tarantool> s:insert{'Tuple with bit value = 10', 0x02}
tarantool> s:insert{'Tuple with bit value = 11', 0x03}
tarantool> s.index.bitset_index:select(0x02, {
         >   iterator = box.index.EQ
         > })
---
- - ['Tuple with bit value = 10', 2]
...
tarantool> s.index.bitset_index:select(0x02, {
         >   iterator = box.index.BITS_ANY_SET
         > })
---
- - ['Tuple with bit value = 10', 2]
  - ['Tuple with bit value = 11', 3]
...
tarantool> s.index.bitset_index:select(0x02, {
         >   iterator = box.index.BITS_ALL_SET
         > })
---
- - ['Tuple with bit value = 10', 2]
  - ['Tuple with bit value = 11', 3]
...
tarantool> s.index.bitset_index:select(0x02, {
         >   iterator = box.index.BITS_ALL_NOT_SET
         > })
---
- - ['Tuple with bit value = 01', 1]
...
```

index_object:get(*key*)

Поиск кортежа по заданному индексу, как описано *выше*.

Параметры

- index_object (*index_object*) – *ссылка на объект*.
- key (*scalar/table*) – значения для сопоставления с ключом индекса

возвращается кортеж, в котором поля ключа в индексе равны переданным значениям ключа.

тип возвращаемого значения кортеж

Возможные ошибки:

- отсутствие такого индекса;
- неправильный тип;
- больше одного кортежа подходят.

Факторы сложности: Размер индекса, тип индекса. См. также *space_object:get()*.

Пример:

```
tarantool> box.space.tester.index.primary:get(2)
---
- [2, 'Music']
...
```

index_object:min([*key*])
Поиск минимального значения в указанном индексе.

Параметры

- index_object (*index_object*) – *ссылка на объект*.
- key (*scalar/table*) – значения для сопоставления с ключом индекса

возвращается кортеж для первого ключа в индексе. Если указано необязательное значение ключа key, будет выведен первый ключ, который больше или равен значению ключа key. Начиная с версии Tarantool 2.0, index_object:min(значение key) не вернет ничего, если значение key не равно значению в индексе.

тип возвращаемого значения кортеж

Возможные ошибки: тип индекса не „TREE“.

Факторы сложности: Размер индекса, тип индекса.

Пример:

```
tarantool> box.space.tester.index.primary:min()
---
- ['Alpha!', 55, 'This is the first tuple!']
...
```

index_object:max([*key*])
Поиск максимального значения в указанном индексе.

Параметры

- index_object (*index_object*) – *ссылка на объект*.
- key (*scalar/table*) – значения для сопоставления с ключом индекса

возвращается кортеж для последнего ключа в индексе. Если указано необязательное значение ключа key, будет выведен последний ключ, который меньше или равен значению ключа key. Начиная с версии Tarantool 2.0, index:max(значение key) не вернет ничего, если значение key не равно значению в индексе.

тип возвращаемого значения кортеж

Возможные ошибки: тип индекса не „TREE“.

Факторы сложности: Размер индекса, тип индекса.

Пример:

```
tarantool> box.space.tester.index.primary:max()
---
- ['Gamma!', 55, 'This is the third tuple!']
...
```

index_object:random(*seed*)
Поиск случайного значения в заданном индексе. Данный метод используется, когда важно получить представление о распределении данных в индексе без необходимости проходить по всему набору данных.

Параметры

- index_object (*index_object*) – *ссылка на объект*.
- seed (*number*) – произвольное неотрицательное целое число

возвращается кортеж для случайного ключа в индексе.

тип возвращаемого значения кортеж

Факторы сложности: Размер индекса, тип индекса.

Примечание про движок базы данных: vinyl не поддерживает `random()`.

Пример:

```
tarantool> box.space.tester.index.secondary:random(1)
---
- ['Beta!', 66, 'This is the second tuple!']
...
```

`index_object:count(`[*key*][, *iterator*]`)`

Итерация по индексу с подсчетом количества кортежей, которые соответствуют паре ключ-значение.

Параметры

- `index_object` (*index_object*) – *ссылка на объект*.
- `key` (*scalar/table*) – значения для сопоставления с ключом индекса
- `iterator` – метод сопоставления

возвращается количество совпадающих кортежей.

тип возвращаемого значения число

Пример:

```
tarantool> box.space.tester.index.primary:count(999)
---
- 0
...
tarantool> box.space.tester.index.primary:count('Alpha!', { iterator = 'LE' })
---
- 1
...
```

`index_object:update(`*key*, *{{operator*, *field_no*, *value}*, *...}*`)`

Обновление кортежа.

То же, что и *box.space...update()*, но поиск ключа происходит в этом индексе, вместо первичного. Данный индекс должен быть уникальным.

Параметры

- `index_object` (*index_object*) – *ссылка на объект*.
- `key` (*scalar/table*) – значения для сопоставления с ключом индекса
- `operator` (string) – тип операции, представленный строкой
- `field_no` (*number*) – к какому полю применяется операция. Номер поля может быть отрицательным, что означает, что позиция рассчитывается с конца кортежа. (#кортеж + отрицательный номер поля + 1)
- `value` (*lua_value*) – какое значение применяется

возвращается

- обновленный кортеж

- nil, если ключ не найден

тип возвращаемого значения tuple or nil

`index_object:delete(`*key*`)`
Удаление кортежа по ключу.

То же, что и *box.space... delete()*, но поиск ключа происходит в этом индексе, вместо первичного. Данный индекс должен быть уникальным.

Параметры

- `index_object` (*index_object*) – *ссылка на объект*.
- `key` (*scalar/table*) – значения для сопоставления с ключом индекса

возвращается удаленный кортеж.

тип возвращаемого значения кортеж

Примечание про движок базы данных: vinyl вернет *nil*, а не удаленный кортеж.

`index_object:alter(`*{options}*`)`
Изменение индекса. В определенных обстоятельствах можно изменять некоторые характеристики индекса, например тип, параметры последовательности и определение его уникальности. Тем не менее, это обычно приводит к перестроению спейса за исключением простого случая, когда значение флага `is_nullable` меняется с `false` на `true`.

Параметры

- `index_object` (*index_object*) – *ссылка на объект*.
- `options` (`table`) – список параметров, аналогичный списку параметров для `create_index`, см. таблицу под названием *Параметры для space_object:create_index()*.

возвращается nil

Возможные ошибки:

- индекс не существует,
- индекс по первичному ключу не может быть неуникальным, то есть нельзя задать `{unique = false}`.

Примечание про движок базы данных: vinyl не поддерживает `alter()` для первичного индекса, если спейс содержит данные.

Пример 1:

Можно добавлять и удалять поля, которые составляют первичный индекс:

```
tarantool> s = box.schema.create_space('test')
---
...
tarantool> i = s:create_index('i', {parts = {{field = 1, type = 'unsigned'}}})
---
...
tarantool> s:insert({1, 2})
---
- [1, 2]
...
tarantool> i:select()
---
```

(continues on next page)

(продолжение с предыдущей страницы)

```
- - [1, 2]
...
tarantool> i:alter({parts = {{field = 1, type = 'unsigned'}, {field = 2, type = 'unsigned
↪'}}})
---
...
tarantool> s:insert({1, 't'})
---
- error: 'Tuple field 2 type does not match one required by operation: expected unsigned'
...
```

Пример 2:

Можно изменять опции индекса для спейсов как в memtx'e, так и в vinyl'e:

```
tarantool> box.space.space55.index.primary:alter({type = 'HASH'})
---
...

tarantool> box.space.vinyl_space.index.i:alter({page_size=4096})
---
...
```

`index_object:drop()`

Удаление индекса. Побочный эффект удаления первичного индекса – все кортежи удалятся.

Параметры

- `index_object` (*index_object*) – *ссылка на объект*.

возвращается nil.

Возможные ошибки:

- индекс не существует,
- первичный индекс невозможно удалить, если существует вторичный индекс.

Пример:

```
tarantool> box.space.space55.index.primary:drop()
---
...
```

`index_object:rename(`*index-name*`)`

Переименование индекса.

Параметры

- `index_object` (*index_object*) – *ссылка на объект*.
- `index-name` (string) – новое имя индекса

возвращается nil

Возможные ошибки: index_object не существует.

Пример:

```
tarantool> box.space.space55.index.primary:rename('secondary')
---
...
```

Факторы сложности: Размер индекса, тип индекса, количество кортежей, к которым получен доступ.

`index_object:bsize()`
Возврат общего количества байтов, занятых индексом.

Параметры

- `index_object` (*index_object*) – *ссылка на объект*.

возвращается количество байтов

тип возвращаемого значения число

`index_object:stat()`
Получение статистики о предпринятых действиях, которые влияют на индекс.

Используется с движком базы данных vinyl.

Подробные данные в выводе `index_object:stat()`:

- `index_object:stat().latency` содержит отметки времени в процентах;
- `index_object:stat().bytes` содержит общее количество байтов;
- `index_object:stat().disk.rows` содержит примерное количество кортежей в каждом диапазоне;
- `index_object:stat().disk.statement` содержит количество вставок, обновлений, обновлений и вставок, удалений (inserts|updates|upserts|deletes);
- `index_object:stat().disk.compaction` содержит количество слияний и их объем;
- `index_object:stat().disk.dump` содержит количество дампов и их объем;
- `index_object:stat().disk.iterator.bloom` содержит количество совпадений и несовпадений по фильтрами Блума;
- `index_object:stat().disk.pages` содержит размер в страницах;
- `index_object:stat().disk.last_level` содержит объем данных на последнем уровне LSM-дерева;
- `index_object:stat().cache.evict` содержит количество освобождений кэша;
- `index_object:stat().range_size` содержит максимальное количество байтов в диапазоне;
- `index_object:stat().dumps_per_compaction` содержит среднее число дампов, которое необходимо для запуска значительного слияния в любом диапазоне LSM-дерева.

С помощью *box.stat.vinyl()* можно получить сводную статистику по индексу.

Параметры

- `index_object` (*index_object*) – *ссылка на объект*.

возвращается статистические данные

тип возвращаемого значения таблица

`index_object:compact()`
Удаление неиспользуемого пространства индекса. Для движка базы данных memtx метод бесполезен; `index_object:compact()` используется только для движка vinyl. Например, на движке vinyl при удалении кортежа память не возвращается незамедлительно. Существует планировщик автоматического восстановления ресурсов на основании таких факторов, как

форма LSM-дерева и усложнение, как описано в разделе *Хранение данных с помощью vinyl*, поэтому выполнять `index_object:compact()` вручную необходимости нет.

возвращается nil (Tarantool возвращает нулевое значение сразу же, не ожидая завершения слияния)

`index_object:user_defined()`

Пользователи могут сами определять любые желаемые функции и связывать их с индексами: фактически они могут создавать собственные методы для работы с индексом. Это можно сделать так:

(1) создать Lua-функцию,

(2) добавить имя функции в заданную глобальную переменную с типом «таблица» (table),

(3) впоследствии в любое время, пока работает сервер, вызвать функцию с помощью `объект_индекса:имя-функции([параметры])`.

Есть три заданные глобальные переменные:

- Метод, добавленный в `box_schema.index_mt`, будет доступен для всех индексов.
- Метод, добавленный в `box_schema.memtx_index_mt`, будет доступен для всех индексов в memtx'e.
- Метод, добавленный в `box_schema.vinyl_index_mt`, будет доступен для всех индексов в vinyl'e.

Можно также сделать задаваемый пользователем метод доступным только для одного индекса путем вызова `getmetatable(объект_индекса)` и последующего добавления имени функции в метатаблицу.

Параметры

- `index_object` (*index_object*) – *ссылка на объект*.
- `any-name` (*any-type*) – то, что определяет пользователь

Пример:

```
-- Доступный для любого индекса спейса memtx, без параметров.
-- После таких запросов значение глобальной переменной global_variable будет 6.
box.schema.space.create('t', {engine='memtx'})
box.space.t:create_index('i')
global_variable = 5
function f() global_variable = global_variable + 1 end
box.schema.memtx_index_mt.counter = f
box.space.t.index.i:counter()
```

Пример:

```
-- Доступный только для индекса box.space.t.index.i, 1 параметр.
-- После таких запросов значение X будет 1005.
box.schema.space.create('t', {engine='memtx', id = 1000})
box.space.t:create_index('i')
X = 0
i = box.space.t.index.i
function f(i_arg, param) X = X + param + i_arg.space_id end
box.schema.memtx_index_mt.counter = f
meta = getmetatable(i)
meta.counter = f
i:counter(5)
```

Пример использования функций box

Данный пример сработает на конфигурации из песочницы, описанной в предисловии, то есть создан спейс под названием tester с первичным числовым ключом. Функция в примере выполнит следующие действия:

- выбрать кортеж, значение ключа в котором равно 1000;
- выдать сообщение об ошибке, если такой кортеж уже существует и содержит 3 поля;
- **вставить или заменить кортеж следующими данными:**
  - поле [1] = 1000
  - поле [2] = UUID
  - поле [3] = количество секунд с 01.01.1970;
- получить поле [3] из того, что заменили;
- преобразовать значение из поля [3] в формат yyyy-mm-dd hh:mm:ss.ffff (год-месяц-день час:минута:секунда.десятитысячные доли секунды);
- вернуть преобразованное значение.

Данная функция использует функции box в Tarantool'е: *box.space...select*, *box.space...replace*, *fiber.time*, *uuid.str*. Данная функция использует Lua-функции os.date() и string.sub().

```
function example()
  local a, b, c, table_of_selected_tuples, d
  local replaced_tuple, time_field
  local formatted_time_field
  local fiber = require('fiber')
  table_of_selected_tuples = box.space.tester:select{1000}
  if table_of_selected_tuples ~= nil then
    if table_of_selected_tuples[1] ~= nil then
      if #table_of_selected_tuples[1] == 3 then
        box.error({code=1, reason='This tuple already has 3 fields'})
      end
    end
  end
  replaced_tuple = box.space.tester:replace
    {1000,  require('uuid').str(), tostring(fiber.time())}
  time_field = tonumber(replaced_tuple[3])
  formatted_time_field = os.date("%Y-%m-%d %H:%M:%S", time_field)
  c = time_field % 1
  d = string.sub(c, 3, 6)
  formatted_time_field = formatted_time_field .. '.' .. d
  return formatted_time_field
end
```

... А вот что происходит, когда вызывается функция:

```
tarantool> box.space.tester:delete(1000)
---
- [1000, '264ee2da03634f24972be76c43808254', '1391037015.6809']
...
tarantool> example(1000)
---
- 2014-01-29 16:11:51.1582
...
```

(continues on next page)

(продолжение с предыдущей страницы)

```
tarantool> example(1000)
---
- error: 'This tuple already has 3 fields'
...
```

Пример с заданным пользователем итератором

Здесь приведен пример того, как создать свой собственный итератор. Функция `paged_iter` представляет собой «функцию с итератором», что поймут только разработчики, которые ознакомились с разделом руководства по Lua Итераторы и замыкания. Она делает постраничную выборку, то есть возвращает 10 кортежей одновременно из таблицы под названием «t», первичный ключ которой определен с помощью `create_index('primary',{parts={1,'string'}})`.

```
function paged_iter(search_key, tuples_per_page)
  local iterator_string = "GE"
  return function ()
  local page = box.space.t.index[0]:select(search_key,
    {iterator = iterator_string, limit=tuples_per_page})
  if #page == 0 then return nil end
  search_key = page[#page][1]
  iterator_string = "GT"
  return page
  end
end
```

Разработчикам, использующим `paged_iter`, необязательно знать, почему она работает, следует лишь понимать, что вызвав функцию в цикле, можно получать 10 кортежей за раз до тех пор, пока кортежи не кончатся.

В данном примере кортежи лишь выводятся по странице за раз. Но легко изменить функцию, например, путем передачи управления после каждой выборки или с помощью прерывания, если кортежи не будут соответствовать дополнительным критериям.

```
for page in paged_iter("X", 10) do
  print("New Page. Number Of Tuples = " .. #page)
  for i = 1, #page, 1 do
    print(page[i])
  end
end
```

Вложенный модуль *box.index* с типом индекса RTREE для поиска в пространственных данных

Вложенный модуль *box.index* может использоваться для поиска в пространственных данных, если тип индекса – RTREE. Существуют операции для поиска *прямоугольников* (геометрические фигуры с 4 углами и 4 сторонами) и *параллелепипедов* (геометрические фигуры с количеством углов более 4 и количеством сторон более 4, которые иногда называются гиперпрямоугольниками). В данном руководстве используется термин *прямоугольник-или-параллелепипед* для всего класса объектов, который включает в себя прямоугольники и параллелепипеды. Примерами иллюстрируются только прямоугольники.

Прямоугольники описаны в соответствии с координатами по оси X (горизонтальной оси) и оси Y (вертикальной оси) на сетке произвольного размера. Ниже представлен рисунок четырех прямоугольников на сетке с 11 горизонтальными точками и 11 вертикальными точками:

```
      X AXIS
           1   2   3   4   5   6   7   8   9   10  11
         1
         2  #-------+                                    <-Прямоугольник №1
Y AXIS   3  |       |
         4  +-------#
         5          #-----------------------+           <-Прямоугольник №2
         6          |                       |
         7          |   #---+               |           <-Прямоугольник №3
         8          |   |   |               |
         9          |   +---#               |
         10         +-----------------------#
         11                                  #           <-Прямоугольник №4
```

Прямоугольники определяются в соответствии со следующей схемой: {верхняя левая координата по оси X, верхняя левая координата по оси Y, нижняя правая координата по оси X, нижняя правая координата по оси Y} – или коротко: {x1,y1,x2,y2}. Таким образом, на рисунке ... Прямоугольник № 1 начинается в точке 1 по оси X и точке 2 по оси Y, а заканчивается в точке 3 по оси X и точке 4 по оси Y, поэтому его координаты будут следующие: {1,2,3,4}. Координаты Прямоугольника № 2: {3,5,9,10}. Координаты Прямоугольника № 3: {4,7,5,9}. И наконец, координаты Прямоугольника № 4: {10,11,10,11}. Прямоугольник № 4, на самом деле, является точкой, поскольку у него нулевая ширина и нулевая высота, так что его можно описать всего двумя числами: {10,11}.

Некоторые отношения между прямоугольниками могут быть описаны так: «Прямоугольник №1 является ближайшим соседом Прямоугольника №2», а «Прямоугольник №3 полностью находится внутри Прямоугольника №2».

Сейчас создадим спейс и добавим RTREE-индекс.

```
tarantool> s = box.schema.space.create('rectangles')
tarantool> i = s:create_index('primary', {
         >   type = 'HASH',
         >   parts = {1, 'unsigned'}
         > })
tarantool> r = s:create_index('rtree', {
         >   type = 'RTREE',
         >   unique = false,
         >   parts = {2, 'ARRAY'}
         > })
```

Поле №1 не имеет значения, мы создаем его лишь потому, что необходим первичный индекс. (RTREE-индексы не могут быть уникальными, поэтому не могут быть первичными индексами.) Второе поле должно быть массивом («array»), что означает, что его значения должны представлять собой точки {x,y} или прямоугольники {x1,y1,x2,y2}. Заполним таблицу, вставив два кортежа с координатами Прямоугольника №2 и Прямоугольника №4.

```
tarantool> s:insert{1, {3, 5, 9, 10}}
tarantool> s:insert{2, {10, 11}}
```

Затем, после описания типов RTREE-итераторов (*RTREE iterator types*), можно произвести поиск прямоугольников с помощью данных запросов:

```
tarantool> r:select({10, 11, 10, 11}, {iterator = 'EQ'})
---
- - [2, [10, 11]]
...
tarantool> r:select({4, 7, 5, 9}, {iterator = 'GT'})
```

(continues on next page)

(продолжение с предыдущей страницы)

```
---
- - [1, [3, 5, 9, 10]]
...
tarantool> r:select({1, 2, 3, 4}, {iterator = 'NEIGHBOR'})
---
- - [1, [3, 5, 9, 10]]
  - [2, [10, 11]]
...
```

Запрос №1 возвращает 1 кортеж, потому что точка {10,11} представляет собой то же, что и прямоугольник {10,11,10,11} («Прямоугольник №4» на рисунке). Запрос № 2 возвращает 1 кортеж, потому что прямоугольник {4,7,5,9}, который был «Прямоугольником №3» на рисунке находится полностью внутри {3,5,9,10}, что представляет собой Прямоугольник № 2. Запрос № 3 возвращает 2 кортежа, потому что итератор NEIGHBOR (сосед) всегда возвращает все кортежи, а первым найденным кортежем будет {3,5,9,10} («Прямоугольник №2» на рисунке), потому что он является ближайшим соседом {1,2,3,4} («Прямоугольник №1» на рисунке).

Теперь создадим спейс и индекс для кубоидов, которые представляют собой прямоугольники-или-параллелепипеды, у которых 6 углов и 6 сторон.

```
tarantool> s = box.schema.space.create('R')
tarantool> i = s:create_index('primary', {parts = {1, 'unsigned'}})
tarantool> r = s:create_index('S', {
         >   type = 'RTREE',
         >   unique = false,
         >   dimension = 3,
         >   parts = {2, 'ARRAY'}
         > })
```

Здесь задается дополнительный параметр“dimension=3“. По умолчанию, измерений 2, поэтому не было необходимости указывать данный параметр в примерах для прямоугольника. Максимальное количество измерений – 20. Что касается вставки и выборки, здесь будет 6 координат. Например:

```
tarantool> s:insert{1, {0, 3, 0, 3, 0, 3}}
tarantool> r:select({1, 2, 1, 2, 1, 2}, {iterator = box.index.GT})
```

Теперь создадим спейс и индекс для пространственных объектов с метрикой расстояния городских кварталов (метрика Манхэттена), которые представляют собой прямоугольники-или-параллелепипеды; соседи для них рассчитываются иным образом.

```
tarantool> s = box.schema.space.create('R')
tarantool> i = s:create_index('primary', {parts = {1, 'unsigned'}})
tarantool> r = s:create_index('S', {
         >   type = 'RTREE',
         >   unique = false,
         >   distance = 'manhattan',
         >   parts = {2, 'ARRAY'}
         > })
```

Здесь задается дополнительный параметр `distance='manhattan'`. По умолчанию, расстояние измеряется по Евклидовой метрике, что лучше всего подходит для измерений по прямой линии. Другой способ расчета расстояния по метрике Манхэттена („manhattan“), который больше подходит, если необходимо следовать линиям сетки, а не по прямой.

```
tarantool> s:insert{1, {0, 3, 0, 3}}
tarantool> r:select({1, 2, 1, 2}, {iterator = box.index.NEIGHBOR})
```

Другие примеры поиска в пространственных данных см. по ссылке R tree index quick start and usage.

Вложенный модуль *box.info*

Вложенный модуль `box.info` предоставляет доступ к информации о переменных экземпляра сервера.

- **cluster.uuid** – это уникальный идентификатор набора реплик (UUID). У каждого экземпляра в наборе реплик будет одно и то же значение `cluster.uuid`. Данное значение также хранится в системном спейсе *box.space._schema*.
- **gc()** возвращает состояние *сборщика мусора в Tarantool'е*, в том числе контрольные точки и их потребителей (пользователи); более подробную информацию см. *ниже*.
- **id** соответствует идентификатору **replication.id** (см. *ниже*).
- **lsn** соответствует регистрационному номеру **replication.lsn** (см. *ниже*).
- **memory()** возвращает статистику об использовании памяти (см. *ниже*).
- **pid** – идентификатор процесса. Это значение также отображается с помощью модуля *tarantool* и по команде `ps -A` в Linux.
- **ro** принимает значение `true`, если экземпляр находится в режиме только для чтения «read-only» (как *read_only* в `box.cfg{}`), или в статусе „orphan“ (одиночный).
- **signature** представляет собой сумму всех значений **lsn** из векторных часов (**vclock**) всех экземпляров в наборе реплик.
- **status** – это текущий статус экземпляра. Он может быть:
  - `running` – экземпляр запущен,
  - `loading` – экземпляр восстанавливается из xlog'ов или snapshot'ов или стартует с нуля (bootstrapping),
  - `orphan` – экземпляр (еще) не подключился к необходимому количеству мастеров (см. *статус orphan*),
  - `hot_standby` – экземпляр является *резервным* для другого экземпляра.
- **uptime** – это количество секунд с момента запуска экземпляра. Данное значение также можно получить с помощью *tarantool.uptime()*.
- **uuid** соответствует идентификатору **replication.uuid** (см. *ниже*).
- **vclock** соответствует часам **replication.downstream.vclock** (см. *ниже*).
- **version** – это версия Tarantool'а. Данное значение также можно отобразить с помощью команды *tarantool -V*.
- **vinyl** возвращает статистику времени работы для движка базы данных vinyl. Данная функция объявлена устаревшей, используйте *box.stat.vinyl()*.

`box.info.memory()`
Функция **memory** в `box.info` дает пользователю `admin` полное представление об экземпляре Tarantool'а.

Примечание: Чтобы получить представление о подсистеме vinyl'а, используйте *box.stat.vinyl()*.

- **memory().cache** – это количество байтов, используемых для кэширования данных пользователей. Движок базы данных memtx не нуждается в кэше, то есть на самом деле это количество байтов в кэше для кортежей движка базы данных vinyl.
- **memory().data** – количество байтов, используемых для хранения данных пользователей (кортежи) в движке memtx и на уровне 0 движка vinyl, не принимая во внимание фрагментацию памяти.
- **memory().index** – количество байтов, используемых для индексирования данных пользователей, включая экстенты для деревьев в memtx'е и vinyl'е, индекс страниц и фильтры Блума в vinyl'е.
- **memory().lua** – количество байтов, используемых Lua-интерпретатором.
- **memory().net** – количество байтов, используемых буферами для сетевого ввода-вывода.
- **memory().tx** – количество байтов, используемых активными транзакциями. Для движка базы данных vinyl это общий размер всех размещаемых объектов (структура `txv`, структура `vy_tx`, структура `vy_read_interval`) и кортежей, прикрепленных к этим объектам.

Пример с минимальным распределением, когда используется только движок базы данных memtx:

```
tarantool> box.info.memory()
---
- cache: 0
  data: 6552
  tx: 0
  lua: 1315567
  net: 98304
  index: 1196032
...
```

`box.info.gc()`

Функция **gc** в `box.info` дает пользователю `admin` полное представление о факторах, которые влияют на *сборщик мусора Tarantool'а*. Сборщик мусора сопоставляет значения vclock (*векторные часы*) пользователей и контрольных точек, поэтому взглянув на `box.info.gc()`, можно понять, почему сборщик мусора не удалил старые WAL-файлы или что он может вскоре удалить.

- **gc().consumers** – список пользователей, запросы которых могут затронуть сборку мусора.
- **gc().checkpoints** – список сохраненных контрольных точек.
- **gc().checkpoints[n].references** – список ссылок на контрольную точку.
- **gc().checkpoints[n].vclock** – значение vclock контрольной точки.
- **gc().checkpoints[n].signature** – сумма компонентов vclock контрольной точки.
- **gc().checkpoint_is_in_progress** – true если идет создание контрольной точки, в противном случае false.
- **gc().vclock** – vclock сборщика мусора.
- **gc().signature** – сумма компонентов контрольной точки сборщика мусора.

`box.info.replication`

Раздел **replication** (репликация) во вложенном модуле `box.info()` содержит статистику по всем экземплярам в наборе реплик относительно текущего экземпляра (см. также *«Мониторинг набора реплик»*):

Далее *n* – это индексный номер одного элемента таблицы, например, `replication[1]`, который содержит данные о экземпляре сервера номер 1, который может быть или не быть тем же самым, что и текущий экземпляр («текущий экземпляр» – это тот, что отвечает на `box.info`).

- `replication[n].id` – это короткий числовой идентификатор экземпляра в наборе реплик. Данное значение хранится в системном спейсе *box.space._cluster*.
- `replication[n].uuid` – это глобально-уникальный идентификатор экземпляра. Данное значение хранится в системном спейсе *box.space._cluster*.
- `replication[n].lsn` – это *номер в журнале* (LSN) для последней записи в *журнале упреждающей записи* (WAL) экземпляра *n*.
- `replication[n].upstream` возникает (т.е. не nil), если текущий экземпляр принимает или готов принимать данные от экземпляра *n*, что обычно означает `replication[n].upstream.status` = `follow`, `replication[n].upstream.peer` = url экземпляра *n*, который отдает данные, `replication[n].lag и idle` = скорость экземпляра, описанная позже. По-другому можно сказать, что `replication[n].upstream` возникает, когда `replication[n].upstream.peer` не принадлежит текущему экземпляру, не является доступным только для чтения и был указан в `box.cfg{replication={...}}`, поэтому он показан в *box.cfg.replication*.
- `replication[n].upstream.status` – это репликационный статус экземпляра:
  - `auth` означает, что экземпляр проходит *аутентификацию*.
  - `connecting` означает, что происходит подключение.
  - `disconnected` означает, что экземпляр не подключен к набору реплик (по причине проблем в сети, а не ошибок репликации).
  - `follow` означает, что текущий экземпляр является репликой (только для чтения, или не только для чтения, но ведет себя как реплика для этого удаленного URI в конфигурации master-master) и получает или может получать данные от мастера экземпляра *n* (upstream).
  - `stopped` означает, что репликация остановилась по причине ошибки репликации (например, *повторяющийся ключ*).
  - `sync` означает, что мастер и реплика в данный момент синхронизируются для получения одинакового набора данных.
- `replication[n].upstream.idle` – это время (в секундах) с момента получения последнего события. Это основной индикатор работоспособности репликации. Подробности в *Monitoring a replica set*.
- `replication[n].upstream.peer` содержит *URI* экземпляра *n*, например 127.0.0.1:3302. Более подробную информацию см. в разделе *Мониторинг набора реплик*.
- `replication[n].upstream.lag` – это разница во времени между локальным временем на экземпляре *n*, зарегистрированным при получении события, и локальное время на другом мастере, зарегистрированное при записи события в *журнал упреждающей записи* на этом мастере. Более подробную информацию см. в разделе *Мониторинг набора реплик*.
- `replication[n].upstream.message` содержит сообщение об ошибке в случае *системного сбоя*, в противном случае – nil.
- `replication[n].downstream` появляется (т.е. не nil), когда имеются данные об экземпляре, который принимает данные от экземпляра *n* или намеревается это делать,что обычно означает `replication[n].downstream.status` = `follow`,

- `replication[n].downstream.vclock` содержит *векторные часы*, которые представляют собой таблицу из пар „**id**, **lsn**“, например `vclock: {1: 3054773, 4: 8938827, 3: 285902018}` (Обратите внимание, что таблица может иметь несколько пар, хотя vclock - это единственное имя).

  Даже если экземпляр *удален*, его значения все равно появятся здесь; однако, его значения будут переопределены, если позже экземпляр присоединится с тем же UUID. Пары векторных часов будут появляться только если `lsn > 0`.

  `replication[n].downstream.vclock` может быть таким же, как и vclock текущего экземпляра (`box.info.vclock`), потому что все значения vclock в кластере известны. Мастер будет знать, что находится в копии vclock реплики, потому что, когда мастер делает изменение данных, он посылает информацию об изменении на реплику (включая векторные часы мастера), и реплика отвечает тем, что находится в ее таблице векторных часов.

- `replication[n].downstream.idle` – это время (в секундах) с момента последней отправки событий экземпляром n через downstream-репликацию.

- `replication[n].downstream.status` – это статус для downstream-репликации:

  – `stopped` означает, что downstream-репликация остановлена,

  – `follow` означает, что downstream-репликация находится в процессе (экземпляр n готов принимать данные от мастера или уже делает это).

- `replication[n].downstream.message` and `replication[n].downstream.system_message` не появятся, пока не возникнет проблем с соединением. Например, если экземпляр n даст сбой, то можно будет увидеть `status = 'stopped'`, `message = 'unexpected EOF when read from socket'`, и `system_message = 'broken pipe'`. См. также *«сбой»*.

`box.info()`

Поскольку содержимое вложенного модуля `box.info` является динамическим, невозможно провести итерацию по ключам с помощью Lua-функции `pairs()`. Для этой цели модуль `box.info()` создает и возвращает Lua-таблицу со всеми ключами и значениями во вложенном модуле.

возвращается ключи и значения во вложенном модуле

тип возвращаемого значения таблица

Пример:

Данный пример приводится для набора со схемой мастер-реплика, который включает в себя один мастер-экземпляр и один реплика-экземпляр. Запрос был отправлен с реплики-экземпляра.

```
tarantool> box.info()
---
- vinyl: []
  version: 2.2.0-482-g8c84932ad
  id: 2
  ro: true
  status: running
  vclock: {1: 9}
  uptime: 356
  lsn: 0
  memory: []
  cluster:
    uuid: e261a5bc-6303-4de3-9873-556f311255e0
  pid: 160
  gc: []
  signature: 9
```

(continues on next page)

(продолжение с предыдущей страницы)

```
  replication:
    1:
      id: 1
      uuid: fce71bb3-0e99-40ec-ab7e-5159487e18d1
      lsn: 9
      upstream:
        status: follow
        idle: 0.035709699994186
        peer: replicator@127.0.0.1:3401
        lag: 0.00016164779663086
      downstream:
        status: follow
        idle: 0.59840899999836
        vclock: {1: 9}
    2:
      id: 2
      uuid: bc4629ce-ea31-4f75-b805-a4807bcacc93
      lsn: 0
  uuid: bc4629ce-ea31-4f75-b805-a4807bcacc93
...
```

Функция *box.once*

box.once(*key*, *function*[, ...])

Выполнение функции при условии, что она раньше не выполнялась. Передаваемое значение проверяется на предмет того, выполнялась ли функция. Если она выполнялась, ничего не происходит. В противном случае вызывается функция.

См. пример использования `box.once()` во время *настройки набора реплик*.

Если в `box.once()` возникает ошибка во время инициализации базы данных, можно повторно запустить невыполненный блок `box.once()`, не останавливая базу данных. Для этого удалите объект `once` из системного спейса *_schema*. Введите команду `box.space._schema:select{}`, найдите объект `once` и удалите его. Например, повторное выполнение блока `key='hello'` :

Когда `box.once()` используется для инициализации, следует подождать, пока база данных не будет в нужном состоянии (только для чтения или для чтения и записи). Для этого см. функции во *вложенном модуле box.ctl*.

```
tarantool> box.space._schema:select{}
---
- - ['cluster', 'b4e15788-d962-4442-892e-d6c1dd5d13f2']
  - ['max_id', 512]
  - ['oncebye']
  - ['oncehello']
  - ['version', 1, 7, 2]
...

tarantool> box.space._schema:delete('oncehello')
---
- ['oncehello']
...

tarantool> box.once('hello', function() end)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
---
...
```

Параметры

- `key` (`string`) – значение для проверки
- `function` (*`function`*) – функция
- `...` – аргументы, которые следует передать в функцию

Примечание: Параметр `key` сохраняется в системном спейсе *`_schema`* после вызова `box.once()`, чтобы предотвратить повторный вызов по ключу. Эти ключи распространяются на набор реплик. Поэтому одновременный вызов `box.once` с одинаковыми ключами на двух экземплярах одного набора реплик может быть успешным, но приведет к конфликту транзакций.

Константа *box.NULL*

Имеется целый ряд серьезных проблем при использовании значения **nil** из Lua в таблицах. Например: вы не можете корректно оценить длину таблицы, не являющейся последовательностью.

Пример:

```
tarantool> t = {0, nil, 1, 2, nil}
---
...

tarantool> t
---
- - 0
  - null
  - 1
  - 2
...

tarantool> #t
---
- 4
...
```

Вывод в консоль `t` обрабатывает значения **nil** в середине и в конце таблицы по-разному. Это вызвано неопределённым поведением.

Примечание: Попытка найти длину для разреженного массива в LuaJIT приводит к другому случаю неопределённого поведения.

Для избежания этой проблемы используйте имеющуюся в Tarantool константу `box.NULL` вместо значения **nil**. `box.NULL` является местозаполнителем для значения **nil** в таблицах с целью сохранения ключа без значения.

Использование box.NULL

`box.NULL` является значением типа cdata, представляющим нулевой указатель (NULL pointer). Оно подобно `msgpack.NULL`, `json.NULL` и `yaml.NULL`. Таким образом, оно является некоторым не **nil** значением, даже если является указателем на NULL.

Используйте `box.NULL` только с NULL, написанным заглавными буквами (`box.null` является ошибкой).

Примечание: Технически, `box.NULL` соответствует `ffi.cast('void *', 0)`.

Пример:

```
tarantool> t = {0, box.NULL, 1, 2, box.NULL}
---
...

tarantool> t
---
- - 0
  - null # cdata
  - 1
  - 2
  - null # cdata
...

tarantool> #t
---
- 5
...
```

Примечание: Заметьте, что `t[2]` демонстрирует один и тот же вывод `null` в обоих примерах. Однако, в данном примере `t[2]` и `t[5]` являются типом cdata, в то время как в предыдущем примере их тип был **nil**.

Важно: Избегайте использования неявных сравнений с обнуляемыми (nullable) значениями при использовании `box.NULL`. В связи со штатным поведением Lua, возвращение любого результата, кроме **false** (ложь) или **nil** (ничто), из выражения условия считается возвращением **true** (истина). Как и упоминалось ранее, `box.NULL` является указателем.

Поэтому выражение `box.NULL` всегда будет расцениваться как **true** (истина) в случае использования в качестве условия в сравнении. Это означает, что код

```
if box.NULL then func() end
```

всегда будет выполнять функцию `func()` (потому, что условие `box.NULL` всегда будет не **false** (ложь) и не **nil** (ничто)).

Различение nil и *box.NULL*

Используйте выражение `x == nil` для проверки того, является ли `x` **nil** или `box.NULL`.

Для выяснения того, является ли `x` в действительности **nil**, но не `box.NULL`, используйте следующее условие:

```
type(x) == 'nil'
```

Если оно истинно (**true**), то `x` – это **nil**, но не“box.NULL“.

Вы можете использовать следующее выражение для `box.NULL`:

```
x == nil and type(x) == 'cdata'
```

Если вышеуказанное выражение истинно (**true**), то `x` – это `box.NULL`.

Примечание: Конвертируя данные в различные форматы (JSON, YAML, msgpack), вы должны ожидать возможного преобразования всех **nil** в разреженных массивах в `box.NULL`. Стоит ответить, что конвертация может происходить неожиданно (например: при отправке данных через *net.box* или при получении данных из *спейсов* и т.п.).

```
tarantool> type(({1, nil, 2})[2])
---
- nil
...

tarantool> type(json.decode(json.encode({1, nil, 2}))[2])
---
- cdata
...
```

Вы должны ожидать подобное поведение и использовать соответствующее выражение условия. Используйте явное сравнение `x == nil` для проверки на отсутствующее значение (NULL) в обнуляемых (nullable) переменных. Оно позволит обнаружить как **nil**, так и `box.NULL`.

Вложенный модуль *box.schema*

Общие сведения

Вложенный модуль `box.schema` содержит функции для определения данных для спейсов, пользователей, ролей, кортежей и последовательностей.

Индекс

Ниже приведен перечень всех функций модуля `box.schema`.

Имя	Использование
box.schema.space.create() или *box.schema.create_space()*	Создание спейса
box.schema.upgrade	Upgrade a database
box.schema.user.create()	Создание пользователя
box.schema.user.drop()	Удаление пользователя
box.schema.user.exists()	Проверка существования пользователя
box.schema.user.grant()	Выдача прав пользователю или роли
box.schema.user.revoke()	Отмена прав пользователя или роли
box.schema.user.password()	Получение хеша пароля пользователя
box.schema.user.passwd()	Ассоциация пароля с пользователем
box.schema.user.info()	Получение описания прав пользователя
box.schema.role.create()	Создание роли
box.schema.role.drop()	Удаление роли
box.schema.role.exists()	Проверка наличия роли
box.schema.role.grant()	Выдача прав роли
box.schema.role.revoke()	Отмена прав роли
box.schema.role.info()	Получение описания прав роли
box.schema.func.create()	Создание кортежа с функцией
box.schema.func.drop()	Удаление кортежа с функцией
box.schema.func.exists()	Проверка наличия кортежа с функцией
box.schema.sequence.create()	Создание нового генератора последовательностей
sequence_object:next()	Генерация и возврат следующего значения
sequence_object:alter()	Изменение параметров последовательности
sequence_object:reset()	Сброс состояния последовательности
sequence_object:set()	Установка нового значения
sequence_object:drop()	Удаление последовательности
space_object:create_index()	Создание индекса

`box.schema.space.create(`*space-name*[, *{options}*]`)`
`box.schema.create_space(`*space-name*[, *{options}*]`)`

Создание *спейса*.

Параметры

- `space-name` (`string`) – имя спейса, которое должно соответствовать *правилам именования объектов*
- `options` (`table`) – см. таблицу «Параметры для box.schema.space.create» ниже

возвращается объект спейса

тип возвращаемого значения пользовательские данные

Можно использовать любой вариант синтаксиса. Например, `s = box.schema.space.create('tester')` эквивалентно `s = box.schema.create_space('tester')`.

Параметры для box.schema.space.create

Имя	Эффект	Type	Значение по умолчанию
engine (движок)	„memtx“ или „vinyl“	string (строка)	„memtx“
field_count (количество полей)	заданное количество *полей*: например, если field_count=5, нельзя вставить кортеж с количеством полей, большим или меньшим, чем 5	число	0, то есть не задано
format (формат)	имена и типы полей: см. наглядные примеры операторов в описании *space_object:format()* и в *box.space._space*. Необязательный параметр, обычно значение не указывается.	таблица	(пустое)
id	уникальный идентификатор: пользователи могут ссылаться на спейсы посредством идентификатора вместо имени	число	идентификатор последнего спейса +1
if_not_exists (если отсутствует)	спейс создается, только если спейса с таким же именем нет в базе данных, в противном случае эффект отсутствует, но ошибка не выдается	boolean (логический)	false (ложь)
is_local	содержимое спейса *реплицируется локально*: изменения сохраняются в *журнале упреждающей записи* локального узла, но не происходит *репликация*.	boolean (логический)	false (ложь)
temporary (временный)	содержимое спейса хранится временно: изменения не хранятся в *журнале упреждающей записи*, и не проводится *репликация*. Примечание по движку базы данных: vinyl не поддерживает временные спейсы.	boolean (логический)	false (ложь)
user (пользователь)	имя пользователя, который считается *владельцем* спейса, для целей авторизации	string (строка)	имя текущего пользователя

Существуют три *варианта синтаксиса* для ссылок на объекты спейса, например, `box.schema.space.drop(`*`id-спейса`*`)` удалит спейс. Однако общий подход заключается в использовании функций, прикрепленных к объектам спейса, например *space_object:drop()*.

Пример

```
tarantool> s = box.schema.space.create('space55')
---
...
tarantool> s = box.schema.space.create('space55', {
         >   id = 555,
         >   temporary = false
         > })
---
- error: Space 'space55' already exists
...
tarantool> s = box.schema.space.create('space55', {
         >   if_not_exists = true
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        > })
---
...
```

Следующим шагом после создания спейса будет *создание индекса* для него, после чего можно будет выполнять вставку, выборку и другие функции *box.space*.

box.schema.user.create(*user-name*[, *{options}*])

Создание пользователя. Чтобы получить информацию о том, как происходит управление данными пользователя в Tarantool'е, см. раздел *Пользователи* и справочник по спейсу *_user*.

Возможные параметры:

- `if_not_exists` (если отсутствует) = `true|false` (правда/ложь, по умолчанию ложь) - логическое значение boolean; `true` (правда) означает, что ошибка не выпадет, если пользователь уже существует,
- `password` (пароль) – строка; указать `password` = *password* неплохо, поскольку в *URI* (унифицированный идентификатор ресурса) обычно нельзя включать имя пользователя без пароля.

Примечание: Максимальное количество пользователей – 32.

Параметры

- `user-name` (`string`) – имя пользователя, которое должно соответствовать *правилам именования объектов*
- `options` (`table`) – `if_not_exists`, `password`

возвращается nil

Примеры:

```
box.schema.user.create('Lena')
box.schema.user.create('Lena', {password = 'X'})
box.schema.user.create('Lena', {if_not_exists = false})
```

box.schema.user.drop(*user-name*[, *{options}*])

Удаление пользователя. Чтобы получить информацию о том, как происходит управление данными пользователя в Tarantool'е, см. раздел *Пользователи* и справочник по спейсу *_user*.

Параметры

- `user-name` (`string`) – имя пользователя
- `options` (`table`) – `if_exists` (если существует) = `true|false` (правда/ложь, по умолчанию ложь) - логическое значение boolean; `true` (правда) означает, что ошибка не выпадет, если такой пользователь не существует.

Примеры:

```
box.schema.user.drop('Lena')
box.schema.user.drop('Lena',{if_exists=false})
```

box.schema.user.exists(*user-name*)

Возврат `true` (правда), если пользователь существует; возврат `false` (ложь), если пользователь отсутствует. Чтобы получить информацию о том, как происходит управление данными пользователя в Tarantool'е, см. раздел *Пользователи* и справочник по спейсу *_user*.

Параметры

- `user-name` (`string`) – имя пользователя

тип возвращаемого значения логическое значение bool

Пример:

```
box.schema.user.exists('Lena')
```

`box.schema.user.grant`(*user-name*, *privileges*, *object-type*, *object-name*[, *{options}*])

`box.schema.user.grant`(*user-name*, *privileges*, *'universe'*[, *nil*, *{options}*])

`box.schema.user.grant`(*user-name*, *role-name*[, *nil*, *nil*, *{options}*])

Выдача *прав* пользователю или другой роли.

Параметры

- `user-name` (`string`) – имя пользователя.
- `privileges` (`string`) – „read“ (чтение) или „write“ (запись), или „execute“ (выполнение), или „create“ (создание), или „alter“ (изменение), или „drop“ (удаление) или их сочетание.
- `object-type` (`string`) – „space“ (спейс) или „function“ (функция), или „sequence“ (последовательность), или „role“ (роль).
- `object-name` (`string`) – имя объекта, на который выдаются права.
- `role-name` (`string`) – имя роли, которая назначается пользователю.
- `options` (`table`) – `grantor`, `if_not_exists`.

Если есть `'function','имя-объекта'`, то должен существовать кортеж в _func с таким именем объекта.

Вариант: вместо `тип-объекта, имя-объекта` введите „universe“, что означает „все типы объектов и все объекты“. В таком случае имя объекта опускается.

Вариант: вместо `права, тип-объекта, имя-объекта` введите `имя-роли` (см. раздел *Роли*).

Вариант: вместо `box.schema.user.grant('имя-пользователя','usage,session','universe', nil, {if_not_exists=true})` введите `box.schema.user.enable('имя-пользователя')`.

Возможные параметры:

- `grantor` = *grantor_ name_ or_ id* – строка или номер для заданного пользователя, выдающего права,
- `if_not_exists` (если отсутствует) = `true|false` (правда/ложь, по умолчанию ложь) - логическое значение boolean; `true` (правда) означает, что ошибка не выпадет, если у пользователя уже есть права.

Пример:

```
box.schema.user.grant('Lena', 'read', 'space', 'tester')
box.schema.user.grant('Lena', 'execute', 'function', 'f')
box.schema.user.grant('Lena', 'read,write', 'universe')
box.schema.user.grant('Lena', 'Accountant')
box.schema.user.grant('Lena', 'read,write,execute', 'universe')
box.schema.user.grant('X', 'read', 'universe', nil, {if_not_exists=true}))
```

`box.schema.user.revoke`(*user-name*, *privileges*, *object-type*, *object-name*[, *{options}*])

`box.schema.user.revoke`(*user-name*, *privileges*, *'universe'*[, *nil*, *{options}*])

box.schema.user.revoke(*user-name*, *role-name*[, *nil*, *nil*, *{options}*])

Отмена *прав* пользователя или другой роли.

Параметры

- `user-name` (`string`) – имя пользователя.
- `privilege` (`string`) – „read“ (чтение) или „write“ (запись), или „execute“ (выполнение), или „create“ (создание), или „alter“ (изменение), или „drop“ (удаление) или их сочетание.
- `object-type` (`string`) – „space“ (спейс) или „function“ (функция), или „sequence“ (последовательность).
- `object-name` (`string`) – имя функции, спейса или последовательности.
- `options` (`table`) – `if_exists`.

The user must exist, and the object must exist, but if the option setting is `{if_exists=true}` then it is not an error if the user does not have the privilege.

Вариант: вместо `тип-объекта, имя-объекта` введите „universe“, что означает „все типы объектов и все объекты“.

Вариант: вместо `права, тип-объекта, имя-объекта` введите `имя-роли` (см. раздел *Роли*).

Вариант: вместо box.schema.user.revoke('*имя-пользователя*','usage, session','universe',nil, {if_not_exists=true}) введите box.schema.user.disable('*имя-пользователя*').

Пример:

```
box.schema.user.revoke('Lena', 'read', 'space', 'tester')
box.schema.user.revoke('Lena', 'execute', 'function', 'f')
box.schema.user.revoke('Lena', 'read,write', 'universe')
box.schema.user.revoke('Lena', 'Accountant')
```

box.schema.user.password(*password*)

Возврат хеша пароля пользователя. Чтобы получить информацию о том, как происходит управление паролями в Tarantool'е, см. раздел *Пароли* и справочник по спейсу *_user*.

Примечание:

- Если у пользователя, который не является пользователем „guest“ нет пароля, **невозможно** подключиться к Tarantool'у через этого пользователя. Пользователь считается только “внутренним”, его нельзя использовать для удаленного подключения. Такие пользователи могут работать, если они определили какие-либо процедуры с помощью *SETUID*, на которые есть доступ у пользователей с внешним подключением. Таким образом, внешние пользователи могут не создавать/удалять объекты, а только вызывать процедуры.
- Для пользователя „guest“ невозможно установить пароль: это бы привело к путанице, поскольку „guest“ является пользователем по умолчанию для любого установленного подключения по *бинарному порту*, а Tarantool не требует пароль при установке *бинарного подключения*. Тем не менее, можно сменить текущего пользователя на пользователя 'guest', предоставив *AUTH-пакет* (пакет авторизации) без пароля или с пустым паролем. Данная функция полезна для пулов соединений, которые хотят повторно использовать соединение для другого пользователя без повторного подключения.

Параметры

- password (string) – пароль для хеширования

тип возвращаемого значения string (строка)

Пример:

```
box.schema.user.password('ЛЕНА')
```

box.schema.user.passwd([*user-name*], *password*)

Ассоциация пароля с авторизованным пользователем или с указанным именем пользователя. Такой пользователь должен существовать и не быть пользователем „guest“.

Если пользователь хочет поменять свой пароль, ему следует использовать синтаксис box.schema.user.passwd(password).

Если администратор хочет поменять пароль других пользователей, ему следует использовать синтаксис box.schema.user.passwd(user-name, password).

Параметры

- user-name (string) – имя пользователя
- password (string) – пароль

Пример:

```
box.schema.user.passwd('ЛЕНА')
box.schema.user.passwd('Lena', 'ЛЕНА')
```

box.schema.user.info([*user-name*])

Return a description of a user's *privileges*.

box.schema.role.create(*role-name*[, *{options}*])

Создание роли. Чтобы получить информацию о том, как происходит управление данными о ролях в Tarantool'е, см. раздел *Роли*.

Параметры

- role-name (string) – имя роли, которое должно соответствовать *правилам именования объектов*
- options (table) – if_not_exists (если отсутствует) = true|false (правда/ложь, по умолчанию ложь) - логическое значение boolean; true (правда) означает, что ошибка не выпадет, если роль уже существует.

возвращается nil

Пример:

```
box.schema.role.create('Accountant')
box.schema.role.create('Accountant', {if_not_exists = false})
```

box.schema.role.drop(*role-name*[, *{options}*])

Удаление роли. Чтобы получить информацию о том, как происходит управление данными о ролях в Tarantool'е, см. раздел *Роли*.

Параметры

- role-name (string) – имя роли
- options (table) – if_exists (если существует) = true|false (правда/ложь, по умолчанию ложь) - логическое значение boolean; true (правда) означает, что ошибка не выпадет, если такая роль не существует.

Пример:

```
box.schema.role.drop('Accountant')
```

box.schema.role.exists(*role-name*)
Возврат `true` (правда), если роль существует; возврат `false` (ложь), если роль отсутствует.

Параметры

- `role-name` (`string`) – имя роли

тип возвращаемого значения логическое значение bool

Пример:

```
box.schema.role.exists('Accountant')
```

box.schema.role.grant(*role-name*, *privilege*, *object-type*, *object-name*[, *option*])
box.schema.role.grant(*role-name*, *privilege*, *'universe'*[, *nil*, *option*])
box.schema.role.grant(*role-name*, *role-name*[, *nil*, *nil*, *option*])
Выдача *прав* роли.

Параметры

- `role-name` (`string`) – имя роли.
- `privilege` (`string`) – „read“ (чтение) или „write“ (запись), или „execute“ (выполнение), или „create“ (создание), или „alter“ (изменение), или „drop“ (удаление) или их сочетание.
- `object-type` (`string`) – „space“ (спейс) или „function“ (функция), или „sequence“ (последовательность), или „role“ (роль).
- `object-name` (`string`) – имя функции, спейса, последовательности или роли.
- `option` (`table`) – `if_not_exists` (если отсутствует) = `true|false` (правда/ложь, по умолчанию ложь) - логическое значение boolean; `true` (правда) означает, что ошибка не выпадет, если у роли уже есть права.

Должна существовать роль, должен существовать объект.

Вариант: вместо `тип-объекта, имя-объекта` введите „universe“, что означает „все типы объектов и все объекты“. В таком случае имя объекта опускается.

Вариант: вместо `privilege, object-type, object-name` введите `role-name`, чтобы назначить роль для роли.

Пример:

```
box.schema.role.grant('Accountant', 'read', 'space', 'tester')
box.schema.role.grant('Accountant', 'execute', 'function', 'f')
box.schema.role.grant('Accountant', 'read,write', 'universe')
box.schema.role.grant('public', 'Accountant')
box.schema.role.grant('role1', 'role2', nil, nil, {if_not_exists=false})
```

box.schema.role.revoke(*role-name*, *privilege*, *object-type*, *object-name*)
Отмена *прав* роли.

Параметры

- `role-name` (`string`) – имя роли.

- `privilege` (`string`) – „read“ (чтение) или „write“ (запись), или „execute“ (выполнение), или „create“ (создание), или „alter“ (изменение), или „drop“ (удаление) или их сочетание.
- `object-type` (`string`) – „space“ (спейс) или „function“ (функция), или „sequence“ (последовательность), или „role“ (роль).
- `object-name` (`string`) – имя функции, спейса, последовательности или роли.

Должна существовать роль, должен существовать объект, но ошибка не выпадет, если у роли нет прав.

Вариант: вместо `тип-объекта, имя-объекта` введите „universe“, что означает „все типы объектов и все объекты“.

Вариант: вместо `privilege, object-type, object-name` введите `role-name`.

Пример:

```
box.schema.role.revoke('Accountant', 'read', 'space', 'tester')
box.schema.role.revoke('Accountant', 'execute', 'function', 'f')
box.schema.role.revoke('Accountant', 'read,write', 'universe')
box.schema.role.revoke('public', 'Accountant')
```

`box.schema.role.info(`*role-name*`)`

Возврат описания прав роли.

Параметры

- `role-name` (`string`) – имя роли.

Пример:

```
box.schema.role.info('Accountant')
```

`box.schema.func.create(`*func-name*`[, {options-without-body}])`

Create a function *tuple*. without including the `body` option. (For functions created with the `body` option, see *box.schema.func.create(func-name [, {options-with-body}])*.

This is called a «not persistent» function because functions without bodies are not persistent. This does not create the function itself – that is done with Lua – but if it is necessary to grant privileges for a function, box.schema.func.create must be done first. For explanation of how Tarantool maintains function data, see the reference for the *box.space._func* space.

Возможные параметры:

- `if_not_exists` (если отсутствует) = `true|false` (правда/ложь, по умолчанию ложь) - логическое значение boolean; `true` (правда) означает, что ошибка не выпадет, если кортеж в `_func` уже существует.
- `setuid` = `true|false` (default = `false`) - boolean; `true` means that Tarantool should treat the function's caller as the function's owner, with owner privileges. `setuid` works only over *binary ports*, `setuid` does not work if the function is invoked via an *admin console* or inside a Lua script.
- `language` = „LUA“|“C“ (default = 'LUA') - string.

Параметры

- `func-name` (`string`) – имя функции, которое должно соответствовать *правилам именования объектов*
- `options` (`table`) – `if_not_exists`, `setuid`, `language`.

возвращается nil

Пример:

```
box.schema.func.create('calculate')
box.schema.func.create('calculate', {if_not_exists = false})
box.schema.func.create('calculate', {setuid = false})
box.schema.func.create('calculate', {language = 'LUA'})
```

`box.schema.func.create(`*func-name*[, *{options-with-body}*]`)`

Create a function *tuple*. including the `body` option. (For functions created without the `body` option, see *box.schema.func.create(func-name [, {options-without-body}])*.

This is called a «persistent» function because only functions with bodies are persistent. This does create the function itself, the body is a function definition. For explanation of how Tarantool maintains function data, see the reference for the *box.space._func* space.

Возможные параметры:

- `if_not_exists` = `true|false` (default = `false`) - boolean; same as for *box.schema.func.create(func-name [, {options-without-body}])*.
- `setuid` = `true|false` (default = `false`) - boolean; same as for *box.schema.func.create(func-name [, {options-without-body}])*.
- `language` = „LUA“|“C“ (default = ‘LUA’) - string. same as for *box.schema.func.create(func-name [, {options-without-body}])*.
- `is_sandboxed` = `true|false` (default = `false`) - boolean; whether the function should be executed in a sandbox.
- `is_deterministic` = `true|false` (default = `false`) - boolean; `true` means that the function should be deterministic, `false` means that the function may or may not be deterministic.
- `body` = function definition (default = nil) - string; the function definition.

Параметры

- `func-name` (`string`) – имя функции, которое должно соответствовать *правилам именования объектов*
- `options` (`table`) – `if_not_exists`, `setuid`, `language`, `is_sandboxed`, `is_deterministic`, `body`.

возвращается nil

C functions are imported from .so files, Lua functions can be defined within `body`. We will only describe Lua functions in this section.

A function tuple with a body is «persistent» because the tuple is stored in a snapshot and is recoverable if the server restarts. All of the option values described in this section are visible in the *box.space._func* system space.

If `is_sandboxed` is true, then the function will be executed in an isolated environment: any operation that accesses the world outside the sandbox will be forbidden or will have no effect. Therefore a sandboxed function can only use modules and functions which cannot affect isolation: assert, error, ipairs, math.*, next, pairs, pcall, print, select, string.*, table.*, tonumber, tostring, type, unpack, *utf8.**, xpcall. Also a sandboxed function cannot refer to global variables – they will be treated as local variables because the sandbox is established with setfenv. So a sandboxed function will happen to be stateless and deterministic.

If `is_deterministic` is true, there is no immediate effect. Tarantool plans to use the is_deterministic value in a future version. A function is deterministic if it always returns the same outputs given the same inputs. It is the function creator's responsibility to ensure that a function is truly deterministic.

Using a persistent Lua function

After a persistent Lua function is created, it can be found in the *box.space._func* system space, and it can be shown with `box.func.`*`func-name`* and it can be invoked by any user with *authorization* to „execute" it. The syntax for invoking is: `box.func.`*`func-name`*`:call([parameters])` or, if the connection is remote, the syntax is as in *net_box:call()*.

Пример:

```
tarantool> lua_code = [[function(a, b) return a + b end]]
tarantool> box.schema.func.create('sum', {body = lua_code})

tarantool> box.func.sum
---
- is_sandboxed: false
  is_deterministic: false
  id: 2
  setuid: false
  body: function(a, b) return a + b end
  name: sum
  language: LUA
...

tarantool> box.func.sum:call({1, 2})
---
- 3
...
```

`box.schema.func.drop(`*func-name*`[, {`*options*`}])`

Удаление кортежа с функцией. Чтобы получить информацию о том, как происходит управление данными функций в Tarantool'е, см. справочник по спейсу *_func*.

Параметры

- `func-name` (`string`) – имя функции
- `options` (`table`) – `if_exists` (если существует) = `true|false` (правда/ложь, по умолчанию ложь) - логическое значение boolean; `true` (правда) означает, что ошибка не выпадет, если кортеж в _func не существует.

Пример:

```
box.schema.func.drop('calculate')
```

`box.schema.func.exists(`*func-name*`)`

Возврат true (правда), если кортеж с функцией существует; возврат false (ложь), если кортеж с функцией отсутствует.

Параметры

- `func-name` (`string`) – имя функции

тип возвращаемого значения логическое значение bool

Пример:

```
box.schema.func.exists('calculate')
```

box.schema.func.reload([*name*])
Перезагрузка модуля на C (со всеми его функциями) без перезапуска сервера.

С точки зрения внутреннего устройства, Tarantool загружает новую копию модуля (библиотека общего пользования `*.so`) и запускает маршрутизацию всех новых запросов на новую версию. Предыдущая версия остается активной до тех пор, пока не завершатся все начатые вызовы. Все библиотеки общего пользования загружены с `RTLD_LOCAL` (см. «man 3 dlopen»), таким образом, множество копий могут работать одновременно без каких-либо проблем.

Примечание: Перезагрузка не сработает, если модуль был загружен из Lua-скрипта с ffi.load().

Параметры

- `name` (`string`) – имя модуля для перезагрузки

Пример:

```
-- перегрузить целиком всё содержимое модуля
box.schema.func.reload('module')
```

Последовательности

Вводная информация о последовательностях дается в разделе *Последовательности* главы «Модель данных». Здесь же приведена подробная информация о каждой функции и каждом параметре.

Все функции, связанные с последовательностями, требуют наличия соответствующих *прав*.

box.schema.sequence.create(*name*[, *options*])
Создание нового генератора последовательностей.

Параметры

- `name` (`string`) – имя последовательности
- `options` (`table`) – см. краткий обзор в *таблице* «Параметры для `box.schema.sequence.create()`» (в разделе *Последовательности* главы «Модель данных»), а более подробную информацию ниже.

возвращается ссылка на новый объект последовательности.

Параметры:

- `start` – НАЧАЛЬНОЕ значение. Тип = целое число, по умолчанию = 1.
- `min` – МИНИМАЛЬНОЕ значение. Тип = целое число, по умолчанию = 1.
- `max` –МАКСИМАЛЬНОЕ значение. Тип = целое число, по умолчанию = 9223372036854775807.

  Есть следующее правило: `min` <= `start` <= `max`. Например, нельзя указать `{start=0}`, поскольку указанное начальное значение (0) будет меньше, чем минимальное значение, используемое по умолчанию (1).

  Есть следующее правило: `min` <= следующее-значение <= `max`. Например, если сгенерированное значение будет 1000, но максимальное значение – 999, это будет считаться переполнением.

  There is a rule: `start` and `min` and `max` must all be <= 9223372036854775807 which is 2^63 - 1 (not 2^64).

- `cycle` – значение ЦИКЛА. Тип = bool (логический), по умолчанию = false (ложь).

  Если следующее значение в генераторе последовательности будет переполнением, это вызовет ошибку – не считая случаев, когда задан цикл (`cycle == true`).

  Если же `cycle == true`, отсчет начинается заново с МИНИМАЛЬНОГО значения или с МАКСИМАЛЬНОГО значения (не с НАЧАЛЬНОГО значения).

- `cache` – значение КЭША. Тип = беззнаковое целое число, по умолчанию = 0.

  В данный момент Tarantool игнорирует это значение, оно зарезервировано для последующего использования.

- `step` – значение УВЕЛИЧЕНИЯ. Тип = целое число, по умолчанию = 1.

  Это значение прибавляется к предыдущему.

`sequence_object:next()`
Генерация и возврат следующего значения.

Простой алгоритм для генерации:

- В первый раз вернуть НАЧАЛЬНОЕ значение.
- Если предыдущее значение плюс значение УВЕЛИЧЕНИЯ меньше, чем МИНИМАЛЬНОЕ значение, или больше, чем МАКСИМАЛЬНОЕ значение, будет переполнение, поэтому либо выдать сообщение об ошибке (если цикл не задан – `cycle` = `false`) или вернуть МАКСИМАЛЬНОЕ значение (если цикл задан – `cycle` = `true` – и `step` < 0), или вернуть МИНИМАЛЬНОЕ значение (если цикл задан – `cycle` = `true` – и `step` > 0).

Если ошибки нет, сохранить результат, который становится «предыдущим значением».

Например, предположим, что для последовательности „S“:

- `min` == -6,
- `max` == -1,
- `step` == -3,
- `start` = -2,
- `cycle` = true,
- предыдущее значение = -2.

Тогда `box.sequence.S:next()` вернет -5, потому что -2 + (-3) == -5.

Затем `box.sequence.S:next()` снова вернет -1, потому что -5 + (-3) < -6, что будет переполнением, которое вызовет цикл, а `max` == -1.

Для данной функции необходимы права на *запись („write“)* на последовательность.

Примечание: Данную функцию не следует использовать в транзакциях между движками (транзакции, в которых используется и движок memtx, и движок vinyl).

Чтобы увидеть предыдущее значение, не изменяя его, сделайте выборку из системного спейса *_sequence_data*.

`sequence_object:alter(`*options*`)`
Функцию `alter()` можно использовать для изменения любых параметров последовательности. Требования и ограничения в данном случае такие же, как для *box.schema.sequence.create()*.

`sequence_object:reset()`
Возврат последовательности в оригинальное состояние. Смысл в том, что последующий вызов `next()` вернет начальное значение `start`. Для данной функции необходимы права на *запись („write“)* на последовательность.

`sequence_object:set(`*new-previous-value*`)`
Установите «предыдущее значение» на `new-previous-value` (новое предыдущее значение). Для данной функции необходимы права на *запись („write“)* на последовательность.

`sequence_object:drop()`
Удаление существующей последовательности.

Пример:

Ниже представлен пример, иллюстрирующий все параметры и операции для последовательностей:

```
s = box.schema.sequence.create(
                'S2',
                {start=100,
                min=100,
                max=tonumber64('9223372036854775807'),
                cache=100000,
                cycle=false,
                step=100
                })
s:alter({step=6})
s:next()
s:reset()
s:set(150)
s:drop()
```

`space_object:create_index(`*... [sequence='...' option] ...*`)`
You can use the `sequence` option when *creating* or *altering* a primary-key index. The sequence becomes associated with the index, so that the next `insert()` will put the next generated number into the primary-key field, if the field value would otherwise be nil.

The syntax may be any of: `sequence = `*`sequence identifier`* or `sequence = {id = `*`sequence identifier`*` }` or `sequence = {field = `*`field number`*` }` or `sequence = {id = `*`sequence identifier`*`, field = `*`field number`*` }` or `sequence = true` or `sequence = {}`. The sequence identifier may be either a number (the sequence id) or a string (the sequence name). The field number may be the ordinal number of any field in the index; default = 1. Examples of all possibilities: `sequence = 1` or `sequence = 'sequence_name'` or `sequence = {id = 1}` or `sequence = {id = 'sequence_name'}` or `sequence = {id = 1, field = 1}` or `sequence = {id = 'sequence_name', field = 1}` or `sequence = {field = 1}` or `sequence = true` or `sequence = {}`. Notice that the sequence identifier can be omitted, if it is omitted then a new sequence is created automatically with default name = *`space-name`*`_seq`. Notice that the field number does not have to be 1, that is, the sequence can be associated with any field in the primary-key index.

Например, если „Q“ – это последовательность, а „T“ – это новый спейс, то сработает:

```
tarantool> box.space.T:create_index('Q',{sequence='Q'})
---
- unique: true
  parts:
  - type: unsigned
    is_nullable: false
    fieldno: 1
```

(continues on next page)

(продолжение с предыдущей страницы)

```
  sequence_id: 8
  id: 0
  space_id: 514
  name: Q
  type: TREE
...
```

(Обратите внимание, что теперь в индексе есть поле идентификатора последовательности `sequence_id`.)

И сработает:

```
tarantool> box.space.T:insert{box.NULL,0}
---
- [1, 0]
...
```

Примечание: The index key type may be either „integer“ or „unsigned“. If any of the sequence options is a negative number, then the index key type should be „integer“.

Users should not insert a value greater than 9223372036854775807, which is 2^63 - 1, in the indexed field. The sequence generator will ignore it.

Последовательность нельзя удалить, если она связана с индексом. Тем не менее, можно использовать *index_object:alter()*, чтобы показать, что последовательность не связана с индексом, например так `box.space.T.index.I:alter({sequence=false})`.

If a sequence was created automatically because the sequence identifier was omitted, then it will be dropped automatically if the index is altered so that `sequence=false`, or if the index is dropped.

`index_object:alter()` can also be used to associate a sequence with an existing index, with the same syntax for options.

When a sequence is used with an index based on a JSON path, inserted tuples must have all components of the path preceding the autoincrement field, and the autoincrement field. To achieve that use `box.NULL` rather than `nil`. Example:

```
s = box.schema.space.create('test')
s:create_index('pk', {parts = {{'[1].a.b[1]', 'unsigned'}}, sequence = true})
s:replace{} -- error
s:replace{{c = {}}} -- error
s:replace{{a = {c = {}}}} -- error
s:replace{{a = {b = {}}}} -- error
s:replace{{a = {b = {nil}}}} -- error
s:replace{{a = {b = {box.NULL}}}} -- ok
```

Вложенный модуль *box.session*

Общие сведения

Вложенный модуль `box.session` позволяет делать запросы состояния сессии, вносить записи во временную Lua-таблицу по отдельной сессии, отправлять экстренные сообщения и настраивать триггеры, которые сработают в начале или окончании сессии.

Сессия – это объект, связанный с каждым подключением клиента.

Индекс

Ниже приведен перечень всех функций и элементов модуля `box.session`.

Имя	Использование
box.session.id()	Получение идентификатора текущей сессии
box.session.exists()	Проверка наличия сессии
box.session.peer()	Получение адреса хоста и порта подключенного узла
box.session.sync()	Получение целочисленной константы sync
box.session.user()	Получение имени текущего пользователя
box.session.type()	Получение типа соединения или повода к действию
box.session.su()	Изменение текущего пользователя
box.session.uid()	Получение идентификатора текущего пользователя
box.session.euid()	Получение идентификатора текущего действующего пользователя
box.session.storage	Таблица с именами и значениями по сессии
box.session.on_connect()	Определение триггера для подключения
box.session.on_disconnect()	Определение триггера для отключения
box.session.on_auth()	Определение триггера для аутентификации
box.session.push()	Отправка внеполосного сообщения

`box.session.id()`

возвращается уникальный идентификатор (ID) для текущей сессии. Результатом может быть 0 или -1, что означает, что сессии нет.

тип возвращаемого значения число

`box.session.exists(`*id*`)`

возвращается true if the session exists, false if the session does not exist.

тип возвращаемого значения boolean

`box.session.peer(`*id*`)`

Данная функция сработает только в том случае, если есть подключенная программа, то есть если было выполнено подключение к отдельному экземпляру Tarantool'а.

возвращается Адрес хоста и порт подключенного узла, например «127.0.0.1:55457». Если существует сессия, но отсутствует подключение к отдельному экземпляру, вернется null. Команда выполняется на экземпляре сервера, поэтому «локальное имя» – это хост и порт экземпляра сервера, а «имя узла» – это хост и порт клиента.

тип возвращаемого значения string (строка)

Возможные ошибки: „session.peer(): сессия отсутствует“

`box.session.sync()`

возвращается значение целочисленной константы `sync`, используемой в бинарном протоколе. Это значение будет недействительным после отключения сессии.

тип возвращаемого значения число

This function is local for the request, i.e. not global for the session. If the connection behind the session is multiplexed, this function can be safely used inside the request processor.

`box.session.user()`

возвращается имя *текущего пользователя*

тип возвращаемого значения string (строка)

box.session.type()

возвращается тип соединения или повод к действию.

тип возвращаемого значения string (строка)

Возможные возвращаемые значения:

- „binary“ (бинарное), если подключение было выполнено по бинарному протоколу, например, к объекту с помощью *box.cfg{listen=...}*;
- „console“ (консоль), если подключение было выполнено по административной консоли, например, к объекту с помощью *console.listen*;
- „repl“ (репликация), если подключение было выполнено напрямую, например, при *использовании Tarantool'a в качестве клиента*;
- „applier“ (наложение), если действие происходит по причине *репликации*, независимо от типа подключения;
- „background“ (в фоне), если действие происходит в *фоновом файбере*, независимо от того, был ли Tarantool *запущен в фоновом режиме*.

`box.session.type()` используется для триггера при замене *on_replace()* на реплике – значение будет „applier“ только в том случае, если триггер был активирован по причине запроса, выполненного на мастере.

box.session.su(*user-name*[, *function-to-execute*])

Изменение *текущего пользователя* Tarantool'a – аналогично Unix-команде `su`.

Или, если указана выполняемая функция (function-to-execute), временное изменение *текущего пользователя* Tarantool'a во время выполнения функции – аналогично Unix-команде `sudo`.

Параметры

- `user-name` (`string`) – целевое имя пользователя
- `function-to-execute` – имя функции или определение функции. Дополнительные параметры могут передаваться в `box.session.su`, они будут интерпретироваться как параметры выполняемой функции.

Пример

```
tarantool> function f(a) return box.session.user() .. a end
---
...

tarantool> box.session.su('guest', f, '-xxx')
---
- guest-xxx
...

tarantool> box.session.su('guest',function(...) return ... end,1,2)
---
- 1
- 2
...
```

box.session.uid()

возвращается ID *текущего пользователя*.

тип возвращаемого значения число

У каждого пользователя есть уникальное имя (узнать с помощью *box.session.user()*) и уникальный идентификатор (узнать с помощью `box.session.uid()`). Значения хранятся вместе в спейсе `_user`.

`box.session.euid()`

возвращается рабочий ID *текущего пользователя*.

Аналогично *box.session.uid()*, за исключением двух случаев:

- Первый случай: если вызов `box.session.euid()` выполняется в рамках функции, вызываемой по *box.session.su(user-name, function-to-execute)* – в таком случае `box.session.euid()` вернет измененный идентификатор пользователя (пользователь, который указан в параметре `user-name` функции `su`), но `box.session.uid()` вернет идентификатор оригинального пользователя (пользователя, который вызывает функцию `su`).
- Второй случай: если вызов `box.session.euid()` выполняется в рамках функции по *box.schema.func.create(function-name, {setuid= true})*, и используется бинарный протокол – в таком случае `box.session.euid()` вернет идентификатор пользователя, который создал функцию «function-name», а `box.session.uid()` вернет идентификатор пользователя, который вызывает эту функцию «function-name».

тип возвращаемого значения число

Пример

```
tarantool> box.session.su('admin')
---
...
tarantool> box.session.uid(), box.session.euid()
---
- 1
- 1
...
tarantool> function f() return {box.session.uid(),box.session.euid()} end
---
...
tarantool> box.session.su('guest', f)
---
- - 1
  - 0
...
```

`box.session.storage`

Lua-таблица с произвольными неупорядоченными именами и значениями по сессии, которая хранится до конца сессии. Например, эту таблицу можно использовать для хранения текущих задач при работе с очередями сообщений в Tarantool'е.

Пример

```
tarantool> box.session.peer(box.session.id())
---
- 127.0.0.1:45129
...
tarantool> box.session.storage.random_memorandum = "Don't forget the eggs"
---
...
tarantool> box.session.storage.radius_of_mars = 3396
```

(continues on next page)

(продолжение с предыдущей страницы)

```
---
...
tarantool> m = ''
---
...
tarantool> for k, v in pairs(box.session.storage) do
         >   m = m .. k .. '='.. v .. ' '
         > end
---
...
tarantool> m
---
- 'radius_of_mars=3396 random_memorandum=Don't forget the eggs. '
...
```

box.session.on_connect([*trigger-function*[, *old-trigger-function*]])

Определение исполняемого триггера во время создания новой сессии при подключению по консоли *console.connect*. Функция с триггером будет первой исполняемой функцией после создания сессии. Если триггер не выполняется и выдает ошибку, эта ошибка отправляется на клиент, и подключение разрывается.

Параметры

- `trigger-function` (*function*) – функция, в которой будет триггер
- `old-trigger-function` (*function*) – существующая функция с триггером, которую заменит новая

возвращается nil или указатель функции

Если указаны параметры (nil, old-trigger-function), старый триггер будет удален.

Если не указан ни один параметр, ответом будет список существующих функций с триггером.

Подробная информация о характеристиках триггера находится в разделе *Триггеры*.

Пример

```
tarantool> function f ()
         >   x = x + 1
         > end
tarantool> box.session.on_connect(f)
```

Предупреждение: Если триггер всегда приводит к ошибке, подключение к серверу для его переустановки может стать невозможным.

box.session.on_disconnect([*trigger-function*[, *old-trigger-function*]])

Определение исполняемого триггера после отключения клиента. Если функция с триггером вызывает ошибку, то ошибка записывается в журнал, в противном случае записей не будет. Триггер вызывается во время сессии клиента и может получить доступ к свойствам сессии, как *box.session.id()*.

Начиная с версии 1.10, функция с триггером вызывается сразу же после прерывания сессии, даже если сделанные запросы не были выполнены.

Параметры

- `trigger-function` (*function*) – функция, в которой будет триггер

- `old-trigger-function` (*function*) – существующая функция с триггером, которую заменит новая

возвращается nil или указатель функции

Если указаны параметры (nil, old-trigger-function), старый триггер будет удален.

Если не указан ни один параметр, ответом будет список существующих функций с триггером.

Подробная информация о характеристиках триггера находится в разделе *Триггеры*.

Пример №1

```
tarantool> function f ()
         >   x = x + 1
         > end
tarantool> box.session.on_disconnect(f)
```

Пример №2

После следующей серии запросов экземпляр Tarantool'а запишет сообщение с помощью модуля *log* при подключении или отключении любого пользователя.

```
function log_connect ()
  local log = require('log')
  local m = 'Connection. user=' .. box.session.user() .. ' id=' .. box.session.id()
  log.info(m)
end

function log_disconnect ()
  local log = require('log')
  local m = 'Disconnection. user=' .. box.session.user() .. ' id=' .. box.session.id()
  log.info(m)
end

box.session.on_connect(log_connect)
box.session.on_disconnect(log_disconnect)
```

Вот что может быть записано в файл журнала при обычной установке:

```
2014-12-15 13:21:34.444 [11360] main/103/iproto I>
    Connection. user=guest id=3
2014-12-15 13:22:19.289 [11360] main/103/iproto I>
    Disconnection. user=guest id=3
```

`box.session.on_auth`([*trigger-function*[, *old-trigger-function*]])

Определение триггера, используемого во время *аутентификации*.

Вызов функции `on_auth` с триггером происходит в следующих обстоятельствах:

(1) Функция *console.connect* включает в себя проверку аутентификации всех пользователей, кроме „guest“. Вызов функции `on_auth` с триггером происходит после триггера `on_connect` только в том случае, если подключение было успешным.

(2) В *бинарном протоколе* есть отдельный *пакет для аутентификации*. В этом случае подключение и аутентификация считаются отдельными действиям.

В отличие от других типов триггеров, вызов функций с триггером `on_auth` происходит **до** события. Таким образом, функция с таким триггером, как `function auth_function () v = box.session.user(); end`, определит `v` как «guest», то есть имя пользователя до проведения аутентификации. Чтобы получить имя пользователя **после** проведения аутентификации, используйте специальный синтаксис: `function auth_function (user_name) v = user_name; end`

Если триггер не выполняется и выдает ошибку, эта ошибка отправляется на клиент, и подключение разрывается.

Параметры

- `trigger-function` (*function*) – функция, в которой будет триггер
- `old-trigger-function` (*function*) – существующая функция с триггером, которую заменит новая

возвращается nil или указатель функции

Если указаны параметры (nil, old-trigger-function), старый триггер будет удален.

Если не указан ни один параметр, ответом будет список существующих функций с триггером.

Подробная информация о характеристиках триггера находится в разделе *Триггеры*.

Пример 1

```
tarantool> function f ()
         >   x = x + 1
         > end
tarantool> box.session.on_auth(f)
```

Пример 2

Более сложный пример с двумя экземплярами сервера.

Первый экземпляр сервера настроен на прослушивание по порту 3301; имя пользователя по умолчанию – „admin“. Есть три триггера `on_auth`:

- В первом триггере есть функция без аргументов, которая только смотрит на `box.session.user()`.
- Во втором триггере есть функция с аргументом `user_name`, которая может смотреть на `box.session.user()` и `user_name`.
- В третьем триггере есть функция с аргументом `user_name` и аргументом `status`, которая может смотреть на `box.session.user()` и `user_name`, и‘‘status‘‘.

Второй экземпляр сервера подключится по *console.connect*, а затем отобразит переменные, определенные функциями с триггером.

```
-- На первом экземпляре сервера, прослушивание на котором настроено на порт 3301
box.cfg{listen=3301}
function function1()
  print('function 1, box.session.user()='..box.session.user())
  end
function function2(user_name)
  print('function 2, box.session.user()='..box.session.user())
  print('function 2, user_name='..user_name)
  end
function function3(user_name, status)
  print('function 3, box.session.user()='..box.session.user())
  print('function 3, user_name='..user_name)
  if status == true then
    print('function 3, status = true, authorization succeeded')
    end
  end
box.session.on_auth(function1)
box.session.on_auth(function2)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
box.session.on_auth(function3)
box.schema.user.passwd('admin')
```

```
-- На втором экземпляре сервера, который подключается по порту 3301
console = require('console')
console.connect('admin:admin@localhost:3301')
```

Теперь результат выглядит следующим образом:

```
function 3, box.session.user()=guest
function 3, user_name=admin
function 3, status = true, authorization succeeded
function 2, box.session.user()=guest
function 2, user_name=admin
function 1, box.session.user()=guest
```

box.session.push(*message*[, *sync*])

Создание внеполосного сообщения. Под внеполосным мы понимаем дополнительное сообщение, которое дополняет то, что отправляется в сети по обычным каналам. Хотя `box.session.push()` можно вызвать в любое время, на практике эта функция используется в сетях, настроенных с помощью *модуля net.box*, и вызывается сервером (на «удаленной системе с базой данных», если использовать нашу терминологию для net.box), а у клиента есть возможность принимать такие сообщения.

Функция возвращает ошибку, если сессия была прервана.

Параметры

- `message` (*any-Lua-type*) – что отправляется
- `sync` (*int*) – необязательный аргумент, который показывает информацию о сессии, полученную из предшествующего вызова *box_session:sync()*. Если не указать, по умолчанию используется текущее значение `box.session.sync()`.

тип возвращаемого значения {nil, ошибка} или true:

- Если результатом будет ошибка, то вернется `nil` вместе с объектом ошибки.
- Если результатом будет не ошибка, то вернется логическое значение `true` (правда).
- Если возвращается `true`, сообщение отправлено в буфер сети в виде *пакета* с кодом IPROTO_CHUNK (0x80).

Единственная задача сервера – вызвать `box.session.push()`, поскольку нет автоматического механизма, который показал бы, что сообщение получено.

Задача клиента заключается в том, чтобы проверять наличие таких сообщений после отправки чего-либо на сервер. Основные клиентские методы – *conn:call*, *conn:eval*, *conn:select*, *conn:insert*, *conn:replace*, *conn:update*, *conn:upsert*, *delete* – могут привести к отправке такого сообщения сервером.

Ситуация 1: когда клиент делает синхронный вызов со значением параметра `{async=false}` по умолчанию. Есть два необязательных дополнительных параметра: `on_push=`*`function-name`* и `on_push_ctx=`*`function-argument`*. Когда клиент получает внеполосное сообщение в сессии, он вызывает «имя-функции(аргумент-функции)». Например, с такими значениями параметров: `{on_push=table.insert, on_push_ctx=messages}` – клиент произведет вставку полученных данных в таблицу под названием „messages“.

Ситуация 2: когда клиент делает асинхронный вызов с измененным значением параметра {async=true}. Здесь не разрешены on_push и on_push_ctx, но сообщения можно увидеть путем вызова pairs() в цикле.

Осложненная ситуация 2: pairs() зависит от времени ожидания. Таким образом, есть необязательный аргумент – время ожидания для итерации. Если время ожидания истечет до получения нового сообщения или окончательного ответа, вернется ошибка. Чтобы проверить наличие ошибки, можно использовать первый параметр в цикле (если цикл начинается с «for i, message in future:pairs()», то первым параметром в цикле будет i). Если это будет box.NULL, то второй параметр (в нашем примере «message») – это объект ошибки.

Пример

```
-- Создайте две оболочки. В оболочке №1 настройте сервер, а
-- в нем функцию, которая содержит box.session.push:
box.cfg{listen=3301}
box.schema.user.grant('guest','read,write,execute','universe')
x = 0
fiber = require('fiber')
function server_function() x=x+1; fiber.sleep(1); box.session.push(x); end

-- В оболочке №2 подключитесь к серверу в качестве клиента, который
-- поддерживает Lua (как второй Tarantool-сервер, работающий
-- в качестве клиента), и создайте таблицу, в которую мы будем получать сообщения:
net_box = require('net.box')
conn = net_box.connect(3301)
messages_from_server = {}

-- В оболочке №2 удаленно вызовите функцию и получите
-- СИНХРОННОЕ внеполосное сообщение:
conn:call('server_function', {},
          {is_async = false,
           on_push = table.insert,
           on_push_ctx = messages_from_server})
messages_from_server
-- Через секунду, во время которой происходит запрос fiber.sleep()
-- в server_function, результат в таблице
--  messages_from_server будет следующим: 1. Проверим:
-- tarantool> messages_from_server
-- ---
-- - - 1
-- ...
-- Хорошо. Это означает, что box.session.push(x) сработала,
-- поскольку мы знаем, что x был 1.

-- В оболочке №2 удаленно вызовите ту же самую функцию
-- для получения АСИНХРОННОГО внеполосного сообщения. При этом мы не можем
-- использовать параметры on_push и on_push_ctx, но можем использовать pairs():
future = conn:call('server_function', {}, {is_async = true})
messages = {}
keys = {}
for i, message in future:pairs() do
    table.insert(messages, message) table.insert(keys, i) end
messages
future:wait_result(1000)
for i, message in future:pairs() do
    table.insert(messages, message) table.insert(keys, i) end
messages
```

(continues on next page)

(продолжение с предыдущей страницы)

```
-- Задержки нет, поскольку conn:call не ждет
-- окончания вызова функции server_function. После первой итерации
-- цикла pairs(), видим, что таблица пуста. Это выглядит так:
-- tarantool> messages
-- ---
-- - - 2
--   - []
-- ...
-- Это нормально, поскольку сервер еще не вызвал
-- box.session.push(). При второй итерации
-- цикла pairs(), видим значение x во время
-- второго вызова box.session.push(). Так:
-- tarantool> messages
-- ---
-- - - 2
--   - &0 []
--   - 2
--   - *0
-- ...
-- Хорошо. Это означает, что сообщение было асинхронным, и
-- box.session.push() выполнила свою задачу.
```

Вложенный модуль *box.slab*

Общие сведения

Вложенный модуль `box.slab` предоставляет доступ к статистике распределения slab. Механизм распределения slab представляет собой основной тип распределения для хранения *кортежей*. Такое распределение можно использовать для отслеживания использования памяти и фрагментации памяти.

Индекс

Ниже приведен перечень всех функций модуля `box.slab`.

Имя	Использование
box.runtime.info()	Отображение отчета по использованию памяти во время исполнения Lua-кода
box.slab.info()	Отображение обобщенного отчета по использованию памяти для распределения slab
box.slab.stats()	Отображение подробного отчета по использованию памяти для распределения slab

`box.runtime.info()`

Отображение отчета по использованию памяти (в байтах) во время исполнения Lua-кода.

возвращается

- `lua` – это размер динамической памяти сборщика мусора в Lua;
- `maxalloc` – это максимальная квота памяти, которую можно выделить для Lua;
- `used` – объем памяти, используемый Lua в данный момент.

тип возвращаемого значения таблица

Пример:

```
tarantool> box.runtime.info()
---
- lua: 913710
  maxalloc: 4398046510080
  used: 12582912
...
tarantool> box.runtime.info().used
---
- used: 12582912
...
```

`box.slab.info()`

Отображение обобщенного отчета по использованию памяти (в байтах) для распределения slab. Данный отчет используется для оценки риска нехватки памяти.

`box.slab.info` выдает несколько показателей:

- items_used_ratio
- arena_used_ratio
- quota_used_ratio

При мониторинге используемой памяти в memtx'е есть два возможных сценария:

1 сценарий: $0.5 <$ `items_used_ratio` < 0.9

	Arena_used_ratio > 0.5	Arena_used_ratio > 0.9
Quota_used_ratio > 0.5		
Quota_used_ratio > 0.9		

Очевидно, память сильно фрагментирована. Проверьте, сколько у вас классов slab, подсчитав количество различных классов с помощью `box.slab.stats()`. Если классов slab много (больше нескольких десятков), то память может закончиться, даже если её занято не так много. На каждом slab может быть использовано мало элементов. Но всякий раз при выделении кортежа, размер которого отличается от любого существующего класса, Tarantool'у может понадобиться новый slab из области распределения slab. И если осталось мало пустых slab, то произойдет попытка увеличения квоты, что, в свою очередь, может привести к ошибке нехватки памяти из-за низкой оставшейся квоты памяти.

2 сценарий: `items_used_ratio` > 0.9

	Arena_used_ratio > 0.5	Arena_used_ratio > 0.9
Quota_used_ratio > 0.5		
Quota_used_ratio > 0.9		

Память заканчивается. Высокие показатели использования памяти. Память не фрагментирована, но каждый уровень механизма распределения slab почти пуст. Следует подумать об увеличении лимита памяти Tarantool'а (“box.cfg.memtx_memory‘).

Вывод: основной показатель нехватки памяти – `quota_used_ratio`. Тем не менее, существует множество абсолютно стабильных установок с высоким показателем `quota_used_ratio` , поэтому

необходимо обращать на это внимание, когда два других показателя также высоки (arena и item used).

возвращается

- `items_size` – это *общий* объем памяти (включая выделенные, но в данный момент свободные slab'ы), который используется только для кортежей, а не для индексов;
- `items_used_ratio` = `items_used` / `items_size`, where `items_size` = `slab_count` * `slab_size` (these are slabs used only for tuples, no indexes);
- `quota_size` – максимальный объем памяти, который механизм распределения slab может использовать как для кортежей, так и для индексов (как настроено в параметре *memtx_memory*, по умолчанию 2^28 байтов = 268 435 456 байтов);
- `quota_used_ratio` = `quota_used` / `quota_size`;
- `arena_used_ratio` = `arena_used` / `arena_size`;
- `items_used` – это *эффективный* объем памяти (не включая выделенные, но в данный момент свободные slab'ы), который используется только для кортежей, а не для индексов;
- `quota_used` – это объем памяти, уже выделенный для распределения slab;
- `arena_size` – это *общий* объем памяти, используемый для кортежей и индексов (включая выделенные, но в данный момент свободные slab'ы);
- `arena_used` – это *эффективный* объем памяти, используемый для кортежей и индексов (не включая выделенные, но в данный момент свободные slab'ы).

тип возвращаемого значения таблица

Пример:

```
tarantool> box.slab.info()
---
- items_size: 228128
  items_used_ratio: 1.8%
  quota_size: 1073741824
  quota_used_ratio: 0.8%
  arena_used_ratio: 43.2%
  items_used: 4208
  quota_used: 8388608
  arena_size: 2325176
  arena_used: 1003632
...

tarantool> box.slab.info().arena_used
---
- 1003632
...
```

`box.slab.stats()`

Отображение подробного отчета об использовании памяти (в байтах) для распределения slab. Отчет разбивается на группы по *размеру элементов данных*, а также по *размеру slab'а* (64 байта, 136 байтов и т.д.). Отчет включает в себя информацию о памяти, выделенной на хранение и кортежей, и индексов.

возвращается

- `mem_free` – это выделенная, но не используемая в данный момент память;
- `mem_used` – это память, используемая для хранения элементов данных (кортежей и индексов);
- `item_count` – это количество хранимых элементов;
- `item_size` – это размер каждого элемента данных;
- `slab_count` – это количество выделенных slab'ов;
- `slab_size` – это размер каждого выделенного slab'а.

тип возвращаемого значения таблица

Пример:

Ниже представлен пример отчета для первой группы:

```
tarantool> box.slab.stats()[1]
---
- mem_free: 16232
  mem_used: 48
  item_count: 2
  item_size: 24
  slab_count: 1
  slab_size: 16384
...
```

В отчете показано, что есть два элемента данных (`item_count` = 2), которые хранятся в одном (`slab_count` = 1) 24-байтовом slab'е (`item_size` = 24), поэтому объем используемой памяти `mem_used` = 2 * 24 = 48 байтов. Кроме того, размер slab'а `slab_size` составляет 16384 байта, из которых 16384 - 48 = 16232 байта свободны (`mem_free`).

В полном отчете будет статистика по использованию памяти во всех группах:

```
tarantool> box.slab.stats()
---
- - mem_free: 16232
    mem_used: 48
    item_count: 2
    item_size: 24
    slab_count: 1
    slab_size: 16384
  - mem_free: 15720
    mem_used: 560
    item_count: 14
    item_size: 40
    slab_count: 1
    slab_size: 16384
  <...>
  - mem_free: 32472
    mem_used: 192
    item_count: 1
    item_size: 192
    slab_count: 1
    slab_size: 32768
  - mem_free: 1097624
    mem_used: 999424
    item_count: 61
    item_size: 16384
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    slab_count: 1
    slab_size: 2097152
...
```

Общий объем используемой памяти `mem_used` для всех групп в данном отчете равен `arena_used` в отчете *box.slab.info()*.

Вложенный модуль *box.space*

Общие сведения

CRUD operations in Tarantool are implemented by the `box.space` submodule. It has the data-manipulation functions `select`, `insert`, `replace`, `update`, `upsert`, `delete`, `get`, `put`. It also has members, such as id, and whether or not a space is enabled. Submodule source code is available in file src/box/lua/schema.lua.

Индекс

Ниже приведен перечень всех функций и элементов модуля `box.space`.

Имя
space_object:auto_increment()
space_object:bsize()
space_object:count()
space_object:create_index() * *Details about index field types* * *Index field types to use in space_object:create_index()* * *A*
space_object:delete()
space_object:drop()
space_object:format()
space_object:frommap()
space_object:get()
space_object:insert()
space_object:len()
space_object:on_replace()
space_object:before_replace()
space_object:pairs()
space_object:put()
space_object:rename()
space_object:replace()
space_object:run_triggers()
space_object:select()
space_object:truncate()
space_object:update()
space_object:upsert()
space_object:user_defined()
space_object:create_check_constraint()
space_object.enabled
space_object.field_count
space_object.id
space_object.index

Имя
box.space._cluster
box.space._func
box.space._index
box.space._vindex
box.space._priv
box.space._vpriv
box.space._schema
box.space._sequence
box.space._sequence_data
box.space._space
box.space._vspace
box.space._user
box.space._ck_constraint
box.space._vuser
box.space._collation
box.space._vcollation

`object space_object`

`space_object:auto_increment(`*tuple*`)`

Вставка нового кортежа, используя первичный ключ с автоматическим увеличением. В спейсе, указанном через space_object должен быть первичный TREE-индекс типа *„unsigned“ или „integer“, или „number“*. Поле первичного ключа будет увеличиваться перед вставкой.

Данный метод объявлен устаревшим с версии 1.7.5 – лучше использовать *последовательности*.

Параметры

- `space_object` (*space_object*) – *ссылка на объект*
- `tuple` (*table/tuple*) – поля кортежа, не включая поле первичного ключа

возвращается вставленный кортеж.

тип возвращаемого значения кортеж

Факторы сложности Размер индекса, тип индекса, количество кортежей, к которым получен доступ, *настройки журнала упреждающей записи (WAL)*.

Возможные ошибки:

- неподходящий тип индекса;
- проиндексированное поле первичного ключа не является числовым.

Пример:

```
tarantool> box.space.tester:auto_increment{'Fld#1', 'Fld#2'}
---
- [1, 'Fld#1', 'Fld#2']
...
tarantool> box.space.tester:auto_increment{'Fld#3'}
---
- [2, 'Fld#3']
...
```

space_object:bsize()

Параметры

- space_object (*space_object*) – *ссылка на объект*

возвращается Количество байтов в спейсе. Это число, которое хранится во внутренней памяти Tarantool'а, представляет собой общее количество байтов во всех кортежах, включая ключи индекса. Для получения информации об измерении размера индекса, см. *index_object:bsize()*.

Пример:

```
tarantool> box.space.tester:bsize()
---
- 22
...
```

space_object:count(*key* [, *iterator*])

Возврат количества кортежей. Если сравнивать с *len()*, то данный метод работает медленнее, поскольку метод count() сканирует весь спейс для подсчета кортежей.

Параметры

- space_object (*space_object*) – *ссылка на объект*
- key (*scalar/table*) – значения поля первичного ключа, которые должны возвращаться в виде Lua-таблицы, если ключ составной
- iterator – метод сопоставления

возвращается Количество кортежей.

Пример:

```
tarantool> box.space.tester:count(2, {iterator='GE'})
---
- 1
...
```

space_object:create_index(*index-name* [, *options*])

Создание индекса. Индекс обязательно должен создаваться для спейса до вставки в него кортежей или выборки. Первый созданный индекс, который будет использоваться в качестве первичного индекса, должен быть уникальным.

Параметры

- space_object (*space_object*) – *ссылка на объект*
- index_name (string) – имя индекса, которое должно соответствовать *правилам именования объектов*
- options (table) – см. «Параметры для space_object:create_index()» ниже

возвращается объект индекса

тип возвращаемого значения объект индекса

Параметры для space_object:create_index()

Имя	Эффект	Type	Значение по умолчанию
type	тип индекса	строка („HASH“ или „TREE“, или „BITSET“, или „RTREE“) Примечание про движок базы данных: vinyl поддерживает только „TREE“	„TREE“
id	уникальный идентификатор	число	идентификатор последнего индекса +1
unique	индекс уникален	boolean (логический)	`true` (правда)
if_not_exists (если отсутствует)	ошибки нет, если имя дублируется	boolean (логический)	`false` (ложь)
parts	номера поля + типы	{field_no, `'unsigned'` or `'string'` or `'integer'` or `'number'` or `'boolean'` or `'varbinary'` or `'array'` or `'scalar'`, and optional collation or is_nullable value or path}	`{field = 1, type = 'unsigned'}`
dimension	только для *RTREE*	число	2
distance	только для RTREE	строка („euclid“ или „manhattan“)	„euclid“ (Евклидова)
bloom_fpr	только для vinyl	число	`vinyl_bloom_fpr`
page_size	только для vinyl	число	`vinyl_page_size`
range_size	только для vinyl	число	`vinyl_range_size`
run_count_per_level	только для vinyl	число	`vinyl_run_count_per_level`
run_size_ratio	только для vinyl	число	`vinyl_run_size_ratio`
sequence	см. раздел об *указании последовательности для create_index()*	строка или число	отсутствует
func	*functional index*	string (строка)	отсутствует

Параметры в вышеуказанной таблице также применимы к *index_object:alter()*.

Примечание про движок базы данных: в vinyl'е есть дополнительные параметры, которые по умолчанию основаны на конфигурационных параметрах *vinyl_bloom_fpr*, *vinyl_page_size*, *vinyl_range_size*, *vinyl_run_count_per_level* и *vinyl_run_size_ratio* – см. описание этих параметров. Текущие значения можно увидеть, сделав выборку из *box.space._index*.

Building or rebuilding a large index will cause occasional *yields* so that other requests will not be

blocked. If the other requests cause an illegal situation such as a duplicate key in a unique index, the index building or rebuilding will fail.

Возможные ошибки:

- слишком много частей;
- индекс „. . .“ уже существует;
- первичный ключ должен быть уникальным.

```
tarantool> s = box.space.tester
---
...
tarantool> s:create_index('primary', {unique = true, parts = { {field = 1, type =
↪'unsigned'}, {field = 2, type = 'string'}} })
---
...
```

Подробнее о типах полей индекса:

Восемь типов полей индекса (unsigned | string | integer | number | boolean | varbinary | array | scalar) отличаются друг от друга возможными значениями и типами индексов, где можно использовать такие поля.

- **unsigned**: беззнаковые целые числа от 0 до 18 446 744 073 709 551 615, т.е. около18 квинтиллионов. Также может называться „uint“ или „num“, но „num“ объявлен устаревшим. Используется в индексах типа TREE или HASH в memtx'e, и в TREE-индексах в vinyl'e.
- **string**: строка, то есть любая последовательность октетов до *максимальной длины*. Также может называться „str“. Используется в индексах типа TREE, HASH или BITSET в memtx'e и в TREE-индексах в vinyl'e. В строке может быть *сортировка*.
- **integer**: целые числа от -9 223 372 036 854 775 808 до 18 446 744 073 709 551 615. Также может называться „int“. Используется в индексах типа TREE или HASH в memtx'e и в TREE-индексах в vinyl'e.
- **number**: целые числа от -9 223 372 036 854 775 808 до 18 446 744 073 709 551 615, числа с плавающей запятой с одинарной точностью или с двойной точностью. Используется в индексах типа TREE или HASH в memtx'e и в TREE-индексах в vinyl'e.
- **boolean**: логическое значение, true (правда) или false (ложь). Используется в индексах типа TREE или HASH в memtx'e и в TREE-индексах в vinyl'e.
- **varbinary**: any set of octets, up to the *maximum length*. Legal in memtx TREE or HASH indexes, and in vinyl TREE indexes. A varbinary byte sequence does not have a *collation* because its contents are not UTF-8 characters.
- **array**: массив чисел. Используется в *RTREE-индексах* в memtx'e.
- **scalar**: null (input with `msgpack.NULL` or `yaml.NULL` or `json.NULL`), booleans (true or false), or integers between -9223372036854775808 and 18446744073709551615, or single-precision floating point numbers, or double-precison floating-point numbers, or strings. When there is a mix of types, the key order is: null, then booleans, then numbers, then strings. Legal in memtx TREE or HASH indexes, and in vinyl TREE indexes.

Кроме того, допускается нулевое значение *nil* для любого типа поля, если указана такая возможность *is_nullable=true*.

Типы полей в индексах для использования в space_object:create_index()

Тип поля для индексирования	Чем может быть	Где может использоваться	Примеры
unsigned	целые числа от 0 до 18 446 744 073 709 551 615	индексы типа TREE или HASH в memtx'e, TREE-индексы в vinyl'e	123456
string	строки – любой набор октетов	индексы типа TREE или HASH в memtx'e TREE-индексы в vinyl'e	„A B C“ „\65 \66 \67“
varbinary	последовательности байтов – любой набор октетов	индексы типа TREE или HASH в memtx'e TREE-индексы в vinyl'e	„\65 \66 \67“
integer	целые числа от -9 223 372 036 854 775 808 до 18 446 744 073 709 551 615	индексы типа TREE или HASH в memtx'e, TREE-индексы в vinyl'e	-2^63
number	целые числа от -9 223 372 036 854 775 808 до 18 446 744 073 709 551 615, числа с плавающей запятой с одинарной точностью или с двойной точностью	индексы типа TREE или HASH в memtx'e, TREE-индексы в vinyl'e	1.234 -44 1.447e+44
boolean	true или false	индексы типа TREE или HASH в memtx'e, TREE-индексы в vinyl'e	false true
array	массив целых чисел от -9 223 372 036 854 775 808 до 9 223 372 036 854 775 807	RTREE-индексы в memtx'e	{10, 11} {3, 5, 9, 10}
scalar	null, booleans (true or false), integers between -9223372036854775808 and 18446744073709551615, single-precision floating point numbers, double-precision floating point numbers, strings	индексы типа TREE или HASH в memtx'e, TREE-индексы в vinyl'e	null true -1 1.234 „“ „ру“

Разрешение использования нулевых значений для индексируемого ключа: /Если тип индекса – TREE, и индекс не является первичным, то оператор `parts={...}` может включать в себя `is_nullable=true` или `is_nullable=false` (по умолчанию). Если значение параметра `is_nullable` – true, то можно вставлять `nil` или аналогичное значение, например `msgpack.NULL` (или можно не вставлять вообще ничего в завершающие ненулевые поля). В рамках индекса такие нулевые значения считаются равными другим нулевым значениям и всегда меньше ненулевых значений. Нулевые значения могут встречаться несколько раз даже в уникальном индексе.

Например:

```
box.space.tester:create_index('I',{unique=true,parts={{field = 2, type = 'number', is_
↪nullable = true}}})
```

Предупреждение: Можно создать множество индексов для одного и того же поля с различными значениями `is_nullable` или вызвать *space_object:format()* со значением `is_nullable`, отличным от используемого для индекса. При наличии несоответствий правило такое: запрещается использовать null кроме случаев, когда `is_nullable=true` для всех индексов и формата спейса.

Использование имен полей вместо номеров полей: в `create_index()` можно использовать имена полей и/или типы полей, описанные в необязательном операторе *space_object:format()*. В следующем примере покажем `format()` для спейса с двумя столбцами под названиями „x“ и „y“, а затем покажем пять вариантов оператора `parts={}` в `create_index()`, сначала для столбца „x“, затем для столбцов „x“ и „y“. Варианты включают в себя пропуск типа, использование номеров и добавление дополнительных фигурных скобок.

```
box.space.tester:format({{name='x', type='scalar'}, {name='y', type='integer'}})
box.space.tester:create_index('I2',{parts={{'x', 'scalar'}}})
box.space.tester:create_index('I3',{parts={{'x','scalar'},{'y','integer'}}})
box.space.tester:create_index('I4',{parts={{1,'scalar'}}})
box.space.tester:create_index('I5',{parts={{1,'scalar'},{2,'integer'}}})
box.space.tester:create_index('I6',{parts={1}})
box.space.tester:create_index('I7',{parts={1,2}})
box.space.tester:create_index('I8',{parts={'x'}})
box.space.tester:create_index('I9',{parts={'x','y'}})
box.space.tester:create_index('I10',{parts={{'x'}}})
box.space.tester:create_index('I11',{parts={{'x'},{'y'}}})
```

Using the path option for map fields (JSON-indexes): To create an index for a field that is a map (a path string and a scalar value), specify the path string during index_create, that is, `parts={` *field-number* `,'` *data-type* `',path = '` *path-name* `' }`. The index type must be `'tree'` or `'hash'` and the field's contents must always be maps with the same path.

```
-- Example 1 -- The simplest use of path:
-- Result will be - - [{'age': 44}]
box.schema.space.create('T')
box.space.T:create_index('I',{parts={{field = 1, type = 'scalar', path = 'age'}}})
box.space.T:insert{{age=44}}
box.space.T:select(44)
-- Example 2 -- path plus format() plus JSON syntax to add clarity
-- Result will be: - [1, {'FIO': {'surname': 'Xi', 'firstname': 'Ahmed'}}]
s = box.schema.space.create('T')
format = {{'id', 'unsigned'}, {'data', 'map'}}
s:format(format)
parts = {{'data.FIO["firstname"]', 'str'}, {'data.FIO["surname"]', 'str'}}
i = s:create_index('info', {parts = parts})
s:insert({1, {FIO={firstname='Ahmed', surname='Xi'}}})
```

Примечание про движок базы данных: vinyl поддерживает только TREE-индексы, и следует создать в vinyl'е вторичные индексы до вставки кортежей.

Using the path option with [*] The string in a path option can contain „[*]“ which is called an array index placeholder. Indexes defined with this are useful for JSON documents that all have the same structure. For example, when creating an index on field#2 for a string document that will

start with `{'data': [{'name': '...'}, {'name': '...'}]`, the parts section in the create_index request could look like: `parts = {{field = 2, type = 'str', path = 'data[*].name'}}`. Then tuples containing names can be retrieved quickly with `index_object:select({key-value})`. In fact a single field can have multiple keys, as in this example which retrieves the same tuple twice because there are two keys „A“ and „B“ which both match the request:

```
s = box.schema.space.create('json_documents')
s:create_index('primarykey')
i = s:create_index('multikey', {parts = {{field = 2, type = 'str', path = 'data[*].name'}}})
s:insert({1,
         {data = {{name='A'},
                  {name='B'}},
          extra_field = 1}})
i:select({''},{iterator='GE'})
```

The result of the select request looks like this:

```
tarantool> i:select({''},{iterator='GE'})
---
- - [1, {'data': [{'name': 'A'}, {'name': 'B'}], 'extra_field': 1}]
- [1, {'data': [{'name': 'A'}, {'name': 'B'}], 'extra_field': 1}]
...
```

Some restrictions exist: () „[*]“ must be alone or must be at the end of a name in the path; () „[*]“ must not appear twice in the path; () if an index has a path with x[*] then no other index can have a path with x.component; () „[*]“ must not appear in the path of a primary-key ; () if an index has `unique=true` and has a path with „[*]“ then duplicate keys from different tuples are disallowed but duplicate keys for the same tuple are allowed; () As with *Using the path option for map fields*, the field's value must have the structure that the path definition implies, or be nil (nil is not indexed).

Making a functional index with space_object:create_index()

Functional indexes are indexes that call a user-defined function for forming the index key, rather than depending entirely on the Tarantool default formation. Functional indexes are useful for condensing or truncating or reversing or any other way that users want to customize the index.

The function definition must expect a tuple (which has the contents of fields at the time a data-change request happens) and must return a tuple (which has the contents that will actually be put in the index).

The space must have a memtx engine. The function must be *persistent* and deterministic. The key parts must not depend on JSON paths. The `create_index` definition must include specification of all key parts, and the function must return a table which has the same number of key parts with the same types. The function must access key-part values by index, not by field name. Functional indexes must not be primary-key indexes. Functional indexes cannot be altered and the function cannot be changed if it is used for an index, so the only way to change them is to drop the index and create it again. Only sandboxed functions are suitable for functional indexes.

Пример:

A function could make a key using only the first letter of a string field.

```
-- Step 1: Make the space.
-- The space needs a primary-key field, which is not the field that we
-- will use for the functional index.
box.schema.space.create('x', {engine = 'memtx'})
box.space.x:create_index('i',{parts={{field = 1, type = 'string'}}})
-- Step 2: Make the function.
```

(continues on next page)

(продолжение с предыдущей страницы)

```
-- The function expects a tuple. In this example it will work on tuple[2]
-- because the key souce is field number 2 in what we will insert.
-- Use string.sub() from the string module to get the first character.
lua_code = [[function(tuple) return {string.sub(tuple[2],1,1)} end]]
-- Step 3: Make the function persistent.
-- Use the box.schema.func.create function for this.
box.schema.func.create('F',
    {body = lua_code, is_deterministic = true, is_sandboxed = true})
-- Step 4: Make the functional index.
-- Specify the fields whose values will be passed to the function.
-- Specify the function.
box.space.x:create_index('j',{parts={{field = 1, type = 'string'}},func = 'F'})
-- Step 5: Test.
-- Insert a few tuples.
-- Select using only the first letter, it will work because that is the key
-- Or, select using the same function as was used for insertion
box.space.x:insert{'a', 'wombat'}
box.space.x:insert{'b', 'rabbit'}
box.space.x.index.j:select('w')
box.space.x.index.j:select(box.func.F:call({{'x', 'wombat'}}));
```

The results of the two `select` requests will look like this:

```
tarantool> box.space.x.index.j:select('w')
---
- - ['a', 'wombat']
...
tarantool> box.space.x.index.j:select(box.func.F:call({{'x','wombat'}}));
---
- - ['a', 'wombat']
...
```

Functions for functional indexes can return multiple keys. Such functions are called «multikey» functions. The `box.func.create` options must include `opts = {is_multikey = true}`. The return value must be a table of tuples. If a multikey function returns N tuples, then N keys will be added to the index.

Пример:

```
s = box.schema.space.create('withdata')
s:format({{name = 'name', type = 'string'},
          {name = 'address', type = 'string'}})
pk = s:create_index('name', {parts = {{field = 1, type = 'string'}}})
lua_code = [[function(tuple)
               local address = string.split(tuple[2])
               local ret = {}
               for _, v in pairs(address) do
                 table.insert(ret, {utf8.upper(v)})
               end
               return ret
             end]]
box.schema.func.create('address',
                       {body = lua_code,
                        is_deterministic = true,
                        is_sandboxed = true,
                        opts = {is_multikey = true}})
```

(continues on next page)

(продолжение с предыдущей страницы)

```
idx = s:create_index('addr', {unique = false,
                              func = 'address',
                              parts = {{field = 1, type = 'string',
                                        collation = 'unicode_ci'}}})
s:insert({"James", "SIS Building Lambeth London UK"})
s:insert({"Sherlock", "221B Baker St Marylebone London NW1 6XE UK"})
idx:select('Uk')
-- Both tuples will be returned.
```

space_object:delete(*key*)

Удаление кортежа по первичному ключу.

Параметры

- space_object (*space_object*) – *ссылка на объект*
- key (*scalar/table*) – значения поля первичного ключа, которые должны возвращаться в виде Lua-таблицы, если ключ составной

возвращается удаленный кортеж.

тип возвращаемого значения кортеж

Факторы сложности: Размер индекса, тип индекса

Примечание про движок базы данных: vinyl вернет nil, а не удаленный кортеж.

Пример:

```
tarantool> box.space.tester:delete(1)
---
- [1, 'My first tuple']
...
tarantool> box.space.tester:delete(1)
---
...
tarantool> box.space.tester:delete('a')
---
- error: 'Supplied key type of part 0 does not match index part type:
  expected unsigned'
...
```

Для получения дополнительной информации о сценариях использования и типичных ошибках, см. *Пример: использование операций с данными* далее в разделе.

space_object:drop()

Удаление спейса. Метод реализуется в фоновом режиме и не блокирует последующие запросы.

Параметры

- space_object (*space_object*) – *ссылка на объект*

возвращается nil

Возможные ошибки: space_object не существует.

Факторы сложности Размер индекса, тип индекса, количество кортежей, к которым получен доступ, настройки журнала упреждающей записи (WAL).

Пример:

```
box.space.space_that_does_not_exist:drop()
```

space_object:format([*format-clause*])

Объявление имен и *типов* полей.

Параметры

- space_object (*space_object*) – *ссылка на объект*
- format-clause (table) – список имен и типов полей

возвращается nil, если не указан оператор формата

Возможные ошибки:

- space_object не существует,
- дублируются имена полей;
- тип не поддерживается.

Как правило, Tarantool допускает поля без имен и без указания типа. Но с помощью format можно, например, задокументировать, что N-ное поле представляет собой поле для фамилии и должно содержать строковое значение. Также оператор формата можно указать в *box.schema.space.create()*.

Оператор формата для каждого поля содержит определение в фигурных скобках: {name='...',type='...'[,is_nullable=...]}, где:

- значение name может представлять собой любую строку при условии, что у двух полей не будет одинаковых имен;
- значением type может быть любой из разрешенных типов: any | unsigned | string | integer | number | boolean | array | map | scalar, но для создания индекса следует использовать только *индексируемые типы*;
- значение необязательного параметра is_nullable может быть true или false (такое же требование, как для *«Параметров для space_ object:create_ index»*). См. также предупреждение в разделе *Разрешение использования нулевых значений для индексируемого ключа*.

В кортежах недопустимы значения неправильного типа; например, после box.space.tester:format({{' ',type='number'}}) (тип = число) запрос box.space.tester:insert{'строка-которая-не-является-числом'} вызовет ошибку.

В кортежах недопустимы нулевые значения, если is_nullable=false, что задано по умолчанию; например, после box.space.tester:format({{' ',type='number',is_nullable=false}}) запрос box.space.tester:insert{nil,2} вызовет ошибку.

В кортежах может быть больше полей, чем описано в операторе формата. Чтобы ограничить количество полей, необходимо указать элемент спейса *field_ count*.

В кортежах может быть меньше полей, чем описано в операторе формата, если пропущенные завершающие поля описаны с помощью is_nullable=true; например после box.space.tester:format({{'a',type='number'},{'b',type='number',is_nullable=true}}) запрос box.space.tester:insert{2} не приведет к ошибке формата.

Можно использовать format для спейса, в котором уже определен формат, заменяя таким образом предыдущие определения при условии, что нет конфликта с существующими данными или определениями индекса.

Можно использовать format для того, чтобы изменить значение флага is_nullable; например, после box.space.tester:format({{' ',type='scalar',is_nullable=false}}) запрос

`box.space.tester:format({{' ',type='scalar',is_nullable=true}})` не вызовет ошибку – и не приведет к перестроению спейса. Но обратное изменение значения `is_nullable` с `true` на `false` может вызвать перестроение и привести к ошибке, если уже есть кортежи с нулевыми значениями.

Пример:

```
box.space.tester:format({{name='surname',type='string'},{name='IDX',type='array'}})
box.space.tester:format({{name='surname',type='string',is_nullable=true}})
```

Можно использовать следующие варианты оператора:

- пропуск и „name=“, и „type=“,
- пропуск „type=“ и
- добавление дополнительных фигурных скобок.

В следующем примере иллюстрируются все варианты, первый для поля с именем „x“, второй – для двух полей с именами „x“ и „y“.

```
box.space.tester:format({{'x'}})
box.space.tester:format({{'x'},{'y'}})
box.space.tester:format({{name='x',type='scalar'}})
box.space.tester:format({{name='x',type='scalar'},{name='y',type='unsigned'}})
box.space.tester:format({{name='x'}})
box.space.tester:format({{name='x'},{name='y'}})
box.space.tester:format({{'x',type='scalar'}})
box.space.tester:format({{'x',type='scalar'},{'y',type='unsigned'}})
box.space.tester:format({{'x','scalar'}})
box.space.tester:format({{'x','scalar'},{'y','unsigned'}})
```

В следующем примере показывается создание спейса, определение формата для него со всеми возможными типа и вставка данных.

```
tarantool> box.schema.space.create('t')
--- ...
tarantool> box.space.t:format({{name='1',type='any'},
         >                     {name='2',type='unsigned'},
         >                     {name='3',type='string'},
         >                     {name='4',type='number'},
         >                     {name='5',type='integer'},
         >                     {name='6',type='boolean'},
         >                     {name='7',type='scalar'},
         >                     {name='8',type='array'},
         >                     {name='9',type='map'}})
--- ...
tarantool> box.space.t:create_index('i',{parts={{field = 2, type = 'unsigned'}}})
--- ...
tarantool> box.space.t:insert{{'a'},      -- any
         >                    1,          -- unsigned
         >                    'W?',       -- string
         >                    5.5,        -- number
         >                    -0,         -- integer
         >                    true,       -- boolean
         >                    true,       -- scalar
         >                    {{'a'}},    -- array
         >                    {val=1}}    -- map
---
```

(continues on next page)

(продолжение с предыдущей страницы)

```
- [['a'], 1, 'W?', 5.5, 0, true, true, [['a']], {'val': 1}]
...
```

Имена, указанные с помощью оператора формата, можно использовать в *space_object:get()*, в *space_object:create_index()*, в *tuple_object[field-name]* и в *tuple_object[field-path]*.

Если оператор формата не указан, то вернется таблица, которая использовалась при предыдущем вызове *объект-спейса*:format(*оператор-формата*). Например, после box.space.tester:format({{'x','scalar'}}), box.space.tester:format() вернет [{'name': 'x', 'type': 'scalar'}].

Formatting or reformatting a large space will cause occasional *yields* so that other requests will not be blocked. If the other requests cause an illegal situation such as a field value of the wrong type, the formatting or reformatting will fail.

space_object:frommap(*map*[, *option*])

Конвертация ассоциативного массива в экземпляр кортежа или в таблицу. Ассоциативный массив должен состоять из пар «имя поля = значение». Имена полей и типы значений должны соответствовать именам и типам, ранее заданным для спейса через *space_object:format()*.

Параметры

- space_object (*space_object*) – *ссылка на объект*
- map (*field-value-pairs*) – ряд пар «поле = значение» в любом порядке.
- option (*boolean*) – единственный возможный параметр {table = true|false}; если параметр не указан, или же {table = false}, то возвращается „cdata“ (то есть кортеж); если {table = true}, то возвращается таблица.

возвращается кортеж или таблица.

тип возвращаемого значения кортеж или таблица

Возможные ошибки: отсутствует объект спейса space_object, или в спейсе нет формата; «unknown field» (неизвестное поле).

Пример:

```
-- Создание формата с двумя полями под названиями 'a' и 'b'.
-- Создание спейса с таким форматом.
-- Создание кортежа на основе ассоциативного массива по данному спейсу.
-- Создание таблицы на основе ассоциативного массива по данному спейсу.
tarantool> format1 = {{name='a',type='unsigned'},{name='b',type='scalar'}}
---
...
tarantool> s = box.schema.create_space('test', {format = format1})
---
...
tarantool> s:frommap({b = 'x', a = 123456})
---
- [123456, 'x']
...
tarantool> s:frommap({b = 'x', a = 123456}, {table = true})
---
- - 123456
  - x
...
```

space_object:get(*key*)
Поиск кортежа в данном спейсе.

Параметры

- space_object (*space_object*) – *ссылка на объект*
- key (*scalar/table*) – значение должно совпасть с индексным ключом, который может быть составным.

возвращается кортеж, ключ индекса в котором совпадает с key или nil.

тип возвращаемого значения кортеж

Возможные ошибки: space_object не существует.

Факторы сложности Размер индекса, тип индекса, количество кортежей, к которым получен доступ, настройки журнала упреждающей записи (WAL).

Функция box.space...select вернет набор кортежей в виде Lua-таблицы; функция box.space...get вернет самое большее один кортеж. Можно получить первый кортеж в спейсе, добавив [1]. Таким образом, box.space.tester:get{1} эквивалентна box.space.tester:select{1}[1], если найден только один кортеж.

Пример:

```
box.space.tester:get{1}
```

Using field names instead of field numbers: *get()* can use field names described by the optional *space_object:format()* clause. This is true because the object returned by get() can be used with most of the features described in the *Submodule box.tuple* description, including *tuple_object[field-name]*.

For example, we can format the *tester* space with a field named *x* and use the name *x* in the index definition:

```
box.space.tester:format({{name='x',type='scalar'}})
box.space.tester:create_index('I',{parts={'x'}})
```

Then, if get or select retrieves a single tuple, we can reference the field „x“ in the tuple by its name:

```
box.space.tester:get{1}['x']
box.space.tester:select{1}[1]['x']
```

space_object:insert(*tuple*)
Вставка кортежа в спейс.

Параметры

- space_object (*space_object*) – *ссылка на объект*
- tuple (*tuple/table*) – вставляемый кортеж.

возвращается вставленный кортеж

тип возвращаемого значения кортеж

Возможные ошибки: ошибка ER_TUPLE_FOUND, если уже существует кортеж с тем же уникальным значением ключа.

Пример:

```
tarantool> box.space.tester:insert{5000,'tuple number five thousand'}
---
- [5000, 'tuple number five thousand']
...
```

Для получения дополнительной информации о сценариях использования и типичных ошибках, см. *Пример: использование операций с данными* далее в разделе.

`space_object:len()`

Возврат количества кортежей в спейсе. Если сравнивать с *count()*, то данный метод работает быстрее, поскольку метод `len()` не сканирует весь спейс для подсчета кортежей.

Параметры

- `space_object` (*space_object*) – *ссылка на объект*

возвращается Количество кортежей в спейсе.

Пример:

```
tarantool> box.space.tester:len()
---
- 2
...
```

Примечание про движок базы данных: vinyl поддерживает `len()`, но результат может быть неточным. Если необходим точный результат, используйте *count()* или *pairs():length()*.

`space_object:on_replace(`[*trigger-function*[, *old-trigger-function*]]`)`

Создание «*триггера* замены». Функция с триггером `trigger-function` будет выполняться в случае операции `replace()` или `insert()`, или `update()`, или `upsert()`, или `delete()` над кортежем в спейсе `<space-name>`.

Параметры

- `trigger-function` (*function*) – функция, в которой будет триггер; для получения информации о параметрах функции с триггером см. Пример №2 ниже
- `old-trigger-function` (*function*) – существующая функция с триггером, которую заменит новая `trigger-function`

возвращается nil или указатель функции

Если указаны параметры (nil, old-trigger-function), старый триггер будет удален.

Если не указан ни один параметр, ответом будет список существующих функций с триггером.

Следует знать, что если активация триггера произошла в случае репликации или определенного вида подключения, функция может ссылаться на *box.session.type()*.

Подробная информация о характеристиках триггера находится в разделе *Триггеры*.

См. также *space_object:before_replace()*.

Пример №1:

```
tarantool> function f ()
         >   x = x + 1
         > end
tarantool> box.space.X:on_replace(f)
```

Пример №2:

В функции с триггером может быть до 4 параметров:

- (кортеж) старое значение до начала запроса,
- (кортеж) новое значение после окончания выполнения запроса,
- (строка) имя спейса,
- (строка) тип запроса: „INSERT“ (вставка), „DELETE“ (удаление), „UPDATE“ (обновление) или „REPLACE“ (замена).

Например, следующий код вызывает вывод nil и „INSERT“ (вставка) при обработке запроса на вставку и вывод [1, „Hi“] и „DELETE“ (удаление) при обработке запроса на удаление:

```
box.schema.space.create('space_1')
box.space.space_1:create_index('space_1_index',{})
function on_replace_function (old, new, s, op) print(old) print(op) end
box.space.space_1:on_replace(on_replace_function)
box.space.space_1:insert{1,'Hi'}
box.space.space_1:delete{1}
```

Пример №3:

Следующая серия запросов создаст спейс, создаст индекс, создаст функцию, которая увеличит содержимое счетчика, создаст триггер, сделает две вставки, удалит спейс и отобразит значение счетчика – 2, поскольку функция выполняется однократно после каждой вставки.

```
tarantool> s = box.schema.space.create('space53')
tarantool> s:create_index('primary', {parts = {{field = 1, type = 'unsigned'}}})
tarantool> function replace_trigger()
         >   replace_counter = replace_counter + 1
         > end
tarantool> s:on_replace(replace_trigger)
tarantool> replace_counter = 0
tarantool> t = s:insert{1, 'First replace'}
tarantool> t = s:insert{2, 'Second replace'}
tarantool> s:drop()
tarantool> replace_counter
```

Примечание: В тригер-функциях для `on_replace` и `before_replace` не следует использовать

- транзакции,
- операции, передающие управление (yield-operations, *явные* или нет),
- действия, которые не разрешено использовать в транзакциях (см. *правило №2*)

потому что все, что выполняется внутри триггеров, уже находится в транзакции.

Пример:

```
tarantool> box.space.test:on_replace(fiber.yield)
tarantool> box.space.test:replace{1, 2, 3}
2020-02-02 21:22:03.073 [73185] main/102/init.lua txn.c:532 E> ER_TRANSACTION_YIELD:␣
↪Transaction has been aborted by a fiber yield
---
- error: Transaction has been aborted by a fiber yield
...
```

space_object:before_replace([*trigger-function*[, *old-trigger-function*]])

Создание «*триггера* замены». Функция с триггером `trigger-function` будет выполняться в случае операции `replace()` или `insert()`, или `update()`, или `upsert()`, или `delete()` над кортежем в спейсе `<space-name>`.

Параметры

- `trigger-function` (*function*) – функция, в которой будет триггер; необязательные параметры функции с триггером см. в описании *on_replace*.
- `old-trigger-function` (*function*) – существующая функция с триггером, которую заменит новая `trigger-function`

возвращается nil или указатель функции

Если указаны параметры `(nil, old-trigger-function)`, старый триггер будет удален.

Если не указан ни один параметр, ответом будет список существующих функций с триггером.

Следует знать, что если активация триггера произошла в случае репликации или определенного вида подключения, функция может ссылаться на *box.session.type()*.

Подробная информация о характеристиках триггера находится в разделе *Триггеры*.

См. также *space_object:on_replace()*.

Администраторы могут создавать триггеры замены с условием после замены `on_replace()` или до замены `before_replace()`. Если созданы оба типа, то все триггеры до замены `before_replace` выполняются до всех триггеров после замены `on_replace`. Функции для обоих типов триггеров `on_replace` и `before_replace` могут вносить изменения в базу данных, но только функции с триггерами до замены `before_replace` могут изменять кортеж, который будет заменен.

Поскольку функция с триггером до замены `before_replace` может вносить дополнительные изменения в старый кортеж, для нее также потребуются дополнительные ресурсы для вызова старого кортежа до внесения изменений. Таким образом, лучше использовать триггер после замены `on_replace`, если нет необходимости изменять старый кортеж. Тем не менее, это применимо только к движку memtx – что касается движка vinyl, такой вызов произойдет для любого типа триггера. (В memtx'е данные кортежа хранятся вместе с ключом индекса, поэтому нет необходимости в дополнительном поиске; для vinyl'а дело обстоит иначе, поэтому нужен дополнительный поиск.)

Если нет необходимости в дополнительных изменениях, следует использовать `on_replace` вместо `before_replace`. Как правило, `before_replace` используется только для определенных сценариев репликации – в части разрешения конфликтов.

Что случится после возврата значения, которое может вернуть функция с триггером `before_replace`, зависит от этого значения. А именно:

- если нет возвращаемого значения, выполнение продолжается со вставкой|заменой нового значения;
- если значение – nil, то кортеж будет удален;
- если значение совпадает со старым, то вызывается функция `on_replace`, и изменение данных не происходит
- если значение совпадает с новым, то считаем, что вызова функции `before_replace` не было;
- если значение другое, выполнение продолжается со вставкой/заменой нового значения.

Тем не менее, если функция с триггером возвращает старый кортеж или вызывает *run_triggers(false)*, это не повлияет на другие триггеры, активируемые в том же запросе вставки, обновления или замены.

Пример:

Далее представлены функции `before_replace`: не возвращает значение, возвращает nil, возвращает совпадающее со старым значение, возвращает совпадающее с новым значение, возвращает другое значение.

```
function f1 (old, new) return end
function f2 (old, new) return nil end
function f3 (old, new) return old end
function f4 (old, new) return new end
function f5 (old, new) return box.tuple.new({new[1],'b'}) end
```

`space_object:pairs(`[*key*[, *iterator*]])

Поиск кортежа или набора кортежей в заданном спейсе и итерация по одному кортежу за раз.

Параметры

- `space_object` (*space_object*) – *ссылка на объект*
- `key` (*scalar/table*) – значение должно совпасть с индексным ключом, который может быть составным
- `iterator` – см. *index_object:pairs*

возвращается итератор, который может использовать в цикле for/end или с функцией totable()

Возможные ошибки:

- отсутствие такого спейса.
- неправильный тип.

Факторы сложности: Размер индекса, тип индекса.

Чтобы посмотреть примеры сложных запросов `pairs`, где можно указать индекс для поиска и используемое условие (например, «больше чем» вместо «равен»), см. раздел далее по тексту *index_object:pairs*.

Для получения информации о внутренней структуре итераторов см. документацию по библиотеке для функционального программирования в Lua «Lua Functional library».

Пример:

```
tarantool> s = box.schema.space.create('space33')
---
...
tarantool> -- в индексе 'X' количество частей по умолчанию {1, 'unsigned'}
tarantool> s:create_index('X', {})
---
...
tarantool> s:insert{0, 'Hello my '}, s:insert{1, 'Lua world'}
---
- [0, 'Hello my ']
- [1, 'Lua world']
...
tarantool> tmp = ''
```

(continues on next page)

(продолжение с предыдущей страницы)

```
---
...
tarantool> for k, v in s:pairs() do
         >   tmp = tmp .. v[2]
         > end
---
...
tarantool> tmp
---
- Hello my Lua world
...
```

space_object:rename(*space-name*)
Переименование спейса.

Параметры

- space_object (*space_object*) – *ссылка на объект*
- space-name (string) – новое имя спейса

возвращается nil

Возможные ошибки: space_object не существует.

Пример:

```
tarantool> box.space.space55:rename('space56')
---
...
tarantool> box.space.space56:rename('space55')
---
...
```

space_object:replace(*tuple*)
space_object:put(*tuple*)
Вставка кортежа в спейс. Если уже существует кортеж с тем же первичным ключом, box.space...:replace() заменит существующий кортеж новым. Варианты синтаксиса (box.space...:replace() и box.space...:put()) приведут к одному результату, но последний иногда используется как противоположность box.space...:get().

Параметры

- space_object (*space_object*) – *ссылка на объект*
- tuple (*table/tuple*) – вставляемый кортеж

возвращается вставленный кортеж.

тип возвращаемого значения кортеж

Возможные ошибки: ошибка ER_TUPLE_FOUND, если уже существует другой кортеж с тем же уникальным значением ключа (это произойдет только в том случае, если есть уникальный вторичный индекс).

Факторы сложности Размер индекса, тип индекса, количество кортежей, к которым получен доступ, настройки журнала упреждающей записи (WAL).

Пример:

```
box.space.tester:replace{5000, 'tuple number five thousand'}
```

Для получения дополнительной информации о сценариях использования и типичных ошибках, см. *Пример: использование операций с данными* далее в разделе.

`space_object:run_triggers(`*true/false*`)`

На тот момент, когда *триггер* определен, он автоматически активируется, то есть он будет исполняться. Триггеры *для замены* можно отключить с помощью `box.space.`*имя-спейса*`:run_triggers(false)` и повторно активировать с помощью `box.space.`*имя-спейса*`:run_triggers(true)`.

возвращается nil

Пример:

Следующая серия запросов ассоциирует существующую функцию с именем F с существующим спейсом с именем T, ассоциирует функцию во второй раз с тем же спейсом (чтобы вызвать ее дважды), отключит все триггеры на T и удалит каждый триггер, заменив его на `nil`.

```
tarantool> box.space.T:on_replace(F)
tarantool> box.space.T:on_replace(F)
tarantool> box.space.T:run_triggers(false)
tarantool> box.space.T:on_replace(nil, F)
tarantool> box.space.T:on_replace(nil, F)
```

`space_object:select(`$\left[$*key*$\left[$, *options* $\right]\right]$`)`

Поиск кортежа или набора кортежей в заданном спейсе. Этот метод не передает управление (детали можно найти в разделе *Кооперативная многозадачность*).

Параметры

- `space_object` (*space_object*) – *ссылка на объект*
- `key` (*scalar/table*) – значение должно совпасть с индексным ключом, который может быть составным.
- `options` (*table/nil*) – ни один, любой или все параметры, которые допускает *index_object:select*: * `options.iterator` (*тип итератора*) * `options.limit` (максимальное количество кортежей) * `options.offset` (количество пропускаемых кортежей)

возвращается кортежи, поля первичного ключа в которых равны полям переданного ключа. Если количество переданных полей меньшей количества полей первичного ключа, сопоставляются только переданные поля, то есть для `select{1, 2}` совпадением будет кортеж с первичным ключом `{1,2,3}`.

тип возвращаемого значения массив кортежей

Запрос выборки `select` также можно выполнить со специальными параметрами индекса, которые указаны в *index_object:select*.

Возможные ошибки:

- отсутствие такого спейса.
- неправильный тип.

Факторы сложности: Размер индекса, тип индекса.

Пример:

```
tarantool> s = box.schema.space.create('tmp', {temporary=true})
---
...
tarantool> s:create_index('primary',{parts = {{field = 1, type = 'unsigned'}, {field = 2,
↪ type = 'string'}} })
---
...
tarantool> s:insert{1,'A'}
---
- [1, 'A']
...
tarantool> s:insert{1,'B'}
---
- [1, 'B']
...
tarantool> s:insert{1,'C'}
---
- [1, 'C']
...
tarantool> s:insert{2,'D'}
---
- [2, 'D']
...
tarantool> -- must equal both primary-key fields
tarantool> s:select{1,'B'}
---
- - [1, 'B']
...
tarantool> -- must equal only one primary-key field
tarantool> s:select{1}
---
- - [1, 'A']
  - [1, 'B']
  - [1, 'C']
...
tarantool> -- must equal 0 fields, so returns all tuples
tarantool> s:select{}
---
- - [1, 'A']
  - [1, 'B']
  - [1, 'C']
  - [2, 'D']
...
tarantool> -- the first field must be greater than 0, and
tarantool> -- skip the first tuple, and return up to
tarantool> -- 2 tuples. This example's options all
tarantool> -- depend on index characteristics so see
tarantool> -- more explanation in index_object:select().
tarantool> s:select({0},{iterator='GT',offset=1,limit=2})
---
- - [1, 'B']
  - [1, 'C']
...
```

Как показано в последнем запросе вышеприведенного примера, чтобы выполнять сложные запросы выборки `select`, где можно указать, в каком индексе производится поиск и с какими условиями (например, «больше, чем» вместо «равный»), а также необходимое количество возвращаемых кортежей, необходимо ознакомиться с *index_ object:select*.

Помните, что из кортежа можно получить поле как по номеру поля, так и по имени поля, что более удобно. См. пример: *использование имен вместо номеров полей*.

Для получения дополнительной информации о сценариях использования и типичных ошибках, см. *Пример: использование операций с данными* далее в разделе.

`space_object:truncate()`

Удаление всех кортежей. Метод выполняется в фоновом режиме и не блокирует последующие запросы.

Параметры

- `space_object` (*space_object*) – *ссылка на объект*

Факторы сложности: Размер индекса, тип индекса, количество кортежей, к которым получен доступ.

возвращается nil

Метод `truncate` может вызвать только тот пользователь, который создал спейс, или другой пользователь через функцию `setuid`, созданную пользователем, который создал спейс. Более подробную информацию о функциях *setuid* можно получить в справочнике по for *box.schema.func.create()*.

Метод `truncate` нельзя вызвать из транзакции.

Пример:

```
tarantool> box.space.tester:truncate()
---
...
tarantool> box.space.tester:len()
---
- 0
...
```

`space_object:update(`*key*, *{{operator*, *field_no*, *value}*, *...}*`)`

Обновление кортежа.

Функция `update` поддерживает операции над полями – присваивание, арифметические операции (если поле числовое), вырезание и вставку фрагментов поля, удаление или вставку поля. Несколько операций можно объединить в отдельный запрос обновления, и в таком случае они будут выполняться атомарно и последовательно. Для каждой операции необходимо указать номер поля. Если выполняются несколько операций, то номер поля для каждой операции считается относительно последнего состояния кортежа, то есть как если бы все предыдущие операции в обновлении с несколькими операциями уже были выполнены. Другими словами, всегда лучше объединить несколько вызовов `update` в один без изменений семантики.

Возможные операторы:

- `+` для сложения (значения должны быть числовыми)
- `-` для вычитания (значения должны быть числовыми)
- `&` для поразрядной операции И (значения должны быть беззнаковыми числами)
- `|` для поразрядной операции ИЛИ (значения должны быть беззнаковыми числами)
- `^` для поразрядной операции Исключающее ИЛИ (значения должны быть беззнаковыми числами)
- `:` для разделения строк

- `!` для вставки
- `#` для удаления
- `=` для присваивания

Для операций `!` и `=` номер поля может быть `-1`, что означает последнее поле в кортеже.

Параметры

- `space_object` (*space_object*) – *ссылка на объект*
- `key` (*scalar/table*) – значения поля первичного ключа, которые должны возвращаться в виде Lua-таблицы, если ключ составной
- `operator` (string) – тип операции, представленный строкой
- `field_no` (*number*) – к какому полю применяется операция. Номер поля может быть отрицательным, что означает, что позиция рассчитывается с конца кортежа. (#кортеж + отрицательный номер поля + 1)
- `value` (*lua_value*) – какое значение применяется

возвращается

- обновленный кортеж
- nil, если ключ не найден

тип возвращаемого значения кортеж или nil

Возможные ошибки: нельзя изменять поле первичного ключа.

Факторы сложности Размер индекса, тип индекса, количество кортежей, к которым получен доступ, настройки журнала упреждающей записи (WAL).

Таким образом, в инструкции:

```
s:update(44, {{'+', 1, 55 }, {'=', 3, 'x'}})
```

значение первичного ключа равно 44, заданы операторы `'+'` и `'='`, что означает *прибавление значение к полю, а затем присваивание значения полю*, первое затронутое поле – это поле 1, к нему прибавляется значение 55, второе затронутое поле – это поле 3, ему присваивается значение `'x'`.

Пример:

Предположим, что изначально есть спейс под названием `tester` с первичным индексом, тип которого – `unsigned`. Есть один кортеж с полем №1 `field[1]` = 999 и полем №2 `field[2]` = `'A'`.

В обновлении: `box.space.tester:update(999, {{'=', 2, 'B'}})` Первый аргумент – это `tester`, то есть обновление происходит в спейсе `tester`. Второй аргумент – 999, то есть затронутый кортеж определяется по значению первичного ключа = 999. Третий аргумент – `=`, то есть будет одна операция – *присваивание полю*. Четвертый аргумент – 2, то есть будет затронуто поле №2 `field[2]`. Пятый аргумент – `'B'`, то есть содержимое `field[2]` изменится на `'B'`. Таким образом, после данного обновления `field[1]` = 999, а `field[2]` = `'B'`.

В обновлении: `box.space.tester:update({999}, {{'=', 2, 'B'}})` Аргументы повторяются за исключением того, что ключ передается в виде Lua-таблицы (в фигурных скобках). В этом нет необходимости, если первичный ключ содержит только одно поле, но было бы необходимо, если бы в первичном ключе было больше одного поля. Таким образом, после данного обновления `field[1]` = 999, а `field[2]` = `'B'` (без изменений).

В обновлении: `box.space.tester:update({999}, {{'=', 3, 1}})` Аргументы повторяются за исключением того, что четвертым аргументом будет 3, то есть будет затронуто поле №3 `field[3]`. Ничего страшного, что до этого поле `field[3]` не существовало. Оно добавится. Таким образом, после данного обновления `field[1]` = 999, `field[2]` = `'B'`, `field[3]` = 1.

В обновлении: `box.space.tester:update({999}, {{'+', 3, 1}})` Аргументы повторяются за исключением того, что третьим аргументом будет `'+'`, то есть будет операция добавления, а не присваивания. Поскольку“field[3]“ ранее содержало значение 1, это означает, что к 1 прибавится 1. Таким образом, после данного обновления `field[1]` = 999, `field[2]` = `'B'`, `field[3]` = 2.

В обновлении: `box.space.tester:update({999}, {{'|', 3, 1}, {'=', 2, 'C'}})` Основная идея состоит в том, чтобы изменить одновременно два поля. Форматами будут `'|'` и `=`, то есть имеем две операции: ИЛИ и присваивание. Четвертый и пятый аргументы означают, что над полем `field[3]` проводится операция ИЛИ со значением 1. Седьмой и восьмой аргументы означают, что полю `field[2]` присваивается `'C'`. Таким образом, после данного обновления `field[1]` = 999, `field[2]` = `'C'`, `field[3]` = 3.

В обновлении: `box.space.tester:update({999}, {{'#', 2, 1}, {'-', 2, 3}})` Основная идея состоит в том, чтобы удалить поле `field[2]`, а затем вычесть 3 из `field[3]`. Но после удаления, произойдет перенумерация, поэтому поле `field[3]` становится `field[2]` до того, как мы вычтем из него 3, вот почему седьмым аргументом будет 2, а не 3. Таким образом, после данного обновления `field[1]` = 999, `field[2]` = 0.

В обновлении: `box.space.tester:update({999}, {{'=', 2, 'XYZ'}})` Создаем длинную строку, чтобы в следующем примере сработало разделение. Таким образом, после данного обновления `field[1]` = 999, `field[2]` = `'XYZ'`.

В обновлении: `box.space.tester:update({999}, {{':', 2, 2, 1, '!!'}})` Третьим аргументом будет `':'`, то есть это пример разделения. Четвертым аргументом будет 2, поскольку изменение произойдет в поле `field[2]`. Пятым аргументом будет 2, поскольку удаление начнется со второго байта. Шестым аргументом будет 1, количество удаляемых байтов – 1. Седьмым аргументом будет `'!!'`, поскольку в данном положении будет добавляться `'!!'`. Таким образом, после данного обновления `field[1]` = 999, `field[2]` = `'X!!Z'`.

Для получения дополнительной информации о сценариях использования и типичных ошибках, см. *Пример: использование операций с данными* далее в разделе.

`space_object:upsert(`*tuple*, *{{operator*, *field_no*, *value}*, *...}*`)`
Обновление или вставка кортежа.

Если существует кортеж, который совпадает с полями ключа `tuple`, запрос приведет к тому же результату, что и *space_object:update()*, и используется параметр `{{operator, field_no, value}, ...}`. Если нет кортежа, который совпадает с полями ключа `tuple`, запрос приведет к тому же результату, что и *space_object:insert()*, и используется параметр `{tuple}`. Однако, в отличие от `insert` или `update`, `upsert` не считывает кортеж и не проверяет на ошибки перед возвратом – это конструктивная особенность, которая увеличивает быстродействие, но требует большей осторожности со стороны пользователя.

Параметры

- `space_object` (*space_object*) – *ссылка на объект*
- `tuple` (*table/tuple*) – вставляемый по умолчанию кортеж, если не найдет аналог
- `operator` (`string`) – тип операции, представленный строкой
- `field_no` (*number*) – к какому полю применяется операция. Номер поля может быть отрицательным, что означает, что позиция рассчитывается с конца

кортежа. (#кортеж + отрицательный номер поля + 1)

- `value` (*lua_value*) – какое значение применяется

возвращается null

Возможные ошибки:

- Нельзя изменять поле первичного ключа.
- Нельзя проводить операцию upsert в спейсе, в котором есть уникальный вторичный индекс.

Факторы сложности Размер индекса, тип индекса, количество кортежей, к которым получен доступ, настройки журнала упреждающей записи (WAL).

Пример:

```
box.space.tester:upsert({12,'c'}, {{'=', 3, 'a'}, {'=', 4, 'b'}})
```

Для получения дополнительной информации о сценариях использования и типичных ошибках, см. *Пример: использование операций с данными* далее в разделе.

`space_object:user_defined()`

Пользователи могут сами определять любые желаемые функции и связывать их со спейсами: фактически они могут создавать собственные методы для работы со спейсом. Это можно сделать так:

(1) создать Lua-функцию,

(2) добавить имя функции в заданную глобальную переменную с типом «таблица» (table),

(3) впоследствии в любое время, пока работает сервер, вызвать функцию с помощью `объект_спейса:имя-функции([параметры])`.

Задана глобальная переменная `box.schema.space_mt`. Метод, добавленный в `box.schema.space_mt`, будет доступен для всех спейсов.

Можно также сделать задаваемый пользователем метод доступным только для одного индекса путем вызова `getmetatable(объект_спейса)` и последующего добавления имени функции в метатаблицу. См. также пример для *index_object:user_defined()*.

Параметры

- `index_object` (*index_object*) – *ссылка на объект*.
- `any-name` (*any-type*) – то, что определяет пользователь

Пример:

```
-- Доступный для любого спейса, без параметров.
-- После таких запросов значение глобальной переменной global_variable будет 6.
box.schema.space.create('t')
box.space.t:create_index('i')
global_variable = 5
function f(space_arg) global_variable = global_variable + 1 end
box.schema.space_mt.counter = f
box.space.t:counter()
```

`space_object:create_check_constraint(`*check_constraint_name*, *expression*`)`

Create a check constraint. A check constraint is a requirement that must be met when a tuple is inserted or updated in a space. Check constraints created with `space_object:create_check_constraint` have the same effect as check constraints created with an SQL CHECK() clause in a *CREATE TABLE statement*.

Параметры

- space_object (*space_object*) – *ссылка на объект*
- check_constraint_name (string) – name of check constraint, which should conform to the *rules for object names*
- expression (string) – SQL code of an expression which must return a boolean result

возвращается check constraint object

тип возвращаемого значения check_constraint_object

The space must be formatted with *space_object:format()* so that the expression can contain field names. The space must be empty. The space must not be a system space.

The expression must return true or false and should be deterministic. The expresion may be any SQL (not Lua) expression containing field names, built-in function names, literals, and operators. Not subqueries. If a field name contains lower case characters, it must be enclosed in «double quotes».

Check constraints are checked before the request is performed, at the same time as Lua *before_replace triggers*. If there is more than one check constraint or before_replace trigger, then they are ordered according to time of creation. (This is a change from the earlier behavior of check constraints, which caused checking before the tuple was formed.)

Check constraints can be dropped with space_object:*check_constraint_name*:drop().

Пример:

```
box.schema.space.create('t')
box.space.t:format({{name = 'f1', type = 'unsigned'},
                    {name = 'f2', type = 'string'},
                    {name = 'f3', type = 'string'}})
box.space.t:create_index('i')
box.space.t:create_check_constraint('c1', [["f2" > 'A']])
box.space.t:create_check_constraint('c2',
                        [["f2"=UPPER("f3") AND NOT "f2" LIKE '__']])
-- This insert will fail, constraint c1 expression returns false
box.space.t:insert{1, 'A', 'A'}
-- This insert will fail, constraint c2 expression returns false
box.space.t:insert{1, 'B', 'c'}
-- This insert will succeed, both constraint expressions return true
box.space.t:insert{1, 'B', 'b'}
-- This update will fail, constraint c2 expression returns false
box.space.t:update(1, {{'=', 2, 'xx'}, {'=', 3, 'xx'}})
```

A list of check constraints is in *space_object._ck_constraint*.

space_object.enabled
: Определение активности спейса. Значение false указывает на отсутствие индекса.

space_object.field_count
: Необходимость подсчета полей всех кортежей в спейсе, который можно изначально задать следующим образом:

```
box.schema.space.create(..., {
    ... ,
    field_count = *field_count_value* ,
    ...
})
```

По умолчанию, будет использоваться значение 0, что указывает на отсутствие необходимости подсчета полей.

Пример:

```
tarantool> box.space.tester.field_count
---
- 0
...
```

space_object.id
: Порядковый номер спейса. На спейс можно ссылаться либо по имени, либо по номеру. Таким образом, если идентификатором спейса `tester` будет `id = 800`, то `box.space.tester:insert{0}` и `box.space[800]:insert{0}` представляют собой равнозначные запросы.

Пример:

```
tarantool> box.space.tester.id
---
- 512
...
```

box.space.index
: Контейнер для всех определенных индексов. Есть Lua-объект типа *box.index* с методами поиска кортежей и итерации по ним в заданном порядке.

Чтобы сбросить, use *box.stat.reset()*.

тип возвращаемого значения таблица

Пример:

```
# checking the number of indexes for space 'tester'
tarantool> local counter=0; for i=0,#box.space.tester.index do
  if box.space.tester.index[i]~=nil then counter=counter+1 end
  end; print(counter)
1
---
...
# checking the type of index 'primary'
tarantool> box.space.tester.index.primary.type
---
- TREE
...
```

box.space._cluster
: `_cluster` – это системный спейс для поддержки *функции репликации*.

box.space._func
: `_func` is a system space with function tuples made by *box.schema.func.create()* or *box.schema.func.create(func-name [, {options-with-body}])*.

Кортежи в данном спейсе включают в себя следующие поля:

- id (цельночисленный идентификатор),
- владелец (целочисленный идентификатор)
- имя функции,
- флаг setuid,

- название языка (необязательно): „LUA“ (по умолчанию) or „C“.
- the body
- the is_deterministic flag
- the is_sandboxed flag
- options

If the function tuple was made in the older way without specification of `body`, then the `_func` space will contain default values for the body and the is_deterministic flag and the is_sandboxed flag. Such function tuples are called «not persistent». You continue to create Lua functions in the usual way, by saying `function function_name () ... end`, without adding anything in the `_func` space. The `_func` space only exists for storing function tuples so that their names can be used within *grant/revoke* functions.

If the function tuple was made the newer way with specification of `body`, then all the fields may contain non-default values. Such functions are called «persistent». They should be invoked with `box.func.`*`func-name`*`:call([parameters])`.

Доступны следующие операции:

- Создание кортежа в `_func` с помощью *box.schema.func.create()*,
- Удаление кортежа в `_func` с помощью *box.schema.func.drop()*,
- Проверка наличия кортежа в `_func` с помощью *box.schema.func.exists()*.

Пример:

В следующем примере создадим функцию с именем ‘f7’, поместим ее в спейс `_func` в Tarantool’е и выдадим права на „выполнение“ этой функции пользователю „guest“.

```
tarantool> function f7()
         >  box.session.uid()
         > end
---
...
tarantool> box.schema.func.create('f7')
---
...
tarantool> box.schema.user.grant('guest', 'execute', 'function', 'f7')
---
...
tarantool> box.schema.user.revoke('guest', 'execute', 'function', 'f7')
---
...
```

`box.space._index`
`_index` – это системный спейс.

Кортежи в данном спейсе включают в себя следующие поля:

- `id` (= идентификатор спейса),
- `iid` (= номер индекса в спейсе),
- `name`,
- `type`,
- `opts` (например, уникальная опция), [`tuple-field-no`, `tuple-field-type` ...].

Вот что при обычной установке включает в себя спейс `_index`:

```
tarantool> box.space._index:select{}
---
- - [272, 0, 'primary', 'tree', {'unique': true}, [[0, 'string']]]
  - [280, 0, 'primary', 'tree', {'unique': true}, [[0, 'unsigned']]]
  - [280, 1, 'owner', 'tree', {'unique': false}, [[1, 'unsigned']]]
  - [280, 2, 'name', 'tree', {'unique': true}, [[2, 'string']]]
  - [281, 0, 'primary', 'tree', {'unique': true}, [[0, 'unsigned']]]
  - [281, 1, 'owner', 'tree', {'unique': false}, [[1, 'unsigned']]]
  - [281, 2, 'name', 'tree', {'unique': true}, [[2, 'string']]]
  - [288, 0, 'primary', 'tree', {'unique': true}, [[0, 'unsigned'], [1, 'unsigned']]]
  - [288, 2, 'name', 'tree', {'unique': true}, [[0, 'unsigned'], [2, 'string']]]
  - [289, 0, 'primary', 'tree', {'unique': true}, [[0, 'unsigned'], [1, 'unsigned']]]
  - [289, 2, 'name', 'tree', {'unique': true}, [[0, 'unsigned'], [2, 'string']]]
  - [296, 0, 'primary', 'tree', {'unique': true}, [[0, 'unsigned']]]
  - [296, 1, 'owner', 'tree', {'unique': false}, [[1, 'unsigned']]]
  - [296, 2, 'name', 'tree', {'unique': true}, [[2, 'string']]]
---
...
```

`box.space._vindex`

`_vindex` – это системный спейс, который реализует виртуальное представление. Структура его кортежей совпадает со структурой кортежей в *_index*, но права доступа на определенные кортежи ограничены в соответствии с правами пользователя. `_vindex` содержит только те кортежи, которые доступны текущему пользователю. Для получения более подробной информации о правах пользователя см. раздел *Управление доступом*.

Если у пользователя есть полный набор прав (как у пользователя „admin“), содержимое `_vindex` совпадает с содержимым `_index`. Если же у пользователя доступ ограничен, `_vindex` содержит только кортежи, которые доступны текущему пользователю.

Примечание:

- `_vindex` – это виртуальное представление системы, поэтому допускаются только запросы на чтение.
- Если спейс `_index` требует наличия соответствующих прав доступа, то любой пользователь всегда может выполнить чтение из `_vindex`.

`box.space._priv`

`_priv` – это системный спейс, где хранятся *права*.

Кортежи в данном спейсе включают в себя следующие поля:

- числовой идентификатор пользователя, который выдал права («grantor_id»),
- числовой идентификатор пользователя, который получил права («grantee_id»),
- the type of object: „space“, „index“, „function“, „sequence“, „user“, „role“, or „universe“,
- числовой идентификатор объекта,
- тип операции: «read» = 1, «write» = 2, «execute» = 4, «create» = 32, «drop» = 64, «alter» = 128, или их комбинация, например «read,write,execute».

Доступны следующие операции:

- Выдача прав с помощью *box.schema.user.grant()*.
- Отмена прав с помощью *box.schema.user.revoke()*.

Примечание:

- Как правило, права выдаются или отменяются владельцем объекта (пользователем, который создал его) или пользователем „admin“.
- До удаления любых объектов или пользователей, убедитесь, что отменили все связанные с ними права.
- Только пользователь *„admin“* может выдавать права на „universe“.
- Только пользователь „admin“ или создатель спейса может удалить, изменить или очистить спейс.
- Только пользователь „admin“ или создатель спейса может изменять change a different user's password.

`box.space._vpriv`

`_vpriv` – это системный спейс, который реализует виртуальное представление. Структура его кортежей совпадает со структурой кортежей в *_priv*, но права доступа на определенные кортежи ограничены в соответствии с правами пользователя. `_vpriv` содержит только те кортежи, которые доступны текущему пользователю. Для получения более подробной информации о правах пользователя см. раздел *Управление доступом*.

Если у пользователя есть полный набор прав (как у пользователя „admin“), содержимое `_vpriv` совпадает с содержимым `_priv`. Если же у пользователя доступ ограничен, `_vpriv` содержит только кортежи, которые доступны текущему пользователю.

Примечание:

- `_vpriv` – это виртуальное представление системы, поэтому допускаются только запросы на чтение.
- Если спейс `_priv` требует наличия соответствующих прав доступа, то любой пользователь всегда может выполнить чтение из `_vpriv`.

`box.space._schema`

`_schema` – это системный спейс.

Этот спейс включает в себя следующие кортежи:

- кортеж `version` с информацией о версии данного экземпляра Tarantool'а,
- кортеж `cluster` с идентификатором набора реплик данного экземпляра,
- кортеж `max_id` с максимальным ID спейса,
- кортежи `once...`, которые соответствуют определенным блокам *box.once()* из *файла инициализации* экземпляра. Первое поле в таких кортежах содержит значение ключа `key` из соответствующего блока `box.once()` с префиксом „once“ (например, *oncehello*), поэтому можно легко найти кортеж, который соответствует определенному блоку `box.once()`.

Пример:

Вот что при обычной установке включает в себя спейс `_schema` (обратите внимание на кортежи для двух блоков `box.once()`: `'oncebye'` и `'oncehello'`):

```
tarantool> box.space._schema:select{}
---
```

(continues on next page)

(продолжение с предыдущей страницы)

```
- - ['cluster', 'b4e15788-d962-4442-892e-d6c1dd5d13f2']
  - ['max_id', 512]
  - ['oncebye']
  - ['oncehello']
  - ['version', 1, 7, 2]
```

box.space._sequence
: _sequence – это системный спейс для поддержки *последовательностей*. Он содержит персистентную информацию, определенную с помощью *box.schema.sequence.create()* или *box.schema.sequence.alter()*.

box.space._sequence_data
: _sequence_data – это системный спейс для поддержки *последовательностей*.

  Каждый кортеж в спейсе _sequence_data содержит два поля:

  - идентификатор последовательности и
  - последнее значение, возвращенное генератором последовательностей (временная информация).

  There is no guarantee that this space will be updated immediately after every data-change request.

box.space._space
: _space – это системный спейс. Он содержит информацию о всех спейсах, хранящихся в данном экземпляре Tarantool - как системные, так и созданные пользователями.

  Кортежи в данном спейсе включают в себя следующие поля:

  - id,
  - owner (= идентификатор пользователя, которому принадлежит спейс),
  - name, engine, field_count,
  - flags (например, временный),
  - format (как задано через *оператор формата*).

  Эти поля определены с помощью *space.create()*.

  Пример №1:

  Следующая функция отобразит все простые поля во всех кортежах спейса _space.

```
function example()
  local ta = {}
  local i, line
  for k, v in box.space._space:pairs() do
    i = 1
    line = ''
    while i <= #v do
      if type(v[i]) ~= 'table' then
        line = line .. v[i] .. ' '
      end
    i = i + 1
    end
    table.insert(ta, line)
  end
  return ta
end
```

Вот что при обычной установке вернет `example()`:

```
tarantool> example()
---
- - '272 1 _schema memtx 0  '
  - '280 1 _space memtx 0  '
  - '281 1 _vspace sysview 0  '
  - '288 1 _index memtx 0  '
  - '296 1 _func memtx 0  '
  - '304 1 _user memtx 0  '
  - '305 1 _vuser sysview 0  '
  - '312 1 _priv memtx 0  '
  - '313 1 _vpriv sysview 0  '
  - '320 1 _cluster memtx 0  '
  - '512 1 tester memtx 0  '
  - '513 1 origin vinyl 0  '
  - '514 1 archive memtx 0  '
...
```

Пример №2:

Следующая серия запросов создаст спейс, используя `box.schema.space.create()` с *оператором формата*, затем выберет кортеж из `_space` для нового спейса. Этот пример иллюстрирует стандартное применение оператора `format`, показывая рекомендованные имена и типы данных для полей.

```
tarantool> box.schema.space.create('TM', {
         >   id = 12345,
         >   format = {
         >     [1] = {["name"] = "field_1"},
         >     [2] = {["type"] = "unsigned"}
         >   }
         > })
---
- index: []
  on_replace: 'function: 0x41c67338'
  temporary: false
  id: 12345
  engine: memtx
  enabled: false
  name: TM
  field_count: 0
- created
...
tarantool> box.space._space:select(12345)
---
- - [12345, 1, 'TM', 'memtx', 0, {}, [{'name': 'field_1'}, {'type': 'unsigned'}]]
...
```

box.space._vspace

`_vspace` – это системный спейс, который реализует виртуальное представление. Структура его кортежей совпадает со структурой кортежей в *_space*, но права доступа на определенные кортежи ограничены в соответствии с правами пользователя. `_vspace` содержит только те кортежи, которые доступны текущему пользователю. Для получения более подробной информации о правах пользователя см. раздел *Управление доступом*.

Если у пользователя есть полный набор прав (как у пользователя „admin“), содержимое `_vspace` совпадает с содержимым `_space`. Если же у пользователя доступ ограничен, `_vspace` содержит только кортежи, которые доступны текущему пользователю.

Примечание:

- `_vspace` – это виртуальное представление системы, поэтому допускаются только запросы на чтение.
- Если спейс `_space` требует наличия соответствующих прав доступа, то любой пользователь всегда может выполнить чтение из `_vspace`.

`box.space._user`

`_user` – это системный спейс, где хранятся имена пользователей и хеши паролей.

Кортежи в данном спейсе включают в себя следующие поля:

- числовой идентификатор кортежа («id»),
- числовой идентификатор создателя кортежа,
- имя,
- тип: „user“ (пользователь) или „role“ (роль),
- пароль по желанию

В спейсе `_user` есть пять специальных кортежей: „guest“, „admin“, „public“, „replication“ и „super“.

Имя	ID	Type	Описание
guest	0	user (пользователь)	Пользователь, который используется по умолчанию при удаленном подключении. Как правило, это не заслуживающий доверия пользователь с небольшим количеством прав.
admin	1	user (пользователь)	Пользователь, который используется по умолчанию при работе с Tarantool'ом как с консолью. Как правило, это *административный пользователь* со всеми правами.
public	2	роль	Заданная *роль*, которая автоматически выдается новым пользователям при их создании методом `box.schema.user.create(имя-пользователя)`. Таким образом, лучше всего выдать права на чтение „read“ спейса „t“ каждому когда-либо созданному пользователю с помощью `box.schema.role.grant('public','read','space','t')`.
replication	3	роль	Заданная *роль*, выдаваемая пользователем „admin“ другим пользователям для использования функций *репликации*.
super	31	роль	Заданная *роль*, выдаваемая пользователем „admin“ другим пользователям для получения всех прав на все объекты. Для роли „super“ такие права выданы на „universe“: чтение, запись, выполнение, создание, удаление, изменение.

Чтобы выбрать кортеж из спейса `_user`, используйте `box.space._user:select()`. Например, при выборке от пользователя с id = 0, который является пользователем „guest“ без пароля по умолчанию, произойдет следующее:

```
tarantool> box.space._user:select{0}
---
```

(continues on next page)

(продолжение с предыдущей страницы)

```
- - [0, 1, 'guest', 'user']
...
```

Предупреждение: Чтобы изменить кортежи в спейсе `_user`, не пользуйтесь стандартными функциями `box.space` для вставки, обновления или удаления. Речь идет об особом спейсе `_user`, поэтому есть особые функции с соответствующей проверкой на ошибки.

Чтобы создать нового пользователя, используйте *box.schema.user.create()*:

```
box.schema.user.create(*имя-пользователя*)
box.schema.user.create(*имя-пользователя*, {if_not_exists = true})
box.schema.user.create(*имя-пользователя*, {password = *пароль*})
```

Чтобы изменить пароль пользователя, воспользуйтесь *box.schema.user.password()*:

```
-- Чтобы изменить пароль текущего пользователя
box.schema.user.passwd(*пароль*)

-- Чтобы изменить пароль другого пользователя
-- (обычно это может делать только 'admin')
box.schema.user.passwd(*имя-пользователя*, *пароль*)
```

Чтобы удалить пользователя, используйте *box.schema.user.drop()*:

```
box.schema.user.drop(*имя-пользователя*)
```

Чтобы проверить, существует ли пользователь, воспользуйтесь *box.schema.user.exists()*, которая вернет `true` (правда) или `false` (ложь):

```
box.schema.user.exists(*имя-пользователя*)
```

Чтобы узнать, какие права есть у пользователя, используйте *box.schema.user.info()*:

```
box.schema.user.info(*имя-пользователя*)
```

Примечание: Максимальное количество пользователей – 32.

Пример:

Ниже представлена сессия, в рамках которой создается новый пользователь с надежным паролем, выбирается кортеж из спейса `_user`, а затем пользователь удаляется.

```
tarantool> box.schema.user.create('JeanMartin', {password = 'Iwtso_6_os$$'})
---
...
tarantool> box.space._user.index.name:select{'JeanMartin'}
---
- - [17, 1, 'JeanMartin', 'user', {'chap-sha1': 't3xjUpQdrt857O+YRvGbMY5py8Q='}]
...
tarantool> box.schema.user.drop('JeanMartin')
---
...
```

box.space._ck_constraint
: _ck_constraint is a system space where check constraints are stored.

  Кортежи в данном спейсе включают в себя следующие поля:

  - the numeric id of the space («space_id»),
  - имя,
  - whether the check is deferred («is_deferred»),
  - the language of the expression, such as „SQL“,
  - the expression («code»)

  Пример:

```
tarantool> box.space._ck_constraint:select()
---
- - [527, 'c1', false, 'SQL', '"f2" > ''A''']
  - [527, 'c2', false, 'SQL', '"f2" == UPPER("f3") AND NOT "f2" LIKE ''__''']
...
```

Пример: использование функций box.space для чтения кортежей из _space

Функция ниже проиллюстрирует, как обращаться ко всем спейсам, и для каждого отобразит примерное количество кортежей и первое поле первого кортежа. В данной функции используются функции из box.space в Tarantool'е: len() и pairs(). Итерация по спейсам закодирована в форме сканирования системного спейса _space, который содержит метаданные. Третье поле в _space содержит имя спейса, поэтому ключевая команда space_name = v[3] означает, что space_name – это поле space_name в кортеже _space, который мы только что получили с помощью pairs(). Функция возвращает таблицу:

```
function example()
  local tuple_count, space_name, line
  local ta = {}
  for k, v in box.space._space:pairs() do
    space_name = v[3]
    if box.space[space_name].index[0] ~= nil then
      tuple_count = '1 or more'
    else
      tuple_count = '0'
    end
    line = space_name .. ' tuple_count =' .. tuple_count
    if tuple_count == '1 or more' then
      for k1, v1 in box.space[space_name]:pairs() do
        line = line .. '. first field in first tuple = ' .. v1[1]
        break
      end
    end
    table.insert(ta, line)
  end
  return ta
end
```

А вот что происходит, когда вызывается функция:

```
tarantool> example()
---
```

(continues on next page)

(продолжение с предыдущей страницы)

```
- - _schema tuple_count =1 or more. first field in first tuple = cluster
  - _space tuple_count =1 or more. first field in first tuple = 272
  - _vspace tuple_count =1 or more. first field in first tuple = 272
  - _index tuple_count =1 or more. first field in first tuple = 272
  - _vindex tuple_count =1 or more. first field in first tuple = 272
  - _func tuple_count =1 or more. first field in first tuple = 1
  - _vfunc tuple_count =1 or more. first field in first tuple = 1
  - _user tuple_count =1 or more. first field in first tuple = 0
  - _vuser tuple_count =1 or more. first field in first tuple = 0
  - _priv tuple_count =1 or more. first field in first tuple = 1
  - _vpriv tuple_count =1 or more. first field in first tuple = 1
  - _cluster tuple_count =1 or more. first field in first tuple = 1
...
```

Пример: использование функций box.space для организации кортежа из _space

Основная цель – отобразить имена и типы полей системного спейса, то есть использование метаданных для поиска метаданных.

Для начала: как можно сделать выборку кортежа из `_space`, который описывает `_space`?

Проще всего проверить постоянные в `box.schema`, что укажет на наличие элемента под названием SPACE_ID == 288. Таким образом, следующие запросы вернут нужный кортеж:

```
box.space._space:select{ 288 }
-- или --
box.space._space:select{ box.schema.SPACE_ID }
```

Также можно обратиться к спейсам в `box.space._index`, что укажет на наличие вторичного индекса с именем „name“ для спейса под номером 288. Таким образом, следующий запрос также вернет нужный кортеж:

```
box.space._space.index.name:select{ '_space' }
```

Однако непросто прочитать информацию из полученного кортежа:

```
tarantool> box.space._space.index.name:select{'_space'}
---
- - [280, 1, '_space', 'memtx', 0, {}, [{'name': 'id', 'type': 'num'}, {'name': 'owner',
        'type': 'num'}, {'name': 'name', 'type': 'str'}, {'name': 'engine', 'type': 'str'},
      {'name': 'field_count', 'type': 'num'}, {'name': 'flags', 'type': 'str'}, {
        'name': 'format', 'type': '*'}]]
...
```

Информация подается бессистемно, поскольку по формату поле №7 содержит рекомендованные имена и типы данных. Как же получить эти данные? Поскольку очевидно, что поле №7 представляет собой ассоциативный массив, цикл *for* проведет организацию данных:

```
tarantool> do
         >   local tuple_of_space = box.space._space.index.name:get{'_space'}
         >   for _, field in ipairs(tuple_of_space[7]) do
         >     print(field.name .. ', ' .. field.type)
         >   end
         > end
id, num
```

(continues on next page)

(продолжение с предыдущей страницы)

```
owner, num
name, str
engine, str
field_count, num
flags, str
format, *
---
...
```

box.space._vuser

_vuser – это системный спейс, который реализует виртуальное представление. Структура его кортежей совпадает со структурой кортежей в *_user*, но права доступа на определенные кортежи ограничены в соответствии с правами пользователя. _vuser содержит только те кортежи, которые доступны текущему пользователю. Для получения более подробной информации о правах пользователя см. раздел *Управление доступом*.

Если у пользователя есть полный набор прав (как у пользователя „admin“), содержимое _vuser совпадает с содержимым _user. Если же у пользователя доступ ограничен, _vuser содержит только кортежи, которые доступны текущему пользователю.

Чтобы посмотреть, как работать с _vuser, *удаленно подключитесь к базе данных Tarantool'a* с помощью tarantoolctl и сделайте выборку кортежей из спейса _user в следующих ситуациях: когда пользователь „guest“ *имеет* и когда он *не имеет* права выполнять чтение данных из базы.

Для начала запустите Tarantool и выдайте пользователю „guest“ права на чтение, запись и выполнение:

```
tarantool> box.cfg{listen = 3301}
---
...
tarantool> box.schema.user.grant('guest', 'read,write,execute', 'universe')
---
...
```

Перейдите на другой терминал, подключитесь к экземпляру Tarantool'a и произведите выборку всех кортежей из спейса _user:

```
$ tarantoolctl connect 3301
localhost:3301> box.space._user:select{}
---
- - [0, 1, 'guest', 'user', {}]
  - [1, 1, 'admin', 'user', {}]
  - [2, 1, 'public', 'role', {}]
  - [3, 1, 'replication', 'role', {}]
  - [31, 1, 'super', 'role', {}]
...
```

Результат включает в себя тот же набор пользователей, как если бы вы выполнили запрос от пользователя „admin“ на своем экземпляре Tarantool'a.

Вернитесь в первый терминал и отмените права на чтение пользователю „guest“:

```
tarantool> box.schema.user.revoke('guest', 'read', 'universe')
---
...
```

Перейдите на другой терминал, остановите сессию (чтобы остановить tarantoolctl, нажмите Ctrl+C или Ctrl+D) и повторите запрос box.space._user:select{}. В доступе отказано:

```
$ tarantoolctl connect 3301
localhost:3301> box.space._user:select{}
---
- error: Read access to space '_user' is denied for user 'guest'
...
```

Тем не менее, если вместо этого произвести выборку из `_vuser`, отображаются данные пользователей, доступные пользователю „guest“:

```
localhost:3301> box.space._vuser:select{}
---
- - [0, 1, 'guest', 'user', {}]
...
```

Примечание:

- `_vuser` – это виртуальное представление системы, поэтому допускаются только запросы на чтение.
- Если спейс `_user` требует наличия соответствующих прав доступа, то любой пользователь всегда может выполнить чтение из `_vuser`.

`box.space._collation`
`_collation` is a system space with a list of *collations*. There are over 270 built-in collations and users may add more. Here is one example:

```
localhost:3301> box.space._collation:select(239)
---
- - [239, 'unicode_uk_s2', 1, 'ICU', 'uk', {'strength': 'secondary'}]
...
```

Explanation of the fields in the example: id = 239 i.e. Tarantool's primary key is 239, name = „unicode_uk_s2“ i.e. according to Tarantool's naming convention this is a Unicode collation + it is for the uk locale + it has secondary strength, owner = 1 i.e. *the admin user*, type = „ICU“ i.e. the rules are according to International Components for Unicode, locale = „uk“ i.e. Ukrainian, opts = „strength:secondary“ i.e. with this collation comparisons use both primary and secondary weights.

`box.space._vcollation`
`_vcollation` is a system space with a list of *collations*. The structure of its tuples is identical to that of *box.space._ collation*, but permissions for certain tuples are limited in accordance with user privileges.

Пример: использование операций с данными

Пример ниже иллюстрирует все возможные сценарии – а также типичные ошибки – для всех *операций с данными* в Tarantool'е: *INSERT*, *DELETE*, *UPDATE*, *UPSERT*, *REPLACE* и *SELECT*.

```
-- Bootstrap the database --
box.cfg{}
format = {}
format[1] = {'field1', 'unsigned'}
format[2] = {'field2', 'unsigned'}
format[3] = {'field3', 'unsigned'}
s = box.schema.create_space('test', {format = format})
-- Create a primary index --
```

(continues on next page)

(продолжение с предыдущей страницы)

```
pk = s:create_index('pk', {parts = {{field = 'field1'}}})
-- Create a unique secondary index --
sk_uniq = s:create_index('sk_uniq', {parts = {{field = 'field2'}}})
-- Create a non-unique secondary index --
sk_non_uniq = s:create_index('sk_non_uniq', {parts = {{field = 'field3'}}, unique = false})
```

INSERT

Операция `insert` (вставка) работает с кортежами с четким форматом и проверяет все ключи на наличие совпадений.

```
tarantool> -- Уникальные индексы: разрешено --
tarantool> s:insert({1, 1, 1})
---
- [1, 1, 1]
...
tarantool> -- Конфликт первичного ключа: ошибка --
tarantool> s:insert({1, 1, 1})
---
- error: Duplicate key exists in unique index 'pk' in space 'test'
...
tarantool> -- Конфликт уникального вторичного ключа: ошибка --
tarantool> s:insert({2, 1, 1})
---
- error: Duplicate key exists in unique index 'sk_uniq' in space 'test'
...
tarantool> -- Ключ {1} присутствует в индексе sk_non_uniq, но он не уникален: разрешено --
tarantool> s:insert({2, 2, 1})
---
- [2, 2, 1]
...
tarantool> s:truncate()
---
...
```

DELETE

`delete` (удаление) работает с полными ключами любого уникального индекса.

`space:delete` – это псевдоним для операции «удалить по первичному ключу».

```
tarantool> -- Вставить некоторые тестовые данные --
tarantool> s:insert{3, 4, 5}
---
- [3, 4, 5]
...
tarantool> s:insert{6, 7, 8}
---
- [6, 7, 8]
...
tarantool> s:insert{9, 10, 11}
---
- [9, 10, 11]
```

(continues on next page)

(продолжение с предыдущей страницы)

```
...
tarantool> s:insert{12, 13, 14}
---
- [12, 13, 14]
...
tarantool> -- Здесь ничего не происходит: нет ключа {4} в индексе pk --
tarantool> s:delete{4}
---
...
tarantool> s:select{}
---
- - [3, 4, 5]
  - [6, 7, 8]
  - [9, 10, 11]
  - [12, 13, 14]
...
tarantool> -- Удалить по первичному ключу: разрешено --
tarantool> s:delete{3}
---
- [3, 4, 5]
...
tarantool> s:select{}
---
- - [6, 7, 8]
  - [9, 10, 11]
  - [12, 13, 14]
...
tarantool> -- Точно удалить по первичному ключу: разрешено --
tarantool> s.index.pk:delete{6}
---
- [6, 7, 8]
...
tarantool> s:select{}
---
- - [9, 10, 11]
  - [12, 13, 14]
...
tarantool> -- Удалить по уникальному вторичному ключу: разрешено --
s.index.sk_uniq:delete{10}
---
- [9, 10, 11]
...
s:select{}
---
- - [12, 13, 14]
...
tarantool> -- Удалить по неуникальному вторичному индексу: ошибка --
tarantool> s.index.sk_non_uniq:delete{14}
---
- error: Get() doesn't support partial keys and non-unique indexes
...
tarantool> s:select{}
---
- - [12, 13, 14]
...
tarantool> s:truncate()
```

(continues on next page)

(продолжение с предыдущей страницы)

```
---
...
```

Ключ должен быть полным: операция `delete` не работает с компонентами ключа.

```
tarantool> s2 = box.schema.create_space('test2')
---
...
tarantool> pk2 = s2:create_index('pk2', {parts = {{field = 1, type = 'unsigned'}, {field = 2, type␣
↪= 'unsigned'}}})
---
...
tarantool> s2:insert{1, 1}
---
- [1, 1]
...
tarantool> -- Delete by a partial key: error --
tarantool> s2:delete{1}
---
- error: Invalid key part count in an exact match (expected 2, got 1)
...
tarantool> -- Delete by a full key: ok --
tarantool> s2:delete{1, 1}
---
- [1, 1]
...
tarantool> s2:select{}
---
- []
...
tarantool> s2:drop()
---
...
```

UPDATE

Как и `delete`, `update` работает с полными ключами любого уникального индекса, а также выполняет операции.

`space:update` – это псевдоним для операции «обновить по первичному ключу».

```
tarantool> -- Вставить некоторые тестовые данные --
tarantool> s:insert{3, 4, 5}
---
- [3, 4, 5]
...
tarantool> s:insert{6, 7, 8}
---
- [6, 7, 8]
...
tarantool> s:insert{9, 10, 11}
---
- [9, 10, 11]
...
tarantool> s:insert{12, 13, 14}
```

(continues on next page)

(продолжение с предыдущей страницы)

```
---
- [12, 13, 14]
...
tarantool> -- Здесь ничего не происходит: нет ключа {4} в индексе pk --
s:update({4}, {{'=', 2, 400}})
---
...
tarantool> s:select{}
---
- - [3, 4, 5]
  - [6, 7, 8]
  - [9, 10, 11]
  - [12, 13, 14]
...
tarantool> -- Обновить по первичному ключу: разрешено --
tarantool> s:update({3}, {{'=', 2, 400}})
---
- [3, 400, 5]
...
tarantool> s:select{}
---
- - [3, 400, 5]
  - [6, 7, 8]
  - [9, 10, 11]
  - [12, 13, 14]
...
tarantool> -- Точно обновить по первичному ключу: разрешено --
tarantool> s.index.pk:update({6}, {{'=', 2, 700}})
---
- [6, 700, 8]
...
tarantool> s:select{}
---
- - [3, 400, 5]
  - [6, 700, 8]
  - [9, 10, 11]
  - [12, 13, 14]
...
tarantool> -- Обновить по уникальному вторичному ключу: разрешено --
tarantool> s.index.sk_uniq:update({10}, {{'=', 2, 1000}})
---
- [9, 1000, 11]
...
tarantool> s:select{}
---
- - [3, 400, 5]
  - [6, 700, 8]
  - [9, 1000, 11]
  - [12, 13, 14]
...
tarantool> -- Обновить по неуникальному вторичному ключу: ошибка --
tarantool> s.index.sk_non_uniq:update({14}, {{'=', 2, 1300}})
---
- error: Get() doesn't support partial keys and non-unique indexes
...
tarantool> s:select{}
```

(continues on next page)

(продолжение с предыдущей страницы)

```
---
- - [3, 400, 5]
  - [6, 700, 8]
  - [9, 1000, 11]
  - [12, 13, 14]
...
tarantool> s:truncate()
---
...
```

UPSERT

`upsert` (обновление и вставка) работает с кортежами с четким форматом и выполняет операции обновления.

Если найден старый кортеж по первичному ключу, то операции обновления применяются к старому кортежу, а новый кортеж игнорируется.

Если старый кортеж не найден, то происходит вставка нового кортежа, а операции обновления **игнорируются**.

Для индексов нет метода `upsert` – это метод для спейса.

```
tarantool> s.index.pk.upsert == nil
---
- true
...
tarantool> s.index.sk_uniq.upsert == nil
---
- true
...
tarantool> s.upsert ~= nil
---
- true
...
tarantool> -- В качестве первого аргумента upsert принимает --
tarantool> -- кортеж с четким форматом, НЕ ключ! --
tarantool> s:insert{1, 2, 3}
---
- [1, 2, 3]
...
tarantool> s:upsert({1}, {{'=', 2, 200}})
---
- error: Tuple field count 1 is less than required by space format or defined indexes
    (expected at least 3)
...
tarantool> s:select{}
---
- - [1, 2, 3]
...
tarantool> s:delete{1}
---
- [1, 2, 3]
...
```

`upsert` превращается в `insert`, когда старый кортеж не найден по первичному ключу.

```
tarantool> s:upsert({1, 2, 3}, {{'=', 2, 200}})
---
...
tarantool> -- Как можно увидеть, произошла вставка {1, 2, 3}, --
tarantool> -- а операции обновления не применились. --
s:select{}
---
- - [1, 2, 3]
...
tarantool> -- Еще одна операция upsert с тем же первичным ключом, --
tarantool> -- но другими значениями прочих полей. --
s:upsert({1, 20, 30}, {{'=', 2, 200}})
---
...
tarantool> -- Старый кортеж был найден по первичному ключу {1}, --
tarantool> -- и применились операции обновления. --
tarantool> -- Новый кортеж игнорируется. --
tarantool> s:select{}
---
- - [1, 200, 3]
...
```

upsert ищет старый кортеж по первичному индексу, НЕ по вторичному. Это может привести к ошибкам с дубликатами, если новый кортеж нарушает уникальность вторичного индекса.

```
tarantool> s:upsert({2, 200, 3}, {{'=', 3, 300}})
---
- error: Duplicate key exists in unique index 'sk_uniq' in space 'test'
...
s:select{}
---
- - [1, 200, 3]
...
tarantool> -- Но сработает, если сохраняется уникальность. --
tarantool> s:upsert({2, 0, 0}, {{'=', 3, 300}})
---
...
tarantool> s:select{}
---
- - [1, 200, 3]
  - [2, 0, 0]
...
tarantool> s:truncate()
---
...
```

REPLACE

replace (замена) работает с кортежами с четким форматом и ищет старый кортеж по первичному ключу нового кортежа.

Если найден старый кортеж, то происходит удаление старого кортежа и вставка нового.

Если старый кортеж не найден, вставляется новый кортеж.

```
tarantool> s:replace{1, 2, 3}
---
- [1, 2, 3]
...
tarantool> s:select{}
---
- - [1, 2, 3]
...
tarantool> s:replace{1, 3, 4}
---
- [1, 3, 4]
...
tarantool> s:select{}
---
- - [1, 3, 4]
...
tarantool> s:truncate()
---
...
```

Как и upsert, replace может нарушить требования уникальности.

```
tarantool> s:insert{1, 1, 1}
---
- [1, 1, 1]
...
tarantool> s:insert{2, 2, 2}
---
- [2, 2, 2]
...
tarantool> -- Такая замена не сработает, поскольку замена новым кортежем {1, 2, 0} --
tarantool> -- старого кортежа по первичному ключу из индекса 'pk' {1, 1, 1}, --
tarantool> -- приведет к созданию дубликата уникального вторичного ключа в индексе 'sk_uniq': --
tarantool> -- ключ {2} используется и в новом кортеже, и в {2, 2, 2}. --
tarantool> s:replace{1, 2, 0}
---
- error: Duplicate key exists in unique index 'sk_uniq' in space 'test'
...
tarantool> s:truncate()
---
...
```

SELECT

select (выборка) работает с любыми индексами (первичными/вторичными) и с любыми ключами (уникальными/неуникальными, полными/компонентами).

Если задан компонент ключа, select выполняет поиск всех ключей, префикс которых совпадает с указанным компонентом ключа.

```
tarantool> s:insert{1, 2, 3}
---
- [1, 2, 3]
...
tarantool> s:insert{4, 5, 6}
---
```

(continues on next page)

(продолжение с предыдущей страницы)

```
- [4, 5, 6]
...
tarantool> s:insert{7, 8, 9}
---
- [7, 8, 9]
...
tarantool> s:insert{10, 11, 9}
---
- [10, 11, 9]
...
tarantool> s:select{1}
---
- - [1, 2, 3]
...
tarantool> s:select{}
---
- - [1, 2, 3]
  - [4, 5, 6]
  - [7, 8, 9]
  - [10, 11, 9]
...
tarantool> s.index.pk:select{4}
---
- - [4, 5, 6]
...
tarantool> s.index.sk_uniq:select{8}
---
- - [7, 8, 9]
...
tarantool> s.index.sk_non_uniq:select{9}
---
- - [7, 8, 9]
  - [10, 11, 9]
...
```

Вложенный модуль *box.stat*

Вложенный модуль `box.stat` предоставляет доступ к статистике Tarantool'а по запросам и использованию сети.

Используйте `box.stat()`, чтобы узнать среднее количество запросов в секунду и общее количество запросов с момента запуска с разбивкой по типу запроса.

Используйте `box.stat.net()`, чтобы просмотреть статистику сетевой активности: количество отправленных и полученных байтов, количество соединений, а также количество активных запросов (текущее, среднее, общее).

Используйте `box.stat.vinyl()`, чтобы просмотреть данные по работе движка базы данных vinyl, например: `box.stat.vinyl().tx` содержит количество коммитов и откатов. Более подробную информацию см. в *конце раздела*.

Используйте `box.stat.reset()`, чтобы сбросить статистику `box.stat()`, `box.stat.net()`, `box.stat.vinyl()` и *box.space.index*.

В таблицах, которые возвращает `box.stat()`:

- `total` обозначает «общее число запросов, обработанных в секунду, с момента запуска сервера»,

- `rps` обозначает «среднее число запросов в секунду за последние 5 секунд».

«ERROR» – это счетчик запросов, которые завершились с ошибкой.

В таблицах, которые возвращает `box.stat.net()`:

- `SENT.rps` и `RECEIVED.rps` – это среднее количество отправленных/полученных байтов в секунду за последние 5 секунд
- `SENT.total` и `RECEIVED.total` – общее число байтов, отправленных/полученных с момента запуска сервера
- `CONNECTIONS.rps` – количество подключений, открытых в секунду, за последние 5 секунд
- `CONNECTIONS.total` – общее количество подключений, открытых с момента запуска сервера
- `REQUESTS.current` – количество запросов, находящихся в обработке, которое может быть ограничено с помощью *box.cfg.net_msg_max*
- `REQUESTS.rps` – число запросов, обработанных в секунду, за последние 5 секунд
- `REQUESTS.total` – общее число запросов, обработанных с момента запуска сервера

```
tarantool> box.stat() -- return 11 tables
---
- DELETE:
    total: 1873949
    rps: 123
  SELECT:
    total: 1237723
    rps: 4099
  INSERT:
    total: 0
    rps: 0
  EVAL:
    total: 0
    rps: 0
  CALL:
    total: 0
    rps: 0
  REPLACE:
    total: 1239123
    rps: 7849
  UPSERT:
    total: 0
    rps: 0
  AUTH:
    total: 0
    rps: 0
  ERROR:
    total: 0
    rps: 0
  EXECUTE:
    total: 0
    rps: 0
  UPDATE:
    total: 0
    rps: 0
...
tarantool> box.stat().DELETE -- total + requests per second from one table
---
```

(continues on next page)

(продолжение с предыдущей страницы)

```
- total: 0
  rps: 0
...
tarantool> box.stat.net() -- 4 tables
---
- SENT:
    total: 0
    rps: 0
  CONNECTIONS:
    current: 0
    rps: 0
    total: 0
  REQUESTS:
    current: 0
    rps: 0
    total: 0
  RECEIVED:
    total: 0
    rps: 0
...
tarantool> box.stat.vinyl().tx.commit -- one item of the vinyl table
---
- 1047632
...
```

Ниже приводится подробная информация о пунктах в `box.stat.vinyl()`.

Подробная информация о box.stat.vinyl().regulator: Регулятор vinyl'а определяет, когда следует предпринимать или отложить действия по дисковому вводу-выводу, путем группировки действий в пакеты так, чтобы обеспечить согласованность и эффективность. Регулятор вызывается планировщиком vinyl'а раз в секунду и обновляет соответствующие переменные при каждом вызове.

- `box.stat.vinyl().regulator.dump_bandwidth` представляет собой предполагаемую среднюю скорость создания дампов. Изначально она составляет 10 485 760 (10 мегабайтов в секунду). Только значительные дампы (более одного мегабайта) используются при оценке.
- `box.stat.vinyl().regulator.dump_watermark` – это точка, когда должно произойти создание дампа. Это значение несколько меньше объема памяти, выделенного для деревьев в vinyl'е, которое указано в параметре *vinyl_memory*.
- `box.stat.vinyl().regulator.write_rate` представляет собой действительную среднюю скорость записи последних данных на диск. Средняя скорость вычисляется в течение 5-секундного интервала, поэтому если за последние 5 секунд ничего не происходило, то `regulator.write_rate = 0`. Скорость `write_rate` может замедлиться во время создания дампа, или если пользователь задал предел *snap_io_rate_limit*.
- `box.stat.vinyl().regulator.rate_limit` – это предел скорости записи в байтах в секунду, который налагается регулятором на основании установленной производительности создания дампов / слияния.

Подробная информация о box.stat.vinyl().disk: Поскольку vinyl является дисковым движком базы данных (в отличие от memtx'а, который представляет собой in-memory движок), он может обрабатывать большие базы данных – однако если база данных больше объема памяти, выделенного для vinyl'а, дисковых операций будет больше.

- `box.stat.vinyl().disk.data` и `box.stat.vinyl().disk.index` содержат объем данных, который поступил в файлы во вложенной директории *vinyl_dir* с именами вида `{lsn}.run` и `{lsn}.index`. Размер файла run зависит от вывода `scheduler.dump_*`.

- `box.stat.vinyl().disk.data_compacted` представляет собой общий размер данных, которые хранятся на последнем уровне LSM-дерева, в байтах. При этом не учитывается сжатие диска. Его можно рассматривать как размер места на диске, которое заняли бы пользовательские данные, если бы не было компрессии, индексирования или увеличения спейса, вызванного конструкцией LSM-дерева.

Подробная информация о box.stat.vinyl().memory: Хотя движок базы данных vinyl не является «in-memory», Tarantool'у всё же требуется память для записи буфера и для кэша:

- `box.stat.vinyl().memory.tuple_cache` содержит количество байтов, используемых для кортежей (данные).
- `box.stat.vinyl().memory.tx` – это транзакционная память, как правило, равная 0.
- `box.stat.vinyl().memory.level0` – это объем памяти уровня 0 «level0», который иногда сокращается до «L0» и представляет собой область, которую vinyl может использовать для хранения данных в оперативной памяти в LSM-дереве.

Таким образом, можно сказать, что «L0 заполняется», когда объем данных в `memory.level0` приближается к максимальному, а именно *regulator.dump_watermark*. Можно ожидать, что «L0 = 0» сразу после создания дампа. Текущий объем в `box.stat.vinyl().memory.page_index` и `box.stat.vinyl().memory.bloom_filter` используется для структур, связанных с индексами. Размер – это количество и размер ключей плюс *vinyl_page_size* плюс *vinyl_bloom_fpr*. Это не счетчик совпадений по фильтру Блума (количество чтений, которых можно избежать, поскольку фильтра Блума предсказывает их наличие в файле типа run) – эта статистика указана в *index_object:stat()*.

Подробная информация о box.stat.vinyl().tx: Информация о запросах, которые влияют на операции транзакций («tx» используется в качестве сокращения слова «транзакция»):

- `box.stat.vinyl().tx.conflict` содержит счетчик конфликтов, которые вызвали откат транзакции.
- `box.stat.vinyl().tx.commit` – это счетчик коммитов (успешно завершенных транзакций). Он включает в себя неявные коммиты, например, любая вставка вызывает коммит, если она не входит в блок begin-end.
- `box.stat.vinyl().tx.rollback` – это счетчик откатов (невыполненные транзакции). Это не просто счетчик явных запросов *box.rollback*, он также включает в себя запросы, которые привели к ошибке. Например, после попытки вставки, в результате которой была выведена ошибка наличия дубликата ключа «Duplicate key exists in unique index», значение счетчика `tx.rollback` увеличивается.
- `box.stat.vinyl().tx.statements`, как правило, будет равен 0.
- `box.stat.vinyl().tx.transactions` содержит количество текущих транзакций.
- `box.stat.vinyl().tx.gap_locks` представляет собой число блокировок разрывов во время выполнения запроса. Чтобы получить низкоуровневое описание имплементации блокировки разрывов в Tarantool'е, см. Блокировка разрывов в менеджере транзакций Vinyl'а.
- `box.stat.vinyl().tx.read_views` показывает, получила ли транзакция статус только для чтения, во избежание временного конфликта. Как правило, 0.

Подробная информация о box.stat.vinyl().scheduler: В основном содержит счетчики, связанные с задачами планировщика по созданию дампов или слиянию: (большинство сбрасываются на 0 при перезапуске сервера или вызове *box.stat.reset()*):

- `box.stat.vinyl().scheduler.compaction_*` содержит объем данных из последних изменений, для которых было произведено *слияние*. Он подразделяется на `scheduler.compaction_input` (объем данных текущего слияния), `scheduler.compaction_queue` (объем данных в ожидании слияния), `scheduler.compaction_time` (общее время, затраченное рабочими потоками на слияние,

в секундах) и `scheduler.compaction_output` (объем данных после слияния, который, предположительно, меньше `scheduler.compaction_input`).

- `box.stat.vinyl().scheduler.tasks_*` содержит информацию о задачах по созданию дампов или слиянию, разделенную на три категории: `scheduler.tasks_inprogress` (текущие), `scheduler.tasks_completed` (успешно завершенные) `scheduler.tasks_failed` (прерванные из-за ошибки).
- `box.stat.vinyl().scheduler_dump_*` содержит объем данных из последних изменений, для которых был создан дамп, включая `dump_time` (общее время, затраченное рабочими потоками на создание дампов, в секундах) и `dump_count` (счетчик созданных дампов), `dump_input` и `dump_output`.

  Понятие «дамп» (dump) объясняется в разделе *Хранение данных с помощью vinyl*:

  > Рано или поздно количество элементов в дереве превысит размер L0. Тогда L0 записывается в файл на диске (который называется забегом – „run“) и освобождается под новые элементы. Эта операция называется „дамп“ (dump).

  Таким образом, можно предсказать создание дампа, если размер L0 (указан в *memory.level0*) приближается к максимальному (указан в *regulator.dump_watermark*), и создание дампа еще не началось. На самом деле Tarantool планирует дамп до достижения предела.

  Дамп также создается во время операции создания *снимка*.

Функция *box.snapshot*

box.snapshot()

Memtx

Создает снимок всех данных и сохраняет его в *memtx_dir*/*<latest-lsn>*.snap. Чтобы сделать снимок, сначала Tarantool входит в режим отложенной сборки мусора по всем данным. В этом режиме *сборщик мусора Tarantool'а* не будет удалять файлы, созданные до начала создания снимка, до тех пор, пока не будет завершено создание снимка. Чтобы сохранить консистентность первичного ключа, используемого для итерации по кортежам, применяется технология копирования при записи. Если главный процесс изменяет часть первичного ключа, страница соответствующего процесса разделяется, и процесс создания снимка получает старую копию страницы. В результате, процесс создания снимка использует многоверсионную параллельную обработку данных, чтобы не скопировать изменения в данных, появившихся уже после начала создания снимка.

Поскольку снимок создается последовательно, можно ожидать высокую скорость записи (в среднем до 80 МБ/секунду на современных дисках), что означает сохранение данных усредненного экземпляра базы данных за несколько минут. Пользователи могут ограничить скорость записи, изменив значение *snap_io_rate_limit*.

Примечание: При условии, что происходят изменения в родительском индексе в ходе многопоточного обновления данных, будет происходить и расщепление страниц, поэтому возникнет необходимость в наличии дополнительной свободной памяти для выполнения этой команды. В среднем, будет достаточно 10% от *memtx_memory*. Оператор подождет окончания создания снимка и вернет результат операции.

Примечание: Обновление: До версии 1.6.6 Tarantool'а процесс создания снимка вызывал клонирование системного процесса (fork), что могло привести к скачкам задержки отклика. Начиная с версии 1.6.6 Tarantool'а, процесс создания снимка создает вид постоянного просмотра, который и записывается в файл снимка с помощью отдельного потока (поток упреждающей записи в журнал).

Хотя `box.snapshot()` не создает ответвление, есть отдельный файбер, который может создавать снимки на регулярной основе – см. обсуждение *демона создания контрольных точек*.

Пример:

```
tarantool> box.info.version
---
- 1.7.0-1216-g73f7154
...
tarantool> box.snapshot()
---
- ok
...
tarantool> box.snapshot()
---
- error: can't save snapshot, errno 17 (File exists)
...
```

Создание снимка не приводит к записи нового журнала упреждающей записи на сервере. После создания снимка старые WAL-файлы можно удалить, если все реплицируемые данные актуальны. Но WAL-файл на момент начала работы `box.snapshot()` следует сохранить на случай восстановления, поскольку он содержит записи журнала после начала работы `box.snapshot()`.

Другим способом сохранения снимка будет отправка сигнала SIGUSR1 процессу. Хотя это может быть удобно, не рекомендуется использовать такой метод в автоматическом процессе: сигнал не дает возможность проверить, был ли корректно сделан снимок.

Vinyl

При использовании vinyl'а вставляемые данные складываются в память до тех пор, пока не будет достигнут предел, установленный в параметре *vinyl_memory*. Затем винил автоматически делает дамп на диск. `box.snapshot()` форсирует создание дампа, чтобы иметь возможность восстановить данные из этой контрольной точки. Файлы снимков хранятся в *space_id*/*index_id*/`*.run`. Таким образом, строго все данные, которые были записаны во время LSN контрольной точки, находятся в **.run* файлах на диске, а все операции, которые происходили после контрольной точки, будут записаны в `*.xlog`. Все файлы дампа, созданные функцией `box.snapshot()`, консистентны и имеют тот же LSN, что и контрольная точка.

На контрольной точке vinyl также пересматривает журнал метаданных `*.vylog`, содержащий операции манипуляции с данными, такие как «создать файл» и «удалить файл». Он проходит по логу, удаляет дублирующие операции из памяти и создает новый файл `*.vylog`, присваивая ему имя в соответствии с *vclock* новой контрольной точки, оставляя только операци создания. Эта процедура очищает `*.vylog` и полезна для восстановления, так как имя лога совпадает с именем подписи контрольной точки.

Вложенный модуль *box.tuple*

Общие сведения

Вложенный модуль `box.tuple` предоставляет доступ только для чтения к пользовательским данным типа кортеж `tuple`. С его помощью для отдельного *кортежа* можно сделать следующее: выборочно искать содержимое поля, получать информацию о размере, проводить итерацию по всем полям и выполнять преобразование в Lua-таблицу.

Индекс

Ниже приведен перечень всех функций модуля `box.tuple`.

Имя	Использование
box.tuple.new()	Создание кортежа
#tuple_object	Подсчет полей кортежа
tuple_object:bsize()	Подсчет байтов в кортеже
tuple_object[field-number]	Получение поля кортежа по номеру
tuple_object[field-name]	Получение поля кортежа по имени
tuple_object[field-path]	Получение полей кортежа или компонентов по пути
tuple_object:find()	Получение номера первого поля, совпадающего с искомым значением
tuple_object:findall()	Получение номеров всех полей, совпадающих с искомым значением
tuple_object:next()	Get the next field value from tuple
tuple_object:ipairs()	Подготовка к итерации
tuple_object:pairs()	Подготовка к итерации
tuple_object:totable()	Получение полей кортежа в виде таблицы
tuple_object:tomap()	Получение полей кортежа в виде таблицы, а также пар ключ-значение
tuple_object:transform()	Удаление (и замена) полей кортежа
tuple_object:unpack()	Получение полей кортежа
tuple_object:update()	Обновление кортежа
tuple_object:upsert()	Update a tuple ignoring errors

`box.tuple.new(`*value*`)`

Создание нового кортежа либо из скаляра, либо из Lua-таблицы. Возможен и вариант получения новых кортежей из запросов *select* или *insert*. или *replace*, или *update* Tarantool'а, которые можно рассматривать в качестве операторов, косвенно выполняющих операцию создания `new()`.

Параметры

- `value` (*lua-value*) – значение, которое станет содержимым кортежа.

возвращается новый кортеж

тип возвращаемого значения кортеж

В следующем примере `x` будет представлять собой новый объект таблицы, который содержит один кортеж, а `t` будет представлять собой объект кортежа. Если ввести команду `t`, будет получен весь кортеж `t`.

Пример:

```
tarantool> x = box.space.tester:insert{
         >   33,
         >   tonumber('1'),
         >   tonumber64('2')
         > }:totable()
---
...
tarantool> t = box.tuple.new{'abc', 'def', 'ghi', 'abc'}
---
...
tarantool> t
---
- ['abc', 'def', 'ghi', 'abc']
...
```

object tuple_object

#<tuple_object>

Оператор # на языке Lua означает «вернуть количество компонентов». Таким образом, если t представляет собой кортеж, то #t вернет количество полей.

тип возвращаемого значения число

В следующем примере создается кортеж под названием t, а затем возвращается количество полей в кортеже t.

```
tarantool> t = box.tuple.new{'Fld#1', 'Fld#2', 'Fld#3', 'Fld#4'}
---
...
tarantool> #t
---
- 4
...
```

tuple_object:bsize()

Если t – это экземпляр кортежа, то t:bsize() вернет количество байтов в кортеже. Как для движка базы данных memtx, так и для движка vinyl максимальное количество, используемое по умолчанию, составляет один мегабайт (*memtx_max_tuple_size* или *vinyl_max_tuple_size*). В каждом поле есть один или более байтов «длины», которые предваряют само содержимое поля, поэтому bsize() вернет значение, которое незначительно больше, чем сумма длин всего содержимого.

Значение не содержит размер кортежа «struct tuple» (чтобы узнать текущий размер данной структуры, посмотрите файл tuple.h в исходном коде Tarantool'а).

возвращается количество байтов

тип возвращаемого значения число

В следующем примере создается кортеж с именем t, в котором три поля, и для каждого поля один байт занимает хранение длины, и три байта занимает хранение содержимого, кроме того, один бит используется на ресурсы, поэтому bsize() вернет 3*(1+3)+1. Такой же размер строки вернула бы функция msgpack.encode({„aaa“,“bbb“,“ccc“}).

```
tarantool> t = box.tuple.new{'aaa', 'bbb', 'ccc'}
---
...
tarantool> t:bsize()
---
- 13
...
```

<tuple_object>(*field-number*)

Если t – это экземпляр кортежа, то t[номер-поля] вернет поле под номером номер-поля в кортеже. Первое поле – это t[1].

возвращается значение поля.

тип возвращаемого значения Lua-значение

В следующем примере создается кортеж под названием t, а затем возвращается второе поле в кортеже t.

```
tarantool> t = box.tuple.new{'Fld#1', 'Fld#2', 'Fld#3', 'Fld#4'}
---
...
tarantool> t[2]
---
- Fld#2
...
```

<tuple_object>(*field-name*)

If `t` is a tuple instance, `t['field-name']` will return the field named „field-name“ in the tuple. Fields have names if the tuple has been retrieved from a space that has an associated *format*. `t[lua-variable-name]` will do the same thing if `lua-variable-name` contains `'field-name'`.

There is a variation which the Lua manual calls «syntactic sugar»: use `t.field-name` as an equivalent of `t['field-name']`.

возвращается значение поля.

тип возвращаемого значения Lua-значение

В следующем примере кортеж под названием `t` возвращается после операции замены, а затем возвращается второе поле с именем „field2“ в кортеже `t`.

```
tarantool> format = {}
---
...
tarantool> format[1] = {name = 'field1', type = 'unsigned'}
---
...
tarantool> format[2] = {name = 'field2', type = 'string'}
---
...
tarantool> s = box.schema.space.create('test', {format = format})
---
...
tarantool> pk = s:create_index('pk')
---
...
tarantool> t = s:replace{1, 'Я'}
---
...
tarantool> t['field2']
---
- Я
...
```

<tuple_object>(*field-path*)

Если `t` – это экземпляр кортежа, то `t['path']` вернет поле или ряд полей, которые находятся в `path`. Параметр `path` должен представлять собой правильную JSON-спецификацию. `path` может содержать имена полей, если кортеж был получен из спейса с заданным *форматом*.

Во избежание неоднозначности Tarantool сначала пытается интерпретировать запрос как *tuple_object[field-number]* или *tuple_object[field-name]*. И только в том случае, если это не удается, Tarantool пытается интерпретировать запрос как `tuple_object[field-path]`.

Путь path должен представлять собой правильную JSON-спецификацию, но в начале может стоять „.“. Символ „.“ означает, что путь выступает в качестве суффикса для кортежа.

При указании пути Tarantool воспользуется им для поиска по телу кортежа и вернет только

тот компонент кортежа, который действительно необходим.

В следующем примере кортеж под названием t возвращается после операции замены, а затем возвращается только необходимый компонент (в данном случае совпадение имени) соответствующего поля. В частности: второе поле, шестой компонент, значение после „value=“.

```
tarantool> format = {}
---
...
tarantool> format[1] = {name = 'field1', type = 'unsigned'}
---
...
tarantool> format[2] = {name = 'field2', type = 'array'}
---
...
tarantool> format[3] = {name = 'field4', type = 'string' }
---
...
tarantool> format[4] = {name = "[2][6]['пш']['Я']", type = 'string'}
---
...
tarantool> s = box.schema.space.create('test', {format = format})
---
...
tarantool> pk = s:create_index('pk')
---
...
tarantool> field2 = {1, 2, 3, "4", {5,6,7}, {пш={Я="п"}, key="V!", value="K!"}}
---
...
tarantool> t = s:replace{1, field2, "123456", "Not K!"}
---
...
tarantool> t["[2][6]['value']"]
---
- K!
...
```

`tuple_object:find(`[*field-number*], *search-value*`)`

`tuple_object:findall(`[*field-number*], *search-value*`)`

Если `t` – это экземпляр кортежа, то `t:find(search-value)` вернет номер первого поля в `t`, которое совпадает с искомым значением, а `t:findall(search-value [, search-value ...])` вернет номера всех колей в `t`, которые совпадают с искомым значением. Можно дополнительно добавить числовой аргумент `field-number` перед search-value, чтобы задать условие “начинать поиск с номера поля `field-number`.”

возвращается номер поля в кортеже.

тип возвращаемого значения число

В следующем примере создается кортеж с именем `t`, а затем: возвращается номер первого поля в `t`, которое совпадает с „a“, затем возвращаются номера всех полей в `t`, которые совпадают с „a“, затем возвращаются номера всех полей в t, которые совпадают с „a“, и находятся на втором месте или далее.

```
tarantool> t = box.tuple.new{'a', 'b', 'c', 'a'}
---
...
```

(continues on next page)

(продолжение с предыдущей страницы)

```
tarantool> t:find('a')
---
- 1
...
tarantool> t:findall('a')
---
- 1
- 4
...
tarantool> t:findall(2, 'a')
---
- 4
...
```

tuple_object:next(*tuple*[, *pos*])

An analogue of the Lua `next()` function, but for a tuple object. When called without arguments, `tuple:next()` returns the first field from a tuple. Otherwise, it returns the field next to the indicated position.

However `tuple:next()` is not really efficient, and it is better to use *tuple:pairs()/ipairs()*.

возвращается field number and field value

тип возвращаемого значения number and field type

```
tarantool> tuple = box.tuple.new({5, 4, 3, 2, 0})
---
...

tarantool> tuple:next()
---
- 1
- 5
...

tarantool> tuple:next(1)
---
- 2
- 4
...

tarantool> ctx, field = tuple:next()
---
...

tarantool> while field do
         > print(field)
         > ctx, field = tuple:next(ctx)
         > end
5
4
3
2
0
---
...
```

tuple_object:pairs()

`tuple_object:ipairs()`

In Lua, lua-table-value:pairs() is a method which returns: `function`, `lua-table-value`, `nil`. Tarantool has extended this so that `tuple-value:pairs()` returns: `function`, `tuple-value`, `nil`. It is useful for Lua iterators, because Lua iterators traverse a value's components until an end marker is reached.

`tuple_object:ipairs()` is the same as `pairs()`, because tuple fields are always integers.

возвращается функция, значение кортежа, nil

тип возвращаемого значения функция, Lua-значение, nil

В следующем примере создается кортеж под названием `t`, а затем все его поля выбираются с помощью Lua-цикла for.

```
tarantool> t = box.tuple.new{'Fld#1', 'Fld#2', 'Fld#3', 'Fld#4', 'Fld#5'}
---
...
tarantool> tmp = ''
---
...
tarantool> for k, v in t:pairs() do
         >   tmp = tmp .. v
         > end
---
...
tarantool> tmp
---
- Fld#1Fld#2Fld#3Fld#4Fld#5
...
```

`tuple_object:totable(`[*start-field-number*[, *end-field-number*]]`)`

Если `t` – это экземпляр кортежа, то `t:totable()` вернет все поля, `t:totable(1)` вернет все поля, начиная с поля №1, `t:totable(1,5)` вернет все поля между полем №1 и полем №5.

Рекомендуется использовать `t:totable()`, а не `t:unpack()`.

возвращается поле или поля из кортежа

тип возвращаемого значения Lua-таблица

В следующем примере создается кортеж под названием `t`, а затем делается выборка всех полей, возвращается результат.

```
tarantool> t = box.tuple.new{'Fld#1', 'Fld#2', 'Fld#3', 'Fld#4', 'Fld#5'}
---
...
tarantool> t:totable()
---
- ['Fld#1', 'Fld#2', 'Fld#3', 'Fld#4', 'Fld#5']
...
```

`tuple_object:tomap(`[*options*]`)`

В Lua-таблице могут быть индексированные значения, которые также называются пары ключ-значение. Например, здесь:

```
a = {}; a['field1'] = 10; a['field2'] = 20
```

`a` – это таблица с «field1: 10» и «field2: 20».

Функция *tuple_object:totable()* вернет только таблицу со значениями. А функция `tuple_object:tomap()` вернет таблицу не только со значениями, но и с парами ключ-значение.

Это сработает только в том случае, если кортеж приходит из спейса, который был форматирован посредством *оператора формата*.

Параметры

- `options` (`table`) – единственный доступный параметр – `names_only`. Если `names_only` принимает значение false или не указан (по умолчанию), то все поля появятся дважды: сначала с числовыми заголовками, а затем с именными заголовками. Если же `names_only` = true, то все поля будут выведены один раз с именными заголовками.

возвращается пары номер-поля:значение и пары ключ:значение из кортежа

тип возвращаемого значения Lua-таблица

В следующем примере возвращается кортеж с именем `t1` из спейса после форматирования, затем таблицы с именами `t1map` и `t1map2` создаются из `t1`.

```
format = {{'field1', 'unsigned'}, {'field2', 'unsigned'}}
s = box.schema.space.create('test', {format = format})
s:create_index('pk',{parts={1,'unsigned',2,'unsigned'}})
t1 = s:insert{10, 20}
t1map = t1:tomap()
t1map_names_only = t1:tomap({names_only=true})
```

`t1map` будет содержать «1: 10», «2: 20», «field1: 10», «field2: 20».

`t1map_names_only` будет содержать «field1: 10» и «field2: 20».

`tuple_object:transform(`*start-field-number*, *fields-to-remove*[, *field-value*, ...]`)`

Если `t` – это экземпляр кортежа, то `t:transform(`*`start-field-number`*`,`*`fields-to-remove`*`)` вернет кортеж, где начиная с поля `start-field-number`, удаляется количество полей (`fields-to-remove`). Дополнительно можно добавить аргументы после `fields-to-remove`, чтобы указать новые значения на замену удаленных.

Если первоначальный кортеж приходит из спейса, который был форматирован посредством *оператора формата*, форматирование возвращаемого кортежа не сохранится.

Параметры

- `start-field-number` (*integer*) – начиная с 1, может быть отрицательным
- `fields-to-remove` (*integer*) –
- `field-value(s)` (*lua-value*) –

возвращается кортеж

тип возвращаемого значения кортеж

В следующем примере создается кортеж под названием `t`, а затем начиная со второго поля, удаляются два поля, а одно новое поле добавляется, затем возвращается результат.

```
tarantool> t = box.tuple.new{'Fld#1', 'Fld#2', 'Fld#3', 'Fld#4', 'Fld#5'}
---
...
tarantool> t:transform(2, 2, 'x')
---
```

(continues on next page)

(продолжение с предыдущей страницы)

```
- ['Fld#1', 'x', 'Fld#4', 'Fld#5']
...
```

`tuple_object:unpack(`[*start-field-number*[, *end-field-number*]])

Если `t` – это экземпляр кортежа, то `t:unpack()` вернет все поля, `t:unpack(1)` вернет все поля, начиная с поля №1, `t:unpack(1,5)` вернет все поля между полем №1 и полем №5.

возвращается поле или поля из кортежа.

тип возвращаемого значения Lua-значение(я)

В следующем примере создается кортеж под названием `t`, а затем делается выборка всех полей, возвращается результат.

```
tarantool> t = box.tuple.new{'Fld#1', 'Fld#2', 'Fld#3', 'Fld#4', 'Fld#5'}
---
...
tarantool> t:unpack()
---
- Fld#1
- Fld#2
- Fld#3
- Fld#4
- Fld#5
...
```

`tuple_object:update(`*{{operator, field_ no, value}, ...}*)

Обновление кортежа.

Эта функция обновляет кортеж, который находится не в спейсе. Ср. функцию `box.space.`*`space-name`*`:update(`*`key`*`, {{`*`format`*`, `*`field_no`*`, `*`value`*`}, ...})`, которая обновляет кортеж в спейсе.

Более подробную информацию см. в описании `operator`, `field_no` и `value` в разделе *box.space.space-name:update{key, format, {field_ number, value}. . .)*.

Если первоначальный кортеж приходит из спейса, который был форматирован посредством *оператора формата*, форматирование возвращаемого кортежа сохранится.

Параметры

- `operator` (`string`) – тип операции, представленный строкой (например, „=“ означает „присвоить новое значение“)
- `field_no` (*`number`*) – к какому полю применяется операция. Номер поля может быть отрицательным, что означает, что позиция рассчитывается с конца кортежа. (#кортеж + отрицательный номер поля + 1)
- `value` (*`lua_value`*) – какое значение применяется

возвращается новый кортеж

тип возвращаемого значения кортеж

В следующем примере создается кортеж под названием `t`, а затем второе поле обновляется до равного „B“.

```
tarantool> t = box.tuple.new{'Fld#1', 'Fld#2', 'Fld#3', 'Fld#4', 'Fld#5'}
---
...
```

(continues on next page)

(продолжение с предыдущей страницы)

```
tarantool> t:update({{'=', 2, 'B'}})
---
- ['Fld#1', 'B', 'Fld#3', 'Fld#4', 'Fld#5']
...
```

`tuple_object:upsert(`*{{operator, field_no, value}, ...}*`)`

The same as `tuple_object:update()`, but ignores errors. In case of an error the tuple is left intact, but an error message is printed. Only client errors are ignored, such as a bad field type, or wrong field index/name. System errors, such as OOM, are not ignored and raised just like with a normal `update()`. Note that only bad operations are ignored. All correct operations are applied.

Параметры

- `operator` (`string`) – operation type represented as a string (e.g. „=“ for „assign new value“)
- `field_no` (*`number`*) – the field to which the operation will be applied. The field number can be negative, meaning the position from the end of tuple. (#tuple + negative field number + 1)
- `value` (*`lua_value`*) – the value which will be applied

возвращается новый кортеж

тип возвращаемого значения кортеж

See the following example where one operation is applied, and one is not.

```
tarantool> t = box.tuple.new({1, 2, 3})
tarantool> t2 = t:upsert({{'=', 5, 100}})
UPSERT operation failed:
ER_NO_SUCH_FIELD_NO: Field 5 was not found in the tuple
---
...

tarantool> t
---
- [1, 2, 3]
...

tarantool> t2
---
- [1, 2, 3]
...

tarantool> t2 = t:upsert({{'=', 5, 100}, {'+', 1, 3}})
UPSERT operation failed:
ER_NO_SUCH_FIELD_NO: Field 5 was not found in the tuple
---
...

tarantool> t
---
- [1, 2, 3]
...

tarantool> t2
---
```

(continues on next page)

(продолжение с предыдущей страницы)

```
- [4, 2, 3]
...
```

Пример

Представленная ниже функция проиллюстрирует, как можно преобразовать кортежи в Lua-таблицы и списки скаляров и обратно:

```
tuple = box.tuple.new({scalar1, scalar2, ... scalar_n}) -- скаляры в кортеж
lua_table = {tuple:unpack()}                            -- кортеж в Lua-таблицу
lua_table = tuple:totable()                             -- кортеж в Lua-таблицу
scalar1, scalar2, ... scalar_n = tuple:unpack()         -- кортеж в скаляры
tuple = box.tuple.new(lua_table)                        -- Lua-таблицу в кортеж
```

Затем она найдет поле, которое содержит значение „b“, удалит это поле из кортежа и отобразит количество байтов, оставшихся в кортеже. Данная функция использует следующие функции `box.tuple` Tarantool'a: `new()`, `unpack()`, `find()`, `transform()`, `bsize()`.

```
function example()
  local tuple1, tuple2, lua_table_1, scalar1, scalar2, scalar3, field_number
  local luatable1 = {}
  tuple1 = box.tuple.new({'a', 'b', 'c'})
  luatable1 = tuple1:totable()
  scalar1, scalar2, scalar3 = tuple1:unpack()
  tuple2 = box.tuple.new(luatable1[1],luatable1[2],luatable1[3])
  field_number = tuple2:find('b')
  tuple2 = tuple2:transform(field_number, 1)
  return 'tuple2 = ' , tuple2 , ' # of bytes = ' , tuple2:bsize()
end
```

... А вот что происходит, когда вызывается функция:

```
tarantool> example()
---
- tuple2 =
- ['a', 'c']
- ' # of bytes = '
- 5
...
```

Управление экземплярами

Общие сведения

Чтобы получить общую информацию и взглянуть на примеры использования, см. раздел *Управление транзакциями*.

Соблюдайте следующие правила в работе с транзакциями:

Правило #1

Запросы в транзакции должны отправляться на сервер в виде единого блока. Недостаточно просто размещать их между началом транзакции и коммитом или откатом. Чтобы убедиться, что они отправ-

ляются в виде единого блока: поместите их в функцию, поместите их на одну строку или используйте символы-разделители, чтобы многостроковые запросы обрабатывались совместно.

Правило #2

Все операции с базой данных в рамках транзакции должны работать с одним движком баз данных. Небезопасно в рамках одной транзакции получать доступ к наборам кортежей, которые определяются по `{engine='vinyl'}`, а также к наборам кортежей, которые определяются по `{engine='memtx'}`.

Правило #3

Requests which cause changes to the data definition – create, alter, drop, truncate – are only allowed with Tarantool version 2.1 or later. Data-definition requests which change an index or change a format, such as *space_object:create_index()* and *space_object:format()*, are not allowed inside transactions except as the first request after `box.begin()`.

Индекс

Ниже приведен перечень всех функций для управления транзакциями.

Имя	Использование
box.begin()	Начало транзакции
box.commit()	Окончание транзакции и сохранение всех изменений
box.rollback()	Окончание транзакции и отмена всех изменений
box.savepoint()	Получение дескриптора точки сохранения
box.rollback_to_savepoint()	Запрещение окончания транзакции и отмена всех изменений, сделанных после точки сохранения
box.atomic()	Выполнение функции как транзакции
box.on_commit()	Определение триггера, активируемого по `box.commit`
box.on_rollback()	Определение триггера, активируемого по `box.rollback`
box.is_in_txn()	Обозначение наличия активной транзакции

`box.begin()`
Начало транзакции. Отключение *неявной передачи управления* до окончания транзакции. Сигнал о записи в *журнал упреждающей записи* будет задержан до окончания транзакции. Фактически файбер, который выполняет функцию `box.begin()`, начинает «активную транзакцию со множеством запросов» с блокировкой всех остальных файберов.

Возможные ошибки: ошибка, если такая операция не допускается, потому что уже есть активная транзакция. ошибка, если по какой-либо причине нельзя выделить память.

`box.commit()`
Окончание транзакции и применение результатов всех операций по изменению данных.

Возможные ошибки: ошибка и прерывание транзакции в случае конфликта. ошибка, если операция не может выполнить запись на диск. ошибка, если по какой-либо причине нельзя выделить память.

`box.rollback()`
Окончание транзакции, но отмена результатов всех операций по изменению данных. Явный вызов

функций не из модуля `box.space`, которые всегда передают управление, например *fiber.sleep()* или *fiber.yield()*, приведет к тому же результату.

`box.savepoint()`
Возврат дескриптора точки сохранения (тип = таблица), который может затем использоваться в *box.rollback_to_savepoint(savepoint)*. Точки сохранения могут быть созданы, пока активна транзакция, и удаляются после окончания транзакции.

возвращается таблица точки сохранения

тип возвращаемого значения Lua-объект

возвращается ошибка, если точку сохранения нельзя указать в отсутствие активной транзакции.

Возможные ошибки: ошибка, если по какой-либо причине нельзя выделить память.

`box.rollback_to_savepoint(`*savepoint*`)`
Запрещение окончания транзакции, но отмена всех изменений и операций *box.savepoint()*, сделанных после точки сохранения.

возвращается ошибка, если точку сохранения нельзя указать в отсутствие активной транзакции.

Возможные ошибки: ошибка, если отсутствует точка сохранения.

Пример:

```
function f()
  box.begin()           -- начало транзакции
  box.space.t:insert{1} -- это не отменится
  local s = box.savepoint()
  box.space.t:insert{2} -- это отменится
  box.rollback_to_savepoint(s)
  box.commit()          -- конец транзакции
end
```

`box.atomic(`*tx-function*`[, `*function-arguments*`])`
Выполнение функции так, как будто функция начинается с явного вызова *box.begin()* и заканчивается неявным вызовом *box.commit()* после успешного выполнения или же заканчивается неявным вызовом *box.rollback()* в случае ошибки.

возвращается результат функции передается в `atomic()` в качестве аргумента.

Возможные ошибки: любая ошибка, которую могут вызвать *box.begin()* и *box.commit()*.

`box.on_commit(`*trigger-function*`[, `*old-trigger-function*`])`
Определения триггера, выполняемого в случае окончания транзакции в связи с *box.commit*.

Функция с триггером может принимать параметр с итератором, как описано в примере к данному разделу.

Функция с триггером не должна получать доступ к любым спейсам базы данных.

Если триггер не сработает и выдаст ошибку, результат будет неблагоприятным, чего следует избегать – используйте Lua-механизм `pcall()` вокруг кода, который может не сработать.

`box.on_commit()` следует вызывать в пределах транзакции, и триггер прекращает существование по окончании транзакции.

Параметры

- `trigger-function` (*function*) – функция, в которой будет триггер
- `old-trigger-function` (*function*) – существующая функция с триггером, которую заменит новая

возвращается nil или указатель функции

Если указаны параметры `(nil, old-trigger-function)`, старый триггер будет удален.

Подробная информация о характеристиках триггера находится в разделе *Триггеры*.

Простой и бесполезный пример: покажет, что произошел коммит:

```
function f()
function f() print('commit happened') end
box.begin() box.on_commit(f) box.commit()
```

Но, конечно, это еще не всё: параметр функции может быть ИТЕРАТОРОМ.

Итератор проходит по результатам каждого запроса изменения спейса в пределах транзакции.

Итератор будет содержать:

- порядковый номер запроса,
- старое значение кортежа до запроса (для запросов вставки это будет нулевое значение nil),
- новое значение кортежа после запроса (для запросов удаления это будет нулевое значение nil),
- и идентификатор спейса.

Более сложный и более полезный пример: покажет результат двух запросов замены:

```
box.space.test:drop()
s = box.schema.space.create('test')
i = box.space.test:create_index('i')
function f(iterator)
  for request_number, old_tuple, new_tuple, space_id in iterator() do
    print('request_number ' .. tostring(request_number))
    print('  old_tuple ' .. tostring(old_tuple[1]) .. ' ' .. old_tuple[2])
    print('  new_tuple ' .. tostring(new_tuple[1]) .. ' ' .. new_tuple[2])
    print('  space_id ' .. tostring(space_id))
  end
end
s:insert{1,'-'}
box.begin() s:replace{1,'x'} s:replace{1,'y'} box.on_commit(f) box.commit()
```

Результат будет выглядеть следующим образом:

```
tarantool> box.begin() s:replace{1,'x'} s:replace{1,'y'} box.on_commit(f) box.commit()
request_number 1
  old_tuple 1 -
  new_tuple 1 x
  space_id 517
request_number 2
  old_tuple 1 x
  new_tuple 1 y
  space_id 517
```

`box.on_rollback(`*trigger-function*`[, `*old-trigger-function*`])`

Определение триггера, выполняемого по окончании транзакции в связи с *box.rollback*.

Используются точно такие же параметры и предупреждения, как в *box.on-commit*.

`box.is_in_txn()`
В процессе транзакции (например, пользователь вызвал *box.begin* и еще не вызвал ни *box.commit*, ни *box.rollback*) возвращается `true`. В остальных случаях возвращается `false`.

Functions for SQL

The `box` module contains two functions related to SQL:

- `box.internal.sql_create_function` – for making Lua functions callable from SQL statements. This, or an SQL statement with the same effect, will be part of the documentation regarding SQL Plus Lua.
- `box.execute` – for making SQL statements callable from Lua functions.

Some SQL statements are illustrated in the *SQL tutorial*.

`box.execute(`*sql-statement*[, *extra-parameters*]`)`
Execute the SQL statement contained in the sql-statement parameter.

Параметры

- `sql-statement` (`string`) – statement, which should conform to the rules for SQL grammar
- `extra-parameters` (`table`) – optional table for placeholders in the statement

возвращается depends on statement

There are two ways to pass extra parameters for `box.execute()`:

- The first way, which is the preferred way, is to put placeholders in the string, and pass a second argument, an *extra-parameters* table. A placeholder is either a question mark «?», or a colon «:» followed by a name. An extra parameter is any Lua expression. If placeholders are question marks, then they will be replaced by extra-parameter values in corresponding positions, that is, the first ? will be replaced by the first extra parameter, the second ? will be replaced by the second extra parameter, and so on. If placeholders are :names, then they will be replaced by extra-parameter values with corrresponding names. For example this request which contains literal values 1 and „x“: `box.execute([[INSERT INTO tt VALUES (1, 'x');]]);` is the same as this request which contains two question-mark placeholders (`?` and `?`) and a two-element extra-parameters table: `x = {1,'x'} box.execute([[INSERT INTO tt VALUES (?, ?);]], x);` and is the same as this request which contains two :name placeholders (`:a` and `:b`) and a two-element extra-parameters table with elements named «a» and «b»: `box.execute([[INSERT INTO tt VALUES (:a, :b); ]], {{[':a']=1},{[':b']='x'}})`
- The second way is to concatenate strings. For example, this Lua script will insert 10 rows with different primary-key values into table t: `for i=1,10,1 do box.execute("insert into t values (" .. i .. ")") end` When creating SQL statements based on user input, application developers should beware of SQL injection.

Since `box.execute()` is an invocation of a Lua function, it either causes an error message or returns a value.

For some statements the returned value will contain a field named «rowcount». For example;

```
tarantool> box.execute([[CREATE TABLE table1 (column1 INT PRIMARY key, column2 VARCHAR(10));
↪]])
---
- rowcount: 1
...
```

(continues on next page)

(продолжение с предыдущей страницы)

```
tarantool> box.execute([[INSERT INTO table1 VALUES (55,'Hello SQL world!');]])
---
- rowcount: 1
...
```

For statements that cause generation of values for PRIMARY KEY AUTOINCREMENT columns, there will also be a field named «autoincrement_ids».

For SELECT statements the returned value will contain a field named «metadata» (a table with column names and data types) and a field named «rows» (a table with the result set). For example:

```
tarantool> box.execute([[SELECT * FROM table1 WHERE column1 > 0;]])
---
- metadata:
  - name: COLUMN1
    type: integer
  - name: COLUMN2
    type: string
  rows:
  - [55, 'Hello SQL world!']
...
```

The result structure contains Tarantool/NoSQL data type names in MsgPack format. For example, for a statement SELECT «x» FROM t WHERE «x»=5; where «x» is an integer column and there is one row, the raw data for the result set will look like this:

```
dd 00 00 00 01                  1-element array
82                              2-element map (for metadata + rows)
a8 6d 65 74 61 64 61 74 61      string = "metadata"
91                              1-element array (for column count)
82                              2-element map (for name + type)
a4 6e 61 6d 65                  string = "name"
a1 78                           string = "x"
a4 74 79 70 6                   string = "type"
a7 69 6e 74 65 67 65 72         string = "integer"
a4 72 6f 77 73                  string = "rows"
91                              1-element array (for row count)
91                              1-element array (for field count)
05                              contents
```

The order of components within a map is not guaranteed.

Alternative: if you are using the Tarantool server as a client, you can switch languages thus:

```
\set language sql
\set delimiter ;
```

Afterwards, you can enter any SQL statement directly without needing `box.execute()`.

There is also an `execute()` function available via *module net.box*, for example after `conn = net_box.connect(url-string)` one can say `conn:execute(sql-statement])`.

5.2.2 Модуль *buffer*

Модуль `buffer` возвращает буфер, допускающий динамическое изменение размера, который используется только в качестве опции для методов *модуля net.box* или *модуля msgpack*.

Как правило, модуль `net.box` возвращает Lua-таблицу. Если используется опция `buffer`, то методы модуля `net.box` возвращают неформатированную строку *строку MsgPack*. Это экономит время работы на сервере, если в клиентском приложении есть собственная процедура декодирования MsgPack-строк.

Буфер использует четыре указателя для управления его мощностью:

- `buf` – указатель на начало буфера
- `rpos` – указатель на начало участка памяти, доступного для чтения данных («read position»)
- `wpos` – указатель на конец участка памяти для чтения и начало участка для записи данных («write position»)
- `epos` – указатель на конец участка для записи данных («end position»)

`buffer.ibuf()`

Создать новый буфер.

Пример:

В этом примере мы покажем, что использование буфера позволит вам сохранить данные в том же формате, в котором они пришли с сервера. Так что если получить данные с сервера нужно только для отправки куда-то дальше, то с буфером это будет гораздо быстрее.

```
box.cfg{listen = 3301}
buffer = require('buffer')
net_box = require('net.box')
msgpack = require('msgpack')

box.schema.space.create('tester')
box.space.tester:create_index('primary')
box.space.tester:insert({1, 'ABCDE', 12345})

box.schema.user.create('usr1', {password = 'pwd1'})
box.schema.user.grant('usr1', 'read,write,execute', 'space', 'tester')

ibuf = buffer.ibuf()

conn = net_box.connect('usr1:pwd1@localhost:3301')
conn.space.tester:select({}, {buffer=ibuf})

msgpack.decode_unchecked(ibuf.rpos)
```

Результат последнего запроса выглядит следующим образом:

```
tarantool> msgpack.decode_unchecked(ibuf.rpos)
---
- {48: [['ABCDE', 12345]]}
- 'cdata<char *>: 0x7f97ba10c041'
...
```

Примечание: До версии 1.7.7 Tarantool'a в данном случае следует использовать функцию `msgpack.ibuf_decode(ibuf.rpos)`. Начиная с версии 1.7.7 Tarantool'a , `ibuf_decode` объявлена устаревшей.

`object buffer_object`

buffer_object:alloc(*size*)
Аллоцировать size байтов для buffer_object'а.

Параметры

- size (*number*) – количество байтов для аллоцирования

возвращает wpos

buffer_object:capacity()
Вернуть мощность buffer_object'а.

возвращает epos - buf

buffer_object:checksize(*size*)
Проверить, доступно ли size байтов для чтения из buffer_object'а.

Параметры

- size (*number*) – память в байтах для проверки

возвращает rpos

buffer_object:pos()
Вернуть размер участка, занятого данными.

возвращает rpos - buf

buffer_object:read(*size*)
Прочитать size байтов из буфера.

buffer_object:recycle()
Очистить слоты памяти, выделенные для buffer_object'а.

```
tarantool> ibuf:recycle()
---
...
tarantool> ibuf.buf, ibuf.rpos, ibuf.wpos, ibuf.epos
---
- 'cdata<char *>: NULL'
- 'cdata<char *>: NULL'
- 'cdata<char *>: NULL'
- 'cdata<char *>: NULL'
...
```

buffer_object:reset()
Очистить слоты памяти, использованные buffer_object'ом. Этот метод позволяет сохранить буфер, но убрать из него все данные. Это полезно, если вы собираетесь использовать буфер дальше.

```
tarantool> ibuf:reset()
---
...
tarantool> ibuf.buf, ibuf.rpos, ibuf.wpos, ibuf.epos
---
- 'cdata<char *>: 0x010cc28030'
- 'cdata<char *>: 0x010cc28030'
- 'cdata<char *>: 0x010cc28030'
- 'cdata<char *>: 0x010cc2c000'
...
```

`buffer_object:reserve(`*size*`)`
: Зарезервировать память для `buffer_object`. Проверить, достаточно ли памяти, чтобы записать `size` байтов после `wpos`. Если нет, `epos` будет сдвигаться, пока `size` байтов не будет доступно.

`buffer_object:size()`
: Вернуть участок, доступный для чтения данных.

  возвращает `wpos - rpos`

`buffer_object:unused()`
: Вернуть участок, доступный для записи данных.

  возвращает `epos - wpos`

Модуль buffer и skip_header

В примере из предыдущего раздела

```
tarantool> msgpack.decode_unchecked(ibuf.rpos)
---
- {48: [['ABCDE', 12345]]}
- 'cdata<char *>: 0x7f97ba10c041'
...
```

было показано, что обычно net.box ответ включает в себя заголовок – 48 (в шестнадцатиричной системе – 30), который является *ключом* для IPROTO_DATA. Но в некоторых ситуациях, например, при передаче буфера в функцию C, которая ожидает массив байт MsgPack без заголовка, заголовок можно пропустить. Это делается путем указания `skip_header=true` в качестве опции *conn.space.space-name:select{...}* или *conn.space.space-name:insert{...}* или *conn.space.space-name:replace{...}* или *conn.space.space-name:update{...}* или *conn.space.space-name:upsert{...}* или *conn.space.space-name:delete{...}*. По умолчанию `skip_header=false`.

Ниже приведен конец того же примера, но с использованием `skip_header=true`.

```
ibuf = buffer.ibuf()

conn = net_box.connect('usr1:pwd1@localhost:3301')
conn.space.tester:select({}, {buffer=ibuf, skip_header=true})

msgpack.decode_unchecked(ibuf.rpos)
```

Результат последнего запроса выглядит следующим образом:

```
tarantool> msgpack.decode_unchecked(ibuf.rpos)
---
- {48: [['ABCDE', 12345]]}
- 'cdata<char *>: 0x7f97ba10c041'
...
```

Заметьте, что заголовок IPROTO_DATA (48) ушел.

Результат остается внутри массива, что видно из того, что он заключен в квадратные скобки. Пропустить заголовок массива можно и функцией `msgpack.decode_array_header()`.

5.2.3 Модуль *clock*

Общие сведения

Модуль `clock` возвращает значения времени, полученных из функции Posix / C CLOCK_GETTIME или аналогичной. Большинство функций модуля возвращают число секунд; функции, названия которых заканчиваются на «64», возвращают 64-разрадяное число наносекунд.

Указатель

Ниже приведен перечень всех функций модуля `clock`.

Имя	Назначение
clock.time() clock.realtime()	Получение физического времени в секундах
clock.time64() clock.realtime64()	Получение физического времени в наносекундах
clock.monotonic()	Получение монотонного времени в секундах
clock.monotonic64()	Получение монотонного времени в наносекундах
clock.proc()	Получение времени процессора в секундах
clock.proc64()	Получение времени процессора в наносекундах
clock.thread()	Получение рабочего времени потока в секундах
clock.thread64()	Получение рабочего времени потока в наносекундах
clock.bench()	Измерение времени, которое функция проводит в процессоре

```
clock.time()
clock.time64()
clock.realtime()
clock.realtime64()
```

Физическое время в секундах. Получено из C-функции clock_gettime(CLOCK_REALTIME). Использование этой функции лучше всего подходит для выяснения официального времени, как установлено системным администратором.

возвращает секунды или наносекунды с начала отсчета (1970-01-01 00:00:00), значение корректируется.

тип возвращаемого значения число или 64-разрядное число

Пример:

```
-- Результатом будет примерное число лет с 1970.
clock = require('clock')
print(clock.time() / (365*24*60*60))
```

См. также *fiber.time64* и стандартную Lua-функцию os.clock.

```
clock.monotonic()
clock.monotonic64()
```

Монотонное время. Получено из C-функции clock_gettime(CLOCK_MONOTONIC). Монотонное время похоже на физическое время, но на него не влияют изменения для перехода на летнее время или изменения, сделанные пользователем. Такую функцию лучше всего использовать для эталонного тестирования, где необходимо рассчитать затраченное время.

возвращает секунды или наносекунды с момента последней загрузки компьютера.

тип возвращаемого значения число или 64-разрядное число

Пример:

```
-- Результатом будет число наносекунд с запуска.
clock = require('clock')
print(clock.monotonic64())
```

clock.proc()
clock.proc64()

Время процессора. Получено из C-функции `clock_gettime(CLOCK_PROCESS_CPUTIME_ID)`. Такую функцию лучше всего использовать для эталонного тестирования, где необходимо рассчитать время, затраченное на процессоре.

возвращает секунды или наносекунды с момента начала работы процессора.

тип возвращаемого значения число или 64-разрядное число

Пример:

```
-- Результатом будет число наносекунд с запуска процессора.
clock = require('clock')
print(clock.proc64())
```

clock.thread()
clock.thread64()

Рабочее время потока. Получено из C-функции `clock_gettime(CLOCK_THREAD_CPUTIME_ID)`. Такую функцию лучше всего использовать для эталонного тестирования, где необходимо рассчитать время, затраченное потоком на процессоре.

возвращает секунды или наносекунды с момента начала работы потока процессора транзакций.

тип возвращаемого значения число или 64-разрядное число

Пример:

```
-- Результатом будет число секунд с момента начала работы потока.
clock = require('clock')
print(clock.thread64())
```

clock.bench(*function*[, ...])

Время, которое функция проводит в процессоре. Данная функция использует `clock.proc()`, то есть рассчитывает затраченное процессором время. Таким образом, она не используется для отображения фактически затраченного времени.

Параметры

- `function` (*function*) – функция или ссылка на функцию
- `...` – значения, которые необходимы для функции.

возвращает таблица. Первый элемент – время работы процессора в секундах, второй элемент – то, что возвращает функция.

Пример:

```
-- Эталонное тестирование функции, которая находится в спящем режиме в течение 10 секунд.
-- NB: bench() не будет рассчитывать время сна.
-- Поэтому вернется значение, которое будет {число менее 10, 88}.
clock = require('clock')
fiber = require('fiber')
function f(param)
  fiber.sleep(param)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
  return 88
end
clock.bench(f, 10)
```

5.2.4 Модуль *console*

Общие сведения

Модуль *console* позволяет одному экземпляру Tarantool'а получать доступ к другому экземпляру Tarantool'а и позволяет одному экземпляру Tarantool'а начать прослушивание по *порту администрирования*.

Указатель

Ниже приведен перечень всех функций модуля `console`.

Имя	Назначение
console.connect()	Подключение к экземпляру
console.listen()	Прослушивание входящих запросов
console.start()	Запуск консоли
console.ac()	Установка флага автодополнения ввода
console.delimiter()	Настройка разделителя
console.get_default_output()	Get default output format
console.set_default_output()	Set default output format

`console.connect(`*uri*`)`

Подключение к экземпляру по *URI*, смена командной строки с „`tarantool>`“ на „*uri*`>`“ и дальнейшая работа в качестве клиента до окончания сессии пользователя или ввода команды `control-D`.

Функция console.connect позволяет одному экземпляру Tarantool'а в интерактивном режиме получать доступ к другому экземпляру Tarantool'а. Последующие запросы на первый взгляд будут обрабатываться локально, но в действительности запросы отправляются на удаленный экземпляр, а локальный экземпляр выступает в виде клиента. После успешного подключения командная строка сменится, и последующие запросы отправляются и выполняются на удаленном экземпляре. Результат выводится на локальный экземпляр. Чтобы вернуться к работе на локальном экземпляре, введите команду `control-D`.

Если экземпляр Tarantool'а по `URI` запрашивает авторизацию, подключение может выглядеть следующим образом: `console.connect('admin:secretpassword@distanthost.com:3301')`.

Нет ограничений по типу вводимых запросов, кроме ограничений по правам на выполняемые запросы – по умолчанию, вход в систему на удаленном экземпляре выполняется от имени пользователя „guest“. Можно разрешить работу на удаленном экземпляре, выдав права: `box.schema.user.grant('guest','execute','universe')`.

Параметры

- `uri` (`string`) – URI удаленного экземпляра

возвращает nil

Возможные ошибки: подключение не будет установлено, если целевой экземпляр Tarantool'а не был инициирован с помощью `box.cfg{listen=...}`.

Пример:

```
tarantool> console = require('console')
---
...
tarantool> console.connect('198.18.44.44:3301')
---
...
198.18.44.44:3301> -- командная строка показывает, что работа идет с удаленным экземпляром
```

`console.listen(`*uri*`)`

Прослушивание по *URI*. Основной способ прослушивания на предмет входящих запросов – по строке информации о подключении, или URI, указанному в `box.cfg{listen=...}`. Другой способ прослушивания – по URI, указанному в `console.listen(...)`. Этот другой способ называется «административным» или просто *«по порту администрирования»*. Такое прослушивание обычно осуществляется по локальному хосту с доменным Unix-сокетом.

Параметры

- `uri` (string) – URI локального экземпляра

«Административный» адрес – это URI для прослушивания. У него нет значения по умолчанию, поэтому следует указать, будет ли подключение производиться по порту администрирования. Параметр выражен URI = Универсальным идентификатором ресурса, например «/tmpdir/unix_domain_socket.sock», или числовым идентификатором TCP-порта. Подключения часто выполняются по telnet. Типичное значение порта: 3313.

Пример:

```
tarantool> console = require('console')
---
...
tarantool> console.listen('unix/:/tmp/X.sock')
... main/103/console/unix/:/tmp/X I> started
---
- fd: 6
  name:
    host: unix/
    family: AF_UNIX
    type: SOCK_STREAM
    protocol: 0
    port: /tmp/X.sock
...
```

`console.start()`

Запуск консоли на текущем интерактивном терминале.

Пример:

`console.start()` специально используется с *файлами инициализации*. Как правило, при запуске экземпляра Tarantool'а с помощью команды `tarantool` *`initialization file`*, консоль не поддерживается. Эту проблему можно решить путем добавления следующих строк в конце файла инициализации:

```
local console = require('console')
console.start()
```

`console.ac(`$\left[$*true*/*false*$\right]$`)`

Установка флага автодополнения ввода. Если значение автодополнения = *true* (правда), и поль-

зователь использует Tarantool в качестве клиента или подключен к Tarantool'у по `console.connect()`, то при нажатии клавиши TAB Tarantool будет автоматически дополнять текст по введенной части. По умолчанию, задано значение *true*.

`console.delimiter(`*marker*`)`
Настройка специального маркера окончания запроса для консоли Tarantool'а.

По умолчанию, маркер окончания запроса представляет собой символ разрыва строки (перевод строки). Нет необходимости в специальных маркерах, поскольку Tarantool может определить, если многостроковый запрос не завершен (например, если видно, что при объявлении функции еще не задано конечное ключевое слово). Тем не менее, в особых случаях или при вводе многострочных запросов в более ранних версиях Tarantool'а, можно изменить маркер окончания запроса. В результате символ разрыва строки не будет означать окончание запроса.

Чтобы вернуться в нормальный режим, введите команду: `console.delimiter('')<marker>`

Параметры

- `marker` (string) – специальный маркер окончания запроса для консоли Tarantool'а

Пример:

```
tarantool> console = require('console'); console.delimiter('!')
---
...
tarantool> function f ()
         > statement_1 = 'a'
         > statement_2 = 'b'
         > end!
---
...
tarantool> console.delimiter('')!
---
...
```

`console.get_default_output()`
Return the current default output format. The result will be `fmt="yaml"`, or it will be `fmt="lua"` if the last *set_default_output* call was `console.set_default_output('lua')`.

`console.set_default_output(`*'yaml'/'lua'*`)`
Set the default output format. The possible values are „yaml" (the default default) or „lua". The output format can be changed within a session by executing `console.eval('\\set output` *yaml* | *lua*`')`; see the description of output format in the *Interactive console* section.

5.2.5 Модуль *crypto*

Общие сведения

«Crypto» – это сокращенно «криптография», что обычно означает производство значения дайджеста из функции (как правило, криптографической хеш-функции – Cryptographic hash function), примененной к строке. Модуль `crypto` Tarantool'а поддерживает десять типов криптографических хеш-функций (AES, DES, DSS, MD4, MD5, MDC2, RIPEMD, SHA-1, SHA-2). В модуле *Модуль digest* также есть некоторые криптографические функции.

Указатель

Ниже приведен перечень всех функций модуля `crypto`.

Имя	Назначение
crypto.cipher.{algorithm}.{cipher_mode}.encrypt()	Шифрование строки
crypto.cipher.{algorithm}.{cipher_mode}.decrypt()	Расшифрование строки
crypto.digest.{algorithm}()	Получение дайджеста
crypto.hmac.{algorithm}()	Получение хеш-ключа

`crypto.cipher.{aes128|aes192|aes256|des}.{cbc|cfb|ecb|ofb}.encrypt(`*string*, *key*, *initialization_vector*`)`
`crypto.cipher.{aes128|aes192|aes256|des}.{cbc|cfb|ecb|ofb}.decrypt(`*string*, *key*, *initialization_vector*`)`

Передача или возврат шифрованного сообщения, полученного из строки, ключа и (необязательно) вектора инициализации. Четыре алгоритма на выбор:

- aes128 - aes-128 (128-битные двоичные строки с использованием AES)
- aes192 - aes-192 (192-битные двоичные строки с использованием AES)
- aes256 - aes-256 (256-битные двоичные строки с использованием AES)
- des - des (56-битные двоичные строки с использованием DES, хотя использование DES не рекомендуется)

Также доступны четыре режима блочного шифрования на выбор:

- cbc - Сцепление блоков шифротекста
- cfb - Обратная связь по шифротексту
- ecb - Электронная кодовая книга
- ofb - Обратная связь по выходу

Для получения дополнительной информации, см. статью о режимах шифрования Encryption Modes

Пример:

```
_16byte_iv='1234567890123456'
_16byte_pass='1234567890123456'
e=crypto.cipher.aes128.cbc.encrypt('string', _16byte_pass, _16byte_iv)
crypto.cipher.aes128.cbc.decrypt(e,  _16byte_pass, _16byte_iv)
```

`crypto.digest.{dss|dss1|md4|md5|mdc2|ripemd160}(`*string*`)`
`crypto.digest.{sha1|sha224|sha256|sha384|sha512}(`*string*`)`

Передача или возврат дайджеста из строки. Выбор из одиннадцати алгоритмов:

- dss - dss (с использованием DSS)
- dss1 - dss (с использованием DSS-1)
- md4 - md4 (128-битные двоичные строки с использованием MD4)
- md5 - md5 (128-битные двоичные строки с использованием MD5)
- mdc2 - mdc2 (с использованием MDC2)
- ripemd160 - ripemd (160-битные двоичные строки с использованием RIPEMD-160)
- sha1 - sha-1 (160-битные двоичные строки с использованием SHA-1)

- sha224 - sha-224 (224-битные двоичные строки с использованием SHA-2)
- sha256 - sha-256 (256-битные двоичные строки с использованием SHA-2)
- sha384 - sha-384 (384-битные двоичные строки с использованием SHA-2)
- sha512 - sha-512(512-битные двоичные строки с использованием SHA-2).

Пример:

```
crypto.digest.md4('string')
crypto.digest.sha512('string')
```

crypto.hmac.{md4|md5|ripemd160}(*key*, *string*)
crypto.hmac.{sha1|sha224|sha256|sha384|sha512}(*key*, *string*)

Передача ключа и строки. Результатом будет код аутентификации сообщения HMAC. 8 алгоритмов на выбор:

- md4 или md4_hex - md4 (128-битные двоичные строки с использованием MD4)
- md5 или md5_hex - md5 (128-битные двоичные строки с использованием MD5)
- ripemd160 или ripemd160_hex - ripemd (160-битные двоичные строки с использованием RIPEMD-160)
- sha1 или sha1_hex - sha-1 (160-битные двоичные строки с использованием SHA-1)
- sha224 или sha224_hex - sha-224 (224-битные двоичные строки с использованием SHA-2)
- sha256 или sha256_hex - sha-256 (256-битные двоичные строки с использованием SHA-2)
- sha384 или sha384_hex - sha-384 (384-битные двоичные строки с использованием SHA-2)
- sha512 или sha512_hex - sha-512(512-битные двоичные строки с использованием SHA-2).

Пример:

```
crypto.hmac.md4('key', 'string')
crypto.hmac.md4_hex('key', 'string')
```

Инкрементальные методы в модуле crypto

Предположим, что вычислен дайджест для строки „A“, затем часть „B“ добавляется в строку, необходим новый дайджест. Новый дайджест можно пересчитать для всей строки „AB“, но быстрее будет взять вычисленный дайджест для „A“ и внести изменения на основании добавленной части „B“. Это называется многошаговым процессом или «инкрементным» хеш-суммированием, которое поддерживает Tarantool поддерживает для всех криптографических функций.

```
crypto = require('crypto')

-- вывести дайджест 'AB' по aes-192 пошагово, затем с инкрементом
key = 'key/key/key/key/key/key/'
iv =  'iviviviviviviviv'
print(crypto.cipher.aes192.cbc.encrypt('AB', key, iv))
c = crypto.cipher.aes192.cbc.encrypt.new(key)
c:init(nil, iv)
c:update('A')
c:update('B')
print(c:result())
c:free()
```

(continues on next page)

(продолжение с предыдущей страницы)

```
-- вывести дайджест 'AB' по sha-256 пошагово, затем с инкрементом
print(crypto.digest.sha256('AB'))
c = crypto.digest.sha256.new()
c:init()
c:update('A')
c:update('B')
print(c:result())
c:free()
```

Получение одинаковых результатов из модулей digest и crypto

Следующие функции равноценны. Например, функция `digest` и функция `crypto` приведут к одному результату.

```
crypto.cipher.aes256.cbc.encrypt('x',b32,b16)==digest.aes256cbc.encrypt('x',b32,b16)
crypto.digest.md4('string') == digest.md4('string')
crypto.digest.md5('string') == digest.md5('string')
crypto.digest.sha1('string') == digest.sha1('string')
crypto.digest.sha224('string') == digest.sha224('string')
crypto.digest.sha256('string') == digest.sha256('string')
crypto.digest.sha384('string') == digest.sha384('string')
crypto.digest.sha512('string') == digest.sha512('string')
```

5.2.6 Модуль *csv*

Общие сведения

Модуль `csv` обрабатывает записи, форматированные в соответствии с правилами CSV (значения, разделенные запятыми).

По умолчанию, используются следующие правила форматирования:

- Escape-последовательности escape sequences в Lua, такие как \n или \10, можно использовать в строках, но не в файлах,
- Запятые обозначают конец поля,
- Символы перевода строки или перевода строки плюс возврата каретки означают конец записи,
- Начальные и конечные пробелы игнорируются,
- Кавычками могут обрамляться поля или компоненты полей,
- При обрамлении кавычками запятые, символы перевода строки и пробелы считаются обычными символами, а двойные кавычки «» считаются одинарными.

Параметры, которые можно передать в функции модуля `csv`:

- `delimiter = строка` (по умолчанию: запятая) – однобайтовый символ для обозначения конца поля
- `quote_char = строка` (по умолчанию: кавычка) – однобайтовый символ для обозначения закрытия строки
- `chunk_size = число` (по умолчанию: 4096) – число символов для одновременного чтения (обычно для эффективности файлового ввода-вывода)

- `skip_head_lines = число` (по умолчанию: 0) – число строк, которые пропускаются в начале (обычно для заголовка)

Указатель

Ниже приведен перечень всех функций модуля `csv`.

Имя	Назначение
csv.load()	Загрузка CSV-файла
csv.dump()	Преобразование входного значения в строку формата CSV
csv.iterate()	Итерация по записям в формате CSV

`csv.load(`*readable*[, *{options}*]`)`

Получение входного значения в формате CSV из `readable` и возврат таблицы в качестве выходного значения. Обычно `readable` представляет собой либо строку, либо открытый для чтения файл. Как правило, параметры *`options`* не указываются.

Параметры

- `readable` (*`object`*) – строка или любой объект с методом read(), форматированный по правилам CSV
- `options` (`table`) – см. *выше*

возвращает загруженное значение

тип возвращаемого значения таблица

Пример:

В читаемой строке 3 поля, поле №2 содержит запятую и пробел, поэтому следует использовать кавычки:

```
tarantool> csv = require('csv')
---
...
tarantool> csv.load('a,"b,c ",d')
---
- - - a
    - 'b,c '
    - d
...
```

В читаемой строке 2-байтный символ = Палочка в кириллице: (Отобразит палочку только в том случае, если кодировка = UTF-8.)

```
tarantool> csv.load('a\\211\\128b')
---
- - - a\211\128b
...
```

Точка с запятой вместо запятой в виде символа разделителя:

```
tarantool> csv.load('a,b;c,d', {delimiter = ';'})
---
- - - a,b
    - c,d
...
```

Читаемый файл `./file.csv` содержит две записи в формате CSV. Объяснение блока `fio` дается в разделе *fio*. Исходный CSV-файл и пример соответственно:

```
tarantool> -- входное значение в файле file.csv:
tarantool> -- a,"b,c ",d
tarantool> -- a\\211\\128b
tarantool> fio = require('fio')
---
...
tarantool> f = fio.open('./file.csv', {'O_RDONLY'})
---
...
tarantool> csv.load(f, {chunk_size = 4096})
---
- - - a
    - 'b,c '
    - d
  - - a\\211\\128b
...
tarantool> f:close()
---
- true
...
```

`csv.dump(`*csv-table*[, *options*, *writable*]`)`

Получение входного значения из таблицы `csv-table` и возврат строки в формате CSV в качестве выходного значения. Или получение входного значения из таблицы `csv-table` и размещение выходного значения в `writable`. Обычно параметры *`options`* не указываются. Как правило, если указан `writable`, то это открытый для чтения файл. *csv.dump()* – это операция, обратная *csv.load()*.

Параметры

- `csv-table` (`table`) – таблица, которую можно форматировать в соответствии с правилами CSV
- `options` (`table`) – необязательно. См. *выше*
- `writable` (*`object`*) – любой объект с методом `write()`

возвращает записанное значение

тип возвращаемого значения строка, которая записывается в объект `writable`, если указан

Пример:

В таблице формата CSV 3 поля, поле №2 содержит «,» поэтому результат включает в себя кавычки

```
tarantool> csv = require('csv')
---
...
tarantool> csv.dump({'a','b,c ','d'})
---
- 'a,"b,c ",d

'
...
```

Круговое преобразование: из строки в таблицу и обратно в строку

```
tarantool> csv_table = csv.load('a,b,c')
---
...
tarantool> csv.dump(csv_table)
---
- 'a,b,c

'
...
```

csv.iterate(*input*, *{options}*)

Создание Lua-функции с итератором для прохода по записям в формате CSV по одному полю за раз. Настоятельно рекомендуется использовать итератор для большого объема данных (10 мегабайт и более).

Параметры

- csv-table (table) – таблица, которую можно форматировать в соответствии с правилами CSV
- options (table) – см. *выше*

возвращает Lua-функция с итератором

тип возвращаемого значения функция с итератором

Пример:

csv.iterate() – это *csv.load()* и *csv.dump()* низкого уровня. Чтобы это доказать, используем функцию, которая совпадает с функцией *csv.load()*, как можно увидеть в исходном коде Tarantool'a (the Tarantool source code).

```
tarantool> load = function(readable, opts)
         >   opts = opts or {}
         >   local result = {}
         >   for i, tup in csv.iterate(readable, opts) do
         >     result[i] = tup
         >   end
         >   return result
         > end
---
...
tarantool> load('a,b,c')
---
- - - a
    - b
    - c
...
```

5.2.7 Module *decimal*

The `decimal` module has functions for working with exact numbers. This is important when numbers are large or even the slighest inaccuracy is unacceptable. For example Lua calculates `0.16666666666667 * 6` with floating-point so the result is 1. But with the decimal module (using `decimal.new` to convert the number to decimal type) `decimal.new('0.16666666666667') * 6` is 1.00000000000002.

To construct a decimal number, bring in the module with `require('decimal')` and then use `decimal.new(n)` or any function in the decimal module: *abs(n)* *exp(n)* *ln(n)* *log10(n)* *new(n)* *precision(n)*

rescale(decimal-number, new-scale) scale(n) sqrt(n) trim(decimal-number), where n can be a string or a non-decimal number or a decimal number. If it is a string or a non-decimal number, Tarantool converts it to a decimal number before working with it. It is best to construct from strings, and to convert back to strings after calculations, because Lua numbers have only 15 digits of precision. Decimal numbers have 38 digits of precision, that is, the total number of digits before and after the decimal point can be 38. Tarantool supports the usual arithmetic and comparison operators + - * / % ^ < > <= >= ~= ==. If an operation has both decimal and non-decimal operands, then the non-decimal operand is converted to decimal before the operation happens.

Use `tostring(decimal-number)` to convert back to a string.

A decimal operation will fail if overflow happens (when a number is greater than 10^38 - 1 or less than -10^38 - 1). A decimal operation will fail if arithmetic is impossible (such as division by zero or square root of minus 1). A decimal operation will not fail if rounding of post-decimal digits is necessary to get 38-digit precision.

`decimal.abs(`*string-or-number-or-decimal-number*`)`
: Returns absolute value of a decimal number. For example if a is -1 then `decimal.abs(a)` returns 1.

`decimal.exp(`*string-or-number-or-decimal-number*`)`
: Returns *e* raised to the power of a decimal number. For example if a is 1 then `decimal.exp(a)` returns 2.71828182845904523536028747135266249 78. Compare `math.exp(1)` from the Lua math library, which returns 2.718281828459.

`decimal.ln(`*string-or-number-or-decimal-number*`)`
: Returns natural logarithm of a decimal number. For example if a is 1 then `decimal.ln(a)` returns 0.

`decimal.log10(`*string-or-number-or-decimal-number*`)`
: Returns base-10 logarithm of a decimal number. For example if a is 100 then `decimal.log10(a)` returns 2.

`decimal.new(`*string-or-number-or-decimal-number*`)`
: Returns the value of the input as a decimal number. For example if a is 1E-1 then `decimal.new(a)` returns 0.1.

`decimal.precision(`*string-or-number-or-decimal-number*`)`
: Returns the number of digits in a decimal number. For example if a is 123.4560 then `decimal.precision(a)` returns 7.

`decimal.rescale(`*decimal-number*, *new-scale*`)`
: Returns the number after possible rounding or padding. If the number of post-decimal digits is greater than new-scale, then rounding occurs. The rounding rule is: round half away from zero. If the number of post-decimal digits is less than new-scale, then padding of zeros occurs. For example if a is -123.4550 then `decimal.rescale(a, 2)` returns -123.46, and `decimal.rescale(a, 5)` returns -123.45500.

`decimal.scale(`*string-or-number-or-decimal-number*`)`
: Returns the number of post-decimal digits in a decimal number. For example if a is 123.4560 then `decimal.scale(a)` returns 4.

`decimal.sqrt(`*string-or-number-or-decimal-number*`)`
: Returns the square root of a decimal number. For example if a is 2 then `decimal.sqrt(a)` returns 1.41421356237309504880168872420969807 86.

`decimal.trim(`*decimal-number*`)`
: Returns a decimal number after possible removing of trailing post-decimal zeros. For example if a is 2.20200 then `decimal.trim(a)` returns 2.202.

5.2.8 Модуль *digest*

Общие сведения

«Дайджест» – это значение, которое возвращает функция (как правило, криптографическая хеш-функция – Cryptographic hash function), примененная к строке. Модуль“digest“ Tarantool’a поддерживает несколько типов криптографических хеш-функций (AES, MD4, MD5, SHA-1, SHA-2, PBKDF2), а также функцию контрольного суммирования (CRC32), две функции для base64 и две некриптографические хеш-функции (guava, murmur). Часть функций модуля digest также включена в модуль *crypto*.

Указатель

Ниже приведен перечень всех функций модуля `digest`.

Имя	Назначение
digest.aes256cbc.encrypt()	Шифрование строки с использованием AES
digest.aes256cbc.decrypt()	Расшифрование строки с использованием AES
digest.md4()	Получение дайджеста с помощью MD4
digest.md4_hex()	Получение шестнадцатеричного дайджеста с помощью MD4
digest.md5()	Получение дайджеста с помощью MD5
digest.md5_hex()	Получение шестнадцатеричного дайджеста с помощью MD5
digest.pbkdf2()	Получение дайджеста с помощью PBKDF2
digest.sha1()	Получение дайджеста с помощью SHA-1
digest.sha1_hex()	Получение шестнадцатеричного дайджеста с помощью SHA-1
digest.sha224()	Получение 224-битного дайджеста с помощью SHA-2
digest.sha224_hex()	Получение 56-байтного шестнадцатеричного дайджеста с помощью SHA-2
digest.sha256()	Получение 256-битного дайджеста с помощью SHA-2
digest.sha256_hex()	Получение 64-байтного шестнадцатеричного дайджеста с помощью SHA-2
digest.sha384()	Получение 384-битного дайджеста с помощью SHA-2
digest.sha384_hex()	Получение 96-байтного шестнадцатеричного дайджеста с помощью SHA-2
digest.sha512()	Получение 512-битного дайджеста с помощью SHA-2
digest.sha512_hex()	Получение 128-байтного шестнадцатеричного дайджеста с помощью SHA-2
digest.base64_encode()	Кодирование строки по стандарту Base64
digest.base64_decode()	Декодирование строки по стандарту Base64
digest.urandom()	Получение массива случайных байтов
digest.crc32()	Получение 32-битной контрольной суммы с помощью CRC32
digest.crc32.new()	Запуск инкрементного вычисления CRC32
digest.guava()	Получение числа с помощью консистентного хеширования
digest.murmur()	Получение дайджеста с помощью MurmurHash
digest.murmur.new()	Запуск инкрементного вычисления с помощью MurmurHash

`digest.aes256cbc.encrypt(`*string*, *key*, *iv*`)`
`digest.aes256cbc.decrypt(`*string*, *key*, *iv*`)`
Возврат 256-битной двоичной строки = дайджест, полученный с помощью AES.

`digest.md4(`*string*`)`
Возврат 128-битной двоичной строки = дайджест, полученный с помощью MD4.

`digest.md4_hex(`*string*`)`
Возврат 32-байтной строки = шестнадцатеричное значение дайджеста, вычисленного с помощью MD4.

`digest.md5(`*string*`)`
Возврат 128-битной двоичной строки = дайджест, полученный с помощью MD5.

`digest.md5_hex(`*string*`)`
Возврат 32-байтной строки = шестнадцатеричное значение дайджеста, вычисленного с помощью MD5.

`digest.pbkdf2(`*string*, *salt*[, *iterations*[, *digest-length*]])
Возврат двоичной строки = дайджест, полученный с помощью PBKDF2. Для эффективности шифрования значение параметра количества итераций `iterations` должно быть как минимум несколько тысяч. Значение параметра `digest-length` определяет длину полученной двоичной строки.

Примечание: `digest.pbkdf2()` yields and should not be used in a transaction (between `box.begin()` and `box.commit()`/`box.rollback()`). PBKDF2 is a time-consuming hash algorithm. It runs in a separate coio thread. While calculations are performed, the fiber that calls `digest.pbkdf2()` yields and another fiber continues working in the tx thread.

`digest.sha1(`*string*`)`
Возврат 160-битной двоичной строки = дайджест, полученный с помощью SHA-1.

`digest.sha1_hex(`*string*`)`
Возврат 40-байтной строки = шестнадцатеричное значение дайджеста, вычисленного с помощью SHA-1.

`digest.sha224(`*string*`)`
Возврат 224-битной двоичной строки = дайджест, полученный с помощью SHA-2.

`digest.sha224_hex(`*string*`)`
Возврат 56-байтной строки = шестнадцатеричное значение дайджеста, вычисленного с помощью SHA-224.

`digest.sha256(`*string*`)`
Возврат 256-битной двоичной строки = дайджест, полученный с помощью SHA-2.

`digest.sha256_hex(`*string*`)`
Возврат 64-байтной строки = шестнадцатеричное значение дайджеста, вычисленного с помощью SHA-256.

`digest.sha384(`*string*`)`
Возврат 384-битной двоичной строки = дайджест, полученный с помощью SHA-2.

`digest.sha384_hex(`*string*`)`
Возврат 96-байтной строки = шестнадцатеричное значение дайджеста, вычисленного с помощью SHA-384.

`digest.sha512(`*string*`)`
Возврат 512-битной двоичной строки = дайджест, полученный с помощью SHA-2.

`digest.sha512_hex(`*string*`)`
Возврат 128-байтной строки = шестнадцатеричное значение дайджеста, вычисленного с помощью SHA-512.

`digest.base64_encode()`
Возврат кодированного по base64 значения обычной строки.

Возможные опции:

- `nopad` – результат не должен включать в себя „=“ для заполнения символами в конце,

- `nowrap` – результат не должен включать в себя символ переноса строки для разделения строк после 72 символов,
- `urlsafe` – результат не должен включать в себя „=“ или символ переноса строки и может содержать „-„ или „_“ взамен „+“ или „/“ в качестве 62 и 63 символа в схеме.

Значения параметров могут быть `true` (правда) или `false` (ложь), по умолчанию используется `false`.

Пример:

```
digest.base64_encode(string_variable,{nopad=true})
```

`digest.base64_decode(`*string*`)`
Возврат обычной строки из кодированного по base64 значения.

`digest.urandom(`*integer*`)`
Возврат массива случайных байтов с длиной = целому числу.

`digest.crc32(`*string*`)`
Возврат 32-битной контрольной суммы с помощью CRC32.

Функции `crc32` и `crc32_update` используют значение многочлена Cyclic Redundancy Check : `0x1EDC6F41` / `4812730177`. (Другие используемые значения: ввод = отраженное значение, вывод = отраженное значение, начальное значение = 0xFFFFFFFF, финальное xor-значение = 0x0.) Если необходима совместимость с другими функциями контрольной суммы на другом языке программирования, убедитесь, что другие функции используют то же значение многочлена.

Например, в Python установите пакет `crcmod` и введите команду:

```
>>> import crcmod
>>> fun = crcmod.mkCrcFun('4812730177')
>>> fun('string')
3304160206L
```

В Perl установите модуль `Digest::CRC` и выполните следующий код:

```
use Digest::CRC;
$d = Digest::CRC->new(width => 32, poly => 0x1EDC6F41, init => 0xFFFFFFFF, refin => 1, refout␣
↪=> 1);
$d->add('string');
print $d->digest;
```

(ожидается выходное значение: 3304160206).

`digest.crc32.new()`
Запуск инкрементного вычисления CRC32. См. примечания по *инкрементным методам*.

`digest.guava(`*state*, *bucket*`)`
Возврат числа с помощью консистентного хеширования.

Функция guava использует алгоритм консистентного хеширование (Consistent Hashing) из библиотеки guava от Google. Первым параметром должен быть хеш-код; вторым параметром должно быть число слотов; возвращается значение в виде целого числа в диапазоне от 0 до указанного числа слотов. Например,

```
tarantool> digest.guava(10863919174838991, 11)
---
- 8
...
```

digest.murmur(*string*)
: Возврат 32-битной двоичной строки = дайджест, полученный с помощью MurmurHash.

digest.murmur.new(*opts*)
: Запуск инкрементного вычисления с помощью MurmurHash. См. примечания по *инкрементным методам*. Например:

```
murmur.new({seed=0})
```

Инкрементальные методы в модуле digest

Предположим, что вычислен дайджест для строки „A“, затем часть „B“ добавляется в строку, необходим новый дайджест. Новый дайджест можно пересчитать для всей строки „AB“, но быстрее будет взять вычисленный дайджест для „A“ и внести изменения на основании добавленной части „B“. Это называется многошаговым процессом или «инкрементным» хеш-суммированием, которое поддерживает Tarantool поддерживает для crc32 и murmur...

```
digest = require('digest')

-- вывести дайджест 'AB' по crc32 пошагово, затем с инкрементом
print(digest.crc32('AB'))
c = digest.crc32.new()
c:update('A')
c:update('B')
print(c:result())

-- вывести дайджест 'AB' по murmur hash пошагово, затем с инкрементом
print(digest.murmur('AB'))
m = digest.murmur.new()
m:update('A')
m:update('B')
print(m:result())
```

Пример

В следующем примере пользователь создает две функции: функцию password_insert(), которая вставляет дайджест слова «**^S^e^c^ret Wordpass**» по SHA-1 в набор кортежей, и функцию password_check(), которая требует ввод пароля.

```
tarantool> digest = require('digest')
---
...
tarantool> function password_insert()
         >   box.space.tester:insert{1234, digest.sha1('^S^e^c^ret Wordpass')}
         >   return 'OK'
         > end
---
...
tarantool> function password_check(password)
         >   local t = box.space.tester:select{12345}
         >   if digest.sha1(password) == t[2] then
         >     return 'Password is valid'
         >   else
         >     return 'Password is not valid'
```

(continues on next page)

(продолжение с предыдущей страницы)

```
         >   end
         > end
---
...
tarantool> password_insert()
---
- 'OK'
...
```

Если затем пользователь вызовет функцию `password_check()` и вводит неверный пароль, результатом будет ошибка.

```
tarantool> password_check('Secret Password')
---
- 'Password is not valid'
...
```

5.2.9 Модуль *errno*

Общие сведения

Модуль `errno`, как правило, используется внутри функции или в рамках Lua-программы совместно с модулем, функции которого могут возвращать ошибки ОС, например *fio*.

Указатель

Ниже приведен перечень всех функций модуля `errno`.

Имя	Назначение
errno()	Получение номера ошибки для последней функции, связанной с ОС
errno.strerror()	Получение сообщения об ошибке для соответствующего номера ошибки

`errno()`
Возврат номера ошибки для последней функции, связанной с операционной системой, или 0. Чтобы вызвать функцию, просто введите команду `errno()` без названия модуля.

тип возвращаемого значения целое число

`errno.strerror(`[*code*]`)`
Возврат строки в ответ на номер ошибки. Строка будет содержать текст традиционного сообщения об ошибке для текущей операционной системы. Если не указан код `code`, то будет выведено сообщение об ошибке для последней функции, связанной с операционной системой, или 0.

Параметры

- `code` (*integer*) – номер ошибки в операционной системе

тип возвращаемого значения строка

Пример:

Данная функция отображает результат вызова *fio.open()*, который вызывает ошибку 2 (`errno.ENOENT`). В результат включен номер ошибки, связанная с ним строка сообщения об ошибке и название ошибки.

```
tarantool> function f()
         >   local fio = require('fio')
         >   local errno = require('errno')
         >   fio.open('no_such_file')
         >   print('errno() = ' .. errno())
         >   print('errno.strerror() = ' .. errno.strerror())
         >   local t = getmetatable(errno).__index
         >   for k, v in pairs(t) do
         >     if v == errno() then
         >       print('errno() constant = ' .. k)
         >     end
         >   end
         > end
---
...

tarantool> f()
errno() = 2
errno.strerror() = No such file or directory
errno() constant = ENOENT
---
...
```

Чтобы увидеть все возможные названия ошибок, которые хранятся в метатаблице `errno`, введите команду `getmetatable(errno)` (выводятся сокращенно):

```
tarantool> getmetatable(errno)
---
- __newindex: 'function: 0x41666a38'
  __call: 'function: 0x41666890'
  __index:
  ENOLINK: 67
  EMSGSIZE: 90
  EOVERFLOW: 75
  ENOTCONN: 107
  EFAULT: 14
  EOPNOTSUPP: 95
  EEXIST: 17
  ENOSR: 63
  ENOTSOCK: 88
  EDESTADDRREQ: 89
  <...>
...
```

5.2.10 Модуль *fiber*

Общие сведения

С помощью модуля `fiber` можно:

- создавать, запускать и управлять *файберами*,
- отправлять и получать сообщения для различных процессов (например, разные соединения, сессии или файберы) по *каналам*, а также
- использовать *механизм синхронизации* для файберов, аналогично работе «условных переменных» и функций операционных систем, таких как `pthread_cond_wait()` плюс

`pthread_cond_signal()`.

Указатель

Ниже приведен перечень всех функций и элементов модуля `fiber`.

Имя	Назначение
fiber.create()	Создание и запуск файбера
fiber.new()	Создание файбера без запуска
fiber.self()	Получение объекта файбера
fiber.find()	Получение объекта файбера по ID
fiber.sleep()	Перевод файбера в режим ожидания
fiber.yield()	Передача управления
fiber.status()	Получение статуса активного файбера
fiber.info()	Получение информации о всех файберах
fiber.kill()	Отмена файбера
fiber.testcancel()	Проверка отмены действующего файбера
fiber_object:id()	Получение ID файбера
fiber_object:name()	Получение имени файбера
fiber_object:name(name)	Назначение имени файбера
fiber_object:status()	Получение статуса файбера
fiber_object:cancel()	Отмена файбера
fiber_object.storage	Локальное хранилище в пределах файбера
fiber_object:set_joinable()	Создание возможности подключения нового файбера
fiber_object:join()	Ожидание статуса „dead“ (недоступен) для файбера
fiber.time()	Получение системного времени в секундах
fiber.time64()	Получение системного времени в микросекундах
fiber.clock()	Получение монотенного времени в секундах
fiber.clock64()	Получение монотонного времени в микросекундах
fiber.channel()	Создание канала связи
channel_object:put()	Отправка сообщения по каналу связи
channel_object:close()	Закрытие канала
channel_object:get()	Перехват сообщения из канала
channel_object:is_empty()	Проверка пустоты канала
channel_object:count()	Подсчет сообщений в канале
channel_object:is_full()	Проверка заполненности канала
channel_object:has_readers()	Проверка пустого канала на наличие читателей в состоянии ожидания
channel_object:has_writers()	Проверка полного канала на наличие писателей в состоянии ожидания
channel_object:is_closed()	Проверка закрытия канала
fiber.cond()	Создание условной переменной
cond_object:wait()	Перевод файбера в режим ожидания до пробуждения другим файбером
cond_object:signal()	Пробуждение отдельного файбера
cond_object:broadcast()	Пробуждение всех файберов

Файберы

Файбер – это набор инструкций, которые выполняются по принципу кооперативной многозадачности. Файберы, управление которых происходит с помощью модуля fiber, связаны с функцией под названием *функция для файбера*, которую задает пользователь.

Существуют три возможных состояния файбера: **running** (активен), **suspended** (приостановлен) или

dead (недоступен). После создания файбера с помощью *fiber.create()* он сразу активен. После создания файбера с помощью *fiber.new()* или передачи управления с помощью *fiber.sleep()* файбер будет приостановлен. По окончании работы (по причине окончания работы соответствующей функции) файбер становится недоступен.

Все файберы составляют часть реестра файберов. Можно производить поиск по реестру с помощью *fiber.find()* по ID файбера (fid), который представляет собой числовой идентификатор.

Неконтролируемый файбер можно остановить с помощью *fiber_object.cancel*. Однако, функция *fiber_object.cancel* консультативна, то есть сработает только в том случае, если неконтролируемый файбер случайно вызовет *fiber.testcancel()*. Большинство функций типа `box.*`, например *box.space...delete()* или *box.space...update()*, действительно вызывают *fiber.testcancel()*, а *box.space...select{}* не вызовет. В действительности неконтролируемый файбер может перестать отвечать, если он производит большое количество вычислений и не проверяет вероятность отмены.

Другой потенциальной проблемой могут стать файберы, которые не включаются в расписание, поскольку они не подписаны ни на какие события, или потому что соответствующие события не происходят. Такие файберы можно в любое принудительно остановить с помощью *fiber.kill()*, потому что функция *fiber.kill()* отправляет асинхронное событие пробуждения на файбер, а *fiber.testcancel()* проверяет наступление такого события пробуждения.

Сборщик мусора собирает недоступные файберы так же, как и все Lua-объекты: сборщик мусора в Lua освобождает память выделенного для файбера пула, сбрасывает все данные файбера и возвращает файбер (который теперь называется каркасом файбера) в пул файберов. Каркас можно использовать повторно при создании другого файбера.

У файбера есть все возможности сопрограммы (coroutine) на языке Lua, и все принципы программирования, которые применяются к сопрограммам на Lua, применимы и к файберам. Однако Tarantool расширил возможности файберов для внутреннего использования. Поэтому, несмотря на возможность и поддержку использования сопрограмм, рекомендуется использовать файберы.

`fiber.create(`*function*[, *function-arguments*]`)`

Создание и запуск файбера. Происходит создание файбера, который незамедлительно начинает работу.

Параметры

- `function` – функция, которая будет связана с файбером
- `function-arguments` – что передается в функцию

Возвращается созданный объект файбера

Тип возвращаемого значения пользовательские данные

Пример:

```
tarantool> fiber = require('fiber')
---
...
tarantool> function function_name()
         >   print("I'm a fiber")
         > end
---
...
tarantool> fiber_object = fiber.create(function_name); print("Fiber started")
I'm a fiber
Fiber started
---
...
```

fiber.new(*function*[, *function-arguments*])

Создание файбера без запуска: файбер создается, но не запускается сразу же, а ожидает, пока создатель файбера (то есть задача, которая вызывает `fiber.new()`) не передаст управление согласно правилам *контроля транзакций*. Файбер создается со статусом „suspended“ (приостановлен). Таким образом, логика `fiber.new()` слегка отличается от *fiber.create()*.

Как правило, `fiber.new()` используется вместе с *fiber_object:set_joinable()* и *fiber_object:join()*.

Параметры

- `function` – функция, которая будет связана с файбером
- `function-arguments` – что передается в функцию

Возвращается созданный объект файбера

Тип возвращаемого значения пользовательские данные

Пример:

```
tarantool> fiber = require('fiber')
---
...
tarantool> function function_name()
         >   print("I'm a fiber")
         > end
---
...
tarantool> fiber_object = fiber.new(function_name); print("Fiber not started yet")
Fiber not started yet
---
...
tarantool> I'm a fiber
---
...
```

fiber.self()

Возвращается объект файбера для запланированного на данный момент файбера.

Тип возвращаемого значения пользовательские данные

Пример:

```
tarantool> fiber.self()
---
- status: running
  name: interactive
  id: 101
...
```

fiber.find(*id*)

Параметры

- `id` – числовой идентификатор файбера.

Возвращается объект файбера для указанного файбера.

Тип возвращаемого значения пользовательские данные

Пример:

```
tarantool> fiber.find(101)
---
- status: running
  name: interactive
  id: 101
...
```

fiber.sleep(*time*)

Передача управления планировщику и переход в режим ожидания на указанное количество секунд. Только текущий файбер можно перевести в режим ожидания.

Параметры

- time – количество секунд в режиме ожидания.

Исключение см. *Пример неудачной передачи управления*

Пример:

```
tarantool> fiber.sleep(1.5)
---
...
```

fiber.yield()

Передача управления планировщику. Работает аналогично *fiber.sleep(0)*.

Исключение см. *Пример неудачной передачи управления*

Пример:

```
tarantool> fiber.yield()
---
...
```

fiber.status(⌊*fiber_object*⌋)

Возврат статуса текущего файбера. Или же, если передается необязательный параметр fiber_object, возврат статуса указанного файбера.

Возвращается статус файбера: “dead” (недоступен), “suspended” (приостановлен) или “running” (активен).

Тип возвращаемого значения строка

Пример:

```
tarantool> fiber.status()
---
- running
...
```

fiber.info()

Возврат информации о всех файберах.

Возвращается количество переключений контекста, обратная трассировка, ID, общий объем памяти, объем используемой памяти, имя каждого файбера.

Тип возвращаемого значения таблица

Пример:

```
tarantool> fiber.info()
---
- 101:
    csw: 7
    backtrace: []
    fid: 101
    memory:
      total: 65776
      used: 0
    name: interactive
...
```

fiber.kill(*id*)

Поиск файбера по числовому идентификатору и его отмена. Другими словами, *fiber.kill()* объединяет в себе *fiber.find()* и *fiber_object:cancel()*.

Параметры

- id – ID файбера для отмены.

Исключение указанный файбер отсутствует, или отмена невозможна.

Пример:

```
tarantool> fiber.kill(fiber.id()) -- функция с self может вызвать окончание программы
---
- error: fiber is cancelled
...
```

fiber.testcancel()

Проверка отмены действующего файбера и выдача исключения, если файбер отменен.

Примечание: Даже при исключении файбер будет отменен. Большинство вызовов проверяют fiber.testcancel(). Однако некоторые функции (id, status, join и т.д.) не вернут ошибку. Мы рекомендуем разработчикам приложений реализовать случайные проверки *fiber.testcancel()* и максимально быстро завершить выполнение файбера, если он был отменен.

Пример:

```
tarantool> fiber.testcancel()
---
- error: fiber is cancelled
...
```

object fiber_object

fiber_object:id()

Параметры

- fiber_object – как правило, это объект, полученный в результате вызова *fiber.create*, *fiber.self* или *fiber.find*

Возвращается ID файбера.

Тип возвращаемого значения число

fiber.self():id() может также быть выражен как fiber.id().

Пример:

```
tarantool> fiber_object = fiber.self()
---
...
tarantool> fiber_object:id()
---
- 101
...
```

`fiber_object:name()`

Параметры

- `fiber_object` – как правило, это объект, полученный в результате вызова *fiber.create*, *fiber.self* или *fiber.find*

Возвращается имя файбера.

Тип возвращаемого значения строка

`fiber.self():name()` может также быть выражен как `fiber.name()`.

Пример:

```
tarantool> fiber.self():name()
---
- interactive
...
```

`fiber_object:name(`*name*[, *options*]`)`

Изменение имени файбера. По умолчанию, файбер в интерактивном режиме экземпляра Tarantool'а называется „interactive“, а новые файберы, созданные с помощью *fiber.create*, называются „lua“. Переименование файберов позволяет легче различать их при использовании *fiber.info*. Максимум 32 символа.

Параметры

- `fiber_object` – как правило, это объект, полученный в результате вызова *fiber.create*, *fiber.self* или *fiber.find*
- `name` (`string`) – новое имя файбера.
- `options` – `truncate=true` – усекает имя до максимальной длины, если оно слишком длинное. Если эта опция установлена в `false` (по умолчанию) и имя слишком длинное, то `fiber.name(new_name)` падает и выдает исключение.

Возвращается nil

Пример:

```
tarantool> fiber.self():name('non-interactive')
---
...
```

`fiber_object:status()`

Возврат статуса указанного файбера.

Параметры

- `fiber_object` – как правило, это объект, полученный в результате вызова *fiber.create*, *fiber.self* или *fiber.find*

Возвращается статус файбера: “dead” (недоступен), “suspended” (приостановлен) или “running” (активен).

Тип возвращаемого значения строка

`fiber.self():status(` может также быть выражен как `fiber.status()`.

Пример:

```
tarantool> fiber.self():status()
---
- running
...
```

`fiber_object:cancel()`

Отмена файбера. Активные и приостановленные файберы можно отменить. После отмены файбера попытки работать с ним вызовут ошибку, например, вызов *fiber_object:name()* вызовет ошибку с указанием недоступности файбера `error: the fiber is dead`. Тем не менее, недоступный файбер может передавать свой ID и статус.

Параметры

- `fiber_object` – как правило, это объект, полученный в результате вызова *fiber.create*, *fiber.self* или *fiber.find*

Возвращается nil

Возможные ошибки: нельзя отменить указанный объект файбера.

Пример:

```
tarantool> fiber.self():cancel() -- функция с self может вызвать окончание программы
---
...
tarantool> fiber.self():cancel()
---
- error: fiber is cancelled
...
tarantool> fiber.self:id()
---
- 163
...
tarantool> fiber.self:status()
---
- dead
...
```

`fiber_object.storage`

Локальное хранилище в пределах файбера. Представляет собой Lua-таблицу, создаваемую при первом обращении к ней. Хранилище может содержать любое количество именованных значений при соблюдении ограничений памяти. Правила именования: *объект_файбера*`.storage.`*имя*, либо *объект_файбера*`.storage['`*имя*`'].`, либо с числом *объект_файбера*`.storage[`*число*`]`. Значения могут быть числовыми или строковыми.

`fiber.storage` уничтожается вместе с файбером, независимо от того, как оно было завершено – через *fiber_object*`:cancel()` или после того, как функция файбера сделала „return“. Более того, хранилище очищается даже для файберов, собранных в пул для обслуживания запросов IProto. Такие файберы никогда не умирают, но тем не менее их хранилище очищается после каждого запроса. Это позволяет использовать `fiber.storage` в качестве полнофункционального хранилища запросов на локальном уровне.

Хранилище можно создать для файбера, созданного как из C, так и из Lua. Например, файбер был создан из C с помощью `fiber.new`, произвел вставку в спейс, в котором есть Lua-триггеры `on_replace`, и один из триггеров может создать `fiber.storage`. Это хранилище будет удалено, когда файбер остановится.

Пример:

```
tarantool> fiber = require('fiber')
---
...
tarantool> function f () fiber.sleep(1000); end
---
...
tarantool> fiber_function = fiber.create(f)
---
...
tarantool> fiber_function.storage.str1 = 'string'
---
...
tarantool> fiber_function.storage['str1']
---
- string
...
tarantool> fiber_function:cancel()
---
...
tarantool> fiber_function.storage['str1']
---
- error: '[string "return fiber_function.storage[''str1'']"]:1: the fiber is dead'
...
```

См. также *box.session.storage*.

`fiber_object:set_joinable(`*true_ or _false*`)`
`fiber_object:set_joinable(true)` делает файбер доступным для присоединения; `fiber_object:set_joinable(false)` делает файбер недоступным для присоединения; по умолчанию, false.

Присоединяемый файбер можно ожидать с помощью *fiber_object:join()*.

Лучше всего вызвать `fiber_object:set_joinable()` до начала выполнения функции с файбером, поскольку в противном случае файбер может стать недоступен до того, как сработает `fiber_object:set_joinable()`. Правильная последовательность может быть такой:

1. Вызов `fiber.new()` вместо `fiber.create()` для создания нового объекта файбера fiber_object.

   Не передавать управление, поскольку это приведет к началу работы функции с файбером.

2. Вызов `fiber_object:set_joinable(true)`, чтобы сделать новый объект файбера fiber_object присоединяемым.

   Сейчас можно передать управление.

3. Вызов `fiber_object:join()`.

   Как правило, следует вызвать `fiber_object:join()`, в противном случае, статус файбера может перейти в „suspended“ (приостановлен) после выполнения функции, а не „dead“ (недоступен).

Параметры

- `true_or_false` – логическое значение, которое изменяет флаг `set_joinable`

Возвращается nil

Пример:

Результат следующего ряда запросов:

- глобальная переменная `d` получит значение 6 (что доказывает, что функция не выполнялась до тех пор, пока значение `d` не стало 1, когда `fiber.sleep(1)` вызвал передачу управления);
- `fiber.status(fi2)` будет приостановлен „suspended“ (что доказывает, что после выполнения функции статус файбера не изменился на недоступный „dead“).

```
fiber=require('fiber')
d=0
function fu2() d=d+5 end
fi2=fiber.new(fu2) fi2:set_joinable(true) d=1 fiber.sleep(1)
print(d)
fiber.status(fi2)
```

`fiber_object:join()`

«Присоединение» присоединяемого файбера. То есть возможность запуска функции с файбером и ожидание перехода файбера в статус недоступности „dead“ (как правило, статус переходит в „dead“, когда заканчивается выполнение функции). Присоединение вызовет передачу управления, таким образом, если файбер находится в приостановленном состоянии, выполнение функции файбера возобновится.

Такое ожидание более удобно, чем переход в цикл с периодической проверкой статуса; тем не менее, это работает, только если файбер был создан с помощью *fiber.new()* и стал доступным для присоединения путем *fiber_ object:set_ joinable()*.

Возвращается два значения. Первое значение логическое. Если первое значение = true (правда), значит присоединение прошло успешно, поскольку функция файбера была выполнена нормально, а второй результат – это возвращаемое значение функции файбера. Если же первое значение = false (ложь), значит присоединение не было осуществлено, поскольку выполнение функции файбера было прервано, а второй результат содержит подробную информацию об ошибке, которую можно распаковать так же, как *результат вызова pcall*.

Тип возвращаемого значения логическое значение +тип результата, или логическое значение + ошибка структуры

Пример:

Результат следующего ряда запросов:

- первый вызов `fiber.status()` возвращает „suspended“ (приостановлен),
- вызов `join()` возвращает true (правда),
- как правило, проходит 5 секунд, и
- второй вызов `fiber.status()` возвращает „dead“ (недоступен).

Это доказывает, что `join()` не возвращает результат, пока функция, которая находится в режиме ожидания в течение 5 секунд, недоступна („dead“).

```
fiber=require('fiber')
function fu2() fiber.sleep(5) end
fi2=fiber.new(fu2) fi2:set_joinable(true)
start_time = os.time()
fiber.status(fi2)
fi2:join()
print('elapsed = ' .. os.time() - start_time)
fiber.status(fi2)
```

fiber.time()

Возвращается текущее системное время (в секундах с начала отсчета) в виде Lua-числа. Время берется из часов событийного цикла, поэтому вызов полезен лишь для создания искусственных ключей кортежа.

Тип возвращаемого значения число

Пример:

```
tarantool> fiber.time(), fiber.time()
---
- 1448466279.2415
- 1448466279.2415
...
```

fiber.time64()

Возвращается текущее системное время (в микросекундах с начала отсчета) в виде 64-битного целого числа. Время берется из часов событийного цикла.

Тип возвращаемого значения cdata

Пример:

```
tarantool> fiber.time(), fiber.time64()
---
- 1448466351.2708
- 1448466351270762
...
```

fiber.clock()

Получение монотонного времени в секундах. Для вычисления таймаутов лучше использовать fiber.clock(), поскольку *fiber.time()* сообщает системное время, а оно может меняться при изменениях в системе.

Возвращается количество секунд в виде числа с плавающей точкой, представляющего собой время с некоторого момента в прошлом, которое гарантированно не изменится в течение всего времени процесса

Тип возвращаемого значения число

Пример:

```
tarantool> start = fiber.clock()
---
...
tarantool> print(start)
248700.58805
---
...
```

(continues on next page)

(продолжение с предыдущей страницы)

```
tarantool> print(fiber.time(), fiber.time()-start)
1600785979.8291 1600537279.241
---
...
```

fiber.clock64()

То же, что и *fiber.clock()*, но в микросекундах.

Возвращается количество секунд в виде 64-битного целого числа, представляющего собой время с некоторого момента в прошлом, которое гарантированно не изменится в течение всего времени процесса

Тип возвращаемого значения cdata

Пример

Создание функции, которая будет связана с файбером. Такая функция содержит бесконечный цикл. Каждая итерация цикла прибавляет 1 к глобальной переменной под названием `gvar`, а затем уходит в режим ожидания на 2 секунды. Ожидание вызывает неявную передачу управления *fiber.yield()*.

```
tarantool> fiber = require('fiber')
tarantool> function function_x()
         >   gvar = 0
         >   while true do
         >     gvar = gvar + 1
         >     fiber.sleep(2)
         >   end
         > end
---
...
```

Создание файбера, ассоциация функции function_x с файбером и запуск function_x. Она сразу же «отсоединится», то есть будет работать отдельно от вызывающего метода.

```
tarantool> gvar = 0

tarantool> fiber_of_x = fiber.create(function_x)
---
...
```

Получение ID файбера (fid) для последующего вывода.

```
tarantool> fid = fiber_of_x:id()
---
...
```

Небольшая остановка, пока работает отсоединенная функция. Затем ... Отображение идентификатора файбера, статуса файбера и переменной gvar (значение gvar немного увеличится в зависимости от длительности паузы). Статус будет «suspended» (приостановлен), потому что файбер практически всё время проводит в режиме ожидания или передачи управления.

```
tarantool> print('#', fid, '. ', fiber_of_x:status(), '. gvar=', gvar)
# 102 .  suspended . gvar= 399
---
...
```

Небольшая остановка, пока работает отсоединенная функция. Затем ... Отмена файбера. Затем снова отображение идентификатора файбера, статуса файбера и переменной gvar (значение gvar немного увеличится в зависимости от длительности паузы). На этот раз статус будет «dead» (недоступен), потому что произошла отмена.

```
tarantool> fiber_of_x:cancel()
---
...
tarantool> print('#', fid, '. ', fiber_of_x:status(), '. gvar=', gvar)
# 102 .  dead . gvar= 421
---
...
```

Пример неудачной передачи управления

Предупреждение: функция *yield()* и любая функция, которая неявно передает управление (например, *sleep()*), может упасть (выдать исключение).

Например, в этой функции есть цикл, который повторяется до тех пор, пока не произойдет *cancel()*. Последнее, что она выведет, это `before yield`, что свидетельствует о том, что функция `yield()` не сработала, цикл не продолжался до тех пор, пока не сработала функция *testcancel()*.

```
fiber = require('fiber')
function function_name()
  while true do
    print('before testcancel')
    fiber.testcancel()
    print('before yield')
    fiber.yield()
  end
end
fiber_object = fiber.create(function_name)
fiber.sleep(.1)
fiber_object:cancel()
```

Каналы

Вызов `fiber.channel()` для выделения спейса и получение объекта канала, который будет называться «channel» в примерах данного раздела.

Вызов других процедур по каналу для отправки сообщений, получения сообщений или проверки статуса канала.

Обмен сообщения происходит синхронно. Сборщик мусора в Lua отмечает или освобождает канал, когда его никто не использует, как и любой другой Lua-объект. Используйте объектно-ориентированный синтаксис, например `channel:put(message)`, а не `fiber.channel.put(message)`.

`fiber.channel(`[*capacity*]`)`
Создание нового канала связи.

Параметры

- `capacity` (*int*) – максимальное количество слотов (спейсы для сообщений `channel:put`), которые можно использовать одновременно. По умолчанию, 0.

возвращает новый канал.

тип возвращаемого значения пользовательские данные, возможно включая строку «channel . . . ».

```
object channel_object
```

`channel_object:put(`*message*[, *timeout*]`)`

Отправка сообщения по каналу связи. Если канал заполнен, `channel:put()` ожидает, пока не освободится слот в канале.

Параметры

- `message` (*lua-value*) – то, что отправляется, как правило, строка, число или таблица
- `timeout` (*number*) – максимальное количество секунд ожидания, чтобы слот освободился

возвращает Если указан параметр времени ожидания timeout, и в канале нет свободного слота в течение указанного времени, возвращается значение `false` (ложь). Если канал закрыт, возвращается значение `false`. В остальных случаях возвращается значение `true` (правда), которое указывает на успешную отправку.

тип возвращаемого значения boolean (логический)

`channel_object:close()`

Закрытие канала. Все, кто находится в режиме ожидания в канале, отключаются. Все последующие операции `channel:get()` вернут нулевое значение `nil`, а все последующие операции `channel:put()` вернут `false` (ложь).

`channel_object:get(`[*timeout*]`)`

Перехват и удаление сообщения из канала. Если канал пуст, `channel:get()` будет ожидать сообщения.

Параметры

- `timeout` (*number*) – максимальное количество секунд ожидания сообщения

возвращает Если указан параметр времени ожидания timeout, и в канале нет сообщения в течение указанного времени, возвращается нулевое значение `nil`. Если канал закрыт, возвращается значение `nil`. В остальных случаях возвращается сообщение, отправленное на канал с помощью `channel:put()`.

тип возвращаемого значения как правило, строка, число или таблица, как определяет `channel:put()`

`channel_object:is_empty()`

Проверка пустоты канала (отсутствие сообщений).

возвращает `true` (правда), если канал пуст. В противном случае, `false` (ложь).

тип возвращаемого значения boolean (логический)

`channel_object:count()`

Определение количества сообщений в канале.

возвращает количество сообщений.

тип возвращаемого значения число

`channel_object:is_full()`

Проверка заполненности канала.

возвращает `true` (правда), если канал заполнен (количество сообщений в канале равно количеству слотов, то есть нет места для новых сообщений). В противном случае, `false` (ложь).

тип возвращаемого значения boolean (логический)

`channel_object:has_readers()`
Проверка пустого канала на наличие читателей в состоянии ожидания сообщения после отправки запросов `channel:get()`.

возвращает `true` (правда), если на канале есть читатели в ожидании сообщения. В противном случае, `false` (ложь).

тип возвращаемого значения boolean (логический)

`channel_object:has_writers()`
Проверка полного канала на наличие писателей в состоянии ожидания после отправки запросов `channel:put()`.

возвращает `true` (правда), если на канале есть писатели в состоянии ожидании. В противном случае, `false` (ложь).

тип возвращаемого значения boolean (логический)

`channel_object:is_closed()`

возвращает `true` (правда), если канал уже закрыт. В противном случае, `false` (ложь).

тип возвращаемого значения boolean (логический)

Пример

В данном примере дается примерное представление о том, как должны выглядеть функции для файберов. Предполагается, что на функции ссылается *fiber.create()*.

```
fiber = require('fiber')
 channel = fiber.channel(10)
 function consumer_fiber()
     while true do
         local task = channel:get()
         ...
     end
 end

 function consumer2_fiber()
     while true do
         -- 10 секунд
         local task = channel:get(10)
         if task ~= nil then
             ...
         else
             -- время ожидания
         end
     end
 end

 function producer_fiber()
     while true do
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        task = box.space...:select{...}
        ...
        if channel:is_empty() then
            -- канал пуст
        end

        if channel:is_full() then
            -- канал полон
        end

        ...
        if channel:has_readers() then
            -- есть файберы
            -- которые ожидают данные
        end
        ...

        if channel:has_writers() then
            -- есть файберы
            -- которые ожидают читателей
        end
        channel:put(task)
    end
end

function producer2_fiber()
    while true do
        task = box.space...select{...}
        -- 10 секунд
        if channel:put(task, 10) then
            ...
        else
            -- время ожидания
        end
    end
end
```

Условные переменные

Вызов `fiber.cond()` используется для создания именованной условной переменной, которая будет называться „cond“ для примеров данного раздела.

Вызов `cond:wait()` используется, чтобы заставить файбер ожидать сигнал, с помощью условной переменной.

Вызов `cond:signal()` используется, чтобы отправить сигнал для пробуждения отдельного файбера, который выполнил запрос `cond:wait()`.

Вызов `cond:broadcast()` используется для отправки сигнала всем файберам, которые выполнили `cond:wait()`.

`fiber.cond()`

Создание новой условной переменной.

возвращает новая условная переменная.

тип возвращаемого значения Lua-объект

object cond_object

cond_object:wait([*timeout*])
Перевод файбера в режим ожидания до пробуждения другим файбером с помощью метода signal() или broadcast(). Переход в режим ожидания вызывает неявную передачу управления *fiber.yield()*.

Параметры

- timeout – количество секунд ожидания, по умолчанию = всегда.

возвращает Если указан параметр времени ожидания timeout, и сигнал не передается в течение указанного времени, wait() вернет значение false (ложь). Если передается signal() или broadcast(), wait() вернет true (правда).

тип возвращаемого значения boolean (логический)

cond_object:signal()
Пробуждение отдельного файбера, который выполнил wait() для той же переменной. Не выполняет передачу управления (yield).

тип возвращаемого значения nil

cond_object:broadcast()
Пробуждение всех файберов, которые выполнили wait() для той же переменной. Не выполняет передачу управления (yield).

тип возвращаемого значения nil

Пример

Предположим, что запущен экземпляр Tarantool'а на прослушивание на localhost по порту 3301. Предположим, что у пользователя guest есть права на подключение. Используем утилиту tarantoolctl для запуска двух клиентов.

В первом терминале введите:

```
$ tarantoolctl connect '3301'
tarantool> fiber = require('fiber')
tarantool> cond = fiber.cond()
tarantool> cond:wait()
```

Задача повиснет, поскольку cond:wait() – без дополнительного аргумента времени ожидания timeout – уйдет в режим ожидания до изменения условной переменной.

Во втором терминале введите:

```
$ tarantoolctl connect '3301'
tarantool> cond:signal()
```

Теперь снова взгляните на терминал №1. Он покажет, что ожидание прекратилось, и функция cond:wait() вернула значение true.

В данном примере показана зависимость от использования глобальной условной переменной с произвольным именем cond. В реальной жизни разработчики следят за использованием различных имен для условных переменных в разных приложениях.

5.2.11 Модуль *fio*

Общие сведения

Tarantool поддерживает файловый ввод-вывод с помощью API, который аналогичен системным вызовам POSIX. Все операции проводятся асинхронно. Несколько файберов могут получать доступ к одному файлу одновременно.

Модуль `fio` включает в себя:

- функции для *стандартных действий с путем к файлу*,
- функции для *проверки наличия и типа директории или файла*,
- функции для *стандартных действий с файлами*, а также
- *постоянные*., которые совпадают с флаговыми значениями POSIX (например, `fio.c.flag.O_RDONLY` = POSIX O_RDONLY).

Указатель

Ниже приведен перечень всех функций и элементов модуля `fio`.

Имя	Назначение
fio.pathjoin()	Формирование пути к файлу из одной или более частей строки
fio.basename()	Получение имени файла
fio.dirname()	Получение имени директории
fio.abspath()	Получение имен директории и файла
fio.path.exists()	Проверка наличия файла или директории
fio.path.is_dir()	Проверка, является ли файл или директория директорией
fio.path.is_file()	Проверка, является ли файл или директория файлом
fio.path.is_link()	Проверка, является ли файл или директория ссылкой
fio.path.lexists()	Проверка наличия файла или директории
fio.umask()	Определение битов маски
fio.lstat() fio.stat()	Получение информации об объекте файла
fio.mkdir() fio.rmdir()	Создание или удаление директории
fio.chdir()	Изменение рабочей директории
fio.listdir()	Вывод списка файлов в директории
fio.glob()	Получение файлов, имена которых совпадают с заданной строкой
fio.tempdir()	Получение имени директории для хранения временных файлов
fio.cwd()	Получение имени текущей рабочей директории
fio.copytree() fio.mktree() fio.rmtree()	Создание и удаление директорий
fio.link() fio.symlink() fio.readlink() fio.unlink()	Создание и удаление ссылок
fio.rename()	Переименование файла или директории
fio.utime()	Изменение времени обновления файла
fio.copyfile()	Копирование файла

Продолжается на следующей странице

Таблица 5 – продолжение с предыдущей страницы

Имя	Назначение
fio.chown() fio.chmod()	Управление правами на использование и правами владения объектами файла
fio.truncate()	Уменьшение размера файла
fio.sync()	Проверка записи изменений на диск
fio.open()	Открытие файла
file-handle:close()	Закрытие файла
file-handle:pread() file-handle:pwrite()	Чтение или запись в файл с произвольным доступом
file-handle:read() file-handle:write()	Чтение или запись в файл не с произвольным доступом
file-handle:truncate()	Изменение размера открытого файла
file-handle:seek()	Изменение позиции в файле
file-handle:stat()	Получение статистики об открытом файле
file-handle:fsync() file-handle:fdatasync()	Проверка записи изменений в открытом файле на диск
fio.c	Таблица переменных, аналогичных флаговым значениям POSIX

Стандартные действия с путем к файлу

`fio.pathjoin`(*partial-string*[, *partial-string* ...])

Конкатенация частей строки, разделенных „/“ для формирования пути к файлу.

Параметры

- `partial-string` (`string`) – одна или более строк для конкатенации.

возвращает путь к файлу

тип возвращаемого значения строка

Пример:

```
tarantool> fio.pathjoin('/etc', 'default', 'myfile')
---
- /etc/default/myfile
...
```

`fio.basename`(*path-name*[, *suffix*])

Удаление из полного пути к файлу всего, за исключением последней части (имени файла). Также удаление суффикса, если он передается.

Параметры

- `path-name` (`string`) – путь к файлу
- `suffix` (`string`) – суффикс

возвращает имя файла

тип возвращаемого значения строка

Пример:

```
tarantool> fio.basename('/path/to/my.lua', '.lua')
---
- my
...
```

fio.dirname(*path-name*)
Удаление последней части (имени файла) из полного пути к файлу.

Параметры

- path-name (string) – путь к файлу

возвращает имя директории, то есть путь к файлу без имени файла.

тип возвращаемого значения строка

Пример:

```
tarantool> fio.dirname('/path/to/my.lua')
---
- '/path/to/'
```

fio.abspath(*file-name*)
Возврат полного пути к файлу на основании последней части (имени файла).

Параметры

- file-name (string) – имя файла

возвращает имя директории, то есть путь к файлу с именем файла.

тип возвращаемого значения строка

Пример:

```
tarantool> fio.abspath('my.lua')
---
- '/path/to/my.lua'
...
```

Проверка наличия и типа директории или файла

Функции в данном разделе подобны некоторым функциям Python os.path.

fio.path.exists(*path-name*)

Параметры

- path-name (string) – путь к директории или файлу.

возвращает true (правда), если путь к файлу ссылается на директорию или файл, которые присутствуют в системе, и не представляет собой нерабочую символьную ссылку; в противном случае, false (ложь)

тип возвращаемого значения boolean (логический)

fio.path.is_dir(*path-name*)

Параметры

- path-name (string) – путь к директории или файлу.

возвращает true (правда), если путь к файлу ссылается на директорию; в противном случае, false (ложь)

тип возвращаемого значения boolean (логический)

`fio.path.is_file(`*path-name*`)`

Параметры

- `path-name` (`string`) – путь к директории или файлу.

возвращает true (правда), если путь к файлу ссылается на файл; в противном случае, false (ложь)

тип возвращаемого значения boolean (логический)

`fio.path.is_link(`*path-name*`)`

Параметры

- `path-name` (`string`) – путь к директории или файлу.

возвращает true (правда), если путь к файлу ссылается на символьную ссылку; в противном случае, false (ложь)

тип возвращаемого значения boolean (логический)

`fio.path.lexists(`*path-name*`)`

Параметры

- `path-name` (`string`) – путь к директории или файлу.

возвращает true (правда), если путь к файлу ссылается на директорию или файл, которые присутствуют в системе, и представляет собой нерабочую символьную ссылку; в противном случае, false (ложь)

тип возвращаемого значения boolean (логический)

Стандартные действия с файлом

`fio.umask(`*mask-bits*`)`
Определение битов маски при создании файлов или директорий. Для получения более подробного описания введите `man 2 umask`.

Параметры

- `mask-bits` (*number*) – биты маски.

возвращает предыдущие биты маски.

тип возвращаемого значения число

Пример:

```
tarantool> fio.umask(tonumber('755', 8))
---
- 493
...
```

`fio.lstat(`*path-name*`)`
`fio.stat(`*path-name*`)`
Возврат информации об объекте файла. Для получения более подробной информации введите `man 2 lstat` или `man 2 stat`.

Параметры

- path-name (string) – путь к файлу.

возвращает (Если ошибки нет) таблица с полями, которые описывают размер блока файла, время создания, размер и прочие атрибуты. (В случае ошибки) возвращаются два значения: null, сообщение об ошибке.

тип возвращаемого значения таблица.

Кроме того, результат `fio.stat('имя-файла')` будет включать в себя методы, которые аналогичны макросам в POSIX:

- `is_blk()` = макрос S_ISBLK в POSIX,
- `is_chr()` = макрос S_ISCHR в POSIX
- `is_dir()` = макрос S_ISDIR в POSIX,
- `is_fifo()` = макрос S_ISFIFO в POSIX,
- `is_link()` = макрос S_ISLINK в POSIX,
- `is_reg()` = макрос S_ISREG в POSIX,
- `is_sock()` = макрос S_ISSOCK в POSIX.

Например, `fio.stat('/'):is_dir()` вернет true.

Пример:

```
tarantool> fio.lstat('/etc')
---
- inode: 1048577
  rdev: 0
  size: 12288
  atime: 1421340698
  mode: 16877
  mtime: 1424615337
  nlink: 160
  uid: 0
  blksize: 4096
  gid: 0
  ctime: 1424615337
  dev: 2049
  blocks: 24
...
```

`fio.mkdir(`*path-name*[, *mode*]`)`
`fio.rmdir(`*path-name*`)`

Создание или удаление директории. Для получения подробной информации введите `man 2 mkdir` или `man 2 rmdir`.

Параметры

- path-name (string) – путь к директории.
- mode (*number*) – Биты режима работы можно передать в виде числа или строковых постоянных, например `S_IWUSR`. Биты режима работы можно комбинировать путем обрамления их в фигурные скобки.

возвращает (Если ошибки нет) true. (В случае ошибки) возвращаются два значения: false, сообщение об ошибке.

тип возвращаемого значения boolean (логический)

Пример:

```
tarantool> fio.mkdir('/etc')
---
- false
...
```

`fio.chdir`(*path-name*)

Изменение рабочей директории. Для получения более подробной информации введите `man 2 chdir`.

Параметры

- `path-name` (`string`) – путь к директории.

возвращает (Если выполнено) true. (Если не выполнено) false.

тип возвращаемого значения boolean (логический)

Пример:

```
tarantool> fio.chdir('/etc')
---
- true
...
```

`fio.listdir`(*path-name*)

Вывод списка файлов в директории. Результат аналогичен результату выполнения команды `ls` в терминале.

Параметры

- `path-name` (`string`) – путь к директории.

возвращает (Если ошибки нет) список файлов. (В случае ошибки) возвращаются два значения: null, сообщение об ошибке.

тип возвращаемого значения таблица

Пример:

```
tarantool> fio.listdir('/usr/lib/tarantool')
---
- - mysql
...
```

`fio.glob`(*path-name*)

Возврат списка файлов, имена которых совпадают с введенной строкой. Список составляется с одним флагом, который контролирует поведение функции: `GLOB_NOESCAPE`. Для получения подробной информации введите `man 3 glob`.

Параметры

- `path-name` (`string`) – путь к файлу, который может содержать специальные символы.

возвращает список файлов, имена которых совпадают с введенной строкой.

тип возвращаемого значения таблица

Возможные ошибки: nil.

Пример:

```
tarantool> fio.glob('/etc/x*')
---
- - /etc/xdg
  - /etc/xml
  - /etc/xul-ext
...
```

`fio.tempdir()`

Возврат имени директории, которую можно использовать для хранения временных файлов.

Пример:

```
tarantool> fio.tempdir()
---
- /tmp/lG31e7
...
```

`fio.cwd()`

Возврат имени текущей рабочей директории.

Пример:

```
tarantool> fio.cwd()
---
- /home/username/tarantool_sandbox
...
```

`fio.copytree(`*from-path*, *to-path*`)`

Копирование всего из директории from-path, включая поддиректории, в to-path. Результат аналогичен результату выполнения команды `cp -r` в терминале. Директория to-path не должна быть пустой.

Параметры

- `from-path` (`string`) – путь.
- `to-path` (`string`) – путь.

возвращает (Если ошибки нет) true. (В случае ошибки) возвращаются два значения: false, сообщение об ошибке.

тип возвращаемого значения boolean (логический)

Пример:

```
tarantool> fio.copytree('/home/original','/home/archives')
---
- true
...
```

`fio.mktree(`*path-name*`)`

Создание пути, включая поддиректории, но без содержимого файла. Результат аналогичен результату выполнения команды `mkdir -p` в терминале.

Параметры

- `path-name` (`string`) – путь.

возвращает (Если ошибки нет) true. (В случае ошибки) возвращаются два значения: false, сообщение об ошибке.

тип возвращаемого значения boolean (логический)

Пример:

```
tarantool> fio.mktree('/home/archives')
---
- true
...
```

`fio.rmtree`(*path-name*)

Удаление указанной директории, включая поддиректории. Результат аналогичен результату выполнения команды `rmdir -r` в терминале. Директория не должна быть пустой.

Параметры

- `path-name` (`string`) – путь.

возвращает (Если ошибки нет) true. (В случае ошибки) возвращаются два значения: null, сообщение об ошибке.

тип возвращаемого значения boolean (логический)

Пример:

```
tarantool> fio.rmtree('/home/archives')
---
- true
...
```

`fio.link`(*src*, *dst*)
`fio.symlink`(*src*, *dst*)
`fio.readlink`(*src*)
`fio.unlink`(*src*)

Функции для создания и удаления ссылок. Для получения подробной информации введите `man readlink`, `man 2 link`, `man 2 symlink`, `man 2 unlink`.

Параметры

- `src` (`string`) – имя существующего файла.
- `dst` (`string`) – связанное имя.

возвращает (Если ошибки нет) `fio.link`, `fio.symlink` и `fio.unlink` возвращают true (правда), `fio.readlink` возвращает ссылку. (В случае ошибки) возвращаются два значения: false|null, сообщение об ошибке.

Пример:

```
tarantool> fio.link('/home/username/tmp.txt', '/home/username/tmp.txt2')
---
- true
...
tarantool> fio.unlink('/home/username/tmp.txt2')
---
- true
...
```

fio.rename(*path-name*, *new-path-name*)
Переименование файла или директории. Для получения подробной информации введите man 2 rename.

Параметры

- path-name (string) – первоначальное имя.
- new-path-name (string) – новое имя.

возвращает (Если ошибки нет) true. (В случае ошибки) возвращаются два значения: false, сообщение об ошибке.

тип возвращаемого значения boolean (логический)

Пример:

```
tarantool> fio.rename('/home/username/tmp.txt', '/home/username/tmp.txt2')
---
- true
...
```

fio.utime(*file-name*[, *accesstime*[, *updatetime*]])
Change the access time and possibly also change the update time of a file. For details type man 2 utime. Times should be expressed as number of seconds since the epoch.

Параметры

- file-name (string) – name.
- accesstime (*number*) – time of last access. default current time.
- updatetime (*number*) – time of last update. default = access time.

возвращает (Если ошибки нет) true. (В случае ошибки) возвращаются два значения: false, сообщение об ошибке.

тип возвращаемого значения boolean (логический)

Пример:

```
tarantool> fio.utime('/home/username/tmp.txt')
---
- true
...
```

fio.copyfile(*path-name*, *new-path-name*)
Копирование файла. Результат аналогичен результату выполнения команды cp в терминале.

Параметры

- path-name (string) – путь к первоначальному файлу.
- new-path-name (string) – путь к новому файлу.

возвращает (Если ошибки нет) true. (В случае ошибки) возвращаются два значения: false, сообщение об ошибке.

тип возвращаемого значения boolean (логический)

Пример:

```
tarantool> fio.copyfile('/home/user/tmp.txt', '/home/usern/tmp.txt2')
---
- true
...
```

fio.chown(*path-name*, *owner-user*, *owner-group*)
fio.chmod(*path-name*, *new-rights*)

Управление правами на использование и правами владения объектами файла. Для получения подробной информации введите `man 2 chown` или `man 2 chmod`.

Параметры

- `owner-user` (`string`) – новый UID пользователя.
- `owner-group` (`string`) – новый UID группы.
- `new-rights` (*number*) – новые права

возвращает null

Пример:

```
tarantool> fio.chmod('/home/username/tmp.txt', tonumber('0755', 8))
---
- true
...
tarantool> fio.chown('/home/username/tmp.txt', 'username', 'username')
---
- true
...
```

fio.truncate(*path-name*, *new-size*)

Уменьшение размера файла до указанного значения. Для получения подробной информации введите `man 2 truncate`.

Параметры

- `path-name` (`string`) –
- `new-size` (*number*) –

возвращает (Если ошибки нет) true. (В случае ошибки) возвращаются два значения: false, сообщение об ошибке.

тип возвращаемого значения boolean (логический)

Пример:

```
tarantool> fio.truncate('/home/username/tmp.txt', 99999)
---
- true
...
```

fio.sync()

Проверка записи изменений на диск. Для получения подробной информации введите `man 2 sync`.

возвращает true – если выполнено, false – если не выполнено.

тип возвращаемого значения boolean (логический)

Пример:

```
tarantool> fio.sync()
---
- true
...
```

fio.open(*path-name*[, *flags*[, *mode*]])

Открытие файла в процессе подготовки к чтению, записи или поиску.

Параметры

- path-name (string) – Полный путь к открываемому файлу.
- flags (*number*) – Флаги могут передаваться в виде числа или в виде строковых постоянных, например „O_RDONLY“, „O_WRONLY“, „O_RDWR“. Флаги можно комбинировать путем обрамления их в фигурные скобки. Все флаги в Linux, как описано на странице руководства по Linux, представлены ниже:
  - O_APPEND (открывать файл в режиме добавления),
  - O_ASYNC (включать ввод-вывод, управляемый сигналом),
  - O_CLOEXEC (устанавливать флаг, связанный с закрытием),
  - O_CREAT (создать файл, если он не существует),
  - O_DIRECT (минимизировать или отключать кэширование),
  - O_DIRECTORY (завершать вызов с ошибкой, если путь не является директорией),
  - O_EXCL (завершать вызов с ошибкой, если файл не может быть создан),
  - O_LARGEFILE (открывать 64-битные файлы),
  - O_NOATIME (не обновлять время последнего доступа к файлу),
  - O_NOCTTY (не делать терминальное устройство управляющим терминалом tty),
  - O_NOFOLLOW (не открывать символьные ссылки),
  - O_NONBLOCK (открывать в неблокирующем режиме),
  - O_PATH (получить путь для низкоуровневого использования),
  - O_SYNC (включить принудительную запись, если возможно),
  - O_TMPFILE (создать безымянный временный файл),
  - O_TRUNC (урезать)

  ... и всегда используется один из флагов:

  - O_RDONLY (только для чтения),
  - O_WRONLY (только для записи) или
  - O_RDWR (для чтения и записи).
- mode (*number*) – Биты режима работы можно передать в виде числа или строковых постоянных, например S_IWUSR. Биты режима работы имеют значение, если указаны флаги O_CREAT или O_TMPFILE. Биты режима работы можно комбинировать путем обрамления их в фигурные скобки.

возвращает (Если ошибки нет) дескриптор файла (далее сокращенно „fh“). (В случае ошибки) возвращаются два значения: null, сообщение об ошибке.

тип возвращаемого значения пользовательские данные

Возможные ошибки: nil.

Пример 1:

```
tarantool> fh = fio.open('/home/username/tmp.txt', {'O_RDWR', 'O_APPEND'})
---
...
tarantool> fh -- отображение дескриптора файла, который возвращает fio.open
---
- fh: 11
...
```

Пример 2:

Using `fio.open()` with `tonumber('N', 8)` to set permissions as an octal number:

```
tarantool> fio.open('x.txt', {'O_WRONLY', 'O_CREAT'}, tonumber('644',8))
 ---
 - fh: 12
 ...
```

`object file-handle`

`file-handle:close()`
Закрытие файла, который был открыт с помощью `fio.open`. Для получения подробной информации введите `man 2 close`.

Параметры

- `fh` (*userdata*) – дескриптор файла, который возвращает `fio.open()`.

возвращает true – если выполнено, false – если не выполнено.

тип возвращаемого значения boolean (логический)

Пример:

```
tarantool> fh:close() -- где fh = дескриптор файла
---
- true
...
```

`file-handle:pread(`*count*, *offset*`)`
`file-handle:pread(`*buffer*, *count*, *offset*`)`
Чтение файла с произвольным доступом независимо от текущего положения в поиске. Для получения подробной информации введите `man 2 pread`.

Параметры

- `fh` (*userdata*) – дескриптор файла, который возвращает `fio.open()`.
- `buffer` – куда считать (если формат – `pread(buffer, count, offset)`)
- `count` (*number*) – количество байтов для чтения
- `offset` (*number*) – смещение в файле – где начинается чтение

Если формат – `pread(count, offset)`, возвращается строка с данными, прочитанными из файла, либо пустая строка, если не выполнено.

Если формат – `pread(buffer, count, offset)`, возвращаются данные в буфер. Буферы можно ввести с помощью *buffer.ibuf*.

Пример:

```
tarantool> fh:pread(25, 25)
---
- |
  elete from t8//
  insert in
...
```

`file-handle:pwrite(`*new-string*, *offset*`)`
`file-handle:pwrite(`*buffer*, *count*, *offset*`)`

Запись в файл с произвольным доступом независимо от текущего положения в поиске. Для получения подробной информации введите `man 2 pwrite`.

Параметры

- `fh` (*userdata*) – дескриптор файла, который возвращает `fio.open()`.
- `new-string` (string) – записываемое значение (если формат – `pwrite(new-string, offset)`)
- `buffer` (*cdata*) – записываемое значение (если формат – `pwrite(buffer, count, offset)`)
- `count` (*number*) – количество байтов для чтения
- `offset` (*number*) – смещение в файле – где начинается запись

возвращает true – если выполнено, false – если не выполнено.

тип возвращаемого значения boolean (логический)

Если формат –`pwrite(new-string, offset)`, строка записывается в файл до конца строки.

Если формат – `pwrite(buffer, count, offset)`, содержимое буфера записывается в файл в объеме, указанном в `count`. Буферы можно ввести с помощью *buffer.ibuf*.

```
tarantool> ibuf = require('buffer').ibuf()
---
...

tarantool> fh:pwrite(ibuf, 1, 0)
---
- true
...
```

`file-handle:read(`$\left[$*count*$\right]$`)`
`file-handle:read(`*buffer*, *count*`)`

Чтение файла не с произвольным доступом. Для получения подробной информации введите `man 2 read` или `man 2 write`.

Примечание: `fh:read` and `fh:write` affect the seek position within the file, and this must be taken into account when working on the same file from multiple fibers. It is possible to limit or prevent file access from other fibers with *fiber.cond()* or *fiber.channel()*.

Параметры

- `fh` (*`userdata`*) – дескриптор файла, который возвращает `fio.open()`.
- `buffer` – куда считать (если формат – `read(buffer, count)`)
- `count` (*`number`*) – количество байтов для чтения

возвращает

- Если формат – `read()` – без `count` – считываются все байты в файле.
- Если формат – `read()` или `read([count])`, возвращается строка с данными, прочитанными из файла, либо пустая строка, если не выполнено.
- Если формат – `read(buffer, count)`, возвращаются данные в буфер. Буферы можно ввести с помощью *buffer.ibuf*.
- В случае ошибки метод возвращает `nil, err` и устанавливает ошибку на `errno`.

```
tarantool> ibuf = require('buffer').ibuf()
---
...

tarantool> fh:read(ibuf:reserve(5), 5)
---
- 5
...

tarantool> require('ffi').string(ibuf:alloc(5),5)
---
- abcde
```

`file-handle:write(`*new-string*`)`
`file-handle:write(`*buffer*, *count*`)`

Запись в файл не с произвольным доступом. Для получения подробной информации введите `man 2 write`.

Примечание: `fh:read` and `fh:write` affect the seek position within the file, and this must be taken into account when working on the same file from multiple fibers. It is possible to limit or prevent file access from other fibers with *fiber.cond()* or *fiber.channel()*.

Параметры

- `fh` (*`userdata`*) – дескриптор файла, который возвращает `fio.open()`.
- `new-string` (`string`) – записываемое значение (если формат – `write(new-string)`)
- `buffer` (*`cdata`*) – записываемое значение (если формат – `write(buffer, count)`)
- `count` (*`number`*) – количество байтов для чтения

возвращает true – если выполнено, false – если не выполнено.

тип возвращаемого значения boolean (логический)

Если формат – `write(new-string)`, строка записывается в файл до конца строки.

Если формат – `write(buffer, count)`, содержимое буфера записывается в файл в объеме, указанном в `count`. Буферы можно ввести с помощью *buffer.ibuf*.

Пример:

```
tarantool> fh:write("new data")
---
- true
...
tarantool> ibuf = require('buffer').ibuf()
---
...
tarantool> fh:write(ibuf, 1)
---
- true
...
```

`file-handle:truncate(`*new-size*`)`

Изменение размера открытого файла. Отличается от функции `fio.truncate`, которая изменяет размер закрытого файла.

Параметры

- `fh` (*userdata*) – дескриптор файла, который возвращает `fio.open()`.

возвращает true – если выполнено, false – если не выполнено.

тип возвращаемого значения boolean (логический)

Пример:

```
tarantool> fh:truncate(0)
---
- true
...
```

`file-handle:seek(`*position*[, *offset-from*]`)`

Изменение положения в файле на указанное. Для получения подробной информации введите `man 2 seek`.

Параметры

- `fh` (*userdata*) – дескриптор файла, который возвращает `fio.open()`.
- `position` (*number*) – искомое положение
- `offset-from` (string) – „SEEK_END“ = конец файла, „SEEK_CUR“ = текущее положение, „SEEK_SET“ = начало файла.

возвращает новое положение, если выполнено

тип возвращаемого значения число

Возможные ошибки: nil.

Пример:

```
tarantool> fh:seek(20, 'SEEK_SET')
---
- 20
...
```

file-handle:stat()
Возврат статистики об открытом файле. Отличается от функции fio.stat, которая возвращает статистику о закрытом файле. Для получения подробной информации введите man 2 stat.

Параметры

- fh (*userdata*) – дескриптор файла, который возвращает fio.open().

возвращает подробная информация о файле.

тип возвращаемого значения таблица

Пример:

```
tarantool> fh:stat()
---
- inode: 729866
  rdev: 0
  size: 100
  atime: 140942855
  mode: 33261
  mtime: 1409430660
  nlink: 1
  uid: 1000
  blksize: 4096
  gid: 1000
  ctime: 1409430660
  dev: 2049
  blocks: 8
...
```

file-handle:fsync()
file-handle:fdatasync()
Проверка записи изменений в открытом файле на диск. Ср. с fio.sync для всех файлов. Для получения подробной информации введите man 2 fsync or man 2 fdatasync.

Параметры

- fh (*userdata*) – дескриптор файла, который возвращает fio.open().

возвращает true – если выполнено, false – если не выполнено.

Пример:

```
tarantool> fh:fsync()
---
- true
...
```

Постоянные для файлового ввода-вывода

fio.c
Таблица с постоянными, которые совпадают с флаговыми значениями в POSIX на целевой платформе (см. man 2 stat).

Пример:

```
tarantool> fio.c
---
- seek:
    SEEK_SET: 0
    SEEK_END: 2
    SEEK_CUR: 1
  mode:
    S_IWGRP: 16
    S_IXGRP: 8
    S_IROTH: 4
    S_IXOTH: 1
    S_IRUSR: 256
    S_IXUSR: 64
    S_IRWXU: 448
    S_IRWXG: 56
    S_IWOTH: 2
    S_IRWXO: 7
    S_IWUSR: 128
    S_IRGRP: 32
  flag:
    O_EXCL: 2048
    O_NONBLOCK: 4
    O_RDONLY: 0
    <...>
...
```

5.2.12 Модуль *fun*

Luafun, также известная как библиотека для функционального программирования в Lua, пользуется преимуществами LuaJIT, чтобы помочь пользователям создавать сложные функции. Модуль включает в себя «последовательные процессоры», такие как `map`, `filter`, `reduce`, `zip` – они берут написанную пользователем функцию в качестве аргумента и применяют ее к каждому элементу в последовательности, что может работать быстрее или более удобно, чем написанный пользователем цикл. Модуль включает в себя «генераторы», такие как `range`, `tabulate` и `rands` – они возвращают ограниченный или неограниченный ряд значений. Модуль включает в себя «преобразователи», «фильтры», «компоновщики» ... или, коротко говоря, все важные функции из таких языков, как Standard ML, Haskell или Erlang.

Вся документация находится по ссылке On the luafun section of github. Однако, первую главу можно пропустить, поскольку установка уже выполнена в пределах Tarantool'a. Единственное, что нужно сделать, – выполнить обычный запрос `require`. После этого сработают все операции, описанные в руководстве по работе с библиотекой для функционального программирования в Lua, при условии, что перед ними указывается имя, возвращенное запросом `require`. Например:

```
tarantool> fun = require('fun')
---
...
tarantool> for _k, a in fun.range(3) do
         >   print(a)
         > end
1
2
3
---
...
```

5.2.13 Модуль *http*

Общие сведения

Модуль `http`, в частности вложенный модуль `http.client` , обеспечивать функциональные возможности HTTP-клиента с поддержкой HTTPS и механизма поддержания в активном состоянии keepalive. Модуль использует процедуры из библиотеки libcurl.

Указатель

Ниже приведен перечень всех функций модуля `http`.

Имя	Назначение
http.client.new()	Создание экземпляра HTTP-клиента
client_object:request()	Выполнение HTTP-запроса
client_object:stat()	Получение таблицы со статистикой

`http.client.new`([*options*])

Создание нового экземпляра HTTP-клиента.

Параметры

- `options` (`table`) – настройки целочисленных значений, которые передаются в libcurl.

Доступны два параметра: `max_connections` и `max_total_connections`.

`max_connections` – это максимальное количество записей в кэше, которое влияет на CURLMOPT_MAXCONNECTS в `libcurl`. По умолчанию -1.

`max_total_connections` is the maximum number of active connections. It affects libcurl CURLMOPT_MAX_TOTAL_CONNECTIONS. It is ignored if the curl version is less than 7.30. The default is 0, which allows libcurl to scale accordingly to easy handle the count.

Обычно значений параметров по умолчанию будет достаточно, но в редких случаях может понадобиться их настройка. На этот случай два совета.

1. You may want to control the maximum number of sockets that a particular HTTP client uses simultaneously. If a system passes many requests to distinct hosts, then libcurl cannot reuse sockets. In this case setting `max_total_connections` may be useful, since it causes curl to avoid creating too many sockets which would not be used anyway.
2. Do not set `max_connections` less than `max_total_connections` unless you are confident about your actions. When `max_connections` is less then `max_total_connections`, in some cases libcurl will not reuse sockets for requests that are going to the same host. If the limit is reached and a new request occurs, then libcurl will first create a new socket, send the request, wait for the first connection to be free, and close it, in order to avoid exceeding the `max_connections` cache size. In the worst case, libcurl will create a new socket for every request, even if all requests are going to the same host. See this Tarantool issue on github for details.

возвращает новый экземпляр HTTP-клиента

тип возвращаемого значения пользовательские данные

Пример:

```
tarantool> http_client = require('http.client').new({max_connections = 5})
---
...
```

object client_object

client_object:request(*method*, *url*, *body*, *opts*)

Если http_client – это экземпляр HTTP-клиента, http_client:request() выполнит HTTP-запрос, и в случае успешного подключения вернет таблицу с информацией о подключении.

Параметры

- method (string) – HTTP-метод, например „GET“, „POST“ или „PUT“
- url (string) – расположение, например „https://tarantool.org/doc“
- body (string) – необязательное начальное сообщение, например „My text string!“
- opts (table) – таблица с параметрами подключения, которые могут содержать любые из следующих компонентов:
  - timeout – количество секунд ожидания API-запроса curl на чтение до превышения времени ожидания
  - ca_path – путь к директории, где хранятся один или более сертификатов для проверки подключенного узла
  - ca_file – путь к SSL-сертификату для проверки подключенного узла
  - verify_host – включение/отключение проверки имени сертификата (CN) для хоста. См. также CURLOPT_SSL_VERIFYHOST
  - verify_peer – включение/отключение проверки SSL-сертификата подключенного узла. См. также CURLOPT_SSL_VERIFYPEER
  - ssl_key – путь к файлу закрытого ключа для клиентского TSL-сертификата и SSL-сертификата. См. также CURLOPT_SSLKEY
  - ssl_cert – путь к файлу клиентского SSL-сертификата. См. также CURLOPT_SSLCERT
  - headers – таблица HTTP-заголовков
  - keepalive_idle – время задержки в секундах, в течение которого операционная система находится в режиме ожидания подключения до отправки сообщений для поддержания в активном состоянии keepalive. См. также CURLOPT_TCP_KEEPIDLE и примечание ниже к keepalive_interval.
  - keepalive_interval – период времени в секундах между отправками сообщений keepalive в операционной системе. См. также CURLOPT_TCP_KEEPINTVL. Если заданы оба параметр keepalive_idle и keepalive_interval, то Tarantool отобразит HTTP-заголовки для keepalive: Connection:Keep-Alive и Keep-Alive:timeout=<keepalive_idle>. В противном случае, Tarantool отправит Connection:close.
  - low_speed_time – установка «времени работы с низкой скоростью» – времени, в течение которого скорость передачи должна быть ниже «предела низкой скорости», чтобы библиотека посчитала работу слишком медленной и завершила ее. См. также CURLOPT_LOW_SPEED_TIME

- `low_speed_limit` – установка «предела низкой скорости» – средней скорости передачи в байтах в секунду, ниже которой должна быть скорость передачи, чтобы библиотека посчитала работу слишком медленной и завершила ее. См. также CURLOPT_LOW_SPEED_LIMIT
- `verbose` – включение/отключение режима отображения подробной информации
- `unix_socket` – имя сокета, которое используется вместо адреса в сети Интернет, для локального соединения. Сборка сервера Tarantool'а должна осуществляться с помощью `libcurl` 7.40 или более поздней версии. См. *второй пример* далее в разделе.
- `max_header_name_len` – максимальная длина имени заголовка. Если имя заголовка больше данного значения, оно усекается до такой длины. По умолчанию, „32“.
- `follow_location` - when the option is set to `true` (default) and the response has a 3xx code, the HTTP client will automatically issue another request to a location that a server sends in the `Location` header. If the new response is 3xx again, the HTTP client will issue still another request and so on in a loop until a non-3xx response will be received. This last response will be returned as a result. Setting this option to `false` allows to disable this behavior. In this case, the HTTP client will return a 3xx response itself.

возвращает информация о подключении со всеми следующими компонентами:

- `status` – статус HTTP-ответа
- `reason` – текст статуса HTTP-ответа
- `headers` – Lua-таблица с нормализованными HTTP-заголовками
- `body` – тело сообщения-ответа
- `proto` – версия протокола

тип возвращаемого значения таблица

Компонент `cookies` содержит Lua-таблицу, ключом в которой является имя файла cookie. Значением же является массив из двух элементов: первый элемент представляет собой значение данных cookie, а второй – массив с параметрами файла cookie. Возможные параметры: «Expires», «Max-Age», «Domain», «Path», «Secure», «HttpOnly», «SameSite». Обратите внимание, что параметр представляет собой строку, в которой знак „=“ разделяет имя параметра и его значение. Дополнительную информацию можно получить здесь.

Пример

Информацию по файлам cookies можно использовать следующим образом:

```
tarantool> require('http.client').get('https://www.tarantool.io/en/').cookies
---
- csrftoken:
  - bWJVkBybvX9LdJ8uLPOTVrit5P3VbRjE3potYVOuUnsSjYT5ahghDVO6tXRkfnOl
  - - Max-Age=31449600
    - Path=/
...

tarantool> cookies = require('http.client').get('https://www.tarantool.io/en/').cookies
---
...
```

(continues on next page)

(продолжение с предыдущей страницы)

```
tarantool> options = cookies['csrftoken'][2]
---
...

tarantool> for _, option in ipairs(options) do
         > if option:startswith('csrftoken cookie's Max-Age = ') then
         > print(option)
         > end
         > end

csrftoken cookie's Max-Age = 31449600
---
...

tarantool>
```

Для запросов существуют следующие ускоренные методы:

- `http_client:get(url, options)` – вспомогательный метод для `http_client:request("GET", url, nil, opts)`
- `http_client:post (url, body, options)` – ускоренный метод для `http_client:request("POST", url, body, opts)`
- `http_client:put(url, body, options)` – ускоренный метод для `http_client:request("PUT", url, body, opts)`
- `http_client:patch(url, body, options)` – ускоренный метод для `http_client:request("PATCH", url, body, opts)`
- `http_client:options(url, options)` – ускоренный метод для `http_client:request("OPTIONS", url, nil, opts)`
- `http_client:head(url, options)` – ускоренный метод для `http_client:request("HEAD", url, nil, opts)`
- `http_client:delete(url, options)` – ускоренный метод для `http_client:request("DELETE", url, nil, opts)`
- `http_client:trace(url, options)` – ускоренный метод для `http_client:request("TRACE", url, nil, opts)`
- `http_client:connect:(url, options)` – ускоренный метод для `http_client:request("CONNECT", url, nil, opts)`

На запросы могут влиять переменные окружения, например, пользователи могут задать прокси-сервер с HTTP, указав `HTTP_PROXY=`*`прокси-сервер`* перед выполнением каких-либо запросов. См. веб-документ по переменным окружения Environment variables libcurl understands.

`client_object:stat()`

Функция `http_client:stat()` возвращает таблицу со статистическими данными:

- `active_requests` – количество активно выполняемых запросов
- `sockets_added` – общее количество сокетов, добавленных в событийный цикл
- `sockets_deleted` – общее количество сокетов, удаленных из событийного цикла
- `total_requests` – общее количество запросов

- `http_200_responses` – общее количество запросов, которые вернули код состояния HTTP 200
- `http_other_responses` – общее количество запросов, которые не вернули код состояния HTTP 200
- `failed_requests` – общее количество невыполненных запросов, включая системные ошибки, ошибки `curl` и HTTP-ошибки

Пример 1:

Подключение к HTTP-серверу, просмотр размера ответа на „GET“-запрос и просмотр статистики по сессии.

```
tarantool> http_client = require('http.client').new()
---
...
tarantool> r = http_client:request('GET','http://tarantool.org')
---
...
tarantool> string.len(r.body)
---
- 21725
...
tarantool> http_client:stat()
---
- total_requests: 1
  sockets_deleted: 2
  failed_requests: 0
  active_requests: 0
  http_other_responses: 0
  http_200_responses: 1
  sockets_added: 2
```

Пример 2:

Запустите два экземпляра Tarantool'а на одном компьютере.

В первом экземпляре Tarantool'а включите прослушивание Unix-сокета:

```
box.cfg{listen='/tmp/unix_domain_socket.sock'}
```

На втором экземпляре Tarantool'а отправьте с помощью `http_client`:

```
box.cfg{}
http_client = require('http.client').new({5})
http_client:put('http://localhost/','body',{unix_socket = '/tmp/unix_domain_socket.sock'})
```

Терминал №1 покажет сообщение об ошибке: «Invalid MsgPack». Данный пример бесполезен, но наглядно демонстрирует синтаксис и получение отправленного сообщения.

5.2.14 Модуль *iconv*

Общие сведения

Модуль `iconv` предоставляет метод конвертации строки с одним типом кодировки в строку с другим типом кодировки, например из ASCII в UTF-8. Он основывается на процедурах с iconv в POSIX.

Точный список доступных кодировок зависит от окружения. Как правило, в список входят ASCII, BIG5, KOI8R, LATIN8, MS-GREEK, SJIS и около 100 других. Чтобы увидеть общий список, введите команду `iconv --list` в терминале.

Указатель

Ниже приведен перечень всех функций модуля `iconv`.

Имя	Назначение
iconv.new()	Создание экземпляра iconv
iconv.converter()	Преобразование строки

`iconv.new(`*to*, *from*`)`

Создание нового iconv-экземпляра.

Параметры

- `to` (`string`) – название будущей кодировки.
- `from` (`string`) – название используемой кодировки.

возвращает новый экземпляр iconv – на самом деле, вызываемая функция

тип возвращаемого значения пользовательские данные

Если значение одного из параметров представляет собой недопустимое имя, появится сообщение об ошибке.

Пример:

```
tarantool> converter = require('iconv').new('UTF8', 'ASCII')
---
...
```

`iconv.converter(`*input-string*`)`

Преобразование.

param string input-string строка для преобразования («из»)

возвращает строка, получаемая в результате преобразования («в»)

Если что-либо в строке input-string нельзя преобразовать, появится сообщение об ошибке, строка останется неизменной.

Пример:

Мы знаем, что кодовая точка для заглавной буквы «Д» в Unicode представляет собой шестнадцатеричное число 0414 в соответствии с таблицей символов Unicode. Таким образом, так она будет выглядеть в UTF-16. Мы знаем, что как правило, Tarantool использует набор символов UTF-8. Поэтому для создания конвертора из UTF-8 в UTF-16 используем string.hex(„Д“), чтобы показать, как выглядит кодировка Д в исходном наборе символов UTF-8, а затем используем string.hex(„Д“-after-conversion), чтобы показать, как она будет выглядеть в целевом наборе символов UTF-16. Поскольку результатом будет 0414, видим, что преобразование с помощью iconv сработало. (В разных реализациях iconv могут использоваться разные имена, например UTF-16BE вместо UTF16BE.)

```
tarantool> string.hex('Д')
---
- d094
...

tarantool> converter = require('iconv').new('UTF16BE', 'UTF8')
---
...

tarantool> utf16_string = converter('Д')
---
...

tarantool> string.hex(utf16_string)
---
- '0414'
...
```

5.2.15 Модуль *json*

Общие сведения

Модуль `json` определяет процедуры работы с форматом JSON. Он создан на основе модуля Lua-CJSON от Mark Pulford. Полное руководство по Lua-CJSON включено в официальную документацию.

Указатель

Ниже приведен перечень всех функций и элементов модуля `json`.

Имя	Назначение
json.encode()	Конвертация Lua-объекта в JSON-строку
json.decode()	Конвертация JSON-строки в Lua-объект
__serialize parameter	Output structure specification
json.cfg()	Change configuration
json.NULL	Аналог «nil» в языке Lua

`json.encode`(*lua-value*[, *configuration*])
Конвертация Lua-объекта в JSON-строку.

Параметры

- `lua_value` – скалярное значение или значение из Lua-таблицы.
- `configuration` – see *json.cfg*

возвращает оригинальное значение, преобразованное в JSON-строку.

тип возвращаемого значения строка

Пример:

```
tarantool> json=require('json')
---
...
```

(continues on next page)

(продолжение с предыдущей страницы)

```
tarantool> json.encode(123)
---
- '123'
...
tarantool> json.encode({123})
---
- '[123]'
...
tarantool> json.encode({123, 234, 345})
---
- '[123,234,345]'
...
tarantool> json.encode({abc = 234, cde = 345})
---
- '{"cde":345,"abc":234}'
...
tarantool> json.encode({hello = {'world'}})
---
- '{"hello":["world"]}'
...
```

json.decode(*string*[, *configuration*])

Конвертация JSON-строки в Lua-объект.

Параметры

- string (string) – строка в формате JSON.
- configuration – see *json.cfg*

возвращает оригинальное содержание в формате Lua-таблицы.

тип возвращаемого значения таблица

Пример:

```
tarantool> json = require('json')
---
...
tarantool> json.decode('123')
---
- 123
...
tarantool> json.decode('[123, "hello"]')
---
- [123, 'hello']
...
tarantool> json.decode('{"hello": "world"}').hello
---
- world
...
```

Чтобы увидеть применение json.decode() в приложении, см. практическое задание *Подсчет суммы по JSON-полям во всех кортежах*.

__serialize parameter:

Структуру JSON-вывода можно указать с помощью __serialize:

- „seq“, „sequence“, „array“ - table encoded as an array

- „map“, „mapping“ - table encoded as a map
- function - the meta-method called to unpack serializable representation of table, cdata or userdata objects

Serializing „A“ and „B“ with different `__serialize` values brings different results:

```
tarantool> json.encode(setmetatable({'A', 'B'}, { __serialize="seq"}))
---
- '["A","B"]'
...
tarantool> json.encode(setmetatable({'A', 'B'}, { __serialize="map"}))
---
- '{"1":"A","2":"B"}'
...
tarantool> json.encode({setmetatable({f1 = 'A', f2 = 'B'}, { __serialize="map"})})
---
- '[{"f2":"B","f1":"A"}]'
...
tarantool> json.encode({setmetatable({f1 = 'A', f2 = 'B'}, { __serialize="seq"})})
---
- '[[]]'
...
```

`json.cfg`(*table*)
: Set values that affect the behavior of *json.encode* and *json.decode*.

  The values are all either integers or boolean `true`/`false`.

Option	Default	Назначение
`cfg.encode_max_depth`	128	Max recursion depth for encoding
`cfg.encode_deep_as_nil`	false	A flag saying whether to crop tables with nesting level deeper than `cfg.encode_max_depth`. Not-encoded fields are replaced with one null. If not set, too deep nesting is considered an error.
`cfg.encode_invalid_numbers`	true	A flag saying whether to enable encoding of NaN and Inf numbers
`cfg.encode_number_precision`	14	Precision of floating point numbers
`cfg.encode_load_metatables`	true	A flag saying whether the serializer will follow *__serialize* metatable field
`cfg.encode_use_tostring`	false	A flag saying whether to use `tostring()` for unknown types
`cfg.encode_invalid_as_nil`	false	A flag saying whether use NULL for non-recognized types
`cfg.encode_sparse_convert`	true	A flag saying whether to handle excessively sparse arrays as maps. See detailed description *below*.
`cfg.encode_sparse_ratio`	2	1/`encode_sparse_ratio` is the permissible percentage of missing values in a sparse array.
`cfg.encode_sparse_safe`	10	A limit ensuring that small Lua arrays are always encoded as sparse arrays (instead of generating an error or encoding as a map)
`cfg.decode_invalid_numbers`	true	A flag saying whether to enable decoding of NaN and Inf numbers
`cfg.decode_save_metatables`	true	A flag saying whether to set metatables for all arrays and maps
`cfg.decode_max_depth`	128	Max recursion depth for decoding

Sparse arrays features:

During encoding, the JSON encoder tries to classify a table into one of four kinds:

- map - at least one table index is not unsigned integer
- regular array - all array indexes are available
- sparse array - at least one array index is missing
- excessively sparse array - the number of values missing exceeds the configured ratio

An array is excessively sparse when **all** the following conditions are met:

- `encode_sparse_ratio` > 0
- `max(table)` > `encode_sparse_safe`
- `max(table)` > `count(table)` * `encode_sparse_ratio`

The JSON encoder will never consider an array to be excessively sparse when `encode_sparse_ratio` = 0. The `encode_sparse_safe` limit ensures that small Lua arrays are always encoded as sparse arrays. By default, attempting to encode an excessively sparse array will generate an error. If `encode_sparse_convert` is set to `true`, excessively sparse arrays will be handled as maps.

json.cfg() example 1:

The following code will encode 0/0 as NaN («not a number») and 1/0 as Inf («infinity»), rather than returning nil or an error message:

```
json = require('json')
json.cfg{encode_invalid_numbers = true}
x = 0/0
y = 1/0
json.encode({1, x, y, 2})
```

Результат запроса `json.encode()` будет следующим:

```
tarantool> json.encode({1, x, y, 2})
---
- '[1,nan,inf,2]
...
```

json.cfg example 2:

To avoid generating errors on attempts to encode unknown data types as userdata/cdata, you can use this code:

```
tarantool> httpc = require('http.client').new()
---
...

tarantool> json.encode(httpc.curl)
---
- error: unsupported Lua type 'userdata'
...

tarantool> json.encode(httpc.curl, {encode_use_tostring=true})
---
- '"userdata: 0x010a4ef2a0"'
...
```

Примечание: To achieve the same effect for only one call to `json.encode()` (i.e. without changing the configuration permanently), you can use `json.encode({1, x, y, 2}, {encode_invalid_numbers = true})`.

Similar configuration settings exist for *MsgPack* and *YAML*.

`json.NULL`

Значение, сопоставимое с нулевым значением «nil» в языке Lua, которое можно использовать в качестве объекта-заполнителя в кортеже.

Пример:

```
-- Когда полю Lua-таблицы присваивается nil, это поле -- null
tarantool> {nil, 'a', 'b'}
---
- - null
  - a
  - b
...
-- Когда полю Lua-таблицы присваивается json.NULL, это поле --  json.NULL
tarantool> {json.NULL, 'a', 'b'}
---
- - null
  - a
```

(continues on next page)

(продолжение с предыдущей страницы)

```
  - b
...
-- Когда JSON-полю присваивается json.NULL, это поле -- null
tarantool> json.encode({field2 = json.NULL, field1 = 'a', field3 = 'c'})
---
- '{"field2":null,"field1":"a","field3":"c"}'
...
```

5.2.16 Module *key_def*

The *key_def* module has a function for making a definition of the field numbers and types of a tuple. The definition is usually used in conjunction with an index definition to extract or compare the index key values.

`key_def.new(`*parts*`)`
Create a new key_def instance.

Параметры

- `parts` (`table`) – field numbers and types. There must be at least one part and it must have at least fieldno and type.

возвращает key_def-object *a key_def object*

The parts table has components which are the same as the `parts` option in *Options for space_object:create_index()*.

fieldno (integer) for example fieldno=1

type (string) for example type="string"

Other components are optional.

Example: `key_def.new({{type = 'unsigned', fieldno = 1}})`

`object key_def_object`
A key_def object is an object returned by *key_def.new()*. It has methods *extract_key()*, *compare()*, *compare_with_key()*, *merge()*, *totable()*.

`key_def_object:extract_key(`*tuple*`)`
Return a tuple containing only the fields of the key_def object.

Параметры

- `tuple` (`table`) – tuple or Lua table with field contents

возвращает the fields that were defined for the key_def object

Example #1:

```
-- Suppose that an item has five fields
-- 1, 99.5, 'X', nil, 99.5
-- and the fields that we care about are
-- #3 (a string) and #1 (an integer).
-- We can define those fields with k = key_def.new
-- and extract the values with k:extract_key.

tarantool> key_def = require('key_def')
---
...
```

(continues on next page)

(продолжение с предыдущей страницы)

```
tarantool> k = key_def.new({{type = 'string', fieldno = 3},
>                           {type = 'unsigned', fieldno =1 }})
---
...

tarantool> k:extract_key({1, 99.5, 'X', nil, 99.5})
---
- ['X', 1]
...
```

Example #2

```
-- Now suppose that the item is a tuple in a space which
-- has an index on field #3 plus field #1.
-- We can use key_def.new with the index definition
-- instead of filling it out as we did with Example #1.
-- The result will be the same.
key_def = require('key_def')
box.schema.space.create('T')
i = box.space.T:create_index('I',{parts={3,'string',1,'unsigned'}})
box.space.T:insert{1, 99.5, 'X', nil, 99.5}
k = key_def.new(i.parts)
k:extract_key(box.space.T:get({'X', 1}))
```

Example #3

```
-- Iterate through the tuples in a secondary non-unique index.
-- extracting the tuples' primary-key values so they can be deleted
-- using a unique index. This code should be part of a Lua function.
local key_def_lib = require('key_def')
local s = box.schema.space.create('test')
local pk = s:create_index('pk')
local sk = s:create_index('test', {unique = false, parts = {
    {2, 'number', path = 'a'}, {2, 'number', path = 'b'}}})
s:insert{1, {a = 1, b = 1}}
s:insert{2, {a = 1, b = 2}}
local key_def = key_def_lib.new(pk.parts)
for _, tuple in sk:pairs({1}) do
    local key = key_def:extract_key(tuple)
    pk:delete(key)
end
```

key_def_object:compare(*tuple_1*, *tuple_2*)

Compare the key fields of tuple_1 to the key fields of tuple_2. This is a tuple-by-tuple comparison so users do not have to write code which compares a field at a time. Each field's type and collation wll be taken into account. In effect it is a comparison of extract_key(tuple_1) with extract_key(tuple_2).

Параметры

- tuple1 (table) – tuple or Lua table with field contents
- tuple2 (table) – tuple or Lua table with field contents

возвращает > 0 if tuple_1 key fields > tuple_2 key fields, = 0 if tuple_1 key fields = tuple_2 key fields, < 0 if tuple_1 key fields < tuple_2 key fields

Пример:

```
-- This will return 0
key_def = require('key_def')
k = key_def.new({{type='string',fieldno=3,collation='unicode_ci'},
                 {type='unsigned',fieldno=1}})
k:compare({1, 99.5, 'X', nil, 99.5}, {1, 99.5, 'x', nil, 99.5})
```

`key_def_object:compare_with_key(`*tuple_1*, *tuple_2*`)`

Compare the key fields of tuple_1 to all the fields of tuple_2. This is the same as *key_def_object:compare()* except that tuple_2 contains only the key fields. In effect it is a comparison of extract_key(tuple_1) with tuple_2.

Параметры

- `tuple1` (`table`) – tuple or Lua table with field contents
- `tuple2` (`table`) – tuple or Lua table with field contents

возвращает > 0 if tuple_1 key fields > tuple_2 fields, = 0 if tuple_1 key fields = tuple_2 fields, < 0 if tuple_1 key fields < tuple_2 fields

Пример:

```
-- This will return 0
key_def = require('key_def')
k = key_def.new({{type='string',fieldno=3,collation='unicode_ci'},
                 {type='unsigned',fieldno=1}})
k:compare_with_key({1, 99.5, 'X', nil, 99.5}, {'x', 1})
```

`key_def_object:merge(`*other_key_def_object*`)`

Combine the main key_def_object with other_key_def_object. The return value is a new key_def_object containing all the fields of the main key_def_object, then all the fields of other_key_def_object which are not in the main key_def_object.

Параметры

- `other_key_def_object` (*key_def_object*) – definition of fields to add

возвращает key_def_object

Пример:

```
-- This will return a key definition with fieldno=3 and fieldno=1.
key_def = require('key_def')
k = key_def.new({{type = 'string', fieldno = 3}})
k2= key_def.new({{type = 'unsigned', fieldno = 1},
                 {type = 'string', fieldno = 3}})
k:merge(k2)
```

`key_def_object:totable()`

Return a table containing what is in the key_def_object. This is the reverse of `key_def.new()`:

- `key_def.new()` takes a table and returns a key_def object,
- `key_def_object:totable()` takes a key_def object and returns a table.

This is useful for input to `_serialize` methods.

возвращает таблица

Пример:

```
-- This will return a table with type='string', fieldno=3
key_def = require('key_def')
k = key_def.new({{type = 'string', fieldno = 3}})
k:totable()
```

5.2.17 Модуль *log*

Общие сведения

Сервер Tarantool'а сохраняет все сообщения об ошибке в файл журнала, указанный в конфигурационном параметре *log*. Сообщения об ошибке могут быть созданы либо системой с помощью внутреннего кода сервера, либо пользователем с помощью функции `log.log_level_function_name`.

Как сказано в описании параметра *log_format*, записи в журнале могут сохраняться в одном из двух форматов:

- „plain“ (по умолчанию) или
- „json“ (более детально с JSON-метками).

Вот как будет выглядеть запись в журнале после выполнения `box.cfg{log_format='plain'}`:

```
2017-10-16 11:36:01.508 [18081] main/101/interactive I> set 'log_format' configuration option to
↪"plain"
```

Вот как будет выглядеть запись в журнале после выполнения `box.cfg{log_format='json'}`:

```
{"time": "2017-10-16T11:36:17.996-0600",
"level": "INFO",
"message": "set 'log_format' configuration option to \"json\"",
"pid": 18081,
"cord_name": "main",
"fiber_id": 101,
"fiber_name": "interactive",
"file": "builtin\/box\/load_cfg.lua",
"line": 317}
```

Указатель

Ниже приведен перечень всех функций и элементов модуля `log`.

Имя	Назначение
log.error() log.warn() log.info() log.verbose() log.debug()	Запись сгенерированного пользователем сообщения в файл журнала
log.logger_pid()	Получение PID регистратора записи в журнале
log.rotate()	Ротация файла журнала

`log.error(`*message*`)`
`log.warn(`*message*`)`
`log.info(`*message*`)`
`log.verbose(`*message*`)`
`log.debug(`*message*`)`
Запись созданного пользователем сообщения в *файл журнала* при условии, что log_level_function_name = `error` или `warn`, или `info`, или `verbose`, или `debug`.

Как поясняется в описании настроек конфигурации *log_level*, есть семь уровней детализации:

- 1 – SYSERROR
- 2 – ERROR – соответствует log.error(...)
- 3 – CRITICAL
- 4 – WARNING – соответствует log.warn(...)
- 5 – INFO – соответствует log.info(...)
- 6 – VERBOSE – соответствует log.verbose(...)
- 7 – DEBUG – соответствует log.debug(...)

Например, если уровень box.cfg.log_level в данный момент – 5 (по умолчанию), то сообщения log.error(...), log.warn(...) и log.info(...) будут записываться в файл журнала. Однако, сообщения log.verbose(...) и log.debug(...) не будут записываться в файл журнала, поскольку они соответствуют более высоким уровням детализации.

Параметры

- message (*any*) – Как правило, строка. Сообщения могут содержать спецификаторы формата в стиле C: %d или %s, то есть log.error('...%d...%s', *x*, *y*) сработает, если x – это число, а y – это строка. В редких случаях сообщения могут представлять собой другие скалярные типы данных или даже таблицы. Поэтому log.error({'x',18.7,true}) сработает.

возвращает nil

Выходное значение будет представлять собой строку в журнале, которая содержит следующее:

- текущая временная отметка,
- название модуля,
- , „W“, „I“, „V“ или „D“ в зависимости от log_level_function_name и
- сообщение.

Вывода не будет, если log_level_function_name соответствует типу больше, чем *log_level*.

log.logger_pid()

возвращает PID регистратора записи в журнале

log.rotate()

Ротация журнала.

возвращает nil

Пример

```
$ tarantool
tarantool> box.cfg{log_level=3, log='tarantool.txt'}
tarantool> log = require('log')
tarantool> log.error('Error')
tarantool> log.info('Info %s', box.info.version)
tarantool> os.exit()
```

```
$ less tarantool.txt
2017-09-20 ... [68617] main/101/interactive C> version 1.7.5-31-ge939c6ea6
2017-09-20 ... [68617] main/101/interactive C> log level 3
2017-09-20 ... [68617] main/101/interactive [C]:-1 E> Error
```

Строке „Error“ в файле `tarantool.txt` предшествует буква «E».

Строка „Info“ отсутствует, потому что `log_level` – 3.

5.2.18 Module *merger*

Общие сведения

The `merger` module takes a stream of tuples and provides access to them as tables.

Указатель

The four functions for creating a merger object instance are:

- *merger.new_tuple_source()*,
- *merger.new_buffer_source()*,
- *merger.new_table_source*,
- *merger.new(merger_source...)*.

The methods for using a merger object are:

- *merger_object:select()*,
- *merger_object:pairs()*.

`merger.new_tuple_source`(*gen*, *param*, *state*)

Create a new merger instance from a tuple source.

A tuple source just returns one tuple.

The generator function `gen()` allows creation of multiple tuples via an iterator.

The `gen()` function should return:

- state, tuple each time it is called and a new tuple is available,
- nil when no more tuples are available.

Параметры

- `gen` – function for iteratively returning tuples
- `param` – parameter for the gen function

возвращает merger-object *a merger object*

Example: see *merger_object:pairs()* method.

`merger.new_buffer_source`(*gen*, *param*, *state*)

Create a new merger instance from a buffer source.

Parameters and return: same as for *merger.new_tuple_source*.

To set up a buffer, or a series of buffers, use *the buffer module*.

merger.new_table_source(*gen*, *param*, *state*)
Create a new merger instance from a table source.

Parameters and return: same as for *merger.new_tuple_source*.

Example: see *merger_object:select()* method.

merger.new(*key_def*, *sources*, *options*)
Create a new merger instance from a merger source.

A merger source is created from a *key_def* object and a set of (tuple or buffer or table or merger) sources. It performs a kind of merge sort. It chooses a source with a minimal / maximal tuple on each step, consumes a tuple from this source, and repeats.

Параметры

- `key_def` – object created with `key_def`
- `source` – parameter for the `gen()` function
- `options` – `reverse=true` if descending, false or nil if ascending

возвращает merger-object *a merger object*

A `key_def` can be cached across requests with the same ordering rules (typically these would be requests accessing the same space).

Example: see *merger_object:pairs()* method.

object merger_object
A merger object is an object returned by:

- *merger.new_tuple_source()* or
- *merger.new_buffer_source()* or
- *merger.new_table_source* or
- *merger.new(merger_source...)*.

It has methods:

- *merger_object:select()* or
- *merger_object:pairs()*.

merger_object:select(⌊*buffer*⌊, *limit*⌋⌋)
Access the contents of a merger object with familiar `select` syntax.

Параметры

- `buffer` – as in `net.box` client *conn:select* method
- `limit` – as in `net.box` client *conn:select* method

возвращает a table of tuples, similar to what `select` would return

Example with new_table_source():

```
-- Source via new_table_source, simple generator function
-- tarantool> s:select()
-- ---
-- - - [100]
--   - [200]
-- ...
merger=require('merger')
```

(continues on next page)

(продолжение с предыдущей страницы)

```
k=0
function merger_function(param)
  k = k + 1
  if param[k] == nil then return nil end
  return box.NULL, param[k]
  end
chunks={}
chunks[1] = {{100}} chunks[2] = {{200}} chunks[3] = nil
s = merger.new_table_source(merger_function, chunks)
s:select()
```

merger_object:pairs()

The `pairs()` method (or the equivalent `ipairs()` `alias` method) returns a luafun iterator. It is a Lua iterator, but also provides a set of handy methods to operate in functional style.

Параметры

- `tuple` (`table`) – tuple or Lua table with field contents

возвращает the tuples that can be found with a standard `pairs()` function

Example with new_tuple_source():

```
-- Source via new_tuple_source, from a space of tables
-- The result will look like this:
-- tarantool> so:pairs():totable()
-- ---
-- - - [100]
--   - [200]
-- ...
merger = require('merger')
box.schema.space.create('s')
box.space.s:create_index('i')
box.space.s:insert({100})
box.space.s:insert({200})
so = merger.new_tuple_source(box.space.s:pairs())
so:pairs():totable()
```

Example with two mergers:

```
-- Source via key_def, and table data

-- Create the key_def object
merger = require('merger')
key_def_lib = require('key_def')
key_def = key_def_lib.new({{
    fieldno = 1,
    type = 'string',
}})
-- Create the table source
data = {{'a'}, {'b'}, {'c'}}
source = merger.new_source_fromtable(data)
i1 = merger.new(key_def, {source}):pairs()
i2 = merger.new(key_def, {source}):pairs()
-- t1 will be 'a' (tuple 1 from merger 1)
t1 = i1:head():totable()
-- t3 will be 'c' (tuple 3 from merger 2)
t3 = i2:head():totable()
```

(continues on next page)

(продолжение с предыдущей страницы)

```
-- t2 will be 'b' (tuple 2 from merger 1)
t2 = i1:head():totable()
-- i1:is_null() will be true (merger 1 ends)
i1:is_null()
-- i2:is_null() will be true (merger 2 ends)
i2:is_null()
```

More examples:

See https://github.com/Totktonada/tarantool-merger-examples which, in addition to discussing the merger API in detail, shows Lua code for handling many more situations than are in this manual's brief examples.

5.2.19 Модуль *msgpack*

Общие сведения

The `msgpack` module decodes *raw MsgPack strings* by converting them to Lua objects, and encodes Lua objects by converting them to raw MsgPack strings. Tarantool makes heavy internal use of MsgPack because tuples in Tarantool are *stored* as MsgPack arrays.

Definitions: MsgPack is short for MessagePack. A «raw MsgPack string» is a byte array formatted according to the MsgPack specification including type bytes and sizes. The type bytes and sizes can be made displayable with *string.hex()*, or the raw MsgPack strings can be converted to Lua objects with `msgpack` methods.

Указатель

Ниже приведен перечень всех функций и элементов модуля `msgpack`.

Имя	Назначение
msgpack.encode(lua_value)	Convert a Lua object to a raw MsgPack string
msgpack.encode(lua_value,ibuf)	Convert a Lua object to a raw MsgPack string in an ibuf
msgpack.decode(msgpack_string)	Convert a raw MsgPack string to a Lua object
msgpack.decode(C_style_string_pointer)	Convert a raw MsgPack string in an ibuf to a Lua object
msgpack.decode_unchecked(mspack_string)	Convert a raw MsgPack string to a Lua object
msgpack.decode_unchecked(C_style_string_pointer)	Convert a raw MsgPack string to a Lua object
msgpack.decode_array_header	Skip array header in a raw MsgPack string
msgpack.decode_map_header	Skip map header in a raw MsgPack string
__serialize parameter	Output structure specification
msgpack.cfg	Изменение конфигурации
msgpack.NULL	Аналог «nil» в языке Lua

`msgpack.encode(`*lua_value*`)`

Convert a Lua object to a raw MsgPack string.

Параметры

- `lua_value` – скалярное значение или значение из Lua-таблицы.

возвращает the original contents formatted as a raw MsgPack string;

тип возвращаемого значения raw MsgPack string

msgpack.encode(*lua_value*, *ibuf*)

Convert a Lua object to a raw MsgPack string in an ibuf, which is a buffer such as *buffer.ibuf()* creates. As with *encode(lua_value)*, the result is a raw MsgPack string, but it goes to the `ibuf` output instead of being returned.

Параметры

- `lua_value` (*lua-object*) – скалярное значение или значение из Lua-таблицы.
- `ibuf` (buffer) – (output parameter) where the result raw MsgPack string goes

возвращает number of bytes in the output

тип возвращаемого значения raw MsgPack string

Example using *buffer.ibuf()* and ffi.string() and *string.hex()*: The result will be „91a161“ because 91 is the MessagePack encoding of «fixarray size 1», a1 is the MessagePack encoding of «fixstr size 1», and 61 is the UTF-8 encoding of „a“:

```
ibuf = require('buffer').ibuf()
msgpack_string_size = require('msgpack').encode({'a'}, ibuf)
msgpack_string = require('ffi').string(ibuf.rpos, msgpack_string_size)
string.hex(msgpack_string)
```

msgpack.decode(*msgpack_string*[, *start_position*])

Convert a raw MsgPack string to a Lua object.

Параметры

- `msgpack_string` (string) – a raw MsgPack string.
- `start_position` (*integer*) – откуда начинать, минимальное значение = 1, максимальное = длина строки, по умолчанию = 1.

возвращает

- (if `msgpack_string` is a valid raw MsgPack string) the original contents of `msgpack_string`, formatted as a Lua object, usually a Lua table, (otherwise) a scalar value, such as a string or a number;
- «next_start_position». Если расшифровка `decode` останавливается после разбора байта N в `msgpack_string`, то «next_start_position» = N + 1, а `decode(msgpack_string, next_start_position)` продолжит разбор с места остановки предыдущего `decode` плюс 1. Как правило, `decode` разбирает всю строку `msgpack_string`, поэтому «next_start_position» будет равняться `string.len(msgpack_string)` + 1.

тип возвращаемого значения Lua object and number

Example: The result will be [„a“] and 4:

```
msgpack_string = require('msgpack').encode({'a'})
require('msgpack').decode(msgpack_string, 1)
```

msgpack.decode(*C_style_string_pointer*, *size*)

Convert a raw MsgPack string, whose address is supplied as a C-style string pointer such as the `rpos` pointer which is inside an ibuf such as *buffer.ibuf()* creates, to a Lua object. A C-style string pointer may be described as `cdata<char *>` or `cdata<const char *>`.

Параметры

- `C_style_string_pointer` (buffer) – a pointer to a raw MsgPack string.

- `size` (*integer*) – number of bytes in the raw MsgPack string

возвращает

- (if C_style_string_pointer points to a valid raw MsgPack string) the original contents of `msgpack_string`, formatted as a Lua object, usually a Lua table, (otherwise) a scalar value, such as a string or a number;
- returned_pointer = a C-style pointer to the byte after what was passed, so that C_style_string_pointer + size = returned_pointer

тип возвращаемого значения table and C-style pointer to after what was passed

Example using *buffer.ibuf* and pointer arithmetic: The result will be [„a“] and 3 and true:

```
ibuf = require('buffer').ibuf()
msgpack_string_size = require('msgpack').encode({'a'}, ibuf)
a, b = require('msgpack').decode(ibuf.rpos, msgpack_string_size)
a, b - ibuf.rpos, msgpack_string_size == b - ibuf.rpos
```

`msgpack.decode_unchecked(`*msgpack_string*[, *start_position*]`)`
Input and output are the same as for *decode(string)*.

`msgpack.decode_unchecked(`*C_style_string_pointer*`)`
Input and output are the same as for *decode(C_style_string_pointer)*, except that `size` is not needed. Some checking is skipped, and `decode_unchecked(C_style_string_pointer)` can operate with string pointers to buffers which `decode(C_style_string_pointer)` cannot handle. For an example see the *buffer* module.

`msgpack.decode_array_header(`*byte-array*, *size*`)`
Call the mp_decode_array function in the MsgPuck library and return the array size and a pointer to the first array component. A subsequent call to `msgpack_decode` can decode the component instead of the whole array.

Параметры

- `byte-array` – a pointer to a raw MsgPack string.
- `size` – a number greater than or equal to the string's length

возвращает

- the size of the array;
- a pointer to after the array header.

```
-- Example of decode_array_header
-- Suppose we have the raw data '\x93\x01\x02\x03'.
-- \x93 is MsgPack encoding for a header of a three-item array.
-- We want to skip it and decode the next three items.
msgpack=require('msgpack'); ffi=require('ffi')
x,y=msgpack.decode_array_header(ffi.cast('char*','\x93\x01\x02\x03'),4)
a=msgpack.decode(y,1);b=msgpack.decode(y+1,1);c=msgpack.decode(y+2,1);
a,b,c
-- The result will be: 1,2,3.
```

`msgpack.decode_map_header(`*byte-array*, *size*`)`
Call the mp_decode_map function in the MsgPuck library and return the map size and a pointer to the first map component. A subsequent call to `msgpack_decode` can decode the component instead of the whole map.

Параметры

- `byte-array` – a pointer to a raw MsgPack string.
- `size` – a number greater than or equal to the raw MsgPack string's length

возвращает

- the size of the map;
- a pointer to after the map header.

```
-- Example of decode_map_header
-- Suppose we have the raw data '\x81\xa2\x41\x41\xc3'.
-- \x81 is MsgPack encoding for a header of a one-item map.
-- We want to skip it and decode the next map item.
msgpack=require('msgpack'); ffi=require('ffi')
x,y=msgpack.decode_map_header(ffi.cast('char*','\x81\xa2\x41\x41\xc3'),5)
a=msgpack.decode(y,3);b=msgpack.decode(y+3,1)
x,a,b
-- The result will be: 1,"AA", true.
```

__serialize parameter:

Структуру MsgPack-вывода можно указать с помощью `__serialize`:

- „seq“, „sequence“, „array“ - table encoded as an array
- „map“, „mappping“ - table encoded as a map
- function - the meta-method called to unpack serializable representation of table, cdata or userdata objects

Serializing „A“ and „B“ with different `__serialize` values brings different results. To show this, here is a routine which encodes `{'A','B'}` both as an array and as a map, then displays each result in hexadecimal.

```
function hexdump(bytes)
    local result = ''
    for i = 1, #bytes do
        result = result .. string.format("%x", string.byte(bytes, i)) .. ' '
    end
    return result
end

msgpack = require('msgpack')
m1 = msgpack.encode(setmetatable({'A', 'B'}, {
                              __serialize = "seq"
                            }))
m2 = msgpack.encode(setmetatable({'A', 'B'}, {
                              __serialize = "map"
                            }))
print('array encoding: ', hexdump(m1))
print('map encoding: ', hexdump(m2))
```

Результат:

```
array encoding: 92 a1 41 a1 42
map encoding:   82 01 a1 41 02 a1 42
```

The MsgPack Specification page explains that the first encoding means:

```
fixarray(2), fixstr(1), "A", fixstr(1), "B"
```

а значение второго результата кодирования:

```
fixmap(2), key(1), fixstr(1), "A", key(2), fixstr(2), "B"
```

Ниже приведены примеры всех стандартных типов: слева отображение в Lua-таблице, а справа – имя и кодировка в формате MsgPack.

Стандартные типы в MsgPack-кодировке

{}	„fixmap“ = 80, если метатаблица – ассоциативный массив „map“, в противном случае, „fixarray“ = 90
„a“	„fixstr“ = a1 61
false	„false“ = c2
true	„true“ = c3
127	„positive fixint“ = 7f
65535	„uint 16“ = cd ff ff
4294967295	„uint 32“ = ce ff ff ff ff
nil	„nil“ = c0
msgpack.NULL	то же, что и nil
[0] = 5	„fixmap(1)“ + „positive fixint“ (для ключа) + „positive fixint“ (для значения) = 81 00 05
[0] = nil	„fixmap(0)“ = 80 – nil не хранится, если это отсутствующее значение ассоциативного массива
1,5	„float 64“ = cb 3f f8 00 00 00 00 00 00

`msgpack.cfg`(*table*)

Some MsgPack configuration settings can be changed.

The values are all either integers or boolean `true`/`false`.

Option	Default	Назначение
`cfg.encode_max_depth`	128	Max recursion depth for encoding
`cfg.encode_deep_as_nil`	false	A flag saying whether to crop tables with nesting level deeper than `cfg.encode_max_depth`. Not-encoded fields are replaced with one null. If not set, too high nesting is considered an error.
`cfg.encode_invalid_numbers`	true	A flag saying whether to enable encoding of NaN and Inf numbers
`cfg.encode_load_metatables`	true	A flag saying whether the serializer will follow *__serialize* metatable field
`cfg.encode_use_tostring`	false	A flag saying whether to use `tostring()` for unknown types
`cfg.encode_invalid_as_nil`	false	A flag saying whether to use NULL for non-recognized types
`cfg.encode_sparse_convert`	true	A flag saying whether to handle excessively sparse arrays as maps. See detailed description *below*
`cfg.encode_sparse_ratio`	2	1/`encode_sparse_ratio` is the permissible percentage of missing values in a sparse array
`cfg.encode_sparse_safe`	10	A limit ensuring that small Lua arrays are always encoded as sparse arrays (instead of generating an error or encoding as a map)
`cfg.decode_invalid_numbers`	true	A flag saying whether to enable decoding of NaN and Inf numbers
`cfg.decode_save_metatables`	true	A flag saying whether to set metatables for all arrays and maps

Sparse arrays features:

During encoding, the MsgPack encoder tries to classify tables into one of four kinds:

- map - at least one table index is not unsigned integer
- regular array - all array indexes are available
- sparse array - at least one array index is missing
- excessively sparse array - the number of values missing exceeds the configured ratio

An array is excessively sparse when **all** the following conditions are met:

- `encode_sparse_ratio` > 0
- `max(table)` > `encode_sparse_safe`
- `max(table)` > `count(table)` * `encode_sparse_ratio`

MsgPack encoder will never consider an array to be excessively sparse when `encode_sparse_ratio = 0`. The `encode_sparse_safe` limit ensures that small Lua arrays are always encoded as sparse arrays. By default, attempting to encode an excessively sparse array will generate an error. If `encode_sparse_convert` is set to `true`, excessively sparse arrays will be handled as maps.

msgpack.cfg() example 1:

If `msgpack.cfg.encode_invalid_numbers = true` (the default), then NaN and Inf are legal values. If that is not desirable, then ensure that `msgpack.encode()` will not accept them, by saying `msgpack.cfg{encode_invalid_numbers = false}`, thus:

```
tarantool> msgpack = require('msgpack'); msgpack.cfg{encode_invalid_numbers = true}
---
...
tarantool> msgpack.decode(msgpack.encode{1, 0 / 0, 1 / 0, false})
---
- [1, -nan, inf, false]
- 22
...
tarantool> msgpack.cfg{encode_invalid_numbers = false}
---
...
tarantool> msgpack.decode(msgpack.encode{1, 0 / 0, 1 / 0, false})
---
- error: ... number must not be NaN or Inf'
...
```

msgpack.cfg example 2:

To avoid generating errors on attempts to encode unknown data types as userdata/cdata, you can use this code:

```
tarantool> httpc = require('http.client').new()
---
...

tarantool> msgpack.encode(httpc.curl)
---
- error: unsupported Lua type 'userdata'
...

tarantool> msgpack.encode(httpc.curl, {encode_use_tostring=true})
```

(continues on next page)

(продолжение с предыдущей страницы)

```
---
- '"userdata: 0x010a4ef2a0"'
...
```

Примечание: To achieve the same effect for only one call to `msgpack.encode()` (i.e. without changing the configuration permanently), you can use `msgpack.encode({1, x, y, 2}, {encode_invalid_numbers = true})`.

Similar configuration settings exist for *JSON* and *YAML*.

`msgpack.NULL`
: Значение, сопоставимое с нулевым значением «nil» в языке Lua, которое можно использовать в качестве объекта-заполнителя в кортеже.

Example

```
tarantool> msgpack = require('msgpack')
---
...
tarantool> y = msgpack.encode({'a',1,'b',2})
---
...
tarantool> z = msgpack.decode(y)
---
...
tarantool> z[1], z[2], z[3], z[4]
---
- a
- 1
- b
- 2
...
tarantool> box.space.tester:insert{20, msgpack.NULL, 20}
---
- [20, null, 20]
...
```

5.2.20 Модуль *net.box*

Общие сведения

Модуль `net.box` включает в себя коннекторы для удаленных систем с базами данных. Одним из вариантов, который рассматривается позднее, является подключение к MySQL, MariaDB или PostgreSQL (см. справочник по *Модулям СУБД SQL*). Другим вариантом, который рассматривается в данном разделе, является подключение к экземплярам Tarantool-сервера по сети.

Можно вызвать следующие методы:

- `require('net.box')` для получения объекта `net.box` (который называется `net_box` для примеров в данном разделе),
- `net_box.connect()` для подключения и получения объекта подключения (который называется `conn` для примеров в данном разделе),

- другие процедуры `net.box()`, передающие `conn:` для выполнения запросов в удаленной системе базы данных,
- `conn:close` для отключения.

Все методы `net.box` безопасны для файберов, то есть можно безопасно обмениваться и использовать один и тот же объект подключения в нескольких файберах одновременно. Фактически так лучше всего работать в Tarantool'е. Когда несколько файберов используют одно соединение, все запросы передаются по одному сетевому сокету, но каждый файбер получает правильный ответ. Уменьшение количества активных сокетов снижает затрату ресурсов на системные вызовы и увеличивает общую производительность сервера. Однако, в некоторых случаях отдельного соединения недостаточно – например, когда необходимо отдавать приоритет разным запросам или использовать различные идентификаторы при аутентификации.

В большинстве методов `net.box` можно использовать заключительный аргумент `{options}`, который может быть:

- `{timeout=...}`. Например, метод с заключительным аргументом `{timeout=1.5}` остановится через 1,5 секунды на локальном узле, хотя это не гарантирует, что выполнение остановится на удаленном сервере.
- `{buffer=...}`. Например, см. *модуль buffer*.
- `{is_async=...}`. Например, метод с заключительным аргументом `{is_async=true}` не будет ждать результата выполнения запроса. См. описание *is_async*.
- `{on_push=... on_push_ctx=...}`. Для получения внеполосных сообщений. См. описание *box.session.push*.

На диаграмме ниже представлены возможные состояния и варианты перехода из одного состояния в другое:

На этой диаграмме:

- Работа начинается с начального состояния „initial“.
- Выполнение метода `net_box.connect()` переводит состояние в „connecting“, создается рабочий файбер.
- Если требуются аутентификация и загрузка схемы, можно позднее повторно войти в состояние загрузки схемы „fetch_schema“ из активного „active“, если запрос не будет выполнен из-за ошибки несовпадения версий схемы, то есть будет вызвана перезагрузка схемы.
- Метод `conn.close()` изменяет состояние на закрытое „closed“ и отключает рабочий процесс. Если транспорт уже находится в состоянии ошибки „error“, `close()` не делает ничего.

Указатель

Ниже приведен перечень всех функций модуля `net.box`.

Имя	Назначение
net_box.connect() net_box.new() net_box.self	Создание подключения
conn:ping()	Выполнение команды проверки состояния PING
conn:wait_connected()	Ожидание активности или закрытия подключения
conn:is_connected()	Проверка активности или закрытия подключения
conn:wait_state()	Ожидание нужного состояния
conn:close()	Закрытие подключения
conn.space.space-name:select{field-value}	Выбор одного или нескольких кортежей
conn.space.space-name:get{field-value}	Выбор кортежа
conn.space.space-name:insert{field-value}	Вставка кортежа
conn.space.space-name:replace{field-value}	Вставка или замена кортежа
conn.space.space-name:update{field-value}	Обновление кортежа
conn.space.space-name:upsert{field-value}	Обновление кортежа
conn.space.space-name:delete{field-value}	Удаление кортежа
conn:eval()	Оценка и выполнение выражения в строке
conn:call()	Вызов хранимой процедуры
conn:timeout()	Установка времени ожидания
conn:on_connect()	Определение триггера на подключение
conn:on_disconnect()	Определение триггера на отключение
conn:on_schema_reload()	Определение триггера при изменении схемы

`net_box.connect(`*URI*`[, {option[s]}])`

`net_box.new(`*URI*`[, {option[s]}])`

Примечание: Имена `connect()` и `new()` являются синонимами: предпочтительным будет `connect()`, а `new()` обеспечивает поддержку обратной совместимости.

Создание нового подключения. Подключение устанавливается по требованию во время первого запроса. Можно повторно установить подключение автоматически после отключения (см. ниже опцию `reconnect_after`). Возвращается объект `conn`, который поддерживает методы создание удаленных запросов, таких как select, update или delete.

Возможные опции:

- *user/password*: есть два способа подключения к удаленному хосту: с использованием *URI* или параметров *user* (пользователь) и *password* (пароль). Например, вместо `connect('username:userpassword@localhost:33301')` можно ввести `connect('localhost:33301', {user = 'имя-пользователя', password='пароль-пользователя'})`.
- *wait_connected*: по умолчанию, создание подключения блокируется до тех пор, пока подключение не будет установлено, но передача `wait_connected=false` заставит метод сразу же вернуться. Передача времени ожидания заставит метод ждать до возвращения (например, `wait_connected=1.5` заставит ожидать подключения максимум 1,5 секунды).

  Примечание: Если присутствует `reconnect_after`, `wait_connected` проигнорирует неустойчивые отказы. Ожидание заканчивается, когда подключение установлено или явным образом закрыто.

- *reconnect_after*: `net.box` автоматически подключается повторно в случае разрыва соединения или провала попытки подключения. В таком случае неустойчивые сетевые отказы становятся очевидными. Повторное подключение выполняется автоматически в фоновом режиме, поэтому запросы/обращения, не выполненные по причине потери соединения, явным образом выполняются повторно. Количество повторов не ограничено, попытки подключения выполняются в течение указанного времени ожидания (например, `reconnect_after=5` – 5 секунд). После явного закрытия подключения или удаления сборщиком мусора в Lua попытки соединения повторно не выполняются.
- *call_16*: [с 1.7.2] по умолчанию, подключения `net.box` соответствуют команде CALL нового бинарного протокола, который не поддерживает обратную совместимость с предыдущими версиями. Команда нового бинарного протокола для вызова CALL больше не ограничивает функцию в возврате массива кортежей и позволяет возвращать произвольный результат в формате MsgPack/JSON, включая scalar (скалярные значения), nil (нулевые значения) и void (пусто). Старый метод CALL оставлен нетронутым для обратной совместимости. В следующей основной версии он будет удален. Все драйверы для языков программирования будут постепенно переведены на использование нового метода CALL. Для подключения к экземпляру Tarantool'а, в котором используется старый метод CALL, укажите `call_16=true`.
- *console*: в зависимости от значения параметра поддерживаются различные методы (как если бы возвращались экземпляры разных классов). Если `console = true`, можно использовать методы `conn`: `close()`, `is_connected()`, `wait_state()`, `eval()` (в этом случае поддерживаются и бинарный сетевой протокол, и протокол Lua-консоли). Если `console = false` (по умолчанию), также можно использовать методы `conn` для работы с базой данных (в этом случае поддерживается только бинарный протокол). Устарел: `console = true` объявлен устаревшим, вместо него следует использовать *console.connect()*.
- *connect_timeout*: количество секунд ожидания до возврата ошибки «error: Connection timed out».

  Параметры

  - `URI` (`string`) – *URI* объекта подключения
  - `options` – возможные опции: *user*, *password*, *wait_connected*, *reconnect_after*, *call_16*, *console* и *connect_timeout*

  возвращает объект подключения

  тип возвращаемого значения пользовательские данные

Примеры:

```
conn = net_box.connect('localhost:3301')
conn = net_box.connect('127.0.0.1:3302', {wait_connected = false})
conn = net_box.connect('127.0.0.1:3303', {reconnect_after = 5, call_16 = true})
```

`object self`

Для локального Tarantool-сервера есть заданный объект всегда установленного подключения под названием *net_box*`.self`. Он создан с целью облегчить полиморфное использование API модуля `net_box`. Таким образом, `conn = net_box.connect('localhost:3301')` можно заменить на `conn = net_box.self`.

Однако, есть важно отличие встроенного подключения от удаленного:

- При встроенном подключении запросы без изменения данных не передают управление. При использовании удаленного подключения любой запрос может передавать управление исходя из *правил неявной передачи управления*, и состояние базы данных может измениться к тому времени, как управление вернется.

- Не учитывается ни один параметр, передаваемый в запросе (`is_async`, `on_push`, `timeout`).

`object conn`

`conn:ping(`*[options]*`)`
Выполнение команды проверки состояния PING.

Параметры

- `options` (`table`) – поддерживается опция `timeout=`*`секунды`*

возвращает true (правда), если выполнено, false (ложь) в случае ошибки

тип возвращаемого значения boolean (логический)

Пример:

```
net_box.self:ping({timeout = 0.5})
```

`conn:wait_connected(`*[timeout]*`)`
Ожидание активности или закрытия подключения.

Параметры

- `timeout` (*`number`*) – в секундах

возвращает true (правда) при подключении, false (ложь), если не выполнено.

тип возвращаемого значения boolean (логический)

Пример:

```
net_box.self:wait_connected()
```

`conn:is_connected()`
Проверка активности или закрытия подключения.

возвращает true (правда) при подключении, false (ложь), если не выполнено.

тип возвращаемого значения boolean (логический)

Пример:

```
net_box.self:is_connected()
```

`conn:wait_state(`*state[s][, timeout]*`)`
[с 1.7.2] Ожидание нужного состояния.

Параметры

- `states` (`string`) – необходимое состояние
- `timeout` (*`number`*) – в секундах

возвращает true (правда) при подключении, false (ложь) при окончании времени ожидания или закрытии подключения

тип возвращаемого значения boolean (логический)

Примеры:

```
-- бесконечное ожидание состояния 'active':
conn:wait_state('active')

-- ожидание в течение максимум 1,5 секунд:
conn:wait_state('active', 1.5)

-- бесконечное ожидание состояния `active` или `fetch_schema`:
conn:wait_state({active=true, fetch_schema=true})
```

`conn:close()`
Закрытие подключения.

Объекты подключения удаляются сборщиком мусора в Lua, как и любой другой Lua-объект, поэтому удалять их явным образом необязательно. Однако, поскольку close() представляет собой системный вызов, лучше всего закрыть соединение явным образом, когда оно больше не используется, с целью ускорения работы сборщика мусора.

Пример:

```
conn:close()
```

`conn.space.<space-name>:select({field-value, ...} [, {options}])`
`conn.space.`*`имя-спейса`*`:select({...})` – это удаленный вызов, аналогичный локальному вызову `box.space.`*`имя-спейса`*`:select{...}` (*детали*). В дополнение см. *Модуль buffer и skip-header*.

Пример:

```
conn.space.testspace:select({1,'B'}, {timeout=1})
```

Примечание: Исходя из *правил неявной передачи управления*, локальный запрос `box.space.`*`имя-спейса`*`:select{...}` не передает управление, а удаленный `conn.space.`*`имя-спейса`*`:select{...}` передаст, поэтому глобальные переменные или кортежи в базе данных могут измениться во время удаленного `conn.space.`*`имя-спейса`*`:select{...}`.

`conn.space.<space-name>:get({field-value, ...} [, {options}])`
`conn.space.`*`имя-спейса`*`:get(...)` – это удаленный вызов, аналогичный локальному вызову `box.space.`*`имя-спейса`*`:get(...)` (*детали*).

Пример:

```
conn.space.testspace:get({1})
```

`conn.space.<space-name>:insert({field-value, ...} [, {options}])`
`conn.space.`*`имя-спейса`*`:insert(...)` – это удаленный вызов, аналогичный локальному вызову `box.space.`*`имя-спейса`*`:insert(...)` (*детали*). В дополнение см. *Модуль buffer и skip-header*.

Пример:

```
conn.space.testspace:insert({2,3,4,5}, {timeout=1.1})
```

`conn.space.<space-name>:replace({field-value, ...} [, {options}])`
`conn.space.`*`имя-спейса`*`:replace(...)` – это удаленный вызов, аналогичный локальному вызову `box.space.`*`имя-спейса`*`:replace(...)` (*детали*). В дополнение см. *Модуль buffer и skip-header*.

Пример:

```
conn.space.testspace:replace({5,6,7,8})
```

`conn.space.<space-name>:update({field-value, ...} [, {options}])`
`conn.space.`*`имя-спейса`*`:update(...)` – это удаленный вызов, аналогичный локальному вызову `box.space.`*`имя-спейса`*`:update(...)`.

Пример:

```
conn.space.Q:update({1},{{'=',2,5}}, {timeout=0})
```

`conn.space.<space-name>:upsert({field-value, ...} [, {options}])`
`conn.space.`*`имя-спейса`*`:upsert(...)` – это удаленный вызов, аналогичный локальному вызову `box.space.`*`имя-спейса`*`:upsert(...)` (*детали*). В дополнение см. *Модуль buffer и skip-header*.

`conn.space.<space-name>:delete({field-value, ...} [, {options}])`
`conn.space.`*`имя-спейса`*`:delete(...)` – это удаленный вызов, аналогичный локальному вызову `box.space.`*`имя-спейса`*`:delete(...)`.(*детали*). В дополнение см. *Модуль buffer и skip-header*.

`conn:eval(`*Lua-string*[, *{arguments}*[, *{options}*]])
`conn:eval(`*`Lua-строка`*`)` оценивает и выполняет выражение в Lua-строке, которое может представлять собой любое выражение или несколько выражений. Требуются *права на выполнение*; если у пользователя таких прав нет, администратор может их выдать с помощью `box.schema.user.grant(`*`имя-пользователя`*`, 'execute', 'universe')`.

Чтобы гарантировать, что `conn:eval` вернет то, что возвращает выражение на Lua, начните Lua-строку со слова «return» (вернуть).

Примеры:

```
tarantool> --Lua-строка
tarantool> conn:eval('function f5() return 5+5 end; return f5();')
---
- 10
...
tarantool> --Lua-строка, {аргументы}
tarantool> conn:eval('return ...', {1,2,{3,'x'}})
---
- 1
- 2
- [3, 'x']
...
tarantool> --Lua-строка, {аргументы}, {парметры}
tarantool> conn:eval('return {nil,5}', {}, {timeout=0.1})
---
- [null, 5]
...
```

`conn:call(`*function-name*[, *{arguments}*[, *{options}*]])
`conn:call('func', {'1', '2', '3'})` – это удаленный вызов, аналогичный `func('1', '2', '3')`. Таким образом, `conn:call` представляет собой удаленный вызов хранимой процедуры. `conn:call` возвращает то, что возвращает функция.

Ограничение: вызванная функция не может вернуть функцию, например, если `func2` определяется как `function func2 () return func end`, то `conn:call(func2)` вернет ошибку «error: unsupported Lua type „function“».

Примеры:

```
tarantool> -- создание 2 функций с conn:eval()
tarantool> conn:eval('function f1() return 5+5 end;')
tarantool> conn:eval('function f2(x,y) return x,y end;')
tarantool> -- вызов первой функции без параметров и опций
tarantool> conn:call('f1')
---
- 10
...
tarantool> -- вызов второй функции с двумя параметрами и одной опцией
tarantool> conn:call('f2',{1,'B'},{timeout=99})
---
- 1
- B
...
```

`conn:timeout(`*timeout*`)`

`timeout(...)` – это надстройка, которая определяет время ожидания для запроса. С версии 1.7.4 этот метод объявлен устаревшим – лучше передать значение времени ожидания с помощью параметра `{options}`.

Пример:

```
conn:timeout(0.5).space.tester:update({1}, {{'=', 2, 15}})
```

Хотя `timeout(...)` объявлен устаревшим, все удаленные вызовы поддерживают его. Использование надстройки обеспечивает совместимость API удаленного соединения с локальным, поэтому отпадает необходимость в отдельном аргументе `timeout`, который проигнорирует локальная версия. После отправки запроса его нельзя отменить с удаленного сервера даже по истечении времени задержки: окончание времени задержки прерывает только ожидание ответа от удаленного сервера, а не сам запрос.

`conn:request(`*... {is_ async=...}*`)`

`{is_async=true|false}` – это опция, которую можно применить во всех запросах `net_box`, включая `conn:call`, `conn:eval` и запросы `conn.space.space-name`.

По умолчанию, `is_async=false`, что означает, что запросы будут синхронными для файбера. Файбер блокируется в ожидании ответа на запрос или до истечения времени ожидания. До версии Tarantool'а 1.10 единственным способом выполнения асинхронных запросов было использование отдельных файберов.

`is_async=true` означает, что запросы будут асинхронными для файбера. Запрос вызывает передачу управления, но файбер не входит в режим ожидания. Сразу же возвращается результат, но это будет не результат запроса, а объект, который может использовать вызывающая программа для получения результат запроса.

У такого сразу же возвращаемого объекта, который мы называем «future» (будущий), есть собственные методы:

- `future:is_ready()` вернет true (правда), если доступен результат запроса,
- `future:result()` используется для получения результата запроса (возвращает ответ на запрос или **nil** в случае, если ответ еще не готов или произошла какая-либо ошибка),
- `future:wait_result(timeout)` будет ждать, когда результат запроса будет доступен, а затем получит его или выдаст ошибку, если по истечении времени ожидания результат не получен.
- `future:discard()` откажется от объекта.

В обычной ситуации пользователь введет команду `future=имя-запроса(... {is_async=true})`, а затем либо цикл с проверкой `future:is_ready()` до тех пор, пока он не вернет true, и получением результата с помощью `request_result=future:result()`, либо же команду `request_result=future:wait_result(...)`. Возможен вариант, когда клиент проверяет наличие внеполосных сообщений от сервера, вызывая в цикле `pairs()` – см. *box.session.push()*.

Можно использовать `future:discard()`, чтобы соединение забыло об ответе – если получен ответ для отброшенного объекта, то он будет проигнорирован, так что размер таблицы запросов будет уменьшен, а другие запросы будут выполняться быстрее.

Пример:

```
tarantool> future = conn.space.tester:insert({900},{is_async=true})
---
...
tarantool> future
---
- method: insert
  response: [900]
  cond: cond
  on_push_ctx: []
  on_push: 'function: builtin#91'
...
tarantool> future:is_ready()
---
- true
...
tarantool> future:result()
---
- [900]
...
```

Как правило, `{is_async=true}` используется только при большой загрузке (более 100 000 запросов в секунду) и большой задержке чтения (более 1 секунды), или же при необходимости отправки нескольких одновременных запросов, которые собирают ответы (что иногда называется «отображение-свертка»).

Примечание: Хотя окончательный результат асинхронного запроса не отличается от результата синхронного запроса, у него другая структура: таблица, а не неупакованные значения.

Триггеры

В модуле `net.box` можно использовать следующие *триггеры*:

`conn:on_connect(`[*trigger-function*[, *old-trigger-function*]])

Определение триггера, исполняемого, когда устанавливается новое соединение (и при условии, что аутентификация и сборка схемы завершены) при таком событии, как `net_box.connect`. Если триггер не срабатывает и выкидывает исключение, статус подключения меняется на „error“.В таком случае соединение прерывается, независимо от значения опции `reconnect_after`. Может вызываться столько раз, сколько раз происходит переподключение, если значение параметра `reconnect_after` больше нуля.

Параметры

- `trigger-function` (*function*) – функция, в которой будет триггер. В качестве первого аргумента берет объект `conn`
- `old-trigger-function` (*function*) – существующая функция с триггером, которую заменит новая

возвращает nil или указатель функции

`conn:on_disconnect(`[*trigger-function*[, *old-trigger-function*]])

Определение триггера, исполняемого после закрытия соединения. Если функция с триггером вызывает ошибку, то ошибка записывается в журнал, в противном случае записей не будет. Выполнение прекращается после явного закрытия соединения или удаления сборщиком мусора в Lua.

Параметры

- `trigger-function` (*function*) – функция, в которой будет триггер. В качестве первого аргумента берет объект `conn`
- `old-trigger-function` (*function*) – существующая функция с триггером, которую заменит новая

возвращает nil или указатель функции

`conn:on_schema_reload(`[*trigger-function*[, *old-trigger-function*]])

Определение триггера, исполняемого во время выполнения определенной операции на удаленном сервере после обновления схемы. Другими словами, если запрос к серверу не выполняется из-за ошибки несовпадения версии схемы, происходит перезагрузка схемы.

Параметры

- `trigger-function` (*function*) – функция, в которой будет триггер. В качестве первого аргумента берет объект `conn`
- `old-trigger-function` (*function*) – существующая функция с триггером, которую заменит новая

возвращает nil или указатель функции

Примечание: Если указаны параметры `(nil, old-trigger-function)`, старый триггер будет удален.

Если не указан ни один параметр, ответом будет список существующих функций с триггером.

Подробная информация о характеристиках триггера находится в разделе *Триггеры*.

Пример

Ниже приводится пример использования большинства методов `net.box`.

Данный пример сработает на конфигурации из песочницы, предполагается, что:

- экземпляр Tarantool'а запущен на `localhost 127.0.0.1:3301`,
- создан спейс под названием `tester` с первичным числовым ключом и кортежем, в котором есть ключ со значением= 800,
- у текущего пользователя есть права на чтение, запись и выполнение.

Ниже приведены команды для быстрой настройки песочницы:

```
box.cfg{listen = 3301}
s = box.schema.space.create('tester')
s:create_index('primary', {type = 'hash', parts = {1, 'unsigned'}})
t = s:insert({800, 'TEST'})
box.schema.user.grant('guest', 'read,write,execute', 'universe')
```

А здесь приведен пример:

```
tarantool> net_box = require('net.box')
---
...
tarantool> function example()
         >   local conn, wtuple
         >   if net_box.self:ping() then
         >     table.insert(ta, 'self:ping() succeeded')
         >     table.insert(ta, '  (no surprise -- self connection is pre-established)')
         >   end
         >   if box.cfg.listen == '3301' then
         >     table.insert(ta,'The local server listen address = 3301')
         >   else
         >     table.insert(ta, 'The local server listen address is not 3301')
         >     table.insert(ta, '(  (maybe box.cfg{...listen="3301"...} was not stated)')
         >     table.insert(ta, '(  (so connect will fail)')
         >   end
         >   conn = net_box.connect('127.0.0.1:3301')
         >   conn.space.tester:delete({800})
         >   table.insert(ta, 'conn delete done on tester.')
         >   conn.space.tester:insert({800, 'data'})
         >   table.insert(ta, 'conn insert done on tester, index 0')
         >   table.insert(ta, '  primary key value = 800.')
         >   wtuple = conn.space.tester:select({800})
         >   table.insert(ta, 'conn select done on tester, index 0')
         >   table.insert(ta, '  number of fields = ' .. #wtuple)
         >   conn.space.tester:delete({800})
         >   table.insert(ta, 'conn delete done on tester')
         >   conn.space.tester:replace({800, 'New data', 'Extra data'})
         >   table.insert(ta, 'conn:replace done on tester')
         >   conn.space.tester:update({800}, {{'=', 2, 'Fld#1'}})
         >   table.insert(ta, 'conn update done on tester')
         >   conn:close()
         >   table.insert(ta, 'conn close done')
         > end
---
...
tarantool> ta = {}
---
...
tarantool> example()
---
...
tarantool> ta
---
- - self:ping() succeeded
  - '  (no surprise -- self connection is pre-established)'
  - The local server listen address = 3301
  - conn delete done on tester.
  - conn insert done on tester, index 0
```

(continues on next page)

(продолжение с предыдущей страницы)

```
  - '  primary key value = 800.'
  - conn select done on tester, index 0
  - '  number of fields = 1'
  - conn delete done on tester
  - conn:replace done on tester
  - conn update done on tester
  - conn close done
...
```

5.2.21 Модуль *os*

Общие сведения

Модуль `os` включает в себя следующие функции: *execute()*, *rename()*, *getenv()*, *remove()*, *date()*, *exit()*, *time()*, *clock()*, *tmpname()*, *environ()*, *setenv()*, *setlocale()*, *difftime()*. Большинство этих функций описаны в Главе 22 руководства по языку Lua Библиотека функций операционной системы.

Указатель

Ниже приведен перечень всех функций модуля `os`.

Имя	Назначение
os.execute()	Выполнение путем передачи в ОС
os.rename()	Переименование файла или директории
os.getenv()	Получение переменной окружения
os.remove()	Удаление файла или директории
os.date()	Получение даты в формате
os.exit()	Выход из программы
os.time()	Получение числа секунд с начала отсчета
os.clock()	Получение числа времени ЦП в секундах с момента начала программы
os.tmpname()	Получение имени временного файла
os.environ()	Получение таблицы со всеми переменными окружения
os.setenv()	Определение переменной окружения
os.setlocale()	Изменение локали
os.difftime()	Получение числа секунд между двумя значениями времени

`os.execute(`*shell-command*`)`

Выполнение путем передачи в ОС.

Параметры

- `shell-command` (`string`) – что выполнить.

Пример:

```
tarantool> os.execute('ls -l /usr')
total 200
drwxr-xr-x   2 root root 65536 Apr 22 15:49 bin
drwxr-xr-x  59 root root 20480 Apr 18 07:58 include
drwxr-xr-x 210 root root 65536 Apr 18 07:59 lib
drwxr-xr-x  12 root root  4096 Apr 22 15:49 local
```

(continues on next page)

(продолжение с предыдущей страницы)

```
drwxr-xr-x   2 root root 12288 Jan 31 09:50 sbin
---
...
```

os.rename(*old-name*, *new-name*)
Переименование файла или директории.

Параметры

- old-name (string) – имя существующего файла или директории,
- new-name (string) – измененное имя файла или директории.

Пример:

```
tarantool> os.rename('local','foreign')
---
- null
- 'local: No such file or directory'
- 2
...
```

os.getenv(*variable-name*)
Получение переменной окружения.

Параметры: (string) variable-name = имя переменной окружения.

Пример:

```
tarantool> os.getenv('PATH')
---
- /usr/local/sbin:/usr/local/bin:/usr/sbin
...
```

os.remove(*name*)
Удаление файла или директории.

Parameters: (string) name = имя файла или директории, которые будут удалены.

Пример:

```
tarantool> os.remove('file')
---
- true
...
```

os.date(*format-string*[, *time-since-epoch*])
Возврат даты в формате.

Parameters: (string) format-string = инструкции; (string) time-since-epoch = число секунд с 1970-01-01. Если не указать time-since-epoch, предполагается использование текущего времени.

Пример:

```
tarantool> os.date("%A %B %d")
---
- Sunday April 24
...
```

os.exit()
Выход из программы. Если выполняется на экземпляре сервера, останавливается работа экземпляра.

Пример:

```
tarantool> os.exit()
user@user-shell:~/tarantool_sandbox$
```

os.time()
Возврат числа секунд с начала отсчета.

Пример:

```
tarantool> os.time()
---
- 1461516945
...
```

os.clock()
Возврат числа времени ЦП в секундах с момента начала программы.

Пример:

```
tarantool> os.clock()
---
- 0.05
...
```

os.tmpname()
Возврат имени временного файла.

Пример:

```
tarantool> os.tmpname()
---
- /tmp/lua_7SW1m2
...
```

os.environ()
Возврат таблицы со всеми переменными окружения.

Пример:

```
tarantool> os.environ()['TERM']..os.environ()['SHELL']
---
- xterm/bin/bash
...
```

os.setenv(*variable-name*, *variable-value*)
Определение переменной окружения.

Пример:

```
tarantool> os.setenv('VERSION','99')
---
-
...
```

`os.setlocale(`[*new-locale-string*]`)`
Изменение локали. Если не указать new-locale-string, вернется текущая локаль.

Пример:

```
tarantool> string.sub(os.setlocale(),1,20)
---
- LC_CTYPE=en_US.UTF-8
...
```

`os.difftime(`*time1*, *time2*`)`
Возврат числа секунд между двумя значениями времени.

Пример:

```
tarantool> os.difftime(os.time() - 0)
---
- 1486594859
...
```

5.2.22 Модуль *pickle*

Указатель

Ниже приведен перечень всех функций модуля `pickle`.

Имя	Назначение
pickle.pack()	Конвертация Lua-переменных в двоичный формат
pickle.unpack()	Конвертация Lua-переменных в двоичный формат

`pickle.pack(`*format*, *argument*[, *argument ...*]`)`
Чтобы использовать примитивы бинарного протокола Tarantool'а из Lua, необходимо конвертировать Lua-переменные в двоичный формат. Прототипом вспомогательной функции `pickle.pack()` выступила функция „pack“ из Perl.

Спецификаторы формата

b, B	конвертирует скалярное Lua-значение в 1-байтное целое число и хранит целое число в полученной строке
s, S	конвертирует скалярное Lua-значение в 2-байтное целое число и хранит целое число в полученной строке, сначала младший байт
i, I	конвертирует скалярное Lua-значение в 4-байтное целое число и хранит целое число в полученной строке, сначала младший байт
l, L	конвертирует скалярное Lua-значение в 8-байтное целое число и хранит целое число в полученной строке, сначала младший байт
n	конвертирует скалярное Lua-значение в 2-байтное целое число и хранит целое число в полученной строке, порядок от старшего к младшему,
N	конвертирует скалярное Lua-значение в 4-байтное целое число и хранит целое число в полученной строке, порядок от старшего к младшему,
q, Q	конвертирует скалярное Lua-значение в 8-байтное целое число и хранит целое число в полученной строке, порядок от старшего к младшему,
f	конвертирует скалярное Lua-значение в 4-байтное число с плавающей запятой и хранит число с плавающей запятой в полученной строке
d	конвертирует скалярное Lua-значение в 8-байтное число двойной точности и хранит число двойной точности в полученной строке
a, A	конвертирует скалярное Lua-значение в последовательность байтов и хранит последовательность в полученной строке

Параметры

- `format` (`string`) – строка со спецификаторами формата
- `argument(s)` (*scalar-value*) – скалярные значения к форматированию

возвращает бинарная строка, которая содержит все аргументы, упакованные в соответствии со спецификаторами формата.

тип возвращаемого значения строка

Скалярное значение может быть либо переменной, либо литеральным значением. Следует помнить, что большие целые числа нужно вводить с *tonumber64() или суффиксами LL или ULL*.

Возможные ошибки: неизвестный спецификатор формата.

Пример:

```
tarantool> pickle = require('pickle')
---
...
tarantool> box.space.tester:insert{0, 'hello world'}
---
- [0, 'hello world']
...
tarantool> box.space.tester:update({0}, {{'=', 2, 'bye world'}})
---
- [0, 'bye world']
...
tarantool> box.space.tester:update({0}, {
         >   {'=', 2, pickle.pack('iiA', 0, 3, 'hello')}
         > })
---
- [0, "\0\0\0\0\x03\0\0\0hello"]
...
```

(continues on next page)

(продолжение с предыдущей страницы)

```
tarantool> box.space.tester:update({0}, {{'=', 2, 4}})
---
- [0, 4]
...
tarantool> box.space.tester:update({0}, {{'+', 2, 4}})
---
- [0, 8]
...
tarantool> box.space.tester:update({0}, {{'^', 2, 4}})
---
- [0, 12]
...
```

pickle.unpack(*format*, *binary-string*)

Противоположность `pickle.pack()`. Внимание: если используется спецификатор формата „A“, он должен идти последним.

Параметры

- `format` (string) –
- `binary-string` (string) –

возвращает Список строк или чисел.

тип возвращаемого значения таблица

Пример:

```
tarantool> pickle = require('pickle')
---
...
tarantool> tuple = box.space.tester:replace{0}
---
...
tarantool> string.len(tuple[1])
---
- 1
...
tarantool> pickle.unpack('b', tuple[1])
---
- 48
...
tarantool> pickle.unpack('bsi', pickle.pack('bsi', 255, 65535, 4294967295))
---
- 255
- 65535
- 4294967295
...
tarantool> pickle.unpack('ls', pickle.pack('ls', tonumber64('18446744073709551615'), 65535))
---
...
tarantool> num, num64, str = pickle.unpack('slA', pickle.pack('slA', 666,
         > tonumber64('666666666666666'), 'string'))
---
...
```

5.2.23 Модуль *socket*

Общие сведения

Модуль `socket` позволяет обмениваться данными с локальным или удаленным хостом по BSD-сокетам в режиме с установлением соединений (TCP) или на основе датаграмм (UDP). Семантика вызовов в API модуля `socket` точно соответствует семантике соответствующих вызовов в POSIX.

Функции для настройки и подключения: `socket`, `sysconnect`, `tcp_connect`. Функции для отправки данных: `send`, `sendto`, `write`, `syswrite`. Функции для получения данных: `recv`, `recvfrom`, `read`. Функции для ожидания отправки/получения данных: `wait`, `readable`, `writable`. Функции для установки флагов: `nonblock`, `setsockopt`. Функции для остановки и отключения: `shutdown`, `close`. Функции для проверки ошибок: `errno`, `error`.

Указатель

Ниже приведен перечень всех функций модуля `socket`.

Имя	Назначение
socket()	Создание сокета
socket.tcp_connect()	Подключение к удаленному хосту с помощью сокета
socket.getaddrinfo()	Получение информации об удаленном узле
socket.tcp_server()	Использование Tarantool'а в качестве TCP-сервера
socket_object:sysconnect()	Подключение к удаленному хосту с помощью сокета
socket_object:send() *socket_object:write()*	Отправка данных по подключенному сокету
socket_object:syswrite()	Запись данных в буфер сокета без блокировки
socket_object:recv()	Чтение с подключенного сокета
socket_object:sysread()	Чтение данных из буфера сокета без блокировки
socket_object:bind()	Привязка сокета к данному хосту/порту
socket_object:listen()	Начало прослушивания входящих соединений
socket_object:accept()	Принятие запроса клиента на соединение + создание подключенного сокета
socket_object:sendto()	Отправка сообщения по UDP-сокету на указанный хост
socket_object:recvfrom()	Получение сообщения по UDP-сокету
socket_object:shutdown()	Отключение передачи данных на чтение, на запись или в обоих направлениях
socket_object:close()	Закрытие сокета
socket_object:error() *socket_object:errno()*	Получение информации о последней ошибке на сокете
socket_object:setsockopt()	Определение флагов сокета
socket_object:getsockopt()	Получение флагов сокета
socket_object:linger()	Установить/убрать флаг SO_LINGER
socket_object:nonblock()	Определить/получить значение флага
socket_object:readable()	Ожидание доступности чего-либо для чтения
socket_object:writable()	Ожидание доступности чего-либо для записи
socket_object:wait()	Ожидание доступности чего-либо для чтения или записи
socket_object:name()	Получение информации о ближней стороне соединения
socket_object:peer()	Получение информации о дальней стороне соединения
socket.iowait()	Ожидание активности чтения/записи
LuaSocket wrapper functions	Несколько методов эмуляции LuaSocket API

Как правило, сессия сокета начинается с функций настройки, определяет один или более флагов, запустит цикл с функциями отправки и получения и закончится функциями завершения – как в примере в конце данного раздела. В течение сессии может быть проверка на ошибки и ожидание синхронизации функции. Чтобы файбер с сокетом не блокировал другие файберы, *правила неявной передачи управления* заставят его передать управление другим процессам в рамках *кооперативной многозадачности*.

Для всех примеров в данном разделе имя сокета будет sock, а вызов функции будет выглядеть как `sock:имя_функции(...)`.

`socket.__call(`*domain*, *type*, *protocol*`)`

Создание нового TCP-сокета или UDP-сокета. Значения аргумента остаются теми же, что и на странице socket(2) руководства по Linux.

возвращает неподключенный сокет или nil.

тип возвращаемого значения пользовательские данные

Пример:

```
socket('AF_INET', 'SOCK_STREAM', 'tcp')
```

`socket.tcp_connect(`*host*[, *port*[, *timeout*]])

Подключение к удаленному хосту с помощью сокета.

Параметры

- `host` (`string`) – URL или IP-адрес
- `port` (*`number`*) – номер порта
- `timeout` (*`number`*) – время ожидания

возвращает подключенный сокет, если нет ошибки.

тип возвращаемого значения пользовательские данные

Пример:

```
socket.tcp_connect('127.0.0.1', 3301)
```

`socket.getaddrinfo(`*host*, *port*[, *timeout*[, *{option-list}*]])

`socket.getaddrinfo(`*host*, *port*[, *{option-list}*])

Функция `socket.getaddrinfo()` используется для поиска информации об удаленном узле, чтобы можно было передать правильные аргументы для `sock:sysconnect()`. Эта функция может использовать конфигурационный параметр *worker_pool_threads*.

Параметры

- `host` (`string`) – URL или IP-адрес
- `port` (*`number`*) – номер порта или строка, указывающая на порт
- `timeout` (*`number`*) – количество секунд ожидания
- `options` (`table`) –
  - `type` – предпочтительный тип сокета
  - `family` – предпочтительное семейство адресов
  - `protocol`
  - `flags` – дополнительные опции (подробнее о них здесь)

возвращает Таблица со следующими полями: «host», «family», «type», «protocol», «port».

тип возвращаемого значения таблица

Пример:

```
tarantool> socket.getaddrinfo('tarantool.org', 'http')
---
- - host: 188.93.56.70
    family: AF_INET
    type: SOCK_STREAM
    protocol: tcp
    port: 80
  - host: 188.93.56.70
    family: AF_INET
    type: SOCK_DGRAM
    protocol: udp
    port: 80
...
-- To find the available values for the options use the following:
tarantool> socket.internal.AI_FLAGS -- or SO_TYPE, or DOMAIN
---
- AI_ALL: 256
  AI_PASSIVE: 1
  AI_NUMERICSERV: 4096
  AI_NUMERICHOST: 4
  AI_V4MAPPED: 2048
  AI_ADDRCONFIG: 1024
  AI_CANONNAME: 2
...
```

socket.tcp_server(*host*, *port*, *handler-function-or-table*[, *timeout*])

Функция socket.tcp_server() заставляет Tarantool выступать в качестве сервера для принятия подключений. Обычно для этой же цели используется *box.cfg{listen=...}*.

Параметры

- host (string) – имя или IP хоста
- port (*number*) – порт хоста, может быть 0
- handler-function-or-table (*function/table*) – что выполнить после подключения
- timeout (*number*) – количество секунд ожидания

Параметр handler-function-or-table может представлять собой просто имя функции или объявление функции: handler_function. Или же может быть таблицей: {handler = *handler_function* [, prepare = *prepare_function*] [, name = *name*] }. Функция handler_function является обязательной, в ней может быть только один параметр = сокет (используется для непрерывной работы после установки соединения), выполняется один раз за соединение после того, как произойдет accept(). Функция prepare_function необязательна; она выполняется однократно перед установкой соединения (bind()) на слушающем сокете и должна возвращать либо значение бэклога, либо ничего. Например:

```
socket.tcp_server('localhost', 3302, function (s) loop_loop() end)
socket.tcp_server('localhost', 3302, {handler=hfunc, name='name'})
socket.tcp_server('localhost', 3302, {handler=hfunc, prepare=pfunc})
```

Более полный пример см. в разделе *Использование tcp_server для получения содержимого файла, отправленного по socat* и *Использование tcp_server с handler и prepare*.

`object socket_object`

`socket_object:sysconnect(`*host*, *port*`)`
Подключение к удаленному хосту с помощью существующего сокета. Значения аргументов будут такие же, как в *tcp_connect()*. Хост должен представлять собой IP-адрес.

Параметры:

- **Либо:**
  - host – строковое представление IPv4 адреса или IPv6 адреса;
  - port – число.
- **Либо:**
  - host – строка, которая содержит «unix/»;
  - port – строка, которая содержит путь к Unix-сокету.
- **Либо:**
  - host – число, 0 (ноль), что означает «все локальные интерфейсы»;
  - port – число. Если номер порта – 0 (ноль), сокет будет привязан к случайному локальному порту.

возвращает значение объекта сокета может изменяться, если будет выполнена функция sysconnect().

тип возвращаемого значения boolean (логический)

Пример:

```
socket = require('socket')
sock = socket('AF_INET', 'SOCK_STREAM', 'tcp')
sock:sysconnect(0, 3301)
```

`socket_object:send(`*data*`)`
`socket_object:write(`*data*`)`
Отправка данных по подключенному сокету.

Параметры

- `data` (`string`) – что отправляется

возвращает количество отправляемых байтов.

тип возвращаемого значения число

Возможные ошибки: nil в случае ошибки.

`socket_object:syswrite(`*size*`)`
Запись максимально возможного количества данных в буфер сокета без блокировки. Используется редко. Для получения подробной информации см. описание.

`socket_object:recv(`*size*`)`
Чтение количества байтов, определенного в `size`, из подключенного сокета. Внутренний буфер опережающего считывания используется для уменьшения использования ресурсов на вызов.

Параметры

- `size` (*integer*) – максимальное количество получаемых байтов. См. *Рекомендованный размер*.

возвращает строка запрошенной длины, если выполнено.

тип возвращаемого значения строка

Возможные ошибки: В случае ошибки возвращается пустая строка, после чего статус, errno, errstr. Если передача данных на запись закрыта с другой стороны, возвращаются оставшиеся для чтения данные из сокета (возможно, пустая строка), после чего идет статус «eof» (конец файла).

`socket_object:read(`*limit*[, *timeout*]`)`

`socket_object:read(`*delimiter*[, *timeout*]`)`

`socket_object:read(`*{options}*[, *timeout*]`)`

Чтение данных из подключенного сокета до выполнения какого-либо условия и возврат прочтенных байтов. Производится чтения количества байтов, которое указано в параметре `limit`, либо до символа-разделителя, либо до истечения времени ожидания. В отличие от `socket_object:recv` (где используется внутренний буфер опережающего считывания), `socket_object:read` зависит от буфера сокета.

Параметры

- `limit` (*integer*) – максимальное количество байтов для чтения, например, 50 означает «остановиться на 50 байтах»
- `delimiter` (`string`) – разделитель, например, „?“ означает «остановиться после знака вопроса»
- `timeout` (*number*) – максимальное количество секунд ожидания, например, 50 означает «остановиться через 50 секунд».
- `options` (`table`) – `chunk=`*предел* и/или `delimiter=`*разделитель*, например, `{chunk=5,delimiter='x'}`.

возвращает пустая строка, если нет данных для чтения, либо нулевое значение nil в случае ошибки, либо строка, ограниченная количеством байтов в `limit`, которая может включать в себя байты, совпадающие с выражением `delimiter`.

тип возвращаемого значения строка

`socket_object:sysread(`*size*`)`

Возврат данных из буфера сокета без блокировки.Если сокет с блокировкой, `sysread()` может блокировать процесс вызова. Используется редко. Для получения подробной информации, см. описание.

Параметры

- `size` (*integer*) – максимальное количество байтов для чтения, например, 50 означает «остановиться на 50 байтах»

возвращает пустая строка, если нет данных для чтения, либо нулевое значение nil в случае ошибки, либо строка, ограниченная количеством байтов в `size`.

тип возвращаемого значения строка

`socket_object:bind(`*host*[, *port*]`)`

Привязка сокета к данному хосту/порту. UDP-сокет после привязки может использоваться для получения данных (см. *socket_object.recvfrom*). TCP-сокет может использоваться для принятия новых соединений после перевода в режим прослушивания.

Параметры

- `host` (`string`) – URL или IP-адрес
- `port` (*`number`*) – номер порта

возвращает true (правда), если выполнено, false (ложь) в случае ошибки. Если возвращается false, используйте *socket_object:errno()* или *socket_object:error()* для получения подробной информации.

тип возвращаемого значения boolean (логический)

`socket_object:listen(`*backlog*`)`
Начало прослушивания входящих соединений.

Параметры

- `backlog` – в Linux очередь запросов `backlog` может быть в `/proc/sys/net/core/somaxconn`, в BSD очередь запросов может представлять собой `SOMAXCONN`.

возвращает true (правда), если выполнено, false (ложь) в случае ошибки.

тип возвращаемого значения boolean (логический).

`socket_object:accept()`
Принятие нового клиентского соединения и создание нового подключенного сокета. Установка блокирующего режима на сокете явным образом после принятия соединения приведет к эффективной работе.

возвращает новый сокет, если выполнено.

тип возвращаемого значения пользовательские данные

Возможные ошибки: nil.

`socket_object:sendto(`*host*, *port*, *data*`)`
Отправка сообщения по UDP-сокету на указанный хост.

Параметры

- `host` (`string`) – URL или IP-адрес
- `port` (*`number`*) – номер порта
- `data` (`string`) – что отправляется

возвращает количество отправляемых байтов.

тип возвращаемого значения число

Возможные ошибки: в случае ошибки возвращает nil, а также может вернуть статус, errno, errstr.

`socket_object:recvfrom(`*size*`)`
Получение сообщения по UDP-сокету.

Параметры

- `size` (*`integer`*) – максимальное количество получаемых байтов. См. *Рекомендованный размер*.

возвращает сообщение, таблица с полями «host», «family» и «port».

тип возвращаемого значения строка, таблица

Возможные ошибки: в случае ошибки возвращает nil, статус, errno, errstr.

Пример:

После `message_content, message_sender = recvfrom(1)` значением `message_content` может быть строка, которая содержит „X“, а значением `message_sender` может быть таблица, которая содержит

```
message_sender.host = '18.44.0.1'
message_sender.family = 'AF_INET'
message_sender.port = 43065
```

`socket_object:shutdown(`*how*`)`
Отключение передачи данных на чтение, на запись или в обоих направлениях.

Параметры

- `how` – socket.SHUT_RD, socket.SHUT_WR, or socket.SHUT_RDWR.

возвращает true (правда) или false (ложь).

тип возвращаемого значения boolean (логический)

`socket_object:close()`
Закрытие (удаление) сокета. Закрытый сокет больше не должен использоваться. Сокет будет закрыт автоматически, когда сборщик мусора Lua удалит данные.

возвращает true (правда), если выполнено, false (ложь) в случае ошибки. Например, если сокет sock уже закрыт, sock:close() вернет false.

тип возвращаемого значения boolean (логический)

`socket_object:error()`
`socket_object:errno()`
Получение информации о последней ошибке на сокете, если таковая была. Ошибки не выдают исключения, поэтому данные функции необходимы.

возвращает результат `sock:errno()`, результат `sock:error()`. Если ошибки нет, то `sock:errno()` вернет 0 и `sock:error()`.

тип возвращаемого значения число, строка

`socket_object:setsockopt(`*level*, *name*, *value*`)`
Определение флагов сокета. Значения аргумента будут такими же, что и на странице getsockopt(2) руководства по Linux. Tarantool принимает следующие:

- SO_ACCEPTCONN
- SO_BINDTODEVICE
- SO_BROADCAST
- SO_DEBUG
- SO_DOMAIN
- SO_ERROR
- SO_DONTROUTE
- SO_KEEPALIVE
- SO_MARK
- SO_OOBINLINE

- SO_PASSCRED
- SO_PEERCRED
- SO_PRIORITY
- SO_PROTOCOL
- SO_RCVBUF
- SO_RCVBUFFORCE
- SO_RCVLOWAT
- SO_SNDLOWAT
- SO_RCVTIMEO
- SO_SNDTIMEO
- SO_REUSEADDR
- SO_SNDBUF
- SO_SNDBUFFORCE
- SO_TIMESTAMP
- SO_TYPE

Установка флага SO_LINGER осуществляется с помощью `sock:linger(active)`.

`socket_object:getsockopt(`*level*, *name*`)`
Получение флагов сокета. Список возможных флагов см. с помощью `sock:setsockopt()`.

`socket_object:linger(`[*active*]`)`
Установить или убрать флаг SO_LINGER. Описание флага см. в руководстве по Linux.

Параметры

- `active` (*boolean*) –

возвращает новые значения active и timeout.

`socket_object:nonblock(`[*flag*]`)`

- `sock:nonblock()` возвращает текущее значение флага.
- `sock:nonblock(false)` устанавливает флаг на false и возвращает false.
- `sock:nonblock(true)` устанавливает флаг на true и возвращает true.

Эту функцию можно использовать до вызова функции, которая в противном случае будет блокировать бесконечно.

`socket_object:readable(`[*timeout*]`)`
Ожидание доступности чего-либо для чтения или до истечения времени ожидания.

возвращает true, если сокет доступен для чтения, false, если истекло время ожидания;

`socket_object:writable(`[*timeout*]`)`
Ожидание доступности чего-либо для записи или до истечения времени ожидания.

возвращает true, если сокет доступен для записи, false, если истекло время ожидания;

`socket_object:wait([timeout])`
Ожидание доступности чего-либо для чтения или записи, или до истечения времени ожидания.

возвращает „R“, если сокет доступен для чтения, „W“, если сокет доступен для записи, „RW“, если сокет доступен и для чтения, и для записи, „“ (пустая строка), если истекло время ожидания;

`socket_object:name()`
Функция `sock:name()` используется для получения информации о ближней стороне соединения. Если сокет привязан к `xyz.com:45`, то `sock:name` вернет информацию о `[host:xyz.com, port:45]`. Аналогичная функция в POSIX – `getsockname()`.

возвращает Таблица со следующими полями: «host», «family», «type», «protocol», «port».

тип возвращаемого значения таблица

`socket_object:peer()`
Функция `sock:peer()` используется для получения информации о дальней стороне соединения. Если TCP-соединение установлено с удаленным хостом `tarantool.org:80`, то `sock:peer()` вернет информацию о `[host:tarantool.org, port:80]`. Аналогичная функция в POSIX – `getpeername()`.

возвращает Таблица со следующими полями: «host», «family», «type», «protocol», «port».

тип возвращаемого значения таблица

`socket.iowait(`*fd*, *read-or-write-flags*`[, `*timeout*`])`
Функция `socket.iowait()` используется для ожидания, пока дескриптор файла не будет активен для чтения или записи.

Параметры

- `fd` – дескриптор файла
- `read-or-write-flags` – „R“ или 1 = чтение, „W“ или 2 = запись, „RW“ или 3 = чтение|запись.
- `timeout` – количество секунд ожидания

Если значение параметра fd – nil, то будет режим ожидания до истечения времени, указанного в параметре timeout. Если timeout – nil или не указан, время ожидания считается бесконечным.

Как правило, возвращается значение совершенного действия („R“ или „W“, или „RW“, или 1, или 2, или 3). Если время ожидания в timeout проходит без действий чтения или записи, возвращается ошибка = ETIMEDOUT.

Пример: `socket.iowait(sock:fd(), 'r', 1.11)`

Функции обертки LuaSocket

LuaSocket API имеет функции, эквивалентные описанным выше, с различными именами и параметрами, например `connect()`, а не `tcp_connect()`, а также `getpeername`, `getsockname`, `setoption`, `settimeout`. Tarantool поддерживает эти функции, так что сторонние пакеты, зависящие от них, будут работать.

Проект LuaSocket находится на github. Описание API находится в руководстве по LuaSocket (нажмите на ссылки «введение» и «ссылка» внизу главной страницы руководства).

Пример для Tarantool - *Использование сокета с функциями обертки LuaSocket*.

Рекомендованный размер

Для `recv` и `recvfrom`: используйте необязательный параметр `size`, чтобы ограничить количество получаемых байтов. Часто используется заданный размер, такой как 512; но во многих случаях лучше использовать предварительно рассчитанный размер, который зависит от контекста – как формат сообщения или состояние сети. Что касается `recvfrom`, следует помнить, что размер больше максимального размера полезного блока данных одного пакета (Maximum Transmission Unit) может вызвать низкоэффективную передачу данных. Что касается Mac OS X, следует отметить, что размер можно настроить с помощью `sysctl net.inet.udp.maxdgram`.

Если размер `size` не задан: Tarantool сделает дополнительный вызов для расчет необходимого количества байтов. Такой дополнительный вызов занимает время, поэтому во избежание низкой эффективности лучше указать `size`.

Если размер `size` задан: в UDP-сокете лишние байты отбрасываются; в TCP-сокете лишние байты не отбрасываются, их можно получить при следующем вызове.

Примеры

Использование TCP-сокета в интернете

В данном примере устанавливается соединение по интернету между экземпляром Tarantool'а и tarantool.org, затем отправляется HTTP-сообщение заголовка «head» и возвращается ответ: «`HTTP/1.1 200 OK`» или что-то другое, если сайт перемещен. Так не слишком удобно взаимодействовать с определенным сайтом, но пример показывает работу системы.

```
tarantool> socket = require('socket')
---
...
tarantool> sock = socket.tcp_connect('tarantool.org', 80)
---
...
tarantool> type(sock)
---
- table
...
tarantool> sock:error()
---
- null
...
tarantool> sock:send("HEAD / HTTP/1.0\r\nHost: tarantool.org\r\n\r\n")
---
- 40
...
tarantool> sock:read(17)
---
- HTTP/1.1 302 Move
...
tarantool> sock:close()
---
- true
...
```

Использование сокета с функциями обертки LuaSocket

Это вариант более раннего примера «Использование TCP-подключения через Интернет». В нем используются *функции обертки LuaSocket*, с слишком коротким временем ожидания, так что, скорее всего, произойдет ошибка «Connection timed out» (Таймаут соединения). Более распространенным способом определения таймаута является использование функции *tcp_connect()*.

```
tarantool> socket = require('socket')
---
...
tarantool> sock = socket.connect('tarantool.org', 80)
---
...
tarantool> sock:settimeout(0.001)
---
- 1
...
tarantool> sock:send("HEAD / HTTP/1.0\r\nHost: tarantool.org\r\n\r\n")
---
- 40
...
tarantool> sock:receive(17)
---
- null
- Connection timed out
...
tarantool> sock:close()
---
- 1
...
```

Использование UDP-сокета на localhost

Ниже приведен пример с датаграммами. Устанавливается два соединения с 127.0.0.1 (localhost): `sock_1` и `sock_2`. С помощью `sock_2` отправляется сообщение на `sock_1`. С помощью `sock_1` получается сообщение. Отображается полученное сообщение. Оба соединения закрываются. Компьютеру так не слишком удобно взаимодействовать с самим собой, но пример показывает работу системы.

```
tarantool> socket = require('socket')
---
...
tarantool> sock_1 = socket('AF_INET', 'SOCK_DGRAM', 'udp')
---
...
tarantool> sock_1:bind('127.0.0.1')
---
- true
...
tarantool> sock_2 = socket('AF_INET', 'SOCK_DGRAM', 'udp')
---
...
tarantool> sock_2:sendto('127.0.0.1', sock_1:name().port,'X')
---
- 1
...
```

(continues on next page)

(продолжение с предыдущей страницы)

```
tarantool> message = sock_1:recvfrom(512)
---
...
tarantool> message
---
- X
...
tarantool> sock_1:close()
---
- true
...
tarantool> sock_2:close()
---
- true
...
```

Использование tcp_server для получения содержимого файла, отправленного по socat

Ниже приведен пример функции `tcp_server`, которая читает строки с клиента и выводит результат. На клиентской стороне утилита `socat` в Linux будет использоваться для отправки целого файла на чтение функции `tcp_server`.

Запустите две оболочки. Первая оболочка будет экземпляром сервера. Вторая оболочка будет клиентом.

В первой оболочке запустите Tarantool и выполните:

```
box.cfg{}
socket = require('socket')
socket.tcp_server('0.0.0.0', 3302,
{
  handler = function(s)
    while true do
      local request
      request = s:read("\n");
      if request == "" or request == nil then
        break
      end
      print(request)
    end
  end,
  prepare = function()
    print('Initialized')
  end
}
)
```

Вышеуказанный код означает:

1. Использовать `tcp_server()` для ожидания подключения с любого хоста по порту 3302.
2. Когда это произойдет, ввести цикл, который читает по сокету и выводит результат чтения. Разделителем для функции чтения будет «\n», поэтому каждое выполнение `read()` выполнит чтение строки до перевода строки, включая перевод строки.

Во второй оболочке создайте файл, который содержит несколько строк. Содержимое не имеет значения. Предположим, что первая строка содержит A, вторая строка содержит B, третья строка содержит

C. Назовите этот файл «tmp.txt».

Во второй оболочке используйте утилиту socat для отправки файла tmp.txt на экземпляр сервера по хосту и порту:

```
$ socat TCP:localhost:3302 ./tmp.txt
```

Теперь смотрите, что происходит в первой оболочке. Выводятся строки «A», «B», «C».

Использование tcp_server с handler и prepare

Ниже приведен пример функции tcp_server с использованием `handler` и `prepare`.

Запустите две оболочки. Первая оболочка будет экземпляром сервера. Вторая оболочка будет клиентом.

В первой оболочке запустите Tarantool и выполните:

```
box.cfg{}
socket = require('socket')
sock = socket.tcp_server(
  '0.0.0.0',
  3302,
  {prepare =
     function(sock)
       print('listening on socket ' .. sock:fd())
       sock:setsockopt('SOL_SOCKET','SO_REUSEADDR',true)
       return 5
     end,
   handler =
    function(sock, from)
      print('accepted connection from: ')
      print('  host: ' .. from.host)
      print('  family: ' .. from.family)
      print('  port: ' .. from.port)
    end
  }
)
```

Вышеуказанный код означает:

1. Использовать `tcp_server()` для ожидания подключения с любого хоста по порту 3302.
2. Указать, что будет первый вызов `prepare`, который покажет что-то о сервере, затем вызовет `setsockopt(...'SO_REUSEADDR'...)` (это та же самая опция, которую Тарантул бы установил, если бы не было `prepare`), а затем вернет 5 (это довольно низкий размер очереди бэклога).
3. Указать, что будут вызовы `handler` по каждому соединению, которые будут отображать что-то о клиенте.

Теперь смотрите, что происходит в первой оболочке. Выведется что-то вроде „listening on socket 12“.

Во второй оболочке запустите Tarantool и выполните:

```
box.cfg{}
require('socket').tcp_connect('127.0.0.1', 3302)
```

Теперь смотрите, что происходит на первой оболочке. На дисплее появится что-то вроде „accepted connection from host: 127.0.0.1 family: AF_INET port: 37186“.

5.2.24 Модуль *strict*

Модуль `strict` включает в себя функции для включения или отключения строгого режима «strict mode». Когда включен строгий режим, попытка использовать необъявленную глобальную переменную приведет к ошибке. Глобальная переменная считается необъявленной, если ей никогда не было присвоено значение. Часто это указывает на ошибку программирования.

По умолчанию, строгий режим отключен, не считая случаев, когда сборка Tarantool'а производилась с помощью `-DCMAKE_BUILD_TYPE=Debug` – см. варианты сборки в разделе *сборка из исходников*.

Пример:

```
tarantool> strict = require('strict')
---
...
tarantool> strict.on()
---
...
tarantool> a = b -- строгий режим включен, поэтому появляется ошибка
---
- error: ... variable ''b'' is not declared'
...
tarantool> strict.off()
---
...
tarantool> a = b -- строгий режим отключен, поэтому ошибки нет
---
...
```

5.2.25 Модуль *string*

Общие сведения

Модуль `string` включает в себя всё из стандартной библиотеки для работы со строками в Lua, а также некоторые расширения специально для Tarantool'а.

В данном разделе мы рассматриваем только дополнительные функции, добавленные разработчиками Tarantool'а.

Указатель

Ниже приведен перечень всех функций библиотеки `string`.

Имя	Назначение
string.ljust()	Выравнивание строки по левому полю
string.rjust()	Выравнивание строки по правому полю
string.hex()	Given a string, return hexadecimal values
string.fromhex()	Given hexadecimal values, return a string
string.startswith()	Проверка, начинается ли строка с заданной подстроки
string.endswith()	Проверка, заканчивается ли строка на заданную подстроку
string.lstrip()	Remove characters from the left of a string
string.rstrip()	Remove characters from the right of a string
string.split()	Разделение строки на таблицу со строками
string.strip()	Удаление пробелов слева и справа от строки

string.ljust(*input-string*, *width*[, *pad-character*])
Возврат строки, выровненной по левому краю, шириной, указанной в width.

Параметры

- input-string (string) – строка для выравнивания по левому краю
- width (*integer*) – ширина строки после выравнивания по левому краю
- pad-character (string) – отдельный символ, по умолчанию = 1 пробел

Возвращается выровненная по левому краю строка (не изменяется, если ширина <= длине строки)

Тип возвращаемого значения строка

Пример:

```
tarantool> string = require('string')
---
...
tarantool> string.ljust(' A', 5)
---
- ' A   '
...
```

string.rjust(*input-string*, *width*[, *pad-character*])
Возврат строки, выровненной по правому краю, шириной, указанной в width.

Параметры

- input-string (string) – строка для выравнивания по правому краю
- width (*integer*) – ширина строки после выравнивания по правому краю
- pad-character (string) – отдельный символ, по умолчанию = 1 пробел

Возвращается выровненная по правому краю строка (не изменяется, если ширина <= длине строки)

Тип возвращаемого значения строка

Пример:

```
tarantool> string = require('string')
---
...
tarantool> string.rjust('', 5, 'X')
---
- 'XXXXX'
...
```

string.hex(*input-string*)
Возврат шестнадцатеричного значения введенной строки.

Параметры

- input-string (string) – обрабатываемая строка

Возвращается шестнадцатеричное число, два символа шестнадцатеричных цифр для каждого введенного символа

Тип возвращаемого значения строка

Пример:

```
tarantool> string = require('string')
---
...
tarantool> string.hex('ABC ')
---
- '41424320'
...
```

string.fromhex(*hexadecimal-input-string*)

Given a string containing pairs of hexadecimal digits, return a string with one byte for each pair. This is the reverse of `string.hex()`. The hexadecimal-input-string must contain an even number of hexadecimal digits.

Параметры

- `hexadecimal-input-string` (string) – string with pairs of hexadecimal digits

Возвращается string with one byte for each pair of hexadecimal digits

Тип возвращаемого значения строка

Пример:

```
tarantool> string = require('string')
---
...
tarantool> string.fromhex('41424320')
---
- 'ABC '
...
```

string.startswith(*input-string*, *start-string*[, *start-pos*[, *end-pos*]])

Возврат true (правда), если `input-string` начинается со `start-string`, в противном случае, возврат false (ложь).

Параметры

- `input-string` (string) – строка, где производится поиск данных из `start-string`
- `start-string` (string) – искомая строка
- `start-pos` (*integer*) – положение: где начинать искать в пределах `input-string`
- `end-pos` (*integer*) – положение: где заканчивать искать в пределах `input-string`

Возвращается true (правда) или false (ложь)

Тип возвращаемого значения boolean (логический)

Значения `start-pos` и `end-pos` могут быть отрицательными, что означает, что положение вычисляется с конца строки.

Пример:

```
tarantool> string = require('string')
---
...
tarantool> string.startswith(' A', 'A', 2, 5)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
---
- true
...
```

string.endswith(*input-string*, *end-string*[, *start-pos*[, *end-pos*]])

Возврат true (правда), если `input-string` заканчивается на `end-string`, в противном случае, возврат false (ложь).

Параметры

- `input-string` (string) – строка, где производится поиск данных из `end-string`
- `end-string` (string) – искомая строка
- `start-pos` (*integer*) – положение: где начинать искать в пределах `input-string`
- `end-pos` (*integer*) – положение: где заканчивать искать в пределах `input-string`

Возвращается true (правда) или false (ложь)

Тип возвращаемого значения boolean (логический)

Значения `start-pos` и `end-pos` могут быть отрицательными, что означает, что положение вычисляется с конца строки.

Пример:

```
tarantool> string = require('string')
---
...
tarantool> string.endswith('Baa', 'aa')
---
- true
...
```

string.lstrip(*input-string*[, *list-of-characters*])

Return the value of the input string, after removing characters on the left. The optional `list-of-characters` parameter is a set not a sequence, so `string.lstrip(...,'ABC')` does not mean strip `'ABC'`, it means strip `'A'` or `'B'` or `'C'`.

Параметры

- `input-string` (string) – обрабатываемая строка
- `list-of-characters` (string) – what characters can be stripped. Default = space.

Возвращается result after stripping characters from input string

Тип возвращаемого значения строка

Пример:

```
tarantool> string = require('string')
---
...
tarantool> string.lstrip(' ABC ')
---
- 'ABC '
...
```

string.rstrip(*input-string*[, *list-of-characters*])
Return the value of the input string, after removing characters on the right. The optional `list-of-characters` parameter is a set not a sequence, so `string.rstrip(...,'ABC')` does not mean strip `'ABC'`, it means strip `'A'` or `'B'` or `'C'`.

Параметры

- `input-string` (`string`) – обрабатываемая строка
- `list-of-characters` (`string`) – what characters can be stripped. Default = space.

Возвращается result after stripping characters from input string

Тип возвращаемого значения строка

Пример:

```
tarantool> string = require('string')
---
...
tarantool> string.rstrip(' ABC ')
---
- ' ABC'
...
```

string.split(*input-string*[, *split-string*[, *max*]])
Разделение `input-string` на одну или более выводимых строк в таблице. Места разделения указаны в `split-string`.

Параметры

- `input-string` (`string`) – строка для разделения
- `split-string` (*integer*) – искомая строка в пределах `input-string`. По умолчанию = пробел.
- `max` (*integer*) – максимальное количество символов-разделителей от начала обрабатываемой строки. Результат содержит не более max + 1 частей.

Возвращается таблица строк, которые были разделены из `input-string`

Тип возвращаемого значения таблица

Пример:

```
tarantool> string = require('string')
---
...
tarantool> string.split("A:B:C:D:F", ":", 2)
---
- - A
  - B
  - C:D:F
...
```

string.strip(*input-string*[, *list-of-characters*])
Return the value of the input string, after removing characters on the left and the right. The optional `list-of-characters` parameter is a set not a sequence, so `string.strip(...,'ABC')` does not mean strip `'ABC'`, it means strip `'A'` or `'B'` or `'C'`.

Параметры

- `input-string` (`string`) – обрабатываемая строка

- `list-of-characters` (string) – what characters can be stripped. Default = space.

Возвращается result after stripping characters from input string

Тип возвращаемого значения строка

Пример:

```
tarantool> string = require('string')
---
...
tarantool> string.strip(' ABC ')
---
- ABC
...
```

5.2.26 Module *swim*

Общие сведения

The `swim` module contains Tarantool's implementation of SWIM – Scalable Weakly-consistent Infection-style Process Group Membership Protocol. It is recommended for any type of Tarantool cluster where the number of nodes can be large. Its job is to discover and monitor the other members in the cluster and keep their information in a «member table». It works by sending and receiving, in a background event loop, periodically, via UDP, messages.

Each message has several parts, including:

- the ping such as «I am checking whether you are alive»,
- the event such as «I am joining»,
- the anti-entropy such as «I know that another member exists»,
- the payload such as «I or another member could have user-generated data».

The maximum message size is about 1500 bytes.

SWIM sends messages periodically to a random subset of the member table. SWIM processes replies from those members asynchronously.

Each entry in the member table has:

- a UUID,
- a status («alive», «suspected», «dead», or «left»).

When a member fails to acknowledge a certain number of pings, its status is changed from «alive» to «suspected», that is, suspected of being dead. But SWIM tries to **avoid false positives** (misidentifying members as dead) which could happen when a member is overloaded and responds to pings too slowly, or when there is network trouble and packets can not go through some channels. When a member is suspected, SWIM randomly chooses other members and sends requests to them: «please ping this suspected member». This is called an **indirect ping**. Thus via different routes and additional hops the suspected member gets additional chances to reply, and thus «refute» the suspicion.

Because selection is random there is an **even network load** of about one message per member per protocol step, regardless of the cluster size. This is a major feature of SWIM. Because the protocol depends on members passing information on, also known as «gossiping», members do not need to broadcast messages to every member, which would cause a network load of N messages per member per protocol step, where N is the number of members in the cluster. However, selection is not entirely random, there is a preference for selecting least-recently-pinged members, like a round-robin.

Regarding the **anti-entropy** part of a message: this is necessary for maintaining the status in entries of the member table. Consider an example where two members, #1 and #2, are both alive. No events happen so only pings are being sent periodically. Then a third member, #3 appears. It knows about one of the existing members, #2. How can it discover the other member? Certainly #1 could notify #2 and #2 could notify #3, but messages go via UDP, so any notification event can be lost. However, regular messages containing «ping» and/or «event» also can contain an «anti-entropy» section, which is taken from a randomly-chosen part of the member table. So for this example, #2 will eventually randomly add to a regular message the anti-entropy note that #1 is alive, and thus #3 will discover #1 even though it did not receive a direct «I am alive» event message from #1.

Regarding the **UUID** part of an entry in the member table: this is necessary for stable identification, because UUID changes more rarely than URI (a combination of IP and port number). But if the UUID does change, SWIM will include both the new and old UUID in messages, so all other members will eventually learn about the new UUID and change the member table accordingly.

Regarding the **payload** part of a message: this is not always necessary, it is a feature which allows passing user-generated information via SWIM instead of via node-to-node communication. The swim module has methods for specifying a «payload», which is arbitrary user data with a maximum size of about 1.2 KB. The payload can be anything, and it will be eventually disseminated over the cluster and available at other members. Each member can have its own payload.

Messages can be **encrypted**. Encryption may not be necessary in a closed network but is necessary for safety if the cluster is on the public Internet. Users can specify an encryption algorithm, an encryption mode, and a private key. All parts of all messages (including ping, acknowledgment, event, payload, URI, and UUID) will be encrypted with that private key, as well as a random public key generated for each message to prevent pattern attacks.

In theory the event dissemination speed (the number of hops to pass information throughout the cluster) is `O(log(cluster_size))`. For that and other theoretical information see the Cornell University paper which originally described SWIM.

`swim.new(`*[cfg]*`)`

Create a new SWIM instance. A SWIM instance maintains a member table and interacts with other members. Multiple SWIM instances can be created in a single Tarantool process.

Параметры

- `cfg` (`table`) – an optional configuration parameter.

  If `cfg` is not specified or is nil, then the new SWIM instance is not bound to a socket and has nil attributes, so it cannot interact with other members and only a few methods are valid until *swim_object:cfg()* is called.

  If `cfg` is specified, then the effect is the same as calling `s = swim.new() s:cfg()`, except for generation. For configuration description see *swim_object:cfg()*.

The generation part of `cfg` can only be specified during `new()`, it cannot be specified later during `cfg()`. Generation is part of *incarnation*. Usually generation is not specified because the default value (a timestamp) is sufficient, but if there is reason to mistrust timestamps (because the time is changed or because the instance is started on a different machine), then users may say `swim.new(`*`generation = {new-value`*. In that case the latest value should be persisted somehow (for example in a file, or in a space, or in a global service), and the new value must be greater than any previous value of generation.

возвращает swim-object *a swim object*

Пример:

```
swim_object = swim.new({uri = 3333, uuid = '00000000-0000-1000-8000-000000000001', heartbeat_
↪rate = 0.1})
```

object swim_object

A swim object is an object returned by *swim.new()*. It has methods: *cfg()*, *delete()*, *is_configured()*, *size()*, *quit()*, *add_member()*, *remove_member()*, *probe_member()*, *broadcast()*, *set_payload()*, *set_payload_raw()*, *set_codec()*, *self()*, *member_by_uuid()*, *pairs()*.

swim_object:cfg(*cfg*)

Configure or reconfigure a SWIM instance.

Параметры

- `cfg` (`table`) – the options to describe instance behavior

The `cfg` table may have these components:

- `heartbeat_rate` (double) – rate of sending round messages, in seconds. Setting `heartbeat_rate` to X does not mean that every member will be checked every X seconds, instead X is the protocol speed. Protocol period depends on member count and heartbeat_rate. Default = 1.
- `ack_timeout` (double) – time in seconds after which a ping is considered to be unacknowledged. Default = 30.
- `gc_mode` (enum) – dead member collection mode.

  If `gc_mode == 'off'` then SWIM never removes dead members from the member table (though users may remove them with *swim_object:remove_member()*), and SWIM will continue to ping them as if they were alive.

  If `gc_mode == 'on'` then SWIM removes dead members from the member table after one round.

  Default = `'on'`.
- `uri` (string or number) – either an `'ip:port'` address, or just a port number (if ip is omitted then 127.0.0.1 is assumed). If `port == 0`, then the kernel will select any free port for the IP address.
- `uuid` (string or cdata struct tt_uuid) – a value which should be unique among SWIM instances. Users may choose any value but the recommendation is: use *box.cfg.instance_uuid*, the Tarantool instance's UUID.

All the `cfg` components are dynamic – `swim_object:cfg()` may be called more than once. If it is not being called for the first time and a component is not specified, then the component retains its previous value. If it is being called for the first time then uri and uuid are mandatory, since a SWIM instance cannot operate without URI and UUID.

`swim_object:cfg()` is atomic – if there is an error, then nothing changes.

возвращает true if configuration succeeds

возвращает nil, `err` if an error occurred. `err` is an error object

Пример:

```
swim_object:cfg({heartbeat_rate = 0.5})
```

After `swim_object:cfg()`, all other `swim_object` methods are callable.

.cfg

Expose all non-nil components of the read-only table which was set up or changed by *swim_object:cfg()*.

Пример:

```
tarantool> swim_object.cfg
---
- gc_mode: off
  uri: 3333
  uuid: 00000000-0000-1000-8000-000000000001
...
```

`swim_object:delete()`
Delete a SWIM instance immediately. Its memory is freed, its member table entry is deleted, and it can no longer be used. Other members will treat this member as „dead“.

After `swim_object:delete()` any attempt to use the deleted instance will cause an exception to be thrown.

возвращает none, this method does not fail

Example: `swim_object:delete()`

`swim_object:is_configured()`
Return false if a SWIM instance was created via *swim.new()* without an optional `cfg` argument, and was not configured with *swim_object:cfg()*. Otherwise return true.

возвращает boolean result, true if configured, otherwise false

Example: `swim_object:is_configured()`

`swim_object:size()`
Return the size of the member table. It will be at least 1 because the «self» member is included.

возвращает integer size

Example: `swim_object:size()`

`swim_object:quit()`
Leave the cluster.

This is a graceful equivalent of *swim_object:delete()* – the instance is deleted, but before deletion it sends to each member in its member table a message, that this instance has left the cluster, and should not be considered dead.

Other instances will mark such a member in their tables as „left“, and drop it after one round of dissemination.

Consequences to the caller are the same as after `swim_object:delete()` – the instance is no longer usable, and an error will be thrown if there is an attempt to use it.

возвращает none, the method does not fail

Example: `swim_object:quit()`

`swim_object:add_member(`*cfg*`)`
Explicitly add a member into the member table.

This method is useful when a new member is joining the cluster and does not yet know what members already exist. In that case it can start interaction explicitly by adding the details about an already-existing member into its member table. Subsequently SWIM will discover other members automatically via messages from the already-existing member.

Параметры

- `cfg` (`table`) – description of the member

The `cfg` table has two mandatory components, `uuid` and `uri`, which have the same format as `uuid` and `uri` in the table for *swim_object:cfg()*.

возвращает true if member is added

возвращает nil, `err` if an error occurred. `err` is an error object

Пример:

```
swim_member_object = swim_object:add_member({uuid = ..., uri = ...})
```

`swim_object:remove_member(`*uuid*`)`

Explicitly and immediately remove a member from the member table.

Параметры

- `uuid` (*string-or-cdata-struct-tt_uuid*) – UUID

возвращает true if member is removed

возвращает nil, `err` if an error occurred. `err` is an error object.

Example: `swim_object:delete('00000000-0000-1000-8000-000000000001')`

`swim_object:probe_member(`*uri*`)`

Send a ping request to the specified `uri` address. If another member is listening at that address, it will receive the ping, and respond with an ACK (acknowledgment) message containing information such as UUID. That information will be added to the member table.

`swim_object:probe_member()` is similar to *swim_object:add_member()*, but it does not require UUID, and it is not reliable because it uses UDP.

Параметры

- `uri` (*string-or-number*) – URI. Format is the same as for `uri` in *swim_object:cfg()*.

возвращает true if member is pinged

возвращает nil, `err` if an error occurred. `err` is an error object.

Example: `swim_object:probe_member(3333)`

`swim_object:broadcast(`$\lfloor$*port*$\rfloor$`)`

Broadcast a ping request to all the network interfaces in the system.

`swim_object:broadcast()` is like *swim_object:probe_member()* to many members at once.

Параметры

- `port` (*number*) – All the sent ping requests have this port as destination port in their UDP headers. By default a currently bound port is used.

возвращает true if broadcast is sent

возвращает nil, `err` if an error occurred. `err` is an error object.

Пример:

```
tarantool> fiber = require('fiber')
---
...
tarantool> swim = require('swim')
---
...
tarantool> s1 = swim.new({uri = 3333, uuid = '00000000-0000-1000-8000-000000000001',␣
↪heartbeat_rate = 0.1})
---
```

(continues on next page)

(продолжение с предыдущей страницы)

```
...
tarantool> s2 = swim.new({uri = 3334, uuid = '00000000-0000-1000-8000-000000000002',␣
↪heartbeat_rate = 0.1})
---
...
tarantool> s1:size()
---
- 1
...
tarantool> s1:add_member({uri = s2:self():uri(), uuid = s2:self():uuid()})
---
- true
...
tarantool> s1:size()
---
- 1
...
tarantool> s2:size()
---
- 1
...

tarantool> fiber.sleep(0.2)
---
...
tarantool> s1:size()
---
- 2
...
tarantool> s2:size()
---
- 2
...
tarantool> s1:remove_member(s2:self():uuid()) s2:remove_member(s1:self():uuid())
---
...
tarantool> s1:size()
---
- 1
...
tarantool> s2:size()
---
- 1
...

tarantool> s1:probe_member(s2:self():uri())
---
- true
...
tarantool> fiber.sleep(0.1)
---
...
tarantool> s1:size()
---
- 2
...
```

(continues on next page)

(продолжение с предыдущей страницы)

```
tarantool> s2:size()
---
- 2
...
tarantool> s1:remove_member(s2:self():uuid()) s2:remove_member(s1:self():uuid())
---
...
tarantool> s1:size()
---
- 1
...
tarantool> s2:size()
---
- 1
...
tarantool> s1:broadcast(3334)
---
- true
...
tarantool> fiber.sleep(0.1)
---
...
tarantool> s1:size()
---
- 2
...

tarantool> s2:size()
---
- 2
...
```

`swim_object:set_payload(`*payload*`)`
Set a payload, as formatted data.

Payload is arbitrary user defined data up to 1200 bytes in size and disseminated over the cluster. So each cluster member will eventually learn what is the payload of other members in the cluster, because it is stored in the member table and can be queried with *swim_member_object:payload()*.

Different members may have different payloads.

Параметры

- `payload` (*object*) – Arbitrary Lua object to disseminate. Set to nil to remove the payload, in which case it will be eventually removed on other instances. The object is serialized in MessagePack.

возвращает true if payload is set

возвращает nil, `err` if an error occurred. `err` is an error object

Пример:

```
swim_object:set_payload({field1 = 100, field2 = 200})
```

`swim_object:set_payload_raw(`*payload*`[`, *size* `])`
Set a payload, as raw data.

Sometimes a payload does not need to be a Lua object. For example, a user may already have a

well formatted MessagePack object and just wants to set it as a payload. Or cdata needs to be exposed.

`set_payload_raw` allows setting a payload as is, without MessagePack serialization.

Параметры

- `payload` (*string-or-cdata*) – any value
- `size` (*number*) – Payload size in bytes. If `payload` is string then `size` is optional, and if specified, then should not be larger than actual `payload` size. If `size` is less than actual `payload` size, then only the first `size` bytes of `payload` are used. If `payload` is cdata then `size` is mandatory.

возвращает true if payload is set

возвращает nil, `err` if an error occurred. `err` is an error object

Пример:

```
tarantool> tarantool> ffi = require('ffi')
---
...
tarantool> fiber = require('fiber')
---
...
tarantool> swim = require('swim')
---
...
tarantool> s1 = swim.new({uri = 0, uuid = '00000000-0000-1000-8000-000000000001',␣
↪heartbeat_rate = 0.1})
---
...
tarantool> s2 = swim.new({uri = 0, uuid = '00000000-0000-1000-8000-000000000002',␣
↪heartbeat_rate = 0.1})
---
...
tarantool> s1:add_member({uri = s2:self():uri(), uuid = s2:self():uuid()})
---
- true
...
tarantool> s1:set_payload({a = 100, b = 200})
---
- true
...
tarantool> s2:set_payload('any payload')
---
- true
...
tarantool> fiber.sleep(0.2)
---
...
tarantool> s1_view = s2:member_by_uuid(s1:self():uuid())
---
...
tarantool> s2_view = s1:member_by_uuid(s2:self():uuid())
---
...
tarantool> s1_view:payload()
---
```

(continues on next page)

(продолжение с предыдущей страницы)

```
- {'a': 100, 'b': 200}
...
tarantool> s2_view:payload()
---
- any payload
...
tarantool> cdata = ffi.new('char[?]', 2)
---
...
tarantool> cdata[0] = 1
---
...
tarantool> cdata[1] = 2
---
...
tarantool> s1:set_payload_raw(cdata, 2)
---
- true
...
tarantool> fiber.sleep(0.2)
---
...
tarantool> cdata, size = s1_view:payload_cdata()
---
...
tarantool> cdata[0]
---
- 1
...
tarantool> cdata[1]
---
- 2
...
tarantool> size
---
- 2
...
```

`swim_object:set_codec(`*codec_cfg*`)`

Enable encryption for all following messages.

For a brief description of encryption algorithms see «enum_crypto_algo» and «enum crypto_mode» in the Tarantool source code file crypto.h.

When encryption is enabled, all the messages are encrypted with a chosen private key, and a randomly generated and updated public key.

Параметры

- `codec_cfg` (`table`) – description of the encryption

The components of the `codec_cfg` table may be:

- `algo` (string) – encryption algorithm name. All the names in *module crypto* are supported: „aes128“, „aes192“, „aes256“, „des“. Specify „none“ to disable encryption.
- `mode` (string) – encryption algorithm mode. All the modes in module `crypto` are supported: „ecb“, „cbc“, „cfb“, „ofb“. Default = „cbc“.

- `key` (cdata or string) – a private secret key which is kept secret and should never be stored hard-coded in source code.
- `key_size` (integer) – size of the key in bytes.

  `key_size` is mandatory if key is cdata.

  `key_size` is optional if key is string, and if `key_size` is shorter than than actual key size then the key is truncated.

All of `algo`, `mode`, `key`, and `key_size` should be the same for all SWIM instances, so that members can understand each others“ messages.

Example;

```
tarantool> tarantool> swim = require('swim')
---
...
tarantool> s1 = swim.new({uri = 0, uuid = '00000000-0000-1000-8000-000000000001'})
---
...
tarantool> s1:set_codec({algo = 'aes128', mode = 'cbc', key = '1234567812345678'})
---
- true
...
```

`swim_object:self()`
Return a *swim member object* (of self) from the member table, or from a cache containing earlier results of `swim_object:self()` or `swim_object:member_by_uuid()` or `swim_object:pairs()`.

возвращает *swim member object*, not nil because self() will not fail

Example: `swim_member_object = swim_object:self()`

`swim_object:member_by_uuid(`*uuid*`)`
Return a *swim member object* (given UUID) from the member table, or from a cache containing earlier results of `swim_object:self()` or `swim_object:member_by_uuid()` or `swim_object:pairs()`.

Параметры

- `uuid` (*string-or-cdata-struct-tt-uuid*) – UUID

возвращает *swim member object*, or nil if not found

Пример:

```
swim_member_object = swim_object:member_by_uuid('00000000-0000-1000-8000-000000000001')
```

`swim_object:pairs()`
Set up an iterator for returning *swim member objects* from the member table, or from a cache containing earlier results of `swim_object:self()` or `swim_object:member_by_uuid()` or `swim_object:pairs()`.

`swim_object:pairs()` should be in a „for“ loop, and there should only be one iterator in operation at one time. (The iterator is implemented in an extra light fashion so only one iterator object is available per SWIM instance.)

Параметры

- `generator+object+key` (*varies*) – as for any Lua pairs() iterators. generator function, iterator object (a swim member object), and initial key (a UUID).

Пример:

```
tarantool> fiber = require('fiber')
---
...
tarantool> swim = require('swim')
---
...
tarantool> s1 = swim.new({uri = 0, uuid = '00000000-0000-1000-8000-000000000001',␣
↪heartbeat_rate = 0.1})
---
...
tarantool> s2 = swim.new({uri = 0, uuid = '00000000-0000-1000-8000-000000000002',␣
↪heartbeat_rate = 0.1})
---
...
tarantool> s1:add_member({uri = s2:self():uri(), uuid = s2:self():uuid()})
---
- true
...
tarantool> fiber.sleep(0.2)
---
...
tarantool> s1:self()
---
- uri: 127.0.0.1:55845
  status: alive
  incarnation: cdata {generation = 1569353431853325ULL, version = 1ULL}
  uuid: 00000000-0000-1000-8000-000000000001
  payload_size: 0
...
tarantool> s1:member_by_uuid(s1:self():uuid())
---
- uri: 127.0.0.1:55845
  status: alive
  incarnation: cdata {generation = 1569353431853325ULL, version = 1ULL}
  uuid: 00000000-0000-1000-8000-000000000001
  payload_size: 0
...
tarantool> s1:member_by_uuid(s2:self():uuid())
---
- uri: 127.0.0.1:53666
  status: alive
  incarnation: cdata {generation = 1569353431865138ULL, version = 1ULL}
  uuid: 00000000-0000-1000-8000-000000000002
  payload_size: 0
...
tarantool> t = {}
---
...
tarantool> for k, v in s1:pairs() do table.insert(t, {k, v}) end
---
...
tarantool> t
---
- - - 00000000-0000-1000-8000-000000000002
    - uri: 127.0.0.1:53666
      status: alive
```

(continues on next page)

(продолжение с предыдущей страницы)

```
      incarnation: cdata {generation = 1569353431865138ULL, version = 1ULL}
      uuid: 00000000-0000-1000-8000-000000000002
      payload_size: 0
  - - 00000000-0000-1000-8000-000000000001
    - uri: 127.0.0.1:55845
      status: alive
      incarnation: cdata {generation = 1569353431853325ULL, version = 1ULL}
      uuid: 00000000-0000-1000-8000-000000000001
      payload_size: 0
...
```

object swim_member_object

Methods *swim_object:member_by_uuid()*, *swim_object:self()*, and *swim_object:pairs()* return swim member objects.

A swim member object has methods for reading its attributes: *status()*, *uuid*, *uri()*, *incarnation()*, *payload_cdata*, *payload_str()*, *payload()*, *is_dropped()*.

swim_member_object:status()

Return the status, which may be „alive“, „suspected“, „left“, or „dead“.

возвращает string „alive“ | „suspected“ | „left“ | dead“

swim_member_object:uuid()

Return the UUID as cdata struct tt_uuid.

возвращает cdata-struct-tt-uuid UUID

swim_member_object:uri()

Return the URI as a string „ip:port“. Via this method a user can learn a real assigned port, if port = 0 was specified in *swim_object:cfg()*.

возвращает string ip:port

swim_member_object:incarnation()

Return a cdata object with the *incarnation*. The cdata object has two attributes: incarnation().generation and incarnation().version.

Incarnations can be compared to each other with any comparison operator (==, <, >, <=, >=, ~=).

Incarnations, when printed, will appear as strings with both generation and version.

возвращает cdata incarnation

swim_member_object:payload_cdata()

Return member's payload.

возвращает pointer-to-cdata payload and size in bytes

swim_member_object:payload_str()

Return payload as a string object. Payload is not decoded. It is just returned as a string instead of cdata. If payload was not specified by *swim_object:set_payload()* or by *swim_object:set_payload_raw()*, then its size is 0 and nil is returned.

возвращает string-object payload, or nil if there is no payload

swim_member_object:payload()

Since the `swim` module is a Lua module, a user is likely to use Lua objects as a payload – tables, numbers, strings etc. And it is natural to expect that *swim_member_object:payload()* should return the same object which was passed into *swim_object:set_payload()* by another instance.

`swim_member_object:payload()` tries to interpret payload as MessagePack, and if that fails then it returns the payload as a string.

`swim_member_object:payload()` caches its result. Therefore only the first call actually decodes cdata payload. All following calls return a pointer to the same result, unless payload is changed with a new incarnation. If payload was not specified (its size is 0), then nil is returned.

`swim_member_object:is_dropped()`

Returns true if this member object is a stray reference to a member which has already been dropped from the member table.

возвращает boolean true if member is dropped, otherwise false

Пример:

```
tarantool> swim = require('swim')
---
...
tarantool> s = swim.new({uri = 0, uuid = '00000000-0000-1000-8000-000000000001'})
---
...
tarantool> self = s:self()
---
...
tarantool> self:status()
---
- alive
...
tarantool> self:uuid()
---
- 00000000-0000-1000-8000-000000000001
...
tarantool> self:uri()
---
- 127.0.0.1:56367
...
tarantool> self:incarnation()
---
- - cdata {generation = 1569354463981551ULL, version = 1ULL}
...
tarantool> self:is_dropped()
---
- false
...
tarantool> s:set_payload_raw('123')
---
- true
...
tarantool> self:payload_cdata()
---
- 'cdata<const char *>: 0x0103500050'
- 3
...
tarantool> self:payload_str()
---
- '123'
...
tarantool> s:set_payload({a = 100})
---
```

(continues on next page)

(продолжение с предыдущей страницы)

```
- true
...
tarantool> self:payload_cdata()
---
- 'cdata<const char *>: 0x0103500050'
- 4
...
tarantool> self:payload_str()
---
- !!binary gaFhZA==
...
tarantool> self:payload()
---
- {'a': 100}
...
```

`swim_member_object:on_member_event(`*trigger-function*[, *ctx*]`)`

Create an «on_member *trigger*». The `trigger-function` will be executed when a member in the member table is updated.

Параметры

- `trigger-function` (*function*) – this will become the trigger function
- `ctx` (*cdata*) – (optional) this will be passed to trigger-function

возвращает nil or function pointer.

The **trigger-function** should have three parameter declarations (Tarantool will pass values for them when it invokes the function):

- the member which is having the member event,
- the event object,
- the `ctx` which will be the same value as what is passed to `swim_object:on_member_event`.

A **member event** is any of:

- appearance of a new member,
- drop of an existing member, or
- update of an existing member.

An **event object** is an object which the trigger-function can use for determining what type of member event has happened. The object's methods – such as `is_new_status()`, `is_new_uri()`, `is_new_incarnation()`, `is_new_payload()`, `is_drop()` – return boolean values.

A member event may have more than one associated **trigger**. Triggers are executed sequentially. Therefore if a trigger function causes yields or sleeps, other triggers may be forced to wait. However, since trigger execution is done in a separate fiber, SWIM itself is not forced to wait.

Example of an on-member trigger function:

```
tarantool> swim = require('swim')

local function on_event(member, event, ctx)
    if event:is_new() then
        ...
    elseif event:is_drop() then
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        ...
    end

    if event:is_update() then
        -- All next conditions can be
        -- true simultaneously.
        if event:is_new_status() then
...
        end
        if event:is_new_uri() then
...
        end
        if event:is_new_incarnation() then
...
        end
        if event:is_new_payload() then
...
        end
    end
end
```

Notice in the above example that the function is ready for the possibility that multiple events can happen simultaneously for a single trigger activation. `is_new()` and `is_drop()` can not both be true, but `is_new()` and `is_update()` can both be true, or `is_drop()` and `is_update()` can both be true. Multiple simultaneous events are especially likely if there are many events and trigger functions are slow – in that case, for example, a member might be added and then updated after a while, and then after a while there will be a single trigger activation.

Also: `is_new()` and `is_new_payload()` can both be true. This case is not due to trigger functions that are slow. It occurs because «omitted payload» and «size-zero payload» are not the same thing. For example: when a ping is received, a new member might be added, but ping messages do not include payload. The payload will appear later in a different message. If that is important for the application, then the function should not assume when `is_new()` is true that the member already has a payload, and should not assume that payload size says something about the payload's presence or absence.

Also: functions should not assume that `is_new()` and `is_drop()` will always be seen. If a new member appears but then is dropped before its appearance has caused a trigger activation, then there will be no trigger activation.

`is_new_generation()` will be true if the generation part of *incarnation* changes. `is_new_version()` will be true if the version part of incarnation changes. `is_new_incarnation()` will be true if either the generation part or the version part of incarnation changes. For example a combination of these methods can be used within a user-defined trigger to check whether a process has restarted, or a member has changed ...

```
swim = require('swim')
s = swim.new()
s:on_member_event(function(m, e)
...
    if e:is_new_incarnation() then
        if e:is_new_generation() then
            -- Process restart.
        end
        if e:is_new_version() then
            -- Process version update. It means
```

(continues on next page)

(продолжение с предыдущей страницы)

```
            -- the member is somehow changed.
        end
    end
end
```

`swim_member_object:on_member_event(`*nil*, *old-trigger*`)`
Delete an on-member trigger.

Параметры

- `old-trigger` (*function*) – old-trigger

The old-trigger value should be the value returned by `on_member_event(trigger-function[, ctx])`.

`swim_member_object:on_member_event(`*new-trigger*, *old-trigger*`[`, *ctx*`])`
This is a variation of `on_member_event(new-trigger, [, ctx])`.

The additional parameter is `old-trigger`. Instead of adding the new-trigger at the end of a list of triggers, this function will replace the entry in the list of triggers that matches old-trigger. The position within a list may be important because triggers are activated sequentially starting with the first trigger in the list.

The old-trigger value should be the value returned by `on_member_event(trigger-function[, ctx])`.

`swim_member_object:on_member_event()`
Return the list of on-member triggers.

SWIM internals

The SWIM internals section is not necessary for programmers who wish to use the SWIM module, it is for programmers who wish to change or replace the SWIM module.

The SWIM wire protocol is open, will be backward compatible in case of any changes, and can be implemented by users who wish to simulate their own SWIM cluster members because they use another language than Lua, or another environment unrelated to Tarantool. The protocol is encoded as MsgPack.

```
SWIM packet structure:

+-----------------Public data, not encrypted------------------+
|                                                             |
|      Initial vector, size depends on chosen algorithm.      |
|                  Next data is encrypted.                    |
|                                                             |
+----------Meta section, handled by transport level-----------+
| map {                                                       |
|     0 = SWIM_META_TARANTOOL_VERSION: uint, Tarantool        |
|                                      version ID,            |
|     1 = SWIM_META_SRC_ADDRESS: uint, ip,                    |
|     2 = SWIM_META_SRC_PORT: uint, port,                     |
|     3 = SWIM_META_ROUTING: map {                            |
|         0 = SWIM_ROUTE_SRC_ADDRESS: uint, ip,               |
|         1 = SWIM_ROUTE_SRC_PORT: uint, port,                |
|         2 = SWIM_ROUTE_DST_ADDRESS: uint, ip,               |
|         3 = SWIM_ROUTE_DST_PORT: uint, port                 |
|     }                                                       |
```

(continues on next page)

(продолжение с предыдущей страницы)

```
| }                                                                |
+------------------Protocol logic section--------------------+
| map {                                                            |
|     0 = SWIM_SRC_UUID: 16 byte UUID,                             |
|                                                                  |
|                 AND                                              |
|                                                                  |
|     2 = SWIM_FAILURE_DETECTION: map {                            |
|         0 = SWIM_FD_MSG_TYPE: uint, enum swim_fd_msg_type,       |
|         1 = SWIM_FD_GENERATION: uint,                            |
|         2 = SWIM_FD_VERSION: uint                                |
|     },                                                           |
|                                                                  |
|               OR/AND                                             |
|                                                                  |
|     3 = SWIM_DISSEMINATION: array [                              |
|         map {                                                    |
|             0 = SWIM_MEMBER_STATUS: uint,                        |
|                                 enum member_status,              |
|             1 = SWIM_MEMBER_ADDRESS: uint, ip,                   |
|             2 = SWIM_MEMBER_PORT: uint, port,                    |
|             3 = SWIM_MEMBER_UUID: 16 byte UUID,                  |
|             4 = SWIM_MEMBER_GENERATION: uint,                    |
|             5 = SWIM_MEMBER_VERSION: uint,                       |
|             6 = SWIM_MEMBER_PAYLOAD: bin                         |
|         },                                                       |
|         ...                                                      |
|     ],                                                           |
|                                                                  |
|               OR/AND                                             |
|                                                                  |
|     1 = SWIM_ANTI_ENTROPY: array [                               |
|         map {                                                    |
|             0 = SWIM_MEMBER_STATUS: uint,                        |
|                                 enum member_status,              |
|             1 = SWIM_MEMBER_ADDRESS: uint, ip,                   |
|             2 = SWIM_MEMBER_PORT: uint, port,                    |
|             3 = SWIM_MEMBER_UUID: 16 byte UUID,                  |
|             4 = SWIM_MEMBER_GENERATION: uint,                    |
|             5 = SWIM_MEMBER_VERSION: uint,                       |
|             6 = SWIM_MEMBER_PAYLOAD: bin                         |
|         },                                                       |
|         ...                                                      |
|     ],                                                           |
|                                                                  |
|               OR/AND                                             |
|                                                                  |
|     4 = SWIM_QUIT: map {                                         |
|         0 = SWIM_QUIT_GENERATION: uint,                          |
|         1 = SWIM_QUIT_VERSION: uint                              |
|     }                                                            |
| }                                                                |
+------------------------------------------------------------------+
```

The **Initial vector section** appears only when encryption is enabled. This section contains a public key. For example, for AES algorithms it is a 16-byte initial vector stored as is. When no encryption is used, the section size is 0.

The later sections (Meta and Protocol Logic) are encrypted as one big data chunk if encryption is enabled.

The **Meta section** handles routing and protocol versions compatibility. It works at the „transport“ level of the SWIM protocol, and is always present. Keys in the meta section are:

- SWIM_META_TARANTOOL_VERSION – mandatory field. Tarantool sets here its version as a 3 byte integer:
  - 1 byte for major,
  - 1 byte for minor,
  - 1 byte for patch.

  For example, Tarantool version 2.1.3 would be encoded like this: `(((2 << 8) | 1) << 8) | 3;`. This field will be used to support multiple versions of the protocol.
- SWIM_META_SRC_ADDRESS and SWIM_META_SRC_PORT – mandatory. source IP address and port. IP is encoded as 4 bytes. «xxx.xxx.xxx.xxx» where each „xxx“ is encoding of one byte. Port is encoded as an integer. Example of how to encode «127.0.0.1:3313»:

```
struct in_addr addr;
inet_aton("127.0.0.1", &addr);
pos = mp_encode_uint(pos, SWIM_META_SRC_ADDRESS);
pos = mp_encode_uint(pos, addr->s_addr);
pos = mp_encode_uint(pos, SWIM_META_SRC_PORT);
pos = mp_encode_uint(pos, 3313);
```

- SWIM_META_ROUTING subsection – not mandatory. Responsible for packet forwarding. Used by SWIM suspicion mechanism. Read about suspicion in the SWIM paper.

  If this subsection is present then the following fields are mandatory:
  - SWIM_ROUTE_SRC_ADDRESS and SWIM_ROUTE_SRC_PORT (source IP address and port) (should be an address of the message originator (can differ from
  - SWIM_META_SRC_ADDRESS and from SWIM_META_SRC_ADDRESS_PORT);
  - SWIM_ROUTE_DST_ADDRESS and SWIM_ROUTE_DST_PORT (destination IP address and port, for the the message's final destination).

  If a message was sent indirectly with the help of SWIM_META_ROUTING, then the reply should be sent back by the same route.

  For an example of how SWIM uses routing for indirect pings ... Assume there are 3 nodes: S1, S2, S3. S1 sends a message to S3 via S2. The following steps are executed in order to deliver the message:

```
S1 -> S2
{ src: S1, routing: {src: S1, dst: S3}, body: ... }
```

  S2 receives the message and sees that routing.dst is not equal to S2, so it is a foreign packet. S2 forwards the packet to S3 preserving all the data including body and routing sections.

```
S2 -> S3
```

  S3 receives the message and sees that routing.dst is equal to S3, so the message is delivered. If S3 wants to answer, it sends a response via the same proxy. It knows that the message was delivered from S2, so it sends an answer via S2.

The **Protocol logic section** handles SWIM logical protocol steps and actions.

- SWIM_SRC_UUID – mandatory field. SWIM uses UUID as a unique identifier of a member, not IP/port. This field stores UUID of sender. Its type is MP_BIN. Size is always 16 bytes. UUID is encoded in host byte order, no bswaps are needed.

Following SWIM_SRC_UUID there are four possible subsections: SWIM_FAILURE_DETECTION, SWIM_DISSEMINATION, SWIM_ANTI_ENTROPY, SWIM_QUIT. Any or all of these subsections may be present. A connector should be ready to handle any combination.

- SWIM_FAILURE_DETECTION subsection – describes a ping or ACK. In the SWIM_FAILURE_DETECTION subsection are:
  - SWIM_FD_MSG_TYPE (0 is ping, 1 is ack);
  - SWIM_FD_GENERATION + SWIM_FD_VERSION (the *incarnation*).
- SWIM_DISSEMINATION subsection – a list of changed cluster members. It may include only a subset of changed cluster members if there are too many changes to fit into one UDP packet.

  In the SWIM_DISSEMINATION subsection are:
  - SWIM_MEMBER_STATUS (mandatory) (0 = alive, 1 = suspected, 2 = dead, 3 = left);
  - SWIM_MEMBER_ADDRESS and SWIM_MEMBER_PORT (mandatory) member IP and port;
  - SWIM_MEMBER_UUID (mandatory) (member UUID);
  - SWIM_MEMBER_GENERATION + SWIM_MEMBER_VERSION (mandatory) (the member *incarnation*);
  - SWIM_MEMBER_PAYLOAD (not mandatory) (member payload) (MessagePack type is MP_BIN).

  Note that absence of SWIM_MEMBER_PAYLOAD means nothing - it is not the same as a payload with zero size.
- SWIM_ANTI_ENTROPY subsection – a helper for the dissemination. It contains all the same fields as the dissemination sub, but all of them are mandatory, including payload even when payload size is 0. Anti-entropy eventually spreads changes which for any reason are not spread by the dissemination.
- SWIM_QUIT subsection – statement that the sender has left the cluster gracefully, for example via *swim_object:quit()*, and should not be considered dead. Sender status should be changed to „left“.

  In the SWIM_QUIT subsection are:
  - SWIM_QUIT_GEMERATOPM + SWIM_QUIT_VERSION (the sender *incarnation*).

The **incarnation** is a 128-bit cdata value which is part of each member's configuration and is present in most messages. It has two parts: generation and version.

Generation is persistent. By default it has the number of microseconds since the epoch (compare the value returned by *clock_realtime64()*). Optionally a user can set generation during *new()*.

Version is volatile. It is initially 0. It is incremented automatically every time that a change occurs.

The incarnation, or sometimes the version alone, is useful for deciding to ignore obsolete messages, for updating a member's attributes on remote nodes, and for refuting messages that say a member is dead.

If the member's incarnation is less than the locally stored incarnation, then the message is obsolete. This can happen because UDP allows reordering and duplication.

If the member's incarnation in a message is greater than the locally stored incarnation, then most of its attributes (IP, port, status) should be updated with the values received in the message. However, the payload attribute should not be updated unless it is present in the message. Because of its relatively large size, payload is not always included in every message.

Refutation usually happens when a false-positive failure detection has happened. In such a case the member thought to be dead receives that information from other members, increases its own incarnation, and spreads a message saying the member is alive (a «refutation»).

Note: in the original version of Tarantool SWIM, and in the original SWIM specification, there is no generation and the incarnation consists of only the version. Generation was added because it is useful for detecting obsolete messages left over from a previous life of an instance that has restarted.

5.2.27 Модуль *table*

Модуль `table` включает в себя всё из стандартной библиотеки для работы с таблицами в Lua, а также некоторые расширения специально для Tarantool'а.

Чтобы убедиться в этом, выполните команду «table»: вы увидите список функций: `clear` (расширение LuaJIT = удаление всех элементов), concat (конкатенация), `copy` (создание копии массива), `deepcopy` (см. описание ниже), `foreach`, `foreachi`, getn (получение количества элементов в массиве), insert (вставка элемента в массив), maxn (получение самого большого индекса) move (перемещение элементов между таблицами), `new` (расширение LuaJIT = возврат новой таблицы с предварительно выделенными элементами), remove (удаление элемента из массива), sort (сортировка элементов массива).

В данном разделе мы рассматриваем только дополнительную функцию, добавленную разработчиками Tarantool'а: `deepcopy`.

`table.deepcopy`(*input-table*)

Возврат детальной копии таблицы – копии, которая включает в себя вложенные структуры любой глубины и не зависит от указателей, копируется содержимое.

Параметры

- `input-table` – таблица для копирования

Возвращается копия таблицы

Тип возвращаемого значения таблица

Пример:

```
tarantool> input_table = {1,{'a','b'}}
---
...

tarantool> output_table = table.deepcopy(input_table)
---
...

tarantool> output_table
---
- - 1
  - - a
    - b
...
```

`table.sort`(*input-table*[, *comparison-function*])

Размещение содержимого введенной таблицы в отсортированном порядке.

В базовой сортировке в Lua, table.sort, есть функция сравнения, которая используется по умолчанию: `function (a, b) return a < b end`.

Эта стандартная функция эффективна. Однако иногда пользователям Tarantool'а может понадобиться эквивалент `table.sort` со следующими функциями:

(1) If the table contains nils, except nils at the end, the results must still be correct. That is not the case with the default tarantool_sort, and it cannot be fixed by making a comparison that checks whether a and b are nil. (Before trying certain Internet suggestions, test with {1, nil, 2, -1, 44, 1e308, nil, 2, nil, nil, 0}.

(2) If strings are to be sorted in a language-aware way, there must be a parameter for collation.

(3) If the table has a mix of types, then they must be sorted as booleans, then numbers, then strings, then byte arrays.

Поскольку все эти функции доступны в спейсах Tarantool'а, решение простое: создайте временный спейс в Tarantool'е, поместите в него содержимое таблицы, извлеките из него кортежи по порядку и перезапишите таблицу.

Тогда `tarantool_sort()` сделает то же самое, что и `table.sort`, но с этими дополнительными функциями. Это не быстрый способ, который требует прав на базу данных, поэтому его следует использовать только при необходимости дополнительных функций.

```
function tarantool_sort(input_table, collation)
      local c = collation or 'binary'
      local tmp_name = 'Temporary_for_tarantool_sort'
      pcall(function() box.space[tmp_name]:drop() end)
      box.schema.space.create(tmp_name, {temporary = true})
      box.space[tmp_name]:create_index('I')
      box.space[tmp_name]:create_index('I2',
                                       {unique = false,
                                        type='tree',
                                        parts={{2, 'scalar',
                                                collation = c,
                                                is_nullable = true}}})
      for i = 1, table.maxn(input_table) do
          box.space[tmp_name]:insert{i, input_table[i]}
      end
      local t = box.space[tmp_name].index.I2:select()
      for i = 1, table.maxn(input_table) do
          input_table[i] = t[i][2]
      end
      box.space[tmp_name]:drop()
    end

Например, предположим, что таблица t = {1, 'A', -88.3, nil, true, 'b', 'B', nil, 'À'}.
После tarantool_sort(t, 'unicode_ci') t содержит {nil, nil, true, -88.3, 1, 'A', 'À', 'b', 'B
↪'}.
```

5.2.28 Модуль *tap*

Общие сведения

Модуль `tap` оптимизирует тестирование других модулей. Он позволяет записывать тесты в TAP-протокол (TAP protocol). Результаты тестов могут подвергаться анализу стандартными TAP-анализаторами, поэтому их можно передавать утилитам, например prove. Таким образом, можно выполнять тестирование, а затем использовать результаты для выведения статистики, принятия решений и т.д.

Указатель

Имя	Назначение
tap.test()	Инициализация
taptest:test()	Создание подтеста и вывод результатов
taptest:plan()	Указание количества проводимых тестов
taptest:check()	Проверка количества выполненных тестов
taptest:diag()	Отображение сообщения диагностики
taptest:ok()	Оценка состояния и отображение сообщения
taptest:fail()	Оценка состояния и отображение сообщения
taptest:skip()	Оценка состояния и отображение сообщения
taptest:is()	Проверка равенства двух аргументов
taptest:isnt()	Проверка отличий двух аргументов
taptest:is_deeply()	Рекурсивная проверка равенства двух аргументов
taptest:like()	Проверка соответствия аргумента шаблону
taptest:unlike()	Проверка отличия аргумента от шаблона
taptest:isnil() *taptest:isstring()* *taptest:isnumber()* *taptest:istable()* *taptest:isboolean()* *taptest:isudata()* *taptest:iscdata()*	Проверка соответствия значения определенному типу
taptest.strict	Flag, true if comparisons with nil should be strict

`tap.test`(*test-name*)
Инициализация.

Результатом `tap.test` является объект, который будет называться taptest в ходе данного разбора, что необходимо для `taptest:plan()` и всех остальных методов.

Параметры

- `test-name` (`string`) – произвольное имя для результата теста.

возвращает taptest

тип возвращаемого значения *table*

```
tap = require('tap')
taptest = tap.test('test-name')
```

`object taptest`

`taptest:test`(*test-name*, *func*)
Создание подтеста (если не указан аргумент `func`) или (если указаны все аргументы) создание подтеста, выполнение тестовой функции и вывод результата.

См. *пример*.

Параметры

- `name` (`string`) – произвольное имя для результата теста.
- `fun` (*`function`*) – выполняемая тестовая логика.

возвращает taptest

тип возвращаемого значения userdata или строка

`taptest:plan(`*count*`)`
Указание количества проводимых тестов.

Параметры

- `count` (*`number`*) –

возвращает nil

`taptest:check()`
Проверка количества выполненных тестов.

Выведенный результат будет включать в себя сообщение: `# bad plan: ...`, если количество выполненных тестов не равно количеству тестов, указанному в `taptest:plan(...)`. (Это собственная функция Tarantool'а: сообщения типа «bad plan» не входят в стандарт TAP13.)

Такую проверку следует проводить только по завершении всех запланированных тестов, поэтому как правило, `taptest:check()` появится лишь в конце скрипта. Тем не менее, в качестве расширения Tarantool'а, `taptest:check()` может появиться в начале любого подтеста. Таким образом, проверка появится в трех случаях:

- при вызове `taptest:check()` в конце скрипта,
- при вызове функции, которая заканчивается вызовом `taptest:check()`,
- или при вызове taptest:test(„...“, имя-функции-подтеста), где функция подтеста не обязана заканчиваться на `taptest:check()`, поскольку ее можно вызвать по окончании подтеста.

возвращает true (правда) или false (ложь).

тип возвращаемого значения boolean (логический)

`taptest:diag(`*message*`)`
Отображение сообщения диагностики.

Параметры

- `message` (`string`) – отображаемое сообщение.

возвращает nil

`taptest:ok(`*condition*, *test-name*`)`
Это базовая функция, которая используется другими функциями. В зависимости от условия `condition`, выводится „ok“ или „not ok“ вместе с отладочной информацией. Отображается сообщение.

Параметры

- `condition` (*`boolean`*) – выражение, которое либо true (правда), либо false (ложь)
- `test-name` (`string`) – имя теста

возвращает true (правда) или false (ложь).

тип возвращаемого значения boolean (логический)

Пример:

```
tarantool> taptest:ok(true, 'x')
ok - x
---
- true
...
tarantool> tap = require('tap')
---
...
tarantool> taptest = tap.test('test-name')
TAP version 13
---
...
tarantool> taptest:ok(1 + 1 == 2, 'X')
ok - X
---
- true
...
```

`taptest:fail(`*test-name*`)`

`taptest:fail('x')` – аналог `taptest:ok(false, 'x')`. Отображается сообщение.

Параметры

- `test-name` (`string`) – имя теста

возвращает true (правда) или false (ложь).

тип возвращаемого значения boolean (логический)

`taptest:skip(`*message*`)`

`taptest:skip('x')` – аналог `taptest:ok(true, 'x' .. '# skip')`. Отображается сообщение.

Параметры

- `test-name` (`string`) – имя теста

возвращает nil

Пример:

```
tarantool> taptest:skip('message')
ok - message # skip
---
- true
...
```

`taptest:is(`*got*, *expected*, *test-name*`)`

Проверка равенства первого аргумента второму аргументу. Отображается подробное сообщение, если результатом будет false (ложь).

Параметры

- `got` (*number*) – фактический результат
- `expected` (*number*) – ожидаемый результат
- `test-name` (`string`) – имя теста

возвращает true (правда) или false (ложь).

тип возвращаемого значения boolean (логический)

`taptest:isnt(`*got*, *expected*, *test-name*`)`
Отрицание *taptest:is()*.

Параметры

- `got` (*number*) – фактический результат
- `expected` (*number*) – ожидаемый результат
- `test-name` (string) – имя теста

возвращает true (правда) или false (ложь).

тип возвращаемого значения boolean (логический)

`taptest:is_deeply(`*got*, *expected*, *test-name*`)`
Рекурсивная версия `taptest:is(...)`, которую можно использовать для сопоставления таблиц, а также скалярных значений.

возвращает true (правда) или false (ложь).

тип возвращаемого значения boolean (логический)

Параметры

- `got` (*lua-value*) – фактический результат
- `expected` (*lua-value*) – ожидаемый результат
- `test-name` (string) – имя теста

`taptest:like(`*got*, *expected*, *test-name*`)`
Проверка совпадения строки с шаблоном. Ok, если найдено совпадение.

возвращает true (правда) или false (ложь).

тип возвращаемого значения boolean (логический)

Параметры

- `got` (*lua-value*) – фактический результат
- `expected` (*lua-value*) – шаблон
- `test-name` (string) – имя теста

```
test:like(tarantool.version, '^[1-9]', "version")
```

`taptest:unlike(`*got*, *expected*, *test-name*`)`
Отрицание *taptest:like()*.

Параметры

- `got` (*number*) – фактический результат
- `expected` (*number*) – шаблон
- `test-name` (string) – имя теста

возвращает true (правда) или false (ложь).

тип возвращаемого значения boolean (логический)

`taptest:isnil(`*value*, *test-name*`)`
`taptest:isstring(`*value*, *test-name*`)`

taptest:isnumber(*value*, *test-name*)
taptest:istable(*value*, *test-name*)
taptest:isboolean(*value*, *test-name*)
taptest:isudata(*value*, *test-name*)
taptest:iscdata(*value*, *test-name*)
Проверка соответствия значения определенному типу. Отображается длинное сообщение, если значение не принадлежит указанному типу.

Параметры

- value (*lua-value*) –
- test-name (string) – имя теста

возвращает true (правда) или false (ложь).

тип возвращаемого значения boolean (логический)

taptest.strict
Set `taptest.strict=true` if *taptest:is()* and *taptest:isnt()* and *taptest:is_deeply()* must be compared strictly with `nil`. Set `taptest.strict=false` if `nil` and `box.NULL` both have the same effect. The default is false. For example, if and only if `taptest.strict=true` has happened, then `taptest:is_deeply({a = box.NULL}, {})` will return `false`.

Пример

Для выполнения данного примера поместите скрипт в файл под названием ./tap.lua, затем сделайте tap.lua выполняемым файлом с помощью команды `chmod a+x ./tap.lua`, а затем выполните его, используя Tarantool в качестве обработчика скриптов после выполнения команды ./tap.lua.

```
#!/usr/bin/tarantool
local tap = require('tap')
test = tap.test("my test name")
test:plan(2)
test:ok(2 * 2 == 4, "2 * 2 is 4")
test:test("some subtests for test2", function(test)
    test:plan(2)
    test:is(2 + 2, 4, "2 + 2 is 4")
    test:isnt(2 + 3, 4, "2 + 3 is not 4")
end)
test:check()
```

Результатом вышеприведенного скрипта будет примерно следующее:

```
TAP version 13
1..2
ok - 2 * 2 is 4
    # Some subtests for test2
    1..2
    ok - 2 + 2 is 4,
    ok - 2 + 3 is not 4
    # Some subtests for test2: end
ok - some subtests for test2
```

5.2.29 Модуль *tarantool*

Выполнив команду `require('tarantool')`, можно получить ответы на вопросы о том, как был собран Tarantool-сервер, например, какие флаги были использованы, или какая версия компилятора использовалась.

Кроме того, можно проверить время работы и версию сервера, а также идентификатор процесса. Эту информацию также можно получить с помощью *box.info()*, но рекомендуется использовать модуль tarantool.

Пример:

```
tarantool> tarantool = require('tarantool')
---
...
tarantool> tarantool
---
- version: 2.3.0-3-g302bb3241
  build:
    target: Linux-x86_64-RelWithDebInfo
    options: cmake . -DCMAKE_INSTALL_PREFIX=/opt/tarantool-install
-DENABLE_BACKTRACE=ON
    mod_format: so
    flags: ' -fexceptions -funwind-tables -fno-omit-frame-pointer
-fno-stack-protector
      -fno-common -fopenmp -msse2 -std=c11 -Wall -Wextra
-Wno-strict-aliasing -Wno-char-subscripts
      -Wno-format-truncation -fno-gnu89-inline -Wno-cast-function-type'
    compiler: /usr/bin/cc /usr/bin/c++
  pid: 'function: 0x40016cd0'
  package: Tarantool
  uptime: 'function: 0x40016cb0'
...
tarantool> tarantool.pid()
---
- 30155
...
tarantool> tarantool.uptime()
---
- 108.64641499519
...
```

5.2.30 Модуль *uuid*

Общие сведения

UUID – это Универсальный уникальный идентификатор (Universally unique identifier). Если значение должно быть уникальным в пределах отдельного компьютера или одной базы данных, лучше использовать простой счетчик вместо UUID, поскольку получение UUID затратно по времени (требуется syscall). Что же касается кластеров компьютеров или широко распространенных приложений, лучше использовать UUID.

Указатель

Ниже приведен перечень всех функций и элементов модуля `uuid`.

Имя	Назначение
uuid.nil	Объект nil
uuid() uuid.bin() uuid.str()	Получение UUID
uuid.fromstr() *uuid.frombin()* *uuid_ object:bin()* *uuid_ object:str()*	Получение конвертированного UUID
uuid_ object:isnil()	Проверка, состоит ли UUID из одних нулей

`uuid.nil`
Объект nil

`uuid.__call()`

возвращает UUID

тип возвращаемого значения cdata.

`uuid.bin()`

возвращает UUID

тип возвращаемого значения 16-байтная строка

`uuid.str()`

возвращает UUID

тип возвращаемого значения 36-байтная двоичная строка

`uuid.fromstr(`*uuid_ str*`)`

Параметры

- `uuid_str` – UUID в 36-байтной шестнадцатеричной строке

возвращает конвертированный UUID

тип возвращаемого значения cdata.

`uuid.frombin(`*uuid_ bin*`)`

Параметры

- `uuid_str` – UUID в 16-байтной двоичной строке

возвращает конвертированный UUID

тип возвращаемого значения cdata.

`object uuid_object`

`uuid_object:bin(`[*byte-order*]`)`
`byte-order` может быть одним из следующих флагов:

- „l“ - порядок от младшего к старшему,
- „b“ - порядок от старшего к младшему,
- „h“ - порядок зависит от хоста (по умолчанию),
- „n“ - порядок зависит от сети

Параметры

- `byte-order` (`string`) – один из `'l'`, `'b'`, `'h'` или `'n'`.

возвращает UUID, сконвертированный из введенного значения формата cdata.

тип возвращаемого значения 16-байтная двоичная строка

`uuid_object:str()`

возвращает UUID, сконвертированный из введенного значения формата cdata.

тип возвращаемого значения 36-байтная шестнадцатеричная строка

`uuid_object:isnil()`
Значение UUID из одних нулей может быть выражено как uuid.NULL или `uuid.fromstr('00000000-0000-0000-0000-000000000000')`. Сравнение со значением из одних нулей также может быть выражено как `uuid_with_type_cdata == uuid.NULL`.

возвращает true (правда), если значение состоит из одних нулей, в противном случае false (ложь).

тип возвращаемого значения bool (логический)

Пример

```
tarantool> uuid = require('uuid')
---
...
tarantool> uuid(), uuid.bin(), uuid.str()
---
- 16ffedc8-cbae-4f93-a05e-349f3ab70baa
- !!binary FvG+Vy1MfUC6kIyeM81DYw==
- 67c999d2-5dce-4e58-be16-ac1bcb93160f
...
tarantool> uu = uuid()
---
...
tarantool> #uu:bin(), #uu:str(), type(uu), uu:isnil()
---
- 16
- 36
- cdata
- false
...
```

5.2.31 Модуль *utf8*

Общие сведения

`utf8` – это модуль Tarantool'а для обработки строк в формате UTF-8. Он содержит некоторые функции, которые совместимы с функциями Lua 5.3, но возможности Tarantool'а намного больше. Например, поскольку Tarantool включает в себя полную копию библиотеки Международных компонентов для Юникода («International Components For Unicode»), доступны также функции сравнения, которые понимают упорядочение символов в кириллице (заглавная буква Ж = строчная буква ж) и японском языке (А в хирагане = А в катакане).

Имя	Назначение
casecmp and *cmp*	Сравнения
lower and *upper*	Замена регистра
isalpha, *isdigit*, *islower* and *isupper*	Определение типа символа
sub	Подстроки
len	Длина в символах
next	Посимвольная итерация

`utf8.casecmp(`*UTF8-string*, *utf8-string*`)`

Параметры

- `string` (*`UTF8-string`*) – строка в формате UTF-8

возвращает -1 означает «меньше», 0 означает «равно», +1 означает «больше»

тип возвращаемого значения число

Сравнение двух строк с Таблицей сортировки символов Юникода по умолчанию (DUCET) для Алгоритма сортировки по Юникоду (Unicode Collation Algorithm). В результате „å“ меньше, чем „B“, хотя значение кодовой точки å (229) больше значения кодовой точки B (66), поскольку алгоритм основывается на значениях Таблица сортировки символов, а не на значениях кодовых точек.

Сравнение осуществляется на основании основного веса. Таким образом, не учитываются элементы, которые влияют на вторичный или последующий вес (такие как «регистр» в латинице или кириллице, или «отличия каны» в японском языке). Если спросить: «Это похоже на сортировку без учета регистра и ударения от компании Майкрософт?» - ответом будет: «Скорее да», хотя Алгоритм сортировки по Юникоду гораздо сложнее, чем это описание.

Пример:

```
tarantool> utf8.casecmp('é','e'),utf8.casecmp('E','e')
---
- 0
- 0
...
```

`utf8.char(`*code-point*`[`, *code-point ...* `])`

Параметры

- `number` (*`code-point`*) – значение кодовой точки в Юникоде, повторяется

возвращает строка в UTF-8

тип возвращаемого значения строка

Число кодовой точки – это значение, которое соответствует символу в Базе данных символов Юникода This is not the same as the byte values of the encoded character, because the UTF-8 encoding scheme is more complex than a simple copy of the code-point number.

Другой способ создать строку с символами Юникода – с помощью механизма экранирования символов \u{шестнадцатеричные-числа}, например, в результате и „\u{41}\u{42}“, и `utf8.char(65, 66)` получим строку „AB“.

Пример:

```
tarantool> utf8.char(229)
---
- å
...
```

utf8.cmp(*UTF8-string*, *utf8-string*)

Параметры

- string (*UTF8-string*) – строка в формате UTF-8

возвращает -1 означает «меньше», 0 означает «равно», +1 означает «больше»

тип возвращаемого значения число

Сравнение двух строк с Таблицей сортировки символов Юникода по умолчанию (DUCET) для Алгоритма сортировки по Юникоду (Unicode Collation Algorithm). В результате „å“ меньше, чем „B“, хотя значение кодовой точки å (229) больше значения кодовой точки B (66), поскольку алгоритм основывается на значениях Таблица сортировки символов, а не на значениях кода.

Сравнение осуществляется на основании не менее трех значений веса. Таким образом, не учитываются элементы, которые влияют на вторичный или последующий вес (такие как «регистр» в латинице или кириллице, или «отличия каны» в японском языке), а верхний регистр следует за нижним.

Пример:

```
tarantool> utf8.cmp('é','e'),utf8.cmp('E','e')
---
- 1
- 1
...
```

utf8.isalpha(*UTF8-character*)

Параметры

- string-or-number (*UTF8-character*) – отдельный символ UTF8, выраженный в виде однобайтной строки или значения кодовой точки

возвращает true (правда) или false (ложь)

тип возвращаемого значения boolean (логический)

Возврат true (правда), если введенный символ является буквенным, в остальных случаях – false (ложь). В целом, символ считается буквенным, если он используется в рамках слова, а не как число или знак пунктуации. Такой символ необязательно должен быть буквой алфавита.

Пример:

```
tarantool> utf8.isalpha('Ж'),utf8.isalpha('å'),utf8.isalpha('9')
---
- true
- true
- false
...
```

utf8.isdigit(*UTF8-character*)

Параметры

- string-or-number (*UTF8-character*) – отдельный символ UTF8, выраженный в виде однобайтной строки или значения кодовой точки

возвращает true (правда) или false (ложь)

тип возвращаемого значения boolean (логический)

Возврат true (правда), если введенный символ является цифрой, в остальных случаях – false (ложь).

Пример:

```
tarantool> utf8.isdigit('Ж'),utf8.isdigit('å'),utf8.isdigit('9')
---
- false
- false
- true
...
```

`utf8.islower(`*UTF8-character*`)`

Параметры

- `string-or-number` (*`UTF8-character`*) – отдельный символ UTF8, выраженный в виде однобайтной строки или значения кодовой точки

возвращает true (правда) или false (ложь)

тип возвращаемого значения boolean (логический)

Возврат true (правда), если введенный символ относится к нижнему регистру, в остальных случаях – false (ложь).

Пример:

```
tarantool> utf8.islower('Ж'),utf8.islower('å'),utf8.islower('9')
---
- false
- true
- false
...
```

`utf8.isupper(`*UTF8-character*`)`

Параметры

- `string-or-number` (*`UTF8-character`*) – отдельный символ UTF8, выраженный в виде однобайтной строки или значения кодовой точки

возвращает true (правда) или false (ложь)

тип возвращаемого значения boolean (логический)

Возврат true (правда), если введенный символ относится к верхнему регистру, в остальных случаях – false (ложь).

Пример:

```
tarantool> utf8.isupper('Ж'),utf8.isupper('å'),utf8.isupper('9')
---
- true
- false
- false
...
```

`utf8.len(`*UTF8-string*`[, `*start-byte*`[, `*end-byte*`]])`

Параметры

- `string` (*UTF8-string*) – строка в формате UTF-8
- `integer` (*end-byte*) – позиция байта первого символа
- `integer` – позиция байта для остановки

возвращает количество символов в строке или же от начала до конца

тип возвращаемого значения число

Позиции байта в начале и в конце могут быть отрицательными, что указывает на отсчет с конца строки, а не с начала.

Если строка содержит последовательность байтов, которая неприменима для UTF-8, каждый байт в неправильной последовательности будет считаться за один символ.

UTF-8 представляет собой схему кодирования изменяемого размера. Как правило, одна буква латиницы занимает один байт, буква кириллицы занимает два байта, а символ из китайского или японского языка занимает три байта, максимальный размер – четыре байта.

Пример:

```
tarantool> utf8.len('G'),utf8.len('ж')
---
- 1
- 1
...

tarantool> string.len('G'),string.len('ж')
---
- 1
- 2
...
```

`utf8.lower(`*UTF8-string*`)`

Параметры

- `string` (*UTF8-string*) – строка в формате UTF-8

возвращает та же строка в нижнем регистре

тип возвращаемого значения строка

Пример:

```
tarantool> utf8.lower('ÅΓЖABCDEFG')
---
- åγжabcdefg
...
```

`utf8.next(`*UTF8-string*`[,` *start-byte* `])`

Параметры

- `string` (*UTF8-string*) – строка в формате UTF-8
- `integer` (*start-byte*) – позиция байта внутри строки, с которой начать выполнение, по умолчанию = 1

возвращает позиция байта следующего символа и значение кодовой точки следующего символа

тип возвращаемого значения таблица

Функция `next` часто используется в цикле для получения символа за раз из строки в формате UTF-8.

Пример:

В строке „åa“ первый символ – „å“, он начинается в позиции 1, занимает два байта, поэтому символ после него будет на позиции 3, значение кодовой точки в Юникоде (десятичное) – 229.

```
tarantool> -- показать позицию следующего символа + кодовую точку первого символа
tarantool> utf8.next('åa', 1)
---
- 3
- 229
...
tarantool> -- (цикл) показать кодовую точку каждого символа
tarantool> for position,codepoint in utf8.next,'åa' do print(codepoint) end
229
97
...
```

`utf8.sub(`*UTF8-string*, *start-character*[, *end-character*]`)`

Параметры

- `string` (*UTF8-string*) – строка в формате UTF-8
- `number` (*end-character*) – позиция первого символа
- `number` – позиция последнего символа

возвращает строка в формате UTF-8, «подстрока» введенного значения

тип возвращаемого значения строка

Позиции символа в начале и в конце могут быть отрицательными, что указывает на отсчет с конца строки, а не с начала.

Значение end-character по умолчанию – длина введенной строки. Таким образом, выполнение `utf8.sub(1, 'abc')` вернет „abc“, т.е. введенную строку.

Пример:

```
tarantool> utf8.sub('åγжabcdefg', 5, 8)
---
- abcd
...
```

`utf8.upper(`*UTF8-string*`)`

Параметры

- `string` (*UTF8-string*) – строка в формате UTF-8

возвращает та же строка в верхнем регистре

тип возвращаемого значения строка

Примечание: В редких случаях результат в верхнем регистре может быть длиннее введенной строки в нижнем регистре, например, `utf8.upper('ß')` вернет „SS“.

Пример:

```
tarantool> utf8.upper('åγжabcdefg')
---
- ÅΓЖABCDEFG
...
```

5.2.32 Модуль *uri*

Общие сведения

URI – это Унифицированный идентификатор ресурса (Uniform Resource Identifier). Согласно стандарту IETF, URI-строка выглядит следующим образом:

```
[схема:]специальная-часть-схемы[#фрагмент]
```

Общий тип, иерархический URI, выглядит так:

```
[схема:][//адрес][путь][?запрос][#фрагмент]
```

Например, строка `'https://tarantool.org/x.html#y'` содержит три компонента:

- `https` – схема,
- `tarantool.org/x.html` – путь,
- `y` – фрагмент.

Модуль Tarantool'a URI включает в себя процедуры для разложения URI-строк на компоненты или объединения компонентов в URI-строку.

Указатель

Ниже приведен перечень всех функций модуля `uri`.

Имя	Назначение
uri.parse()	Получение таблицы URI-компонентов
uri.format()	Создание URI из компонентов

`uri.parse(`*URI-string*`)`

Параметры

- `URI-string` – Унифицированный идентификатор ресурса

возвращает таблица с компонентами URI. Доступные компоненты: fragment (фрагмент), host (хост), login (имя для входа), password (пароль), path (путь), query (запрос), scheme (схема), service (сервис).

тип возвращаемого значения Таблица

Пример:

```
tarantool> uri = require('uri')
---
...

tarantool> uri.parse('http://x.html#y')
```

(continues on next page)

(продолжение с предыдущей страницы)

```
---
- host: x.html
  scheme: http
  fragment: y
...
```

uri.format(*URI-components-table*[, *include-password*])

Параметры

- URI-components-table – ряд пар ключ-значение, одна для каждого компонента
- include-password – логическое значение. Если указать значение true, то компонент пароля отображается открытым текстом, в остальных случаях не отображается.

возвращает URI-строка. Таким образом, uri.format() – это операция, обратная uri.parse().

тип возвращаемого значения строка

Пример:

```
tarantool> uri.format({host = 'x.html', scheme = 'http', fragment = 'y'})
---
- http://x.html#y
...
```

5.2.33 Модуль *xlog*

Модуль xlog включает в себя одну функцию: pairs(). Ее можно использовать для чтения *файлов снимка* или *файлов журнала упреждающей записи (WAL)* в Tarantool'е. Описание формата файла дается в разделе *Персистентность данных и формат WAL-файла*.

xlog.pairs([*file-name*])

Открытие файла и итерация по одной записи файла за раз.

возвращает итератор, который можно использовать в цикле for / end.

тип возвращаемого значения итератор

Возможные ошибки: Файл не содержит снимок в правильном формате или информацию журнала упреждающей записи.

Пример:

В данном примере производится чтение первого WAL-файла, который был создан в директории *wal_dir* в рамках наших *упражнений в «Руководстве для начинающих»*.

Каждый результат из pairs() выводится в формате MsgPack, поэтому его структуру можно указать с помощью *__serialize*.

```
xlog = require('xlog')
t = {}
for k, v in xlog.pairs('00000000000000000000.xlog') do
  table.insert(t, setmetatable(v, { __serialize = "map"}))
end
return t
```

Первые строки результата будут выглядеть следующим образом:

```
(...)
---
- - {'BODY':   {'space_id': 272, 'index_base': 1, 'key': ['max_id'],
                 'tuple': [['+', 2, 1]]},
     'HEADER': {'type': 'UPDATE', 'timestamp': 1477846870.8541,
                 'lsn': 1, 'server_id': 1}}
  - {'BODY':   {'space_id': 280,
                  'tuple': [512, 1, 'tester', 'memtx', 0, {}, []]},
     'HEADER': {'type': 'INSERT', 'timestamp': 1477846870.8597,
                 'lsn': 2, 'server_id': 1}}
```

5.2.34 Модуль *yaml*

Общие сведения

Модуль `yaml` берет строки в формате YAML и декодирует их или берет ряд значений в ином формате и кодирует их в формат YAML.

Указатель

Ниже приведен перечень всех функций и элементов модуля `yaml`.

Имя	Назначение
yaml.encode()	Конвертация Lua-объекта в YAML-строку
yaml.decode()	Конвертация YAML-строки в Lua-объект
__serialize parameter	Output structure specification
yaml.cfg()	Change configuration
yaml.NULL	Аналог «nil» в языке Lua

`yaml.encode(`*lua_value*`)`
Конвертация Lua-объекта в YAML-строку.

Параметры

- `lua_value` – скалярное значение или значение из Lua-таблицы.

возвращает оригинальное значение, преобразованное в YAML-строку.

тип возвращаемого значения строка

`yaml.decode(`*string*`)`
Конвертация YAML-строки в Lua-объект.

Параметры

- `string` – строка в формате YAML.

возвращает оригинальное содержание в формате Lua-таблицы.

тип возвращаемого значения таблица

__serialize parameter:

The YAML output structure can be specified with `__serialize`:

- „seq“, „sequence“, „array“ - table encoded as an array

- „map“, „mappping“ - table encoded as a map
- function - the meta-method called to unpack serializable representation of table, cdata or userdata objects

„seq“ or „map“ also enable the flow (compact) mode for the YAML serializer (flow=»[1,2,3]» vs block=» - 1n - 2n - 3n»).

Serializing „A“ and „B“ with different `__serialize` values brings different results:

```
tarantool> yaml.encode(setmetatable({'A', 'B'}, { __serialize="seq"}))
---
- '["A","B"]'
...
tarantool> yaml.encode(setmetatable({'A', 'B'}, { __serialize="map"}))
---
- '{"1":"A","2":"B"}'
...
tarantool> yaml.encode({setmetatable({f1 = 'A', f2 = 'B'}, { __serialize="map"})})
---
- '[{"f2":"B","f1":"A"}]'
...
tarantool> yaml.encode({setmetatable({f1 = 'A', f2 = 'B'}, { __serialize="seq"})})
---
- '[[]]'
...
```

`yaml.cfg`(*table*)

Set values affecting the behavior of encode and decode functions.

The values are all either integers or boolean `true`/`false`.

Option	Default	Назначение
`cfg.encode_invalid_numbers`	true	A flag saying whether to enable encoding of NaN and Inf numbers
`cfg.encode_number_precision`	14	Precision of floating point numbers
`cfg.encode_load_metatables`	true	A flag saying whether the serializer will follow *__serialize* metatable field
`cfg.encode_use_tostring`	false	A flag saying whether to use `tostring()` for unknown types
`cfg.encode_invalid_as_nil`	false	A flag saying whether to use NULL for non-recognized types
`cfg.encode_sparse_convert`	true	A flag saying whether to handle excessively sparse arrays as maps. See detailed description *below*
`cfg.encode_sparse_ratio`	2	1/`encode_sparse_ratio` is the permissible percentage of missing values in a sparse array
`cfg.encode_sparse_safe`	10	A limit ensuring that small Lua arrays are always encoded as sparse arrays (instead of generating an error or encoding as map)
`cfg.decode_invalid_numbers`	true	A flag saying whether to enable decoding of NaN and Inf numbers
`cfg.decode_save_metatables`	true	A flag saying whether to set metatables for all arrays and maps

Sparse arrays features:

During encoding, The YAML encoder tries to classify table into one of four kinds:

- map - at least one table index is not unsigned integer
- regular array - all array indexes are available
- sparse array - at least one array index is missing
- excessively sparse array - the number of values missing exceeds the configured ratio

An array is excessively sparse when **all** the following conditions are met:

- `encode_sparse_ratio` > 0
- `max(table)` > `encode_sparse_safe`
- `max(table)` > `count(table)` * `encode_sparse_ratio`

The YAML encoder will never consider an array to be excessively sparse when `encode_sparse_ratio = 0`. The `encode_sparse_safe` limit ensures that small Lua arrays are always encoded as sparse arrays. By default, attempting to encode an excessively sparse array will generate an error. If `encode_sparse_convert` is set to `true`, excessively sparse arrays will be handled as maps.

yaml.cfg() example 1:

The following code will encode 0/0 as NaN («not a number») and 1/0 as Inf («infinity»), rather than returning nil or an error message:

```
yaml = require('yaml')
yaml.cfg{encode_invalid_numbers = true}
x = 0/0
y = 1/0
yaml.encode({1, x, y, 2})
```

The result of the `yaml.encode()` request will look like this:

```
tarantool> yaml.encode({1, x, y, 2})
---
- '[1,nan,inf,2]
...
```

yaml.cfg example 2:

To avoid generating errors on attempts to encode unknown data types as userdata/cdata, you can use this code:

```
tarantool> httpc = require('http.client').new()
---
...

tarantool> yaml.encode(httpc.curl)
---
- error: unsupported Lua type 'userdata'
...

tarantool> yaml.encode(httpc.curl, {encode_use_tostring=true})
---
- '"userdata: 0x010a4ef2a0"'
...
```

Примечание: To achieve the same effect for only one call to `yaml.encode()` (i.e. without changing the configuration permanently), you can use `yaml.encode({1, x, y, 2}, {encode_invalid_numbers =`

true}).

Similar configuration settings exist for *JSON* and *MsgPack*.

yaml.NULL
: Значение, сопоставимое с нулевым значением «nil» в языке Lua, которое можно использовать в качестве объекта-заполнителя в кортеже.

Пример

```
tarantool> yaml = require('yaml')
---
...
tarantool> y = yaml.encode({'a', 1, 'b', 2})
---
...
tarantool> z = yaml.decode(y)
---
...
tarantool> z[1], z[2], z[3], z[4]
---
- a
- 1
- b
- 2
...
tarantool> if yaml.NULL == nil then print('hi') end
hi
---
...
```

Набор YAML-стилей можно указать с помощью `__serialize`:

- `__serialize="sequence"` для массива последовательности блоков,
- `__serialize="seq"` для массива последовательности потоков,
- `__serialize="mapping"` для ассоциативного массива последовательности блоков,
- `__serialize="map"` для ассоциативного массива последовательности потоков.

Сериализация „A“ и“ B“ различными значениями `__serialize` приводит к различным результатам:

```
tarantool> yaml = require('yaml')
---
...

tarantool> yaml.encode(setmetatable({'A', 'B'}, { __serialize="sequence"}))
---
- '---

  - A

  - B

  ...

  '
```

(continues on next page)

(продолжение с предыдущей страницы)

```
...

tarantool> yaml.encode(setmetatable({'A', 'B'}, { __serialize="seq"}))
---
- '--- [''A'', ''B'']

  ...

  '
...

tarantool> yaml.encode({setmetatable({f1 = 'A', f2 = 'B'}, { __serialize="map"})})
---
- '---

  - {''f2'': ''B'', ''f1'': ''A''}

  ...

  '
...
```

5.2.35 Other package components

All the Tarantool modules are, at some level, inside a package which, appropriately, is named `package`. There are also miscellaneous functions and variables which are outside all modules.

Имя	Назначение
tonumber64()	Конвертация строки или Lua-числа в 64-битное целое число
dostring()	Анализ и выполнение произвольного Lua-кода
package.path	Where Tarantool looks for Lua additions
package.cpath	Where Tarantool looks for C additions
package.loaded	What Tarantool has already looked for and found
package.setsearchroot	Set the root path for a directory search
package.searchroot	Get the root path for a directory search

`tonumber64`(*value*)

Конвертация строки или Lua-числа в 64-битное целое число. Входное значение может быть выражено десятичным, двоичным (например, 0b1010) или шестнадцатеричным (например, -0xffff) числом. Результат может использоваться в арифметике, причем скорее в 64-битной целочисленной арифметике, а не в арифметике в системе с плавающей запятой. (Операции с неконвертированными Lua-числами выполняются в арифметике в системе с плавающей запятой.) Функция `tonumber64()` в Tarantool'е является глобальной.

Пример:

```
tarantool> type(123456789012345), type(tonumber64(123456789012345))
---
- number
- number
...
tarantool> i = tonumber64('1000000000')
---
```

(continues on next page)

(продолжение с предыдущей страницы)

```
...
tarantool> type(i), i / 2, i - 2, i * 2, i + 2, i % 2, i ^ 2
---
- number
- 500000000
- 999999998
- 2000000000
- 1000000002
- 0
- 1000000000000000000
...
```

Warning: There is an underlying LuaJIT library that operates with C rules. Therefore you should expect odd results if you compare unsigned and signed (for example 0ULL > -1LL is false), or if you use numbers outside the 64-bit integer range (for example 9223372036854775808LL is negative). Also you should be aware that `type(`*`number-literal-ending-in-ULL`*`)` and `type(tonumber64(`*`number-with-more-than-14-digits`*`))` is cdata, not a Lua arithmetic type, which prevents direct use with some functions in Lua libraries such as math. See the LuaJIT reference and look for the phrase «64 bit integer arithmetic». and the phrase «64 bit integer comparison». Or see the comments on Issue#4089.

`dostring(`*lua-chunk-string*`[, `*lua-chunk-string-argument ...*`])`

Анализ и выполнение произвольного Lua-кода. Данная функция используется преимущественно для определения и выполнения Lua-кода без необходимости внесения изменений в глобальное Lua-окружение.

Параметры

- `lua-chunk-string` (`string`) – Lua-код
- `lua-chunk-string-argument` (*`lua-value`*) – ноль или другие скалярные значения, которые заменяются или к которым прибавляются значения.

возвращает то, что возвращает Lua-код.

Возможные ошибки: Ошибка компиляции появляется как Lua-ошибка.

Пример:

```
tarantool> dostring('abc')
---
error: '[string "abc"]:1: ''='' expected near ''<eof>'''
...
tarantool> dostring('return 1')
---
- 1
...
tarantool> dostring('return ...', 'hello', 'world')
---
- hello
- world
...
tarantool> dostring([[
         >   local f = function(key)
         >     local t = box.space.tester:select{key}
         >     if t ~= nil then
         >       return t[1]
         >     else
```

(continues on next page)

(продолжение с предыдущей страницы)

```
         >       return nil
         >     end
         >   end
         >   return f(...)]], 1)
---
- null
...
```

`package.path`
: This is a string that Tarantool uses to search for Lua modules, especially imporant for `require()`. See *Modules, rocks and applications*.

`package.cpath`
: This is a string that Tarantool uses to search for C modules, especially imporant for `require()`. See *Modules, rocks and applications*.

`package.loaded`
: This is a string that shows what Lua or C modules Tarantool has loaded, so that their functions and members are available. Initially it has all the pre-loaded modules, which don't need `require()`.

`package.setsearchroot(`[*search-root*]`)`
: Set the search root. The search root is the root directory from which dependencies are loaded.

  Параметры

  - `search-root` (`string`) – the path. Default = current directory.

  The search-root string must contain a relative or absolute path. If it is a relative path, then it will be expanded to an absolute path. If search-root is omitted, or is box.NULL, then the search root is reset to the current directory, which is found with debug.sourcedir().

  Пример:

  Suppose that a Lua file `myapp/init.lua` is the project root. Suppose the current path is `/home/tara`. Add this as the first line of `myapp/init.lua`: `package.setsearchroot()` Start the project with `$ tarantool myapp/init.lua` The search root will be the default, made absolute: `/home/tara/myapp`. Within the Lua application all dependencies will be searched relative to `/home/tara/myapp`.

`package.searchroot()`
: Return a string with the current search root. After `package.setsearchroot('/home')` the returned string will be `/home'`.

5.2.36 Коды ошибок базы данных

В текущей версии бинарного протокола в ответы сервера не включены сообщения об ошибках, которые как правило, содержат больше информации, чем коды ошибок. Само сообщение может содержать имя файла, подробное описание причины или код ошибки операционной системы. Однако все такие сообщения регистрируются в журнале ошибок. Ниже приведены общие описания некоторых распространенных кодов. Полный список ошибок можно найти в файле errcode.h в исходном дереве.

Список кодов ошибок

ER_NONMASTER	(Репликация) Экземпляр сервера не может вносить изменения в данные, если он не является мастером.
ER_ILLEGAL_PARAMS	Недопустимые параметры. Некорректное протокольное сообщение.
ER_MEMORY_ISSUE	Нехватка оперативной памяти: достижение предела памяти *memtx_memory*.
ER_WAL_IO	Запись на диск не удалась. Может означать, что не удалось записать изменение в журнале упреждающей записи. Некоторая ошибка на диске.
ER_KEY_PART_COUNT	Количество частей ключа не совпадает с количеством частей индекса
ER_NO_SUCH_SPACE	Указанный спейс отсутствует.
ER_NO_SUCH_INDEX	Указанного индекса нет в указанном спейсе.
ER_PROC_LUA	Возникла ошибке в Lua-процедуре.
ER_FIBER_STACK	При создании нового файбера был достигнут предел рекурсии. Обычно это указывает на то, что хранимая процедура слишком часто рекурсивно вызывает себя.
ER_UPDATE_FIELD	Возникла ошибка во время обновления поля.
ER_TUPLE_FOUND	В уникальном индексе есть повторяющийся ключ.

5.2.37 Обработка ошибок

Ниже представлены несколько процедур для более надежного вызова Lua-функций в случае ошибок, в частности, ошибок базы данных.

1. Вызов с помощью pcall.

   Используйте механизмы Lua для «Обработки ошибок и исключений», в частности `pcall`. То есть вместо простого вызова функции с помощью
   `box.space.`*`имя-спейса`*`:`*`имя-функции`*`()`
   выполните
   `if pcall(box.space.`*`имя-спейса`*`.`*`имя-функции`*`, box.space.`*`имя-спейса`*`) ...`

   Для некоторых функций модуля box в Tarantool'е pcall также вернет описание ошибки, включая имя файла и номер строки в исходном коде Tarantool'а. Например:

   ```
   x, y = pcall(function() box.schema.space.create('') end)
   y:unpack()
   ```

   Чтобы увидеть применение pcall в приложении, см. практическое задание *Подсчет суммы по JSON-полям во всех кортежах*.

2. Проверка и вызов ошибки с помощью box.error.

   В модуле box.error предусмотрена функция *box.error(code, errtext [, errtext ...])*, чтобы создать ошибку и передать ее.

   Чтобы найти последнюю ошибку, в модуле box.error предусмотрена функция *box.error.last()*. (Также можно найти текст последней ошибки операционной системы для определенной функции – *errno.strerror([code])*.)

3. Запись в журнал.

   Записывайте сообщения в журнал с помощью *модуля log*.

И отфильтровывайте автоматически созданные сообщения с помощью конфигурационного параметра *log*.

Как правило, встроенные функции Tarantool'а, которые предназначены для возврата объектов, вернут либо объект, либо нулевое значение nil, либо Lua-ошибку. Например, рассмотрим программу *fio_read.lua* из рекомендаций по разработке:

```
#!/usr/bin/env tarantool

local fio = require('fio')
local errno = require('errno')
local f = fio.open('/tmp/xxxx.txt', {'O_RDONLY' })
if not f then
    error("Failed to open file: "..errno.strerror())
end
local data = f:read(4096)
f:close()
print(data)
```

После вызова функции, который может не сработать, как fio.open() выше, обычно можно увидеть такой синтаксис, как `if not f then ...` или `if f == nil then ...`, который проверяет на типичные отказы. Но если есть ошибка синтаксиса, например, fio.opex вместо fio.open, то появится Lua-ошибка, и f не изменится. Если речь идет о проверке таких очевидных ошибок, программист вероятно будет использовать pcall().

Все функции в модулях Tarantool'а должны работать таким образом, если в руководстве явно не говорится об обратном.

5.2.38 Средства отладки

Общие сведения

Пользователи Tarantool'а могут воспользоваться преимуществами встроенных средств отладки, которые составляют часть:

- Lua (библиотека отладки, см. подробное описание ниже) и
- LuaJit (функции отладки debug.*).

Библиотека `debug` предоставляет интерфейс для отладки Lua-программ. Все функции этой библиотеки содержатся в таблице `debug`. В функциях для работы с потоками есть дополнительный первый параметр, в котором указывается необходимый поток. По умолчанию, это всегда текущий поток.

Примечание: Библиотеку следует использовать только для отладки и профилирования, а не в качестве программного средства, поскольку данные функции выполняются слишком долго. Кроме того, некоторые из этих функций могут привести к нарушению работы безопасного в других отношениях кода.

Указатель

Ниже приведен перечень всех функций библиотеки `debug`.

Имя	Назначение
debug.debug()	Вход в интерактивный режим
debug.getfenv()	Получение среды объекта
debug.gethook()	Получение текущих настроек ловушки потока
debug.getinfo()	Получение информации о функции
debug.getlocal()	Получение имени и значения локальной переменной
debug.getmetatable()	Получение метатаблицы объекта
debug.getregistry()	Получение таблицы реестра
debug.getupvalue()	Получение имени и значения сопоставляющего значения
debug.setfenv()	Определение среды объекта
debug.sethook()	Определение данной функции в качестве ловушки
debug.setlocal()	Присваивание значения локальной переменной
debug.setmetatable()	Определение метатаблицы объекта
debug.setupvalue()	Присваивание значения сопоставляющему значению
debug.sourcedir()	Get the source directory name
debug.sourcefile()	Get the source file name
debug.traceback()	Получение обратной трассировки стека вызовов

`debug.debug()`

Вход в интерактивный режим и выполнение каждой строки, которую печатает пользователь. Пользователь может, в частности, проверять глобальные и локальные переменные, изменять их значения и вычислять выражения.

Введите `cont` для выхода из данной функции, чтобы вызывающий клиент мог продолжить выполнение.

Примечание: Команды для `debug.debug()` не вложены лексически в какую-либо функцию, поэтому у них нет прямого доступа к локальным переменным.

`debug.getfenv(`*object*`)`

Параметры

- `object` – объект, для которого будет получена среда

возвращает среда объекта `object`

`debug.gethook(`[*thread*]`)`

возвращает текущие настройки ловушки потока `thread` в виде трех значений:

- текущая функция-ловушка
- текущая маска ловушки
- текущий счетчик ловушки, как определяет функция `debug.sethook()`

`debug.getinfo(`[*thread*]`,` *function*[`,` *what*]`)`

Параметры

- `function` – функция, по которой будет получена информация
- `what` (`string`) – какую информацию о функции `function` вернуть

возвращает таблица с информацией о функции `function`

Можно передать функцию `function` напрямую или же передать число, которое указывает на функцию, выполняемую на уровне `function` стека вызовов данного потока `thread`: уровень 0 –

это текущая функция (сама функция `getinfo()`), уровень 1 – это функция, которая вызвала `getinfo()`, и т.д. Если для функции `function` указано число больше числа активных функций, `getinfo()` вернет `nil`.

По умолчанию, `what` – это вся доступная информация, кроме таблицы допустимых строк. Если задать опцию `f`, добавится поле под названием `func` с самой функцией. Если задать опцию `L`, добавится поле под названием `activelines` с таблицей доступных строк.

`debug.getlocal(`[*thread*], *level*, *local*`)`

Параметры

- `level` (*number*) – уровень стека
- `local` (*number*) – индекс локальной переменной

возвращает имя и значение локальной переменной с индексом `local` функции на уровне `level` стека или `nil`, если нет локальной переменной с указанным индексом; появится ошибка, если уровень `level` вне диапазона

Примечание: Можно вызвать `debug.getinfo()` для проверки доступности уровня.

`debug.getmetatable(`*object*`)`

Параметры

- `object` – объект, для которого будет получена метатаблица

возвращает метатаблица объекта `object` или `nil`, если метатаблица отсутствует

`debug.getregistry()`

возвращает таблица реестра

`debug.getupvalue(`*func*, *up*`)`

Параметры

- `func` (*function*) – функция, для которой будет получено сопоставляющее значение
- `up` (*number*) – индекс сопоставляющего значения функции

возвращает имя и значение сопоставляющего значения с индексом `up` функции `func` или `nil`, если нет сопоставляющего значения в пределах заданного индекса

`debug.setfenv(`*object*, *table*`)`
Определение среды объекта `object` для таблицы `table`.

Параметры

- `object` – объект, среда которого будет изменена
- `table` (`table`) – таблица для определения среды объекта

возвращает объект `object`

`debug.sethook(`[*thread*], *hook*, *mask*[, *count*]`)`
Определение данной функции в качестве ловушки. При вызове без аргументов ловушка отключается.

Параметры

- `hook` (*function*) – функция, которая будет определена в качестве ловушки

- `mask` (`string`) – описание того, когда будет вызвана ловушка `hook`; может принимать следующие значения: * `c` – ловушка“hook“ вызывается каждый раз, когда Lua вызывает функцию * `r` – ловушка `hook` вызывается каждый раз, когда Lua возвращается из функции * `l` – ловушка `hook` вызывается каждый раз, когда Lua переходит на новую строку кода
- `count` (*number*) – описание того, когда будет вызвана ловушка `hook`; если отличается от нуля, ловушка `hook` вызывается после каждой инструкции `count`.

`debug.setlocal`([*thread*], *level*, *local*, *value*)

Присвоение значения `value` локальной переменной с индексом `local` функции на уровне `level` стека

Параметры

- `level` (*number*) – уровень стека
- `local` (*number*) – индекс локальной переменной
- `value` – значение, присваиваемое локальной переменной

возвращает имя локальной переменной или `nil`, если локальная переменная с заданным индексом отсутствует; возникает ошибка, если уровень `level` вне диапазона

Примечание: Можно вызвать `debug.getinfo()` для проверки доступности уровня.

`debug.setmetatable`(*object*, *table*)

Определение метатаблицы объекта `object` для таблицы `table`.

Параметры

- `object` – объект, метатаблица которого будет изменена
- `table` (`table`) – таблица для определения метатаблицы объекта

`debug.setupvalue`(*func*, *up*, *value*)

Присвоение значения `value` сопоставляющему значению с индексом `up` функции `func`.

Параметры

- `func` (*function*) – функция, для которой будет определено сопоставляющее значение
- `up` (*number*) – индекс сопоставляющего значения функции
- `value` – значение, присваиваемое сопоставляющему значению функции

возвращает имя сопоставляющего значения или `nil`, если сопоставляющее значение с данным индексом отсутствует

`debug.sourcedir`([*level*])

Параметры

- `level` (*number*) – the level of the call stack which should contain the path (default is 2)

возвращает a string with the relative path to the source file directory

Instead of `debug.sourcedir()` one can say `debug.__dir__` which means the same thing.

Determining the real path to a directory is only possible if the function was defined in a Lua file (this restriction may not apply for loadstring() since Lua will store the entire string in debug info).

If `debug.sourcedir()` is part of a `return` argument, then it should be inside parentheses: `return (debug.sourcedir())`.

`debug.sourcefile(`[*level*]`)`

Параметры

- `level` (*number*) – the level of the call stack which should contain the path (default is 2)

возвращает a string with the relative path to the source file

Instead of `debug.sourcefile()` one can say `debug.__file__` which means the same thing.

Determining the real path to a file is only possible if the function was defined in a Lua file (this restriction may not apply to `loadstring()` since Lua will store the entire string in debug info).

If `debug.sourcefile()` is part of a `return` argument, then it should be inside parentheses: `return (debug.sourcefile())`.

`debug.traceback(`[*thread*]`,` [*message*][`,` *level*]`)`

Параметры

- `message` (string) – необязательное сообщение, добавленное к началу обратной трассировки
- `level` (*number*) – указывает на каком уровне начинать обратную трассировку (по умолчанию, 1)

возвращает строка с обратной трассировкой стека вызовов

Debug example:

Make a file in the /tmp directory named example.lua, containing:

```
function w()
  print(debug.sourcedir())
  print(debug.sourcefile())
  print(debug.traceback())
  print(debug.getinfo(1)['currentline'])
end
w()
```

Execute `tarantool /tmp/example.lua`. Expect to see this:

```
/tmp
/tmp/example.lua
stack traceback:
    /tmp/example.lua:4: in function 'w'
    /tmp/example.lua:7: in main chunk
5
```

5.3 Справочник по сторонним библиотекам

В данном справочнике описаны сторонние Lua-модули для Tarantool'а.

5.3.1 Модули СУБД SQL

В данном разделе справочника рассматривается внедрение и использование двух уже созданных модулей: сторонние библиотеки СУБД SQL для MySQL и PostgreSQL.

Для вызова другой СУБД из Tarantool'а нужно: другая СУБД и Tarantool. Модуль, который соединяет другую СУБД может называться коннектором. В модуле есть библиотека общего пользования, которая может называться драйвером.

Tarantool предоставляет модули-коннекторы для СУБД вместе с менеджером модулей для Lua под названием LuaRocks.

Модули Tarantool'а позволяют подключаться к SQL-серверам и выполнять SQL-запросы так же, как это делает клиент MySQL или PostgreSQL. Операторы SQL доступны как Lua-методы. Таким образом, Tarantool может служить Lua-коннектором для MySQL или Lua-коннектором для PostgreSQL, что было бы полезно, даже если бы Tarantool больше ничего не умел. Но конечно же, Tarantool также представляет собой СУБД, поэтому модуль используется для любых операций, таких как копирование и ускорение базы данных, которые максимально эффективно, если приложение может работать как с SQL, так и с Tarantool в пределах одной Lua-процедуры. Методы подключения / выборки / вставки / и т.д. аналогичны методам модуля *net.box*.

С точки зрения пользователя, модули для MySQL и PostgreSQL очень похожи, поэтому следующие разделы – «Пример для MySQL» и «Пример для PostgreSQL» – слегка избыточны.

Пример для MySQL

В данном примере предполагается, что установлены MySQL 5.5, MySQL 5.6 или MySQL 5.7. Последние версии MariaDB также подойдут, используется коннектор к MariaDB для C. Самым важным пакетом будет пакет для разработчиков клиента MySQL, который обычно называется libmysqlclient-dev. Наиболее важным файлом из этого пакета будет файл libmysqlclient.so или с похожим названием. Можно использовать " find" или " whereis", чтобы узнать, в каких директориях установлены эти файлы.

Также нужно будет установить библиотеку общего пользования Tarantool'а с драйвером для MySQL, загрузить ее и использовать для подключения к экземпляру MySQL-сервера. После этого можно передавать любой оператор MySQL на экземпляр сервера и получать результаты, включая наборы результатов.

Установка

Проверьте инструкции по загрузке и установке бинарного пакета, которые применимы к среде, где установлен Tarantool. Помимо установки `tarantool`, установите `tarantool-dev`. Например, в Ubuntu добавьте строку:

```
$ sudo apt-get install tarantool-dev
```

Что касается библиотеки общего пользования с драйвером для MySQL, ее можно установить двумя способами:

Из LuaRocks

Начните с установки luarocks. Убедитесь, что tarantool указан в серверах, как описано на странице сторонних модулей Tarantool'а rocks.tarantool.org. Затем выполните:

```
luarocks install mysql [MYSQL_LIBDIR = path]
                       [MYSQL_INCDIR = path]
                       [--local]
```

Пример:

```
$ luarocks install mysql MYSQL_LIBDIR=/usr/local/mysql/lib
```

Из GitHub

Перейдите по ссылке github.com/tarantool/mysql. Следуя инструкциям, введите команду:

```
$ git clone https://github.com/tarantool/mysql.git
 $ cd mysql && cmake . -DCMAKE_BUILD_TYPE=RelWithDebInfo
 $ make
 $ make install
```

На данном этапе желательно проверить, что после установки появился файл под названием `driver.so`, а также проверить, что этот файл находится в директории, которую можно найти по запросу `require`.

Подключение

Начните с выполнения запроса `require` для драйвера mysql. В дальнейших примерах у него будет имя `mysql`.

```
mysql = require('mysql')
```

Теперь выполните:

```
*имя_подключения* = mysql.connect(*параметры подключения*)
```

Параметры подключения включены в таблицу. Доступные параметры:

- `host = `*`имя-хоста`* – строка, значение по умолчанию = „localhost“
- `port = `*`номер-порта`* – число, значение по умолчанию = 3306
- `user = `*`имя-пользователя`* – строка, значение по умолчанию – имя пользователя в операционной системе
- `password = `*`пароль`* – строка, по умолчанию пустая
- `db = `*`имя-базы-данных`* – строка, по умолчанию пустая
- `raise = `*`true/false`* – логическое значение, по умолчанию, false (ложь)

Имена параметров, за исключением *raise*, похожи на имена, которые используются в MySQL-клиенте mysql, для получения подробной информации см. руководство по MySQL по ссылке dev.mysql.com/doc/refman/5.6/en/connecting.html. Значение параметра *raise* следует указать как true, если ошибки должны возникать при обнаружении. Чтобы подключиться по Unix-сокету, а не по TCP, укажите `host = 'unix/'` и `port = `*`имя-сокета`*.

Пример с использованием таблицы, заключенной в {фигурные скобки}:

```
conn = mysql.connect({
    host = '127.0.0.1',
    port = 3306,
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    user = 'p',
    password = 'p',
    db = 'test',
    raise = true
 })
-- ИЛИ
conn = mysql.connect({
    host = 'unix/',
    port = '/var/run/mysqld/mysqld.sock'
})
```

Пример с созданием функции, которая определяет параметры в отдельных строках:

```
tarantool> -- Функция подключения. Использование: conn = mysql_connect()
tarantool> function mysql_connection()
         >   local p = {}
         >   p.host = 'widgets.com'
         >   p.db = 'test'
         >   conn = mysql.connect(p)
         >   return conn
         > end
---
...
tarantool> conn = mysql_connect()
---
...
```

Предполагаем, что в дальнейших примерах будет использоваться имя „conn“.

Как проверить связь

Чтобы убедиться, что подключение работает, следует использовать запрос:

```
*имя-соединение*:ping()
```

Пример:

```
tarantool> conn:ping()
---
- true
...
```

Исполнение оператора

Для всех операторов MySQL запрос будет:

```
*имя-соединения*:execute(*sql-оператор* [, *параметры*])
```

где `sql-statement` – это строка, а необязательные параметры – это дополнительные значения, которыми можно заменить любые знаки вопроса («?») в SQL-операторе.

Пример:

```
tarantool> conn:execute('select table_name from information_schema.tables')
---
- - table_name: ALL_PLUGINS
  - table_name: APPLICABLE_ROLES
  - table_name: CHARACTER_SETS
  <...>
- 78
...
```

Закрытие соединения

Чтобы закрыть сессию, которую открыли с помощью `mysql.connect`, используется следующий запрос:

```
*имя-соединения*:close()
```

Пример:

```
tarantool> conn:close()
---
...
```

Для получения дополнительной информации, включая примеры редко используемых запросов, см. файл README.md по ссылке github.com/tarantool/mysql.

Пример

Пример выполняется на машине с ОС Ubuntu 12.04 (Precise Pangolin), где Tarantool установлен в поддиректорию /usr, а копия MySQL установлена в ~/mysql-5.5. Экземпляр сервера mysqld уже запущен на localhost 127.0.0.1.

```
$ export TMDIR=~/mysql-5.5
$ # Check that the include subdirectory exists by looking
$ # for .../include/mysql.h. (If this fails, there's a chance
$ # that it's in .../include/mysql/mysql.h instead.)
$ [ -f $TMDIR/include/mysql.h ] && echo "OK" || echo "Error"
OK

$ # Check that the library subdirectory exists and has the
$ # necessary .so file.
$ [ -f $TMDIR/lib/libmysqlclient.so ] && echo "OK" || echo "Error"
OK

$ # Check that the mysql client can connect using some factory
$ # defaults: port = 3306, user = 'root', user password = '',
$ # database = 'test'. These can be changed, provided one uses
$ # the changed values in all places.
$ $TMDIR/bin/mysql --port=3306 -h 127.0.0.1 --user=root \
    --password= --database=test
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 25
Server version: 5.5.35 MySQL Community Server (GPL)
...
Type 'help;' or '\h' for help. Type '\c' to clear ...
```

(continues on next page)

(продолжение с предыдущей страницы)

```
$ # Insert a row in database test, and quit.
mysql> CREATE TABLE IF NOT EXISTS test (s1 INT, s2 VARCHAR(50));
Query OK, 0 rows affected (0.13 sec)
mysql> INSERT INTO test.test VALUES (1,'MySQL row');
Query OK, 1 row affected (0.02 sec)
mysql> QUIT
Bye

$ # Install luarocks
$ sudo apt-get -y install luarocks | grep -E "Setting up|already"
Setting up luarocks (2.0.8-2) ...

$ # Set up the Tarantool rock list in ~/.luarocks,
$ # following instructions at rocks.tarantool.org
$ mkdir ~/.luarocks
$ echo "rocks_servers = {[[http://rocks.tarantool.org/]]}" >> \
    ~/.luarocks/config.lua

$ # Ensure that the next "install" will get files from Tarantool
$ # master repository. The resultant display is normal for Ubuntu
$ # 12.04 precise
$ cat /etc/apt/sources.list.d/tarantool.list
deb http://tarantool.org/dist/2.1/ubuntu/ precise main
deb-src http://tarantool.org/dist/2.1/ubuntu/ precise main

$ # Install tarantool-dev. The displayed line should show version = 2.1
$ sudo apt-get -y install tarantool-dev | grep -E "Setting up|already"
Setting up tarantool-dev (2.1.0.222.g48b98bb~precise-1) ...
$

$ # Use luarocks to install locally, that is, relative to $HOME
$ luarocks install mysql MYSQL_LIBDIR=/usr/local/mysql/lib --local
Installing http://rocks.tarantool.org/mysql-scm-1.rockspec...
... (more info about building the Tarantool/MySQL driver appears here)
mysql scm-1 is now built and installed in ~/.luarocks/

$ # Ensure driver.so now has been created in a place
$ # tarantool will look at
$ find ~/.luarocks -name "driver.so"
~/.luarocks/lib/lua/5.1/mysql/driver.so

$ # Change directory to a directory which can be used for
$ # temporary tests. For this example we assume that the name
$ # of this directory is /home/pgulutzan/tarantool_sandbox.
$ # (Change "/home/pgulutzan" to whatever is the user's actual
$ # home directory for the machine that's used for this test.)
$ cd /home/pgulutzan/tarantool_sandbox

$ # Start the Tarantool server instance. Do not use a Lua initialization file.

$ tarantool
tarantool: version 2.1.0-222-g48b98bb
type 'help' for interactive help
tarantool>
```

Настройте Tarantool и загрузите модуль mysql. Убедитесь, что Tarantool не выбрасывает ошибку в ответ на вызов «require()».

```
tarantool> box.cfg{}
...
tarantool> mysql = require('mysql')
---
...
```

Создайте Lua-функцию, которая подключится к экземпляру MySQL-сервера (используя значения по умолчанию для параметров порта, пользователя и пароля), выберите одну строку и выведите ее на экран. Описание используемых здесь типов операторов вы можете найти в практикуме по Lua в руководстве пользователя Tarantool'а.

```
tarantool> function mysql_select ()
         >   local conn = mysql.connect({
         >     host = '127.0.0.1',
         >     port = 3306,
         >     user = 'root',
         >     db = 'test'
         >   })
         >   local test = conn:execute('SELECT * FROM test WHERE s1 = 1')
         >   local row = ''
         >   for i, card in pairs(test) do
         >       row = row .. card.s2 .. ' '
         >       end
         >   conn:close()
         >   return row
         > end
---
...
tarantool> mysql_select()
---
- 'MySQL row '
...
```

Просмотрите результат. В нем есть строка «MySQL row». Это и есть строка, которая была вставлена в базу данных MySQL. А сейчас она выделена с помощью Tarantool-клиента.

Пример для PostgreSQL

В данном примере предполагается, что установлены PostgreSQL 8 или PostgreSQL 9. Более поздние версии также должны сработать. Самым важным пакетом будет пакет для разработчиков клиента PostgreSQL, который обычно называется libpq-dev. На Ubuntu его можно установить следующим образом:

```
$ sudo apt-get install libpq-dev
```

Однако, не все платформы одинаковы, поэтому в данном примере предполагается, что пользователь должен проверить наличие нужных PostgreSQL-файлов, а также явным образом прописать, где они находятся, для сборки драйвера Tarantool/PostgreSQL. Для поиска директорий, где установлены PostgreSQL-файлы, можно воспользоваться командами `find` или `whereis`.

Также нужно будет установить библиотеку общего пользования Tarantool'а с драйвером для PostgreSQL, загрузить ее и использовать для подключения к экземпляру PostgreSQL-сервера. После этого можно передавать любой оператор PostgreSQL на экземпляр сервера и получать результаты.

Установка

Проверьте инструкции по загрузке и установке бинарного пакета, которые применимы к среде, где установлен Tarantool. Помимо установки `tarantool`, установите `tarantool-dev`. Например, в Ubuntu добавьте строку:

```
$ sudo apt-get install tarantool-dev
```

Что касается библиотеки общего пользования с драйвером для PostgreSQL, ее можно установить двумя способами:

Из LuaRocks

Начните с установки luarocks. Убедитесь, что tarantool указан в серверах, как описано на странице сторонних модулей Tarantool'а rocks.tarantool.org. Затем выполните:

```
luarocks install pg [POSTGRESQL_LIBDIR = *путь*]
                    [POSTGRESQL_INCDIR = *путь*]
                    [--local]
```

Пример:

```
$ luarocks install pg POSTGRESQL_LIBDIR=/usr/local/postgresql/lib
```

Из GitHub

Перейдите по ссылке github.com/tarantool/pg. Следуя инструкциям, введите команду:

```
$ git clone https://github.com/tarantool/pg.git
$ cd pg && cmake . -DCMAKE_BUILD_TYPE=RelWithDebInfo
$ make
$ make install
```

На данном этапе желательно проверить, что после установки появился файл под названием `driver.so`, а также проверить, что этот файл находится в директории, которую можно найти по запросу `require`.

Подключение

Начните с выполнения запроса `require` для драйвера pg. В дальнейших примерах у него будет имя `pg`.

```
pg = require('pg')
```

Теперь выполните:

```
*имя_подключения* = pg.connect(*параметры подключения*)
```

Параметры подключения включены в таблицу. Доступные параметры:

- `host = `*`имя-хоста`* – строка, значение по умолчанию = „localhost“
- `port = `*`номер-порта`* – число, значение по умолчанию = 5432

- `user = имя-пользователя` – строка, значение по умолчанию – имя пользователя в операционной системе
- `pass = пароль` или `password = пароль` – строка, по умолчанию пустая
- `db = имя-базы-данных` – строка, по умолчанию пустая

Имена параметров похожи на имена, которые используются в PostgreSQL.

Пример с использованием таблицы, заключенной в {фигурные скобки}:

```
conn = pg.connect({
    host = '127.0.0.1',
    port = 5432,
    user = 'p',
    password = 'p',
    db = 'test'
})
```

Пример с созданием функции, которая определяет параметры в отдельных строках:

```
tarantool> function pg_connect()
         >   local p = {}
         >   p.host = 'widgets.com'
         >   p.db = 'test'
         >   p.user = 'postgres'
         >   p.password = 'postgres'
         >   local conn = pg.connect(p)
         >   return conn
         > end
---
...
tarantool> conn = pg_connect()
---
...
```

Предполагаем, что в дальнейших примерах будет использоваться имя „conn“.

Как проверить связь

Чтобы убедиться, что подключение работает, следует использовать запрос:

```
*имя-соединение*:ping()
```

Пример:

```
tarantool> conn:ping()
---
- true
...
```

Исполнение оператора

Для всех операторов PostgreSQL запрос будет:

```
*имя-соединения*:execute(*sql-оператор* [, *параметры*])
```

где `sql-statement` – это строка, а необязательные параметры – это дополнительные значения, которыми можно заменить любые местозаполнители ($1 $2 $3 и т.д.) в SQL-операторе.

Пример:

```
tarantool> conn:execute('select tablename from pg_tables')
---
- - tablename: pg_statistic
  - tablename: pg_type
  - tablename: pg_authid
  <...>
...
```

Закрытие соединения

Чтобы закрыть сессию, которую открыли с помощью `pg.connect`, используется следующий запрос:

```
*имя-соединения*:close()
```

Пример:

```
tarantool> conn:close()
---
...
```

Для получения дополнительной информации, включая примеры редко используемых запросов, см. файл README.md по ссылке github.com/tarantool/pg.

Пример

Пример выполняется на машине с ОС Ubuntu 12.04 (Precise Pangolin), где Tarantool установлен в поддиректорию /usr, а копия PostgreSQL установлена в /usr. Экземпляр сервера PostgreSQL уже запущен на localhost 127.0.0.1.

```
$ # Check that the include subdirectory exists
$ # by looking for /usr/include/postgresql/libpq-fe-h.
$ [ -f /usr/include/postgresql/libpq-fe.h ] && echo "OK" || echo "Error"
OK

$ # Check that the library subdirectory exists and has the necessary .so file.
$ [ -f /usr/lib/x86_64-linux-gnu/libpq.so ] && echo "OK" || echo "Error"
OK

$ # Check that the psql client can connect using some factory defaults:
$ # port = 5432, user = 'postgres', user password = 'postgres',
$ # database = 'postgres'. These can be changed, provided one changes
$ # them in all places. Insert a row in database postgres, and quit.
$ psql -h 127.0.0.1 -p 5432 -U postgres -d postgres
Password for user postgres:
psql (9.3.10)
SSL connection (cipher: DHE-RSA-AES256-SHA, bits: 256)
Type "help" for help.

postgres=# CREATE TABLE test (s1 INT, s2 VARCHAR(50));
CREATE TABLE
```

(continues on next page)

(продолжение с предыдущей страницы)

```
postgres=# INSERT INTO test VALUES (1,'PostgreSQL row');
INSERT 0 1
postgres=# \q
$

$ # Install luarocks
$ sudo apt-get -y install luarocks | grep -E "Setting up|already"
Setting up luarocks (2.0.8-2) ...

$ # Set up the Tarantool rock list in ~/.luarocks,
$ # following instructions at rocks.tarantool.org
$ mkdir ~/.luarocks
$ echo "rocks_servers = {[[http://rocks.tarantool.org/]]}" >> \
       ~/.luarocks/config.lua

$ # Ensure that the next "install" will get files from Tarantool master
$ # repository. The resultant display is normal for Ubuntu 12.04 precise
$ cat /etc/apt/sources.list.d/tarantool.list
deb http://tarantool.org/dist/2.0/ubuntu/ precise main
deb-src http://tarantool.org/dist/2.0/ubuntu/ precise main

$ # Install tarantool-dev. The displayed line should show version = 2.0
$ sudo apt-get -y install tarantool-dev | grep -E "Setting up|already"
Setting up tarantool-dev (2.0.4.222.g48b98bb~precise-1) ...
$

$ # Use luarocks to install locally, that is, relative to $HOME
$ luarocks install pg POSTGRESQL_LIBDIR=/usr/lib/x86_64-linux-gnu --local
Installing http://rocks.tarantool.org/pg-scm-1.rockspec...
... (more info about building the Tarantool/PostgreSQL driver appears here)
pg scm-1 is now built and installed in ~/.luarocks/

$ # Ensure driver.so now has been created in a place
$ # tarantool will look at
$ find ~/.luarocks -name "driver.so"
~/.luarocks/lib/lua/5.1/pg/driver.so

$ # Change directory to a directory which can be used for
$ # temporary tests. For this example we assume that the
$ # name of this directory is $HOME/tarantool_sandbox.
$ # (Change "$HOME" to whatever is the user's actual
$ # home directory for the machine that's used for this test.)
cd $HOME/tarantool_sandbox

$ # Start the Tarantool server instance. Do not use a Lua initialization file.

$ tarantool
tarantool: version 2.0.4-412-g803b15c
type 'help' for interactive help
tarantool>
```

Настройте Tarantool и загрузите модуль pg. Убедитесь, что Tarantool не выбрасывает ошибку в ответ на вызов «require()».

```
tarantool> box.cfg{}
...
```

(continues on next page)

(продолжение с предыдущей страницы)

```
tarantool> pg = require('pg')
---
...
```

Создайте Lua-функцию, которая подключится к PostgreSQL-серверу (используя значения по умолчанию для параметров порта, пользователя и пароля), выберите одну строку и выведите ее на экран. Описание используемых здесь типов операторов вы можете найти в практикуме по Lua в руководстве пользователя Tarantool'а.

```
tarantool> function pg_select ()
         >   local conn = pg.connect({
         >     host = '127.0.0.1',
         >     port = 5432,
         >     user = 'postgres',
         >     password = 'postgres',
         >     db = 'postgres'
         >   })
         >   local test = conn:execute('SELECT * FROM test WHERE s1 = 1')
         >   local row = ''
         >   for i, card in pairs(test) do
         >       row = row .. card.s2 .. ' '
         >       end
         >   conn:close()
         >   return row
         > end
---
...
tarantool> pg_select()
---
- 'PostgreSQL row '
...
```

Просмотрите результат. В нем есть строка «PostgreSQL row». Это и есть строка, которая была вставлена в базу данных PostgreSQL. А сейчас она выделена с помощью Tarantool-клиента.

5.3.2 Модуль *expiriationd*

Рассмотрим исходный код `expiriationd` – пример Lua-модуля для промышленной эксплуатации, который работает с Tarantool'ом – Tarantool предоставляет его с лицензией Artistic на GitHub. Программа `expiriationd.lua` довольно объемная (около 500 строк), поэтому здесь мы остановимся на пунктах, знания о которых можно расширить, позднее изучив программу полностью.

```
task.worker_fiber = fiber.create(worker_loop, task)
log.info("expiration: task %q restarted", task.name)
...
fiber.sleep(expiriationd.constants.check_interval)
...
```

Если в Tarantool'е упоминается «демон», то речь идет об использовании *файбера*. Программа создает файбер и передает управление так, что он периодически запускается, уходит в режим ожидания, а затем повторяет эти действия.

```
for _, tuple in scan_space.index[0]:pairs(nil, {iterator = box.index.ALL}) do
...
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    expiration_process(task, tuple)
...
    /* expiration_process() contains:
    if task.is_tuple_expired(task.args, tuple) then
    task.expired_tuples_count = task.expired_tuples_count + 1
    task.process_expired_tuple(task.space_id, task.args, tuple) */
```

Команду «for» можно перевести как «выполнить итерацию по индексу сканируемого спейса», а внутри – если кортеж «неактуален» (например, если в кортеже есть поле метки времени, которое меньше текущего времени), то обработать кортеж как неактуальный кортеж.

```
-- функция обработки неактуального кортежа по умолчанию
local function default_tuple_drop(space_id, args, tuple)
    box.space[space_id]:delete(construct_key(space_id, tuple))
end
    /* construct_key() contains:
    local function construct_key(space_id, tuple)
        return fun.map(
            function(x) return tuple[x.fieldno] end,
           box.space[space_id].index[0].parts
        ):totable()
    end */
```

В конечном итоге, обработка неактуального кортежа приводит к `default_tuple_drop()`, что приводит к удалению кортежа из первоначального спейса. Сначала используется модуль *fun*, в частности fun.map. Учитывая, что index[0] всегда является первичным ключом спейса, а index[0].parts[N].fieldno всегда является номером поля для компонента ключа N, функция fun.map() создает таблицу из первичных значений кортежа. Результат fun.map() передается в *space_ object:delete()*.

```
local function expirationd_run_task(name, space_id, is_tuple_expired, options)
...
```

На этом этапе ясно, что `expirationd.lua` запускает фоновый процесс (файбер), который выполняет итерацию по всем кортежам в спейсе, в рамках кооперативной многозадачности уходит в режим ожидания, чтобы другие файберы могли работать одновременно с ним, а когда находит неактуальный кортеж, удаляет его из спейса. Теперь функцию «`expirationd_run_task()`» можно использовать в тестировании, где создаются образцы данных, некоторое время работает демон, и выводятся результаты.

Если вы хотите увидеть, как все работает, обратите внимание на нижеприведенные шаги по включению `expirationd` в тестирование.

1. Найдите `expirationd.lua`. Можно воспользоваться стандартным способом, поскольку модуль включен в общий список модулей, но для этой цели просто скопируйте содержимое `expirationd.lua` в директорию в Lua-пути (введите `print(package.path)`, чтобы увидеть Lua-путь).
2. Запустите Tarantool-сервер, как описано выше.
3. Выполните следующие запросы:

```
fiber = require('fiber')
expd = require('expirationd')
box.cfg{}
e = box.schema.space.create('expirationd_test')
e:create_index('primary', {type = 'hash', parts = {1, 'unsigned'}})
e:replace{1, fiber.time() + 3}
e:replace{2, fiber.time() + 30}
function is_tuple_expired(args, tuple)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
  if (tuple[2] < fiber.time()) then return true end
  return false
  end
expd.run_task('expirationd_test', e.id, is_tuple_expired)
retval = {}
fiber.sleep(2)
expd.task_stats()
fiber.sleep(2)
expd.task_stats()
expd.kill_task('expirationd_test')
e:drop()
os.exit()
```

Запросы в работе с базой данных (`cfg`, *space.create*, *create_index*) уже должны быть вам знакомы.

В `expirationd` передается функция is_tuple_expired, которая задает следующее условие: если второе поле кортежа меньше *текущего времени* , вернуть true (правда), в противном случае, вернуть false (ложь).

Ключ к запуску модуля – `expd = require('expirationd')`. Функция require – это именно то, что выполняет чтение в программе. Она появится и в дальнейших примерах в данном руководстве, когда будет необходимо запустить модуль, который не входит в ядро Tarantool'а, но находится в Lua-пути (`package.path`) или же C-пути (`package.cpath`). После того, как Lua-переменной expd присваивается значение модуля `expirationd`, можно вызвать функцию модуля `run_task()`.

После ухода *в режим ожидания* на две секунды, когда проводится итерация по спейсам, `expd.task_stats()` выведет отчет о количестве неактуальных кортежей – «expired_count: 0».

После ожидания в течение еще двух секунд `expd.task_stats()` выведет отчет о количестве неактуальных кортежей – «expired_count: 1». Это показывает, что функция `is_tuple_expired()` с течением времени вернула «true» для одного из кортежей, поскольку поле метки времени было дольше трех секунд.

Конечно, `expirationd` можно настроить на выполнение различных задач с помощью разных параметров, что будет очевидно после более детального изучения исходного кода. В частности, важны опции `{options}`, которые можно добавить в качестве последнего параметра в `expirationd.run_task`:

- `force` (логическое значение) – выполнение задачи даже на реплике. По умолчанию: `force=false`, поэтому, как правило, `expirationd` не учитывает реплики.
- `tuples_per_iteration` (целое число) – количество кортежей, которые проверяются за одну итерацию. По умолчанию: `tuples_per_iteration=1024`.
- `full_scan_time` (число) – число секунд на полное сканирование диска. По умолчанию: `full_scan_time=3600`.
- `vinyl_assumed_space_len` (целое число) – предполагаемый размер спейса vinyl'а, используется только для первой итерации. По умолчанию: `vinyl_assumed_space_len=10000000`.
- `vinyl_assumed_space_len_factor` (целое число) – коэффициент перерасчета размера спейса vinyl'а. По умолчанию: `vinyl_assumed_space_len_factor=2`. (Размер спейса vinyl'а не так легко рассчитать, поэтому для первой итерации используется «предполагаемый» размер, на второй итерации – «предполагаемый» размер, помноженный на «коэффициент», на третьей итерации – «предполагаемый» размер, дважды помноженный на «коэффициент» и так далее.)

5.3.3 Модуль *membership*

Этот модуль представляет собой библиотеку `membership` для Tarantool'а на основе протокола gossip.

Эта библиотека создает сеть из нескольких экземпляров Tarantool. Сеть сама контролирует себя, помогает участникам обнаружить всех остальных в группе и получать уведомления об изменениях своего статуса с низкой задержкой. Модуль основан на концепциях из Consul или, точнее, алгоритма SWIM.

Модуль `membership` работает по протоколу UDP и может производить операции даже до инициализации box.cfg.

Структура членов данных

Члены-данные представлены в виде таблиц со следующими полями:

- `uri` (строка) – это унифицированный идентификатор ресурса.
- `status` (строка) – это строка, которая принимает одно из следующих значений.
  - `alive`: член группы, который отвечает на сообщения проверки связи, работоспособен в статусе `alive`.
  - `suspect`: если какой-либо член группы не может получить ответ от какого-либо другого участника, первый член группы просит трех других активных членов группы в статусе `alive` отправить сообщение проверки связи соответствующему участнику. Если ответа нет, последний получает статус сомнительного, то есть `suspect`.
  - `dead`: член группы в статусе `suspect` получает статус вышедшего из строя `dead` по истечении времени ожидания.
  - `left`: член группы получает статус выбывшего `left` после выполнения функции *leave()*.

    Примечание: Протокол gossip гарантирует, что каждый член группы узнает о любом изменении статуса в двух циклах связи.

- `incarnation` (число) – это значение, которое увеличивается каждый раз, когда экземпляр получает статус `suspect`, `dead` или обновляет полезную нагрузку.
- `payload` (таблица) – это вспомогательные данные, которыми могут воспользоваться различные модули.
- `timestamp` (число) – это значение `fiber.time64()`, которое:
  - соответствует последнему обновлению параметра `status` или `incarnation`;
  - всегда локально;
  - не зависит от настроек часов других членов группы.

Ниже приведен пример таблицы:

```
tarantool> membership.myself()
---
uri: localhost:33001
status: alive
incarnation: 1
payload:
    uuid: 2d00c500-2570-4019-bfcc-ab25e5096b73
timestamp: 1522427330993752
...
```

Справочник по API

Ниже приведен список простых функций, функций шифрования, подписки и параметры модуля `membership`.

Имя	Назначение
Простые функции	
init(advertise_host, port)	Инициализация модуля `membership`.
myself()	Получение структуры данных текущего экземпляра.
get_member(uri)	Получение структуры данных для указанного URI.
members()	Получение таблицы со всеми членами группы, известными текущему экземпляру.
pairs()	Сокращение для `pairs(membership.members())`.
add_member(uri)	Добавление члена в группу.
probe_uri(uri)	Проверка принадлежности члена к группе.
broadcast()	Обнаружение участников в локальной сети путем отправки широковещательного сообщения UDP.
set_payload(key, value)	Обновление `myself().payload` и распространение информации.
leave()	Корректное исключение из группы.
is_encrypted()	Проверка, включено ли шифрование.
Функции шифрования	
set_encryption_key(key)	Установка ключа для низкоуровневого шифрования сообщений.
get_encryption_key()	Получение используемого ключа шифрования.
Функции подписки	
subscribe()	Подписка на обновления членов таблицы.
unsubscribe()	Удаление подписки.
Параметры	
PROTOCOL_PERIOD_SECONDS	Время отправки сообщений проверки связи напрямую.
ACK_TIMEOUT_SECONDS	Время ожидания сообщения подтверждения.
ANTI_ENTROPY_PERIOD_SECONDS	Период синхронизации во избежание энтропии.
SUSPECT_TIMEOUT_SECONDS	Время ожидания, чтобы перевести члена группы из статуса `suspect` в `dead`.
NUM_FAILURE_DETECTION_SUBGROUPS	Число членов группы, которые отправляют сообщения проверки связи члену группы в статусе `suspect`.

Простые функции:

`membership.init(`*advertise_host*, *port*`)`

Инициализация модуля `membership`. Привязывает UDP-сокет к `0.0.0.0:<port>`, задает значение параметра `advertise_uri` = `<advertise_host>:<port>` (передаваемый хост, порт) и значение параметра `incarnation` = `1`.

Функцию `init()` можно вызвать несколько раз, старый сокет будет закрыт, откроется новый сокет.

Если значение параметра `advertise_uri` изменится во время очередного выполнения `init()`, старый URI считается недоступным со статусом `DEAD`. Чтобы корректно исключить члена из группы, используйте функцию *leave()*.

Параметры

- `advertise_host` (`string`) – имя хоста или IP-адрес, передаваемый другим членам группы
- `port` (*number*) – привязываемый UDP-порт

возвращает `true` (правда)

тип возвращаемого значения boolean (логический)

вызывает подключенный сокет, если нет ошибки

`membership.myself()`

возвращает *структура данных члена группы* для текущего экземпляра.

тип возвращаемого значения таблица

`membership.get_member(`*uri*`)`

Параметры

- `uri` (`string`) – `advertise_uri` для указанного члена группы

возвращает *структура данных* экземпляра с указанным URI.

тип возвращаемого значения таблица

`membership.members()`

Получение всех членов группы, известных текущему экземпляру.

Редактирование этой таблицы ни на что не вляет.

возвращает таблица с URI в качестве ключей и *структурой данных члена группы* в качестве значений.

тип возвращаемого значения таблица

`membership.pairs()`

Сокращение для `pairs(membership.members())`.

возвращает Lua-итератор

Можно использовать следующим образом:

```
for uri, member in memberhip.pairs()
  -- что-то сделать
end
```

`membership.add_member(`*uri*`)`

Добавление в группу члена с указанным URI и передача информации об этом событии другим членам группы. Достаточно добавить члена группы в один экземпляр, так как все остальные экземпляры в группе со временем получат информацию об этом. Не имеет значения, кто кого добавляет.

Параметры

- `uri` (`string`) – параметр `advertise_uri` добавляемого члена группы

возвращает `true` (правда) или нулевое значение `nil` в случае ошибки

тип возвращаемого значения boolean (логический)

вызывает ошибка анализа, если URI нельзя проанализировать

`membership.probe_uri(`*uri*`)`

Отправка сообщения члену группы, чтобы убедиться, что он включен в группу. Если экземпляр активен со статусом `alive`, но не включен в группу, происходит его добавление. Если он уже включен в группу, ничего не происходит.

Параметры

- `uri` (string) – параметр `advertise_uri` члена группы, которому отправляются сообщения проверки связи

возвращает `true` (правда), если ответ возвращается в течение 0.2 секунды, в остальных случаях `no response` (нет ответа)

тип возвращаемого значения boolean (логический)

вызывает `ping was not sent` (сообщение проверки связи не отправлено), если имя хоста не разрешено

`membership.broadcast()`

Обнаружение членов группы в локальной сети путем отправки широковещательного сообщения UDP во все сети, обнаруженные с помощью вызова `getifaddrs()` на языке C.

возвращает `true` (правда), если сообщение отправлено, `false` (ложь), если `getaddrinfo()` не выполнена.

тип возвращаемого значения boolean (логический)

`membership.set_payload(`*key*, *value*`)`

Обновление `myself().payload` и распространение соответствующей информации вместе со статусом члена группы.

Увеличивает значение параметра `incarnation`.

Параметры

- `key` (string) – ключ, задаваемый в таблице payload
- `value` – дополнительные данные

возвращает `true` (правда)

тип возвращаемого значения boolean (логический)

`membership.leave()`

Корректное исключение из группы `membership`. Узел получает статус выбывшего `left`, другие члены группы не будут пытаться снова подключить его.

возвращает `true` (правда)

тип возвращаемого значения boolean (логический)

`membership.is_encrypted()`

возвращает `true` (правда), если шифрование включено, `false` в противном случае.

тип возвращаемого значения boolean (логический)

Функции шифрования:

`membership.set_encryption_key(`*key*`)`

Установка ключа, который используется для низкоуровневого шифрования сообщений. Ключ автоматически обрезается или дополняется до 32 байтов. Если значения ключа `key` нулевое `nil`, шифрование будет отключено.

Модуль Tarantool `crypto.cipher.aes256.cbc` занимается шифрованием.

Чтобы обеспечить правильную связь, все члены группы должны быть настроены на использование одного и того же ключа шифрования. В противном случае члены группы получат статус либо `dead`, либо `non-decryptable` (невозможно расшифровать).

Параметры

- `key` (string) – ключ шифрования

возвращает nil.

membership.get_encryption_key()
Получение используемого ключа шифрования.

возвращает ключ шифрования или нулевое значение nil, если шифрование отключено.

тип возвращаемого значения строка

Функции подписки:

membership.subscribe()
Подписка на обновления членов таблицы.

возвращает объект fiber.cond, который передается при каждом изменении таблицы.

тип возвращаемого значения объект

membership.unsubscribe(*cond*)
Удаление подписки на cond, получаемый с помощью функции *subscribe()*.

Достоверность cond не проверяется.

Параметры

- cond – объект fiber.cond, получаемый с помощью функции *subscribe()*

возвращает nil.

Ниже приведен перечень параметров membership. Их можно задать следующим образом:

```
options = require('membership.options')
options.<параметр> = <значение>
```

options.PROTOCOL_PERIOD_SECONDS
Период отправки сообщение проверки связи напрямую. Обозначается как T' в протоколе SWIM.

options.ACK_TIMEOUT_SECONDS
Время ожидания сообщения подтверждения после отправки сообщения проверки связи. Если ответ запаздывает, вызывается алгоритм косвенной проверки связи.

options.ANTI_ENTROPY_PERIOD_SECONDS
Период выполнения алгоритма синхронизации во избежание энтропии из протокола SWIM.

options.SUSPECT_TIMEOUT_SECONDS
Время ожидания, чтобы перевести члена группы из статуса suspect в dead.

options.NUM_FAILURE_DETECTION_SUBGROUPS
Число членов группы, которые пытаются отправить сообщения проверки связи члену группы в статусе suspect. Обозначается как k в протоколе SWIM.

5.3.4 Модуль *vshard*

В модуле vshard реализована функция продвинутого шардинга (сегментирования), которая основывается на понятии *виртуального сегмента* и позволяет осуществлять горизонтальное масштабирование в Tarantool'е.

Стоит начать с *Руководства по быстрому запуску* – или же сразу переходить к углубленному изучению документации по vshard:

Введение

С ростом проекта масштабируемость баз данных часто становится одной из наиболее серьезных проблем. Если отдельный сервер не может справиться с нагрузкой, необходимо применять средства масштабирования.

Шардинг, или сегментирование, представляет собой архитектуру базы данных, которая дает возможность горизонтального масштабирования, что подразумевает под собой секционирование набора данных и их распределение по нескольким серверам.

С помощью модуля vshard кортежи набора данных распределяются по множеству узлов, на каждом из которых находится экземпляр сервера базы данных Tarantool'а. Каждый экземпляр обрабатывает лишь подмножество от общего количества данных, поэтому увеличение нагрузки можно компенсировать добавлением новых серверов. Первоначальный набор данных секционируется на множество частей, то есть каждая часть хранится на отдельном сервере.

Модуль `vshard` основан на концепции *виртуальных сегментов*: набор кортежей распределяется на большое количество абстрактных виртуальных узлов (**виртуальных сегментов**, или просто **сегментов** далее по тексту), а не на малое количество физических узлов.

Секционирование набора данных осуществляется с помощью **сегментных ключей** (идентификаторов сегментов). Хеширование сегментного ключа в большое количество сегментов позволяет незаметно для пользователя изменять количество серверов в кластере. **Механизм балансирования** распределяет сегменты между шардами при добавлении или удалении каких-либо серверов.

Для сегментов предусмотрены **состояния**, поэтому можно легко отслеживать состояние сервера. Например, активен ли экземпляр сервера и доступен ли он для всех типов запросов, или же произошел отказ, и сервер принимает только запросы на чтение.

Модуль `vshard` предоставляет общедоступные и внутренние API роутера и хранилища для приложений с поддержкой шардинга.

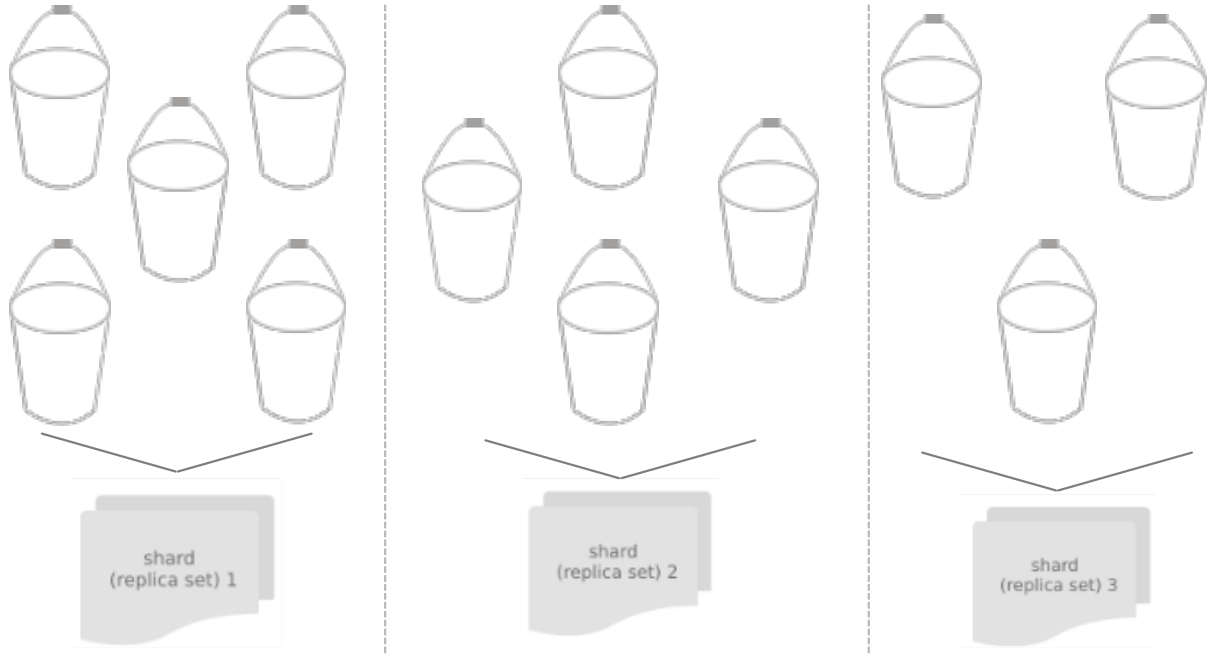
Архитектура

Общие сведения

Рассмотрим распределенный Tarantool-кластер, состоящий из подкластеров под названием **шарды**, в каждом из которых хранится некоторая часть данных. Каждый шард, в свою очередь, представляет собой **набор реплик**, состоящую из нескольких **реплик**, одна из которых служит ведущим узлом, обрабатывающим все запросы на чтение и запись.

Весь набор данных при шардинге распределяется на заданное количество *виртуальных сегментов* (далее по тексту просто **сегменты**). Каждому из них присваивается уникальный номер от 1 до N, где N – это общее количество сегментов. Специально выбирается количество сегментов на несколько порядков больше, чем потенциальное количество кластерных узлов даже с учетом будущего масштабирования кластера. Например, если предполагается M узлов, набор данных может быть разделен на 100 * M или даже 1000 * M сегментов. Особое внимание следует уделить выбору количества сегментов: слишком большое число может потребовать дополнительную память для хранения информации о маршрутизации; слишком маленькое может привести к снижению степени детализации балансировки.

Каждый шард хранит уникальное подмножество сегментов. Один сегмент не может относиться к нескольким шардам одновременно, как показано на схеме ниже:



Такая схема распределения сегментов по шардам хранится в таблице в одном из системных пространств Tarantool'а, при этом в каждом шарде содержится только определенную часть схемы, которая покрывает присвоенные этому шарду сегменты.

Помимо таблицы, **идентификатор сегмента** также хранится в специальном поле каждого кортежа каждой таблицы, участвующей в шардинге.

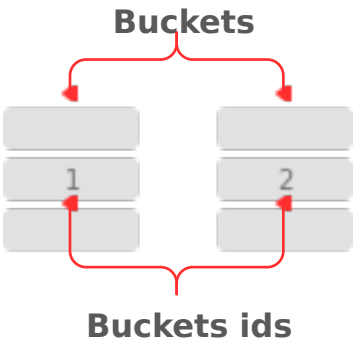
Как только шард получает любой запрос (за исключением SELECT) от приложения, этот шард сверяет идентификатор сегмента, указанный в запросе, с таблицей идентификаторов сегментов, которые принадлежат данному узлу. Если указанный идентификатор сегмента недействителен, то запрос завершается со следующей ошибкой: «wrong bucket" (неверный сегмент). В противном случае запрос выполняется, и всем создаваемым данным присваивается указанный в запросе идентификатор сегмента. Обратите внимание, что запрос должен изменять только данные с тем же идентификатором сегмента, что и в запросе.

Хранение идентификаторов сегментов как в самих данных, так и в таблице обеспечивает согласованность данных независимо от логики приложения и прозрачность балансировки для приложения. Хранение таблицы соответствий в системном спейсе обеспечивает последовательность шардинга в случае восстановления после отказа, так как у всех реплик в шарде будет одно исходное состояние таблицы.

Виртуальные сегменты

Набор данных при шардинге распределяется на большое количество абстрактных узлов, которые называются **виртуальные сегменты** (далее по тексту просто **сегменты**).

Секционирование набора данных происходит с помощью сегментного ключа (или **идентификатора сегмента** (bucket id) в терминах Tarantool'а). Идентификатор сегмента – это число от 1 до N, где N – это общее количество сегментов.



В каждом наборе реплик есть уникальное подмножество сегментов. Один сегмент не может относиться к нескольким наборам реплик одновременно.

Общее количество сегментов определяет администратор, который настраивает первоначальную конфигурацию кластера.

В каждом спейсе, который будет разделен на шарды, должно быть числовое поле с идентификаторами сегментов. Это поле должно соответствовать следующим требованиям:

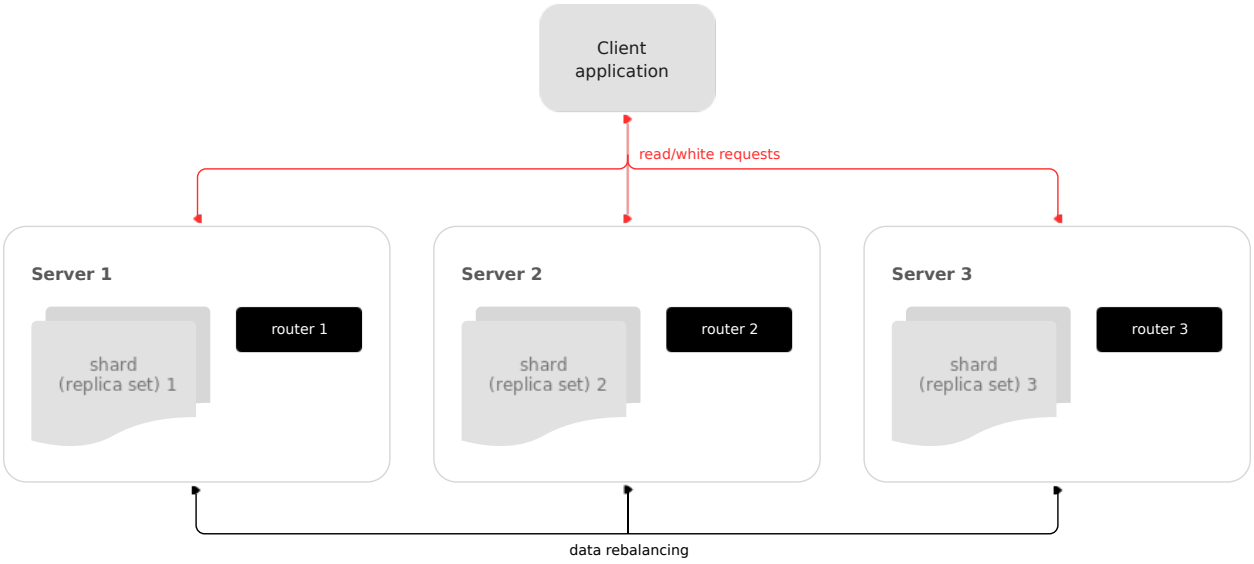
- Тип данных поля может быть: unsigned (без знака), number (число) или integer (целое число).
- Поле не должно быть нулевым.
- Поле должно быть проиндексировано с помощью *shard_index*. Имя по умолчанию для этого индекса: `bucket_id`.

См. *пример конфигурации*.

Структура

Сегментированный кластер в Tarantool'е состоит из:

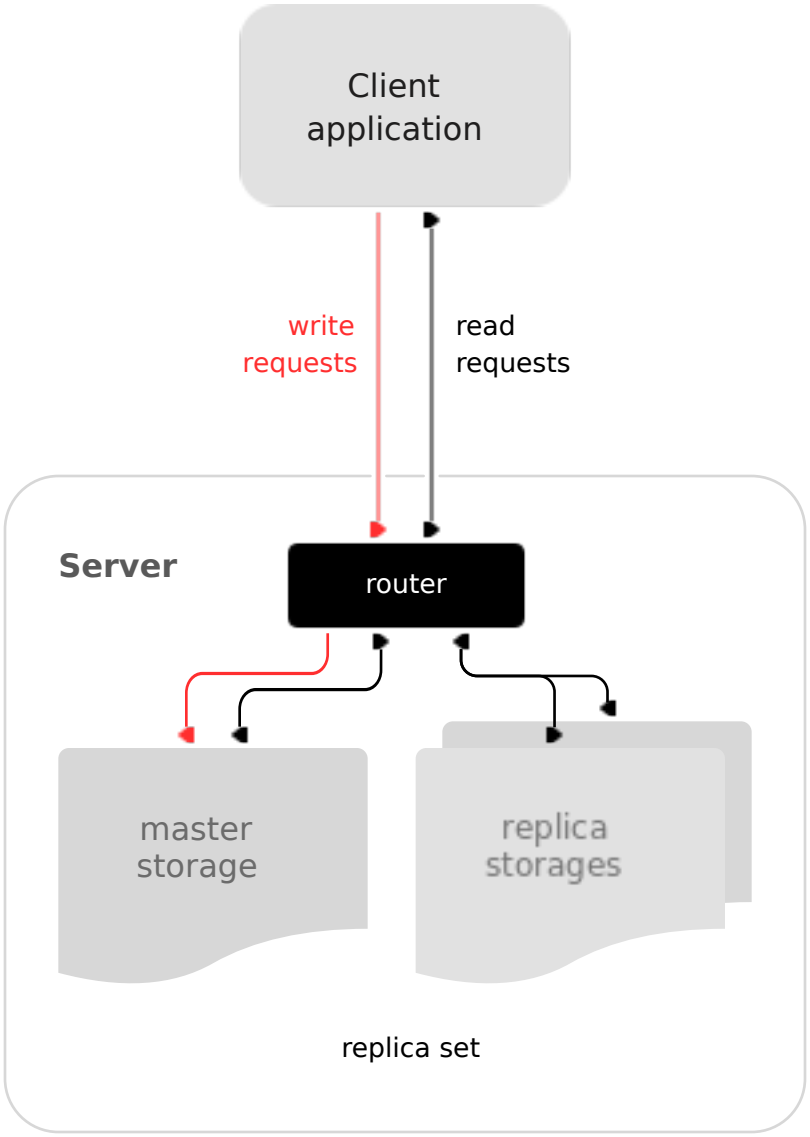
- хранилищ,
- роутеров
- и балансировщика.



Хранилище

Хранилище (storage) – это узел, который хранит подмножество набора данных. Несколько реплицируемых (для резерва) хранилищ составляют **набор реплик** (также называемый **шардом**).

У каждого хранилища в наборе реплик есть роль: **мастер** или **реплика**. Мастер обрабатывает запросы на чтение и запись. Реплика обрабатывает запросы на чтение, но не может обрабатывать запросы на запись.



Роутер

Роутер (router) – это автономный компонент ПО, который обеспечивает маршрутизацию запросов чтения и записи от клиентского приложения к шардам.

Все запросы из приложения приходят в сегментированный кластер через роутер (`router`). Роутер сохраняет топологию сегментированного кластера прозрачной для приложения, не сообщая приложению:

- номер и местоположение шардов,

- процесс балансировки данных,
- наличие отказа и восстановление после отказа реплики.

Роутер также может самостоятельно вычислить идентификатор сегмента при условии, что приложение четко определяет правила вычисления идентификатора сегмента на основе данных запроса. Для этого роутеру необходимо знать схему данных.

У роутера нет постоянного статуса, он не хранит топологию кластера и не выполняет балансировку данных. Роутер – это автономный компонент ПО, который может работать на уровне хранилища или на уровне приложения в зависимости от функций приложения.

Роутер поддерживает постоянный пул соединений со всеми хранилищами, созданными при запуске, что помогает избежать ошибок конфигурации. После создания пула роутер кэширует текущее состояние таблицы `_vbucket`, чтобы ускорить маршрутизацию. Если сегмент был перемещен в другое хранилище в результате балансировки, или же один из шардов переключается на реплику, роутер обновит таблицу маршрутизации так, чтобы это было понятно приложению.

Шардинг не интегрирован ни в одну систему централизованного хранения конфигураций. Предполагается, что само приложение обрабатывает взаимодействие с такой системой и передает параметры шардинга. При этом конфигурацию можно изменить динамически, например, при добавлении или удалении одного или нескольких шардов:

1. Чтобы добавить новый шард в кластер, системный администратор сначала изменяет конфигурацию всех роутеров, а затем конфигурацию всех хранилищ.
2. Новый шард становится доступен для балансировки на уровне хранилища.
3. В результате балансировки один из виртуальных сегментов перемещается на новый шард.
4. При попытке доступа к виртуальному сегменту роутер получает специальный код ошибки, который указывает новое местоположение сегмента.

CRUD-операции: create, replace, update, delete (создание, замена, обновление, удаление)

CRUD-операции могут:

- либо выполняться в рамках хранимой процедуры в хранилище,
- либо запускаться приложением.

В любом случае приложение должно включать идентификатор рабочего сегмента в запрос. При выполнении запроса вставки INSERT идентификатор сегмента хранится в созданном кортеже. В других случаях проверяется, совпадает ли указанный идентификатор рабочего сегмента с идентификатором сегмента кортежа, в который вносятся изменения.

SELECT-запросы

Поскольку хранилище не знает о соответствии идентификатора сегмента и первичного ключа, все запросы выборки SELECT в хранимых процедурах внутри хранилища выполняются только локально. SELECT-запросы, которые были инициализированы приложением, направляются на роутер. И если приложение передало идентификатор сегмента, роутер использует его для вычисления шарда.

Вызов хранимых процедур

Существует несколько способов вызвать хранимые процедуры в наборах реплик кластера. Хранимые процедуры можно вызвать:

- либо на определенном виртуальном сегменте, расположенном в наборе реплик (в этом случае необходимо различать процедуры чтения и записи, так как процедуры записи не применимы к перемещаемым сегментам),
- либо без указания определенного сегмента.

Все проверки правильности маршрутизации, выполняемые для шардированных DML-операций, распространяются и на хранимые процедуры, связанные с сегментами.

Балансировщик

Балансировщик представляет собой фоновый процесс балансировки, который обеспечивает равномерное распределение сегментов по шардам. Во время балансировки происходит миграция сегментов по наборам реплик.

Балансировщик периодически «просыпается» и перераспределяет данные из наиболее загруженных узлов в менее загруженные узлы. Балансировка начинается, когда **предел дисбаланса** в наборе реплик превышает предел дисбаланса, указанный в конфигурации.

Предел дисбаланса рассчитывается следующим образом:

```
|эталонное_число_сегментов - текущее_число_сегментов| / эталонное_число_сегментов * 100
```

Миграция сегментов

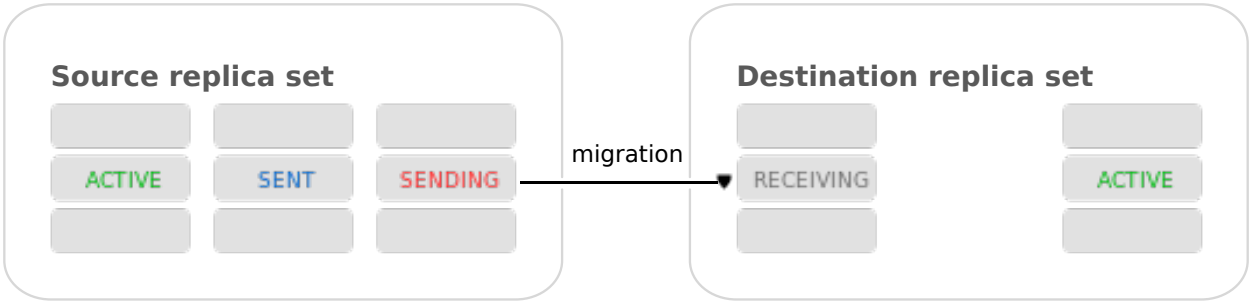
Набор реплик, из которого переносится сегмент, называется **исходный** (source); а набор реплик, куда переносится сегмент, называется **целевой** (destination).

Блокировка набора реплик позволяет набору реплик оставаться невидимым для балансировщика. Набор реплик с блокировкой не может ни принимать новые сегменты, ни мигрировать свои собственные.

Во время миграции у сегмента могут быть разные статусы:

- ACTIVE (активный) – сегмент доступен для запросов чтения и записи.
- PINNED (закрепленный) – сегмент заблокирован для миграции в другой набор реплик. Во всем остальном закрепленные сегменты аналогичны активным сегментам.
- SENDING (отправляемый) – в настоящий момент сегмент копируется в целевой набор реплик; запросы на чтение в исходный набор реплик обрабатываются.
- RECEIVING (принимающий) – происходит наполнение сегмента; все запросы отклоняются.
- SENT (отправленный) – сегмент был перенесен в целевой набор реплик. Роутер использует статус SENT, чтобы определить новое местонахождение сегмента. Сегмент в статусе SENT переходит в статус мусора GARBAGE автоматически через количество секунд, указанное в BUCKET_SENT_GARBAGE_DELAY, по умолчанию равное *0,5 секунды*.
- GARBAGE (мусор) – произошла миграция сегмента в целевой набор реплик во время балансировки; или же принимающий сегмент был в статусе RECEIVING, но произошла ошибка во время миграции.

Сегменты в статусе мусора GARBAGE удаляются сборщиком мусора.



Миграция происходит следующим образом:

1. В целевом наборе реплик создается новый сегмент, который получает статус RECEIVING (принимающий), начинается копирование данных, и сегмент отклоняет все запросы.
2. Отправляемый сегмент в исходном наборе реплик получает статус SENDING и продолжает обрабатывать запросы на чтение.
3. После копирования данных сегмент в исходном наборе реплик получает статус отправленного (SENT) и перестает принимать запросы.
4. Сегмент в целевом наборе реплик переходит в активный статус (ACTIVE) и начинает принимать все запросы.

Примечание: Есть специальная ошибка `vshard.error.code.TRANSFER_IS_IN_PROGRESS`, которая возвращается в том случае, если запрос пытается выполнить действие, неприменимое к перемещаемому сегменту. В этом случае необходимо повторить попытку выполнения запроса.

Системный спейс *_bucket*

Системный спейс `_bucket` в каждом наборе реплик хранит идентификаторы сегментов данного набора реплик. Спейс содержит следующие поля:

- `bucket` – идентификатор сегмента
- `status` – статус сегмента
- `destination` – UUID целевого набора реплик

Пример `_bucket.select{}`:

```
---
- - [1, ACTIVE, abfe2ef6-9d11-4756-b668-7f5bc5108e2a]
  - [2, SENT, 19f83dcb-9a01-45bc-a0cf-b0c5060ff82c]
...
```

После миграции сегмента UUID целевого набора реплик вносится в таблицу. Пока сегмент еще находится в исходном наборе реплик, значение UUID целевого набора реплик равно `NULL`.

Таблица маршрутизации

Таблица маршрутизации роутера отображает все идентификаторы сегментов с соответствующими наборами реплик. Она обеспечивает консистентность шардинга в случае отказа.

Роутер поддерживает постоянный пул соединений со всеми хранилищами, созданными при запуске, что помогает избежать ошибки конфигурации. После создания пула соединений роутер кэширует текущее состояние таблицы маршрутизации, чтобы ускорить ее. Если произошла миграция сегмента в другое хранилище после балансировки или же отказ, который вызвал переключение шарда на другую реплику, файбер обнаружения (`discovery fiber`) в роутере обновит таблицу маршрутизации автоматически.

Поскольку идентификатор сегмента явно указан как в данных, так и в таблице отображения на роутере, данные сохраняются независимо от логики приложения. Это также обеспечивает прозрачность балансировки для приложения.

Обработка запросов

Запросы в базу данных можно производить из приложения или с помощью хранимых процедур. В любом случае идентификатор сегмента следует явным образом указать в запросе.

Сначала все запросы направляются в роутер. Роутер поддерживает только операцию вызова, которая выполняется с помощью функции `vshard.router.call()`:

```
result = vshard.router.call(<идентификатор_сегмента>, <режим>, <имя_функции>, {<список_аргументов>}
↪, {<опции>})
```

Запросы обрабатываются следующим образом:

1. Роутер использует идентификатор сегмента для поиска набора реплик с соответствующим сегментом в таблице маршрутизации.

   Если роутер не содержит информацию о соответствии идентификатора сегмента набору реплик (файбер обнаружения еще не заполнил таблицу), роутер выполняет запросы ко всем хранилищам, чтобы обнаружить местонахождение сегмента.

2. После обнаружения сегмента шард проверяет:

   - хранится ли сегмент в системном спейсе `_bucket` набора реплик;
   - находится ли сегмент в статусе ACTIVE (активный) или PINNED (закрепленный) (если выполняется запрос на чтение, то сегмент может находиться в состоянии отправки SENDING).

3. Если проверка пройдена, запрос выполняется. В противном случае, выполнение запроса прекращается с ошибкой: **“`wrong bucket`”** (несоответствующий сегмент).

Глоссарий

Вертикальное масштабирование Добавление мощности в отдельный сервер: использование более мощного процессора, добавление оперативной памяти, добавление хранилищ и т.д.

Горизонтальное масштабирование Добавление дополнительных серверов в пул ресурсов, последующее секционирование и распределение набора данных по серверам.

Шардинг Архитектура базы данных, которая допускает секционирование набора данных по сегментному ключу и распределение набора данных по нескольким серверам. Шардинг представляет собой частный случай горизонтального масштабирования.

Узел Виртуальный или физический экземпляр сервера.

Кластер Набор узлов, которые составляют отдельную группу.

Хранилище Узел, который хранит подмножество данных из набора.

Набор реплик Ряд узлов, на которых хранятся копии набора данных. У каждого хранилища в наборе реплик есть роль: мастер или реплика.

Мастер Хранилище в наборе реплик, которое обрабатывает запросы на чтение и запись.

Реплика Хранилище в наборе реплик, которое обрабатывает только запросы на чтение.

Запросы на чтение Запросы только на чтение, то есть выборка.

Запросы на запись Операции по изменению данных, то есть запросы на создание, замену, обновление и удаление данных.

Сегменты (виртуальные сегменты) Абстрактные виртуальные узлы, на которые производится секционирование набора данных по сегментному ключу (идентификатору сегмента).

Идентификатор сегмента A sharding key defining which bucket belongs to which replica set. A bucket id may be calculated from a *hash key*.

Роутер Прокси-сервер, который отвечает за запросы маршрутизации от приложения к узлам в кластере.

Администрирование

Установка

Пакет `vshard` распространяется отдельно от основного пакета Tarantool'а. Для установки выполните команду:

```
$ tarantoolctl rocks install vshard
```

Примечание: Для работы с модулем `vshard` необходимо, чтобы были установлены: Tarantool версии 1.9+., пакет программ для разработки Tarantool'а, `git`, `cmake` и `gcc`.

Настройка

Любой рабочий сегментированный кластер состоит из:

- одного или нескольких наборов реплик с двумя или несколькими *хранилищами* в каждом,
- одного или нескольких *роутеров*.

Количество хранилищ в наборе реплик определяет коэффициент избыточности данных. Рекомендуемое значение: 3 или более. Количество роутеров не ограничено, потому что у роутеров нет состояния. Рекомендуем увеличивать количество роутеров, если существующий экземпляр роутера ограничен возможностями процессора или ввода-вывода.

`vshard` поддерживает работу с несколькими роутерами в отдельном экземпляре Tarantool'а. Каждый роутер может подключиться к любому кластеру `vshard`. Несколько роутеров могут быть подключены к одному кластеру.

Поскольку приложения роутера (`router`) и хранилища (`storage`) выполняют совершенно разные наборы функций, их следует разворачивать на различных экземплярах Tarantool'а. Хотя технически возможно разместить приложение роутера на каждом узле типа хранилища, такой подход крайне не рекомендуется, и его следует избегать при развертывании в производственной среде.

Все хранилища можно развернуть, используя один набор файлов экземпляра (конфигурационных файлов).

Самоопределение в настоящий момент осуществляется с помощью `tarantoolctl`:

```
$ tarantoolctl имя_экземпляра
```

Все роутеры также можно развернуть, используя один набор файлов экземпляра (конфигурационных файлов).

Топология всех узлов кластера должна быть одинаковой. Администратор должен убедиться, что конфигурации совпадают. Рекомендуем использовать инструмент управления конфигурациями, такой как Ansible или Puppet, во время развертывания кластера.

Шардинг не интегрирован ни в одну систему для централизованного управления конфигурациями. Предполагается, что само приложение отвечает за взаимодействие с такой системой и передачу параметров шардинга.

Пример настройки простого сегментированного кластера можно найти *здесь*.

Вес реплики

Роутер отправляет все запросы чтения и записи только на мастер-экземпляр. Задав вес реплики, можно разрешить отправку запросов только на чтение не только на мастер-экземпляр, но и на доступную реплику, которая находится ближе всего к роутеру. Вес используется для определения расстояния между репликами в наборе реплик.

Например, вес можно использовать для определения физического расстояния между роутером и каждой репликой в наборе реплик. В таком случае запросы на чтение будут отправляться на ближайшую реплику (с наименьшим весом).

Кроме того, можно задать вес реплик, чтобы определить наиболее мощную реплику, которая может обрабатывать наибольшее количество запросов в секунду.

Основная идея состоит в том, чтобы указать зону для каждого `роутера` и каждой реплики, и таким образом составить матрицу относительных весов зоны. Этот подход позволяет устанавливать разный вес в разных зонах для одного набора реплик.

Чтобы задать вес, используйте атрибут zone (зона) для каждой реплики во время конфигурации:

```
local cfg = {
   sharding = {
      ['...uuid_набора_реплик...'] = {
         replicas = {
            ['...uuid_реплики...'] = {
                 ...,
                 zone = <число или строка>
            }
         }
      }
   }
}
```

Затем укажите относительный вес для каждой пары зон в параметре `weights` (вес) в `vshard.router.cfg`. Например:

```
weights = {
    [1] = {
        [2] = 1, -- Роутеры 1 зоны видят вес 2 зоны = 1.
        [3] = 2, -- Роутеры 1 зоны видят вес 3 зоны = 2 .
        [4] = 3, -- ...
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    },
    [2] = {
        [1] = 10,
        [2] = 0,
        [3] = 10,
        [4] = 20,
    },
    [3] = {
        [1] = 100,
        [2] = 200, -- Роутеры 3 зоны видят вес 2 зоны = 200.
                   -- Обратите внимание, что этот вес не равен весу 2 зоны (= 2),
                   -- который видят роутеры 1 зоны (= 1).
        [4] = 1000,
    }
}

local cfg = vshard.router.cfg({weights = weights, sharding = ...})
```

Вес набора реплик

Вес набора реплик не равноценен весу реплики. Вес набора реплик определяет производительность набора реплик: чем больше вес, тем больше сегментов может хранить набор реплик. Общий размер всех сегментированных спейсов в наборе реплик также определяет его производительность.

Вес набора реплик можно рассматривать как относительный объем данных в наборе реплик. Например, если `replicaset_1 = 100`, и `replicaset_2 = 200`, второй набор реплик хранит в два раза больше сегментов, чем первый. По умолчанию веса всех наборов реплик равны.

Вес можно использовать, к примеру, чтобы хранить преобладающий объем данных в наборе реплик с большим объемом памяти.

Процесс балансировки

Существует **эталонное число** сегментов в наборе реплик («эталонный» в данном случае значит идеальный). Если во всем наборе реплик это число остается неизменным, то сегменты распределяются равномерно.

Эталонное число рассчитывается автоматически с учетом количества сегментов в кластере и веса наборов реплик.

Балансировка начинается, когда **предел дисбаланса в наборе реплик** превышает предел дисбаланса, *указанный в конфигурации*.

Предел дисбаланса набора реплик рассчитывается следующим образом:

```
|эталонное_число_сегментов - текущее_число_сегментов| / эталонное_число_сегментов * 100
```

Например: Пользователь указал, что количество сегментов = 3000, а вес 3 наборов реплик составляет 1, 0,5 и 1,5. В результате получаем следующее эталонное число сегментов для наборов реплик: 1 набор реплик – 1000, 2 набор реплик – 500, 3 набор реплик – 1500.

Такой подход позволяет назначить нулевой вес для набора реплик, который запускает миграцию сегментов на оставшиеся узлы кластера. Это также позволяет добавить новый набор реплик с нулевой нагрузкой, который запускает миграцию сегментов из загруженных наборов реплик в набор реплик с нулевой нагрузкой.

Примечание: Новому набору реплик с нулевой нагрузкой следует присвоить вес, чтобы начать процесс балансировки.

При добавлении нового шарда конфигурацию можно обновить динамически:

1. Конфигурацию следует сначала обновить на всех роутерах, а затем на всех хранилищах.
2. Новый шард становится доступен для балансирования на уровне хранилища.
3. В результате балансировки происходит миграция сегментов на новый шард.
4. Если происходит запрос к перемещенному сегменту, `роутер` получает код ошибки с информацией о новом местонахождении сегмента.

В это время новый шард уже включен в пул соединений `роутера`, поэтому переадресация видима для приложения.

Параллельная балансировка

Балансировщик в `vshard` первоначально был довольно прост: один процесс на одном узле, который рассчитывал *маршруты* отправки сегментов, сколько их отправлять и куда. Узлы применяли эти маршруты один за другим последовательно.

К сожалению, такая простая схема работала недостаточно быстро, особенно для Vinyl'a, где затраты ресурсов на чтение диска были сопоставимы с сетевыми затратами. На самом деле, механизм применения маршрутов в балансировщике Vinyl'a большую часть времени был в режиме ожидания.

Теперь каждый узел может параллельно посылать несколько сегментов по кругу в несколько пунктов назначения или всего в один.

Чтобы определять степень параллельности, используется новая опция *rebalancer_max_sending*. Задавать ее можно в конфигурации хранилища в корневой таблице:

```
cfg.rebalancer_max_sending = 5
vshard.storage.cfg(cfg, box.info.uuid)
```

Этот параметр не учитывается для роутеров.

Примечание: Задав `cfg.rebalancer_max_sending = N`, вы вряд ли получите N-кратное ускорение. На это влияют многие факторы: сеть, диск, количество других файберов в системе.

Пример №1:

У вас уже есть 10 наборов реплик, добавили новый. Теперь все 10 наборов реплик будут пытаться отправить сегменты на новый.

Предположим, каждый набор реплик может отправить до 5 сегментов одновременно. В этом случае будет довольно большая нагрузка на новый набор реплик: одновременная загрузка 50 сегментов. Если узлу нужно выполнить какую-то другую работу, возможно, такая большая нагрузка нежелательна. Кроме того, слишком большое количество параллельно загружаемых сегментов может привести к задержкам самого процесса балансировки.

Чтобы исправить это, можно установить меньшее значение `rebalancer_max_sending` для старых наборов реплик или же уменьшить `rebalancer_max_receiving` для нового набора реплик. В последнем случае будет происходить управление загрузкой на старых узлах, и вы увидите это в логах.

Важно значение параметра `rebalancer_max_sending`, если у вас есть ограничение на максимальное количество сегментов, которые могут быть одновременно доступны только для чтения в кластере. Как упоминалось выше, во время отправки сегмент не принимает новые запросов на запись.

Пример №2:

> У вас есть 100 000 сегментов, и каждый сегмент хранит ~ 0,001% ваших данных. В кластере 10 наборов реплик. И нельзя позволить себе заблокировать для записи > 0,1% данных. Таким образом, не следует устанавливать значение `rebalancer_max_sending` > 10 на этих узлах. Тогда балансировщик не будет посылать более 100 сегментов одновременно по всему кластеру.

Если значение `max_sending` задано слишком высоко, а `max_receiving` слишком низко, то некоторые сегменты будут пытаться переместиться – и не смогут. При этом будут расходоваться сетевые ресурсы и время. Важно настроить эти параметры так, чтобы они не конфликтовали друг с другом.

Блокировка набора реплик и закрепление корзины

Блокировка набора реплик делает набор реплик невидимым для балансировщика: заблокированный набор реплик не может ни принимать новые сегменты, ни мигрировать собственные сегменты.

В результате закрепления сегмента определенный сегмент блокируется для миграции: закрепленный сегмент остается в наборе реплик, в котором он закреплен, до отмены закрепления.

Закрепление всех сегментов в наборе реплик не означает блокирование набора реплик. Даже после закрепления всех сегментов незаблокированный набор реплик может принимать новые сегменты.

Блокировка набора реплик используется, к примеру, чтобы выделить для тестирования набор реплик из наборов реплик, используемых в производстве, или чтобы сохранить некоторые метаданные приложения, которые в течение некоторого времени не должны быть сегментированы. Закрепление сегмента используется в похожих случаях, но в меньшем масштабе.

Блокировка набора реплик и закрепление всех сегментов означает изоляцию целого набора реплик.

Заблокированные наборы реплик и закрепленные сегменты влияют на алгоритм балансировки, так как `балансировщик` должен игнорировать заблокированные наборы реплик и учитывать закрепленные сегменты при попытке достичь наилучшего возможного баланса.

Это нетривиальная задача, поскольку пользователь может закрепить слишком много сегментов в наборе реплик, так что становится невозможным достижение идеального баланса. Например, рассмотрим следующий кластер (предположим, что все веса наборов реплик равны 1).

Начальная конфигурация:

```
rs1: bucket_count = 150 -- число сегментов
rs2: bucket_count = 150, pinned_count = 120 -- число сегментов, число закрепленных сегментов
```

Добавление нового набора реплик:

```
rs1: bucket_count = 150
rs2: bucket_count = 150, pinned_count = 120
rs3: bucket_count = 0
```

Идеальным балансом было бы 100 - 100 - 100, чего невозможно достичь, поскольку набор реплик `rs2` содержит 120 закрепленных сегментов. The best possible balance here is the following:

```
rs1: bucket_count = 90
rs2: bucket_count = 120, pinned_count 120
rs3: bucket_count = 90
```

`Балансировщик` переместил максимально возможное количество сегментов из `rs2`, чтобы уменьшить дисбаланс. В то же время он учел одинаковый вес respected `rs1` и `rs3`.

Алгоритмы реализации блокировки и закрепления совершенно разные, хотя с точки зрения функций они похожи.

Заблокированный набор реплик и балансировка

Заблокированные наборы реплик просто не участвуют в балансировке. Это означает, что даже если фактическое общее количество сегментов не равно эталонному числу, дисбаланс нельзя исправить из-за блокировки. Когда балансировщик обнаруживает, что один из наборов реплик заблокирован, он пересчитывает эталонное число сегментов неблокированных наборов реплик, как если бы заблокированный набор реплик и его сегменты вообще не существовали.

Закрепленный набор реплик и балансировка

Балансировка наборов реплик с закрепленными сегментами требует более сложного алгоритма. Здесь `pinned_count[o]` – это число закрепленных сегментов, а `etalon_count` – это эталонное число сегментов для набора реплик:

1. Балансировщик рассчитывает эталонное число сегментов, как если бы все сегменты не были закреплены. Затем балансировщик проверяет каждый набор реплик и сопоставляет эталонное число сегментов с числом закрепленных сегментов в наборе реплик. Если `pinned_count < etalon_count`, незаблокированные наборы реплик (на данном этапе все заблокированные наборы реплик уже отфильтрованы) с закрепленными сегментами могут получать новые сегменты.
2. Если же `pinned_count > etalon_count`, дисбаланс исправить нельзя, так как балансировщик не может вывести закрепленные сегменты из этого набора реплик. В таком случае эталонное число обновляется как равное числу закрепленных сегментов. Наборы реплик с `pinned_count > etalon_count` не обрабатываются балансировщиком‘, а число закрепленных сегментов вычитается из общего числа сегментов. Балансировщик пытается вывести как можно больше сегментов из таких наборов реплик.
3. Эта процедура перезапускается с шага 1 для наборов реплик с `pinned_count >= etalon_count` до тех пор, пока не будет выполнено условие `pinned_count <= etalon_count` для всех наборов реплик. Процедура также перезапускается при изменении общего числа сегментов.

Псевдокод для данного алгоритма будет следующим:

```
function cluster_calculate_perfect_balance(replicasets, bucket_count)
        -- балансировка сегментов с использованием веса рабочих наборов реплик --
end;

cluster = <all of the non-locked replica sets>;
bucket_count = <the total number of buckets in the cluster>;
can_reach_balance = false
while not can_reach_balance do
        can_reach_balance = true
        cluster_calculate_perfect_balance(cluster, bucket_count);
        foreach replicaset in cluster do
                if replicaset.perfect_bucket_count <
                   replicaset.pinned_bucket_count then
                        can_reach_balance = false
                        bucket_count -= replicaset.pinned_bucket_count;
                        replicaset.perfect_bucket_count =
```

(continues on next page)

(продолжение с предыдущей страницы)

```
                                replicaset.pinned_bucket_count;
                end;
        end;
end;
cluster_calculate_perfect_balance(cluster, bucket_count);
```

Сложность алгоритма составляет `O(N^2)`, где N – количество наборов реплик. На каждом шаге алгоритм либо завершает вычисление, либо игнорирует хотя бы один новый набор реплик, перегруженный закрепленными сегментами, и обновляет эталонное число сегментов в других наборах реплик.

Ссылка в сегменте

Ссылка в сегменте – это счетчик в оперативной памяти, который похож на *закрепление сегмента* со следующими отличиями:

1. Ссылка в сегменте никогда не сохраняется. Ссылки предназначены для запрета передачи сегментов во время выполнения запроса, но при перезапуске все запросы отбрасываются.
2. Есть 2 типа ссылок в сегменте: только чтение (RO) и чтение-запись (RW).

   If a bucket has RW refs, it cannot be moved. However, when the rebalancer needs it to be sent, it locks the bucket for new write requests, waits until all current requests are finished, and then sends the bucket.

   Если в сегменте есть ссылки типа RO, его можно отправить, но нельзя удалить. Такой сегмент может даже перейти в статус мусора GARBAGE или отправки SENT, но его данные сохраняются до тех пор, пока не уйдет последний читатель.

   В одном сегменте могут быть ссылки как типа RO, так и типа RW.
3. Ссылки в сегменте исчисляются.

Методы *vshard.storage.bucket_ref/unref()* вызываются автоматически при использовании *vshard.router.call()* или *vshard.storage.call()*. При использовании API, например `r = vshard.router.route() r:callro/callrw`, следует дополнительно вызвать метод `bucket_ref()` в рамках функции. Кроме того, следует убедиться, что после `bucket_ref()` вызывается `bucket_unref()`, иначе сегмент нельзя перемещать из хранилища до перезапуска экземпляра.

Чтобы узнать количество ссылок в сегменте, используйте *vshard.storage.buckets_info([идентификатор_сегмента])* (параметр `идентификатор_сегмента` необязателен).

Пример:

```
vshard.storage.buckets_info(1)
---
- 1:
    status: active
    ref_rw: 1
    ref_ro: 1
    ro_lock: true
    rw_lock: true
    id: 1
```

Определение спейса

Схема базы данных хранится на хранилищах, а роутеры ничего не знают о спейсах и кортежах.

В приложении хранилища следует определить спейсы с помощью box.once(). Например:

```
box.once("testapp:schema:1", function()
    local customer = box.schema.space.create('customer')
    customer:format({
        {'customer_id', 'unsigned'},
        {'bucket_id', 'unsigned'},
        {'name', 'string'},
    })
    customer:create_index('customer_id', {parts = {'customer_id'}})
    customer:create_index('bucket_id', {parts = {'bucket_id'}, unique = false})

    local account = box.schema.space.create('account')
    account:format({
        {'account_id', 'unsigned'},
        {'customer_id', 'unsigned'},
        {'bucket_id', 'unsigned'},
        {'balance', 'unsigned'},
        {'name', 'string'},
    })
    account:create_index('account_id', {parts = {'account_id'}})
    account:create_index('customer_id', {parts = {'customer_id'}, unique = false})
    account:create_index('bucket_id', {parts = {'bucket_id'}, unique = false})
    box.snapshot()

    box.schema.func.create('customer_lookup')
    box.schema.role.grant('public', 'execute', 'function', 'customer_lookup')
    box.schema.func.create('customer_add')
end)
```

Примечание: В каждом спейсе, который вы планируете шардировать, должно быть поле с *идентификаторами сегментов*, проиндексированное с помощью *shard index*.

Добавление данных

Все DML-операции с данными следует выполнять через роутер. Роутер поддерживает только вызов *CALL* через идентификатор сегмента bucket_id:

```
result = vshard.router.call(идентификатор_сегмента, режим, функция, аргументы)
```

vshard.router.call() направляет вызов result = func(unpack(args)) на шард, который обслуживает идентификатор сегмента bucket_id.

Идентификатор сегмента bucket_id – это обычное число в диапазоне 1...*bucket_count*». Этот номер можно произвольным образом назначить с помощью клиентского приложения. Сегментированный кластер Tarantool использует этот номер в качестве непрозрачного уникального идентификатора для распределения данных по множествам реплик. Мы гарантируем, что все записи с одним и тем же bucket_id будут храниться в одном и том же наборе реплик.

Настройка и перезапуск хранилища

В случае отказа мастера в наборе реплик рекомендуется:

1. Переключить одну из реплик в режим мастера, что позволит новому мастеру обрабатывать все входящие запросы.
2. Обновить конфигурацию всех членов кластера, в результате чего все запросы будут перенаправлены на новый мастер.

Мониторинг состояния мастера и переключение режимов экземпляров можно осуществлять с помощью внешней утилиты.

Для проведения запланированного остановки мастера в наборе реплик рекомендуется:

1. Обновить конфигурацию мастера и подождать синхронизации всех реплик, в результате чего все запросы будут перенаправлены на новый мастер.
2. Переключить другой экземпляр в режим мастера.
3. Обновить конфигурацию всех узлов.
4. Отключить старый мастер.

Для проведения запланированной остановки набора реплик рекомендуется:

1. Произвести миграцию всех сегментов в другие хранилища кластера.
2. Обновить конфигурацию всех узлов.
3. Отключить набор реплик.

В случае отказа всего набора реплик некоторая часть набора данных становится недоступной. Тем временем `роутер` пытается повторно подключиться к мастеру отказавшего набора реплик. Таким образом, после того, как набор реплик снова запущен, кластер автоматически восстанавливается.

Файберы

Поиск сегментов, восстановление сегментов и балансировка сегментов выполняются автоматически и не требуют ручного вмешательства.

С технической точки зрения есть несколько файберов, которые отвечают за различные типы действий:

- файбер **обнаружения** на `роутере` выполняет поиск сегментов в фоновом режиме
- файбер **восстановления после отказа** на `роутере` поддерживает соединения с репликами
- файбер **сборки мусора** на каждом мастер-хранилище удаляет содержимое перемещенных сегментов
- файбер **восстановления сегмента** на каждом мастер-хранилище восстанавливает сегменты в статусах отправки SENDING и получения RECEIVING в случае перезагрузки
- **балансировщик** на отдельном мастер-хранилище среди множества наборов реплик выполняет процесс балансировки.

Для получения подробной информации см. разделы *Процесс балансировки* и *Миграция сегментов*.

Сборщик мусора

Файбер **сборщик мусора** работает в фоновом режиме на мастер-хранилищах в каждом наборе реплик. Он начинает удалять содержимое сегмента в состоянии мусора GARBAGE по частям. Когда сегмент пуст, запись о нем удаляется из системного спейса `_bucket`.

Восстановление сегмента

Файбер **восстановления сегмента** работает на мастер-хранилищах. Он помогает восстановить сегменты в статусах отправки SENDING и получения RECEIVING в случае перезагрузки.

Сегменты в статусе SENDING восстанавливаются следующим образом:

1. Сначала система ищет сегменты в статусе SENDING.
2. Если такой сегмент обнаружен, система отправляет запрос в целевой набор реплик.
3. Если сегмент в целевом наборе реплик находится в активном статусе ACTIVE, исходный сегмент удаляется из исходного узла.

Сегменты в статусе RECEIVING удаляются без дополнительных проверок.

Восстановление после отказа

Файбер **восстановления после отказа** работает на каждом роутере. Если мастер набора реплик становится недоступным, файбер перенаправляет запросы на чтение к репликам. Запросы на запись отклоняются с ошибкой до тех пор, пока мастер не будет доступен.

Руководство по быстрому запуску

Чтобы получить инструкции по установке, обратитесь к *руководству по установке vshard*.

Предварительно настроенный кластер можно найти в директории `example/` репозитория vshard. Этот пример включает в себя 5 экземпляров Tarantool'а и 2 набора реплик:

- `router_1` – экземпляр роутера (`router`)
- `storage_1_a` – экземпляр хранилища (`storage`), **мастер первого** набора реплик
- `storage_1_b` – экземпляр хранилища (`storage`), **реплика** из **первого** набора реплик
- `storage_2_a` – экземпляр хранилища (`storage`), **мастер второго** набора реплик
- `storage_2_b` – экземпляр хранилища (`storage`), **реплика** из **второго** набора реплик

Управление всеми экземплярами осуществляется с помощью утилиты `tarantoolctl`.

Измените директорию `example/` и используйте команду `make` для запуска кластера:

```
$ cd example/
$ make
tarantoolctl stop storage_1_a  # stop the first storage instance
Stopping instance storage_1_a...
tarantoolctl stop storage_1_b
<...>
rm -rf data/
tarantoolctl start storage_1_a # start the first storage instance
Starting instance storage_1_a...
Starting configuration of replica 8a274925-a26d-47fc-9e1b-af88ce939412
I am master
Taking on replicaset master role...
Run console at unix/:./data/storage_1_a.control
started
mkdir ./data/storage_1_a
<...>
```

(continues on next page)

(продолжение с предыдущей страницы)

```
tarantoolctl start router_1 # start the router
Starting instance router_1...
Starting router configuration
Calling box.cfg()...
<...>
Run console at unix/:./data/router_1.control
started
mkdir ./data/router_1
Waiting cluster to start
echo "vshard.router.bootstrap()" | tarantoolctl enter router_1
connected to unix/:./data/router_1.control
unix/:./data/router_1.control> vshard.router.bootstrap()
---
- true
...
unix/:./data/router_1.control>
tarantoolctl enter router_1 # enter the admin console
connected to unix/:./data/router_1.control
unix/:./data/router_1.control>
```

Некоторые команды `tarantoolctl`:

- `tarantoolctl start router_1` – запуск экземпляра роутера
- `tarantoolctl enter router_1` – вход в административную консоль

Полный список команд `tarantoolctl` для управления экземплярами Tarantool'а можно найти в *справочнике по tarantoolctl*.

Необходимо знать следующие команды `make`:

- `make start` – запуск всех экземпляров Tarantool'а
- `make stop` – остановка всех экземпляров Tarantool'а
- `make logcat` – вывод журналов всех экземпляров
- `make enter` – вход в административную консоль на роутере `router_1`
- `make clean` – очистка всех персистентных данных
- `make test` – запуск набора тестов (можно также выполнить `test-run.py` в директории с тестами `test`)
- `make` – выполнить `make stop`, `make clean`, `make start` и `make enter`

Например, для запуска всех экземпляров используйте `make start`:

```
$ make start
$ ps x|grep tarantool
46564   ??  Ss     0:00.34 tarantool storage_1_a.lua <running>
46566   ??  Ss     0:00.19 tarantool storage_1_b.lua <running>
46568   ??  Ss     0:00.35 tarantool storage_2_a.lua <running>
46570   ??  Ss     0:00.20 tarantool storage_2_b.lua <running>
46572   ??  Ss     0:00.25 tarantool router_1.lua <running>
```

Для выполнения команд в административной консоли, используйте *общедоступный API*:

```
unix/:./data/router_1.control> vshard.router.info()
---
```

(continues on next page)

(продолжение с предыдущей страницы)

```
- replicasets:
    ac522f65-aa94-4134-9f64-51ee384f1a54:
      replica: &0
        network_timeout: 0.5
        status: available
        uri: storage@127.0.0.1:3303
        uuid: 1e02ae8a-afc0-4e91-ba34-843a356b8ed7
      uuid: ac522f65-aa94-4134-9f64-51ee384f1a54
      master: *0
    cbf06940-0790-498b-948d-042b62cf3d29:
      replica: &1
        network_timeout: 0.5
        status: available
        uri: storage@127.0.0.1:3301
        uuid: 8a274925-a26d-47fc-9e1b-af88ce939412
      uuid: cbf06940-0790-498b-948d-042b62cf3d29
      master: *1
  bucket:
    unreachable: 0
    available_ro: 0
    unknown: 0
    available_rw: 3000
  status: 0
  alerts: []
...
```

Образец конфигурации

Конфигурация простого сегментированного кластера может выглядеть следующим образом:

```
local cfg = {
    memtx_memory = 100 * 1024 * 1024,
    replication_connect_quorum = 0,
    bucket_count = 10000,
    rebalancer_disbalance_threshold = 10,
    rebalancer_max_receiving = 100,
    sharding = {
        ['cbf06940-0790-498b-948d-042b62cf3d29'] = {
            replicas = {
                ['8a274925-a26d-47fc-9e1b-af88ce939412'] = {
                    uri = 'storage:storage@127.0.0.1:3301',
                    name = 'storage_1_a',
                    master = true
                },
                ['3de2e3e1-9ebe-4d0d-abb1-26d301b84633'] = {
                    uri = 'storage:storage@127.0.0.1:3302',
                    name = 'storage_1_b'
                }
            },
        },
        ['ac522f65-aa94-4134-9f64-51ee384f1a54'] = {
            replicas = {
                ['1e02ae8a-afc0-4e91-ba34-843a356b8ed7'] = {
                    uri = 'storage:storage@127.0.0.1:3303',
                    name = 'storage_2_a',
```

(continues on next page)

(продолжение с предыдущей страницы)

```
                    master = true
                },
                ['001688c3-66f8-4a31-8e19-036c17d489c2'] = {
                    uri = 'storage:storage@127.0.0.1:3304',
                    name = 'storage_2_b'
                }
            },
        },
    },
}
```

Данный кластер включает в себя один роутер (`router`) и два хранилища (`storage`). Каждое хранилище `storage` включает в себя один мастер и одну реплику. Поле `sharding` (шардинг) определяет логическую топологию сегментированного кластера Tarantool'а. Все остальные поля передаются в `box.cfg()` в неизменном виде. Для получения подробной информации см. раздел *Справочник по настройке*.

На роутерах вызовите `vshard.router.cfg(cfg)`:

```
cfg.listen = 3300

-- Запуск базы данных с шардингом
vshard = require('vshard')
vshard.router.cfg(cfg)
```

На хранилищах вызовите `vshard.storage.cfg(cfg, uuid_экземпляра)`:

```
-- Получение имени экземпляра
local MY_UUID = "de0ea826-e71d-4a82-bbf3-b04a6413e417"

-- Вызов поставщика конфигурации
local cfg = require('localcfg')

-- Запуск базы данных с шардингом
vshard = require('vshard')
vshard.storage.cfg(cfg, MY_UUID)
```

`vshard.storage.cfg()` автоматически вызывает `box.cfg()` и настраивает порт для прослушивания и параметры репликации.

Образец конфигурации можно посмотреть в файлах `router.lua` и `storage.lua` в директории `example/` репозитория vshard.

Справочник по настройке

Базовые параметры

- *sharding*
- *weights*
- *shard_index*
- *bucket_count*
- *collect_bucket_garbage_interval*
- *collect_lua_garbage*

- *sync_timeout*
- *rebalancer_disbalance_threshold*
- *rebalancer_max_receiving*
- *rebalancer_max_sending*
- *discovery_mode*

`sharding`
: Поле, которое определяет логическую топологию сегментированного кластера Tarantool'a.

  Тип: таблица

  По умолчанию: false (ложь)

  Динамический: да

`weights`
: Поле, которое определяет конфигурацию относительного веса для каждой пары зон в наборе реплик. См. раздел *Вес реплики*.

  Тип: таблица

  По умолчанию: false (ложь)

  Динамический: да

`shard_index`
: Название или id TREE-индекса по *идентификатору сегмента*. Спейсы без этого индекса не задействованы в шардированном кластере Tarantool'a и при необходимости могут быть использованы как обычные спейсы. Необходимо указать первую часть индекса, остальные части являются необязательными.

  Тип: непустая строка или неотрицательное целое число

  По умолчанию: «bucket_id» (идентификатор сегмента)

  Динамический: нет

`bucket_count`
: Общее число сегментов в кластере.

  Это число должно быть на несколько порядков больше, чем потенциальное число узлов кластера, учитывая потенциальное масштабирование в обозримом будущем.

  Пример:

  Если предполагаемое количество узлов равно M, тогда набор данных должен быть разделен на 100M или даже 1000M сегментов, в зависимости от запланированного масштабирования. Это число, безусловно, больше потенциального числа узлов кластера в проектируемой системе.

  Следует помнить, что слишком большое число сегментов может привести к необходимости выделять больше памяти для хранения информации о маршрутизации. С другой стороны, недостаточное число сегментов может привести к снижению степени детализации при балансировке.

Тип: число

По умолчанию: 3000

Динамический: нет

`collect_bucket_garbage_interval`

Интервал между действиями сборщика мусора в секундах.

Тип: число

По умолчанию: 0.5

Динамический: да

`collect_lua_garbage`

Если задано значение true (правда), периодически вызывается Lua-функция `collectgarbage()`.

Тип: логический

По умолчанию: нет

Динамический: да

`sync_timeout`

Время ожидания синхронизации старого мастера с репликами перед сменой мастера. Используется при переключении мастера или при вызове функции `sync()` вручную.

Тип: число

По умолчанию: 1

Динамический: да

`rebalancer_disbalance_threshold`

Максимальный предел дисбаланса сегментов в процентах. Предел вычисляется для каждого набора реплик по следующей формуле:

```
|эталонное_число_сегментов - фактическое_число_сегментов| / эталонное_число_сегментов * 100
```

Тип: число

По умолчанию: 1

Динамический: да

`rebalancer_max_receiving`

Максимальное количество сегментов, которые может получить параллельно один набор реплик. Это число должно быть ограничено, так как при добавлении нового набора реплик в кластер балансировщик отправляет очень большое количество сегментов из существующих наборов реплик в новый набор реплик. Это создает большую нагрузку на новый набор реплик.

Пример:

Предположим, `rebalancer_max_receiving` = 100, число сегментов в `bucket_count` = 1000. Есть 3 набора реплик с 333, 333 и 334 сегментами соответственно. При добавлении нового набора реплик `эталонное_число_сегментов` становится равным 250. Вместо того, чтобы сразу получить все 250 сегментов, новый набор реплик получит последовательно 100, 100 и 50 сегментов.

Тип: число
По умолчанию: 100
Динамический: да

`rebalancer_max_sending`
Степень параллельности для *параллельной балансировки*.

Используется только для хранилищ, для роутеров игнорируется.

Максимальное значение: 15.

Тип: число
По умолчанию: 1
Динамический: да

`discovery_mode`
Режим работы файбера обнаружения сегментов: `on`/`off`/`once`. *Подробнее*.

Тип: строка
По умолчанию: «on»
Динамический: да

Функции набора реплик

- *uuid*
- *weight*

`uuid`
Уникальный идентификатор набора реплик.

Тип:
По умолчанию:
Динамическое:

`weight`
Вес набора реплик. Для получения подробной информации см. раздел *Вес набора реплик*.

Тип:
По умолчанию: 1

Динамическое:

API reference

В этом разделе представлен общедоступный и внутренний API для *роутера* и для *хранилища*.

Раздел	Методы
Общедоступный API роутера	• *vshard.router.bootstrap()* • *vshard.router.cfg(cfg)* • *vshard.router.new(name, cfg)* • *vshard.router.call(bucket_id, mode, function_name, {argument_list}, {options})* • *vshard.router.callro(bucket_id, function_name, {argument_list}, {options})* • *vshard.router.callrw(bucket_id, function_name, {argument_list}, {options})* • *vshard.router.callre(bucket_id, function_name, {argument_list}, {options})* • *vshard.router.callbro(bucket_id, function_name, {argument_list}, {options})* • *vshard.router.callbre(bucket_id, function_name, {argument_list}, {options})* • *vshard.router.route(bucket_id)* • *vshard.router.routeall()* • *vshard.router.bucket_id_strcrc32(key)* • *vshard.router.bucket_id_mpcrc32(key)* • *vshard.router.bucket_count()* • *vshard.router.sync(timeout)* • *vshard.router.discovery_wakeup()* • *vshard.router.discovery_set()* • *vshard.router.info()* • *vshard.router.buckets_info()* • *replicaset_object:call()* • *replicaset_object:callro()* • *replicaset_object:callrw()* • *replicaset_object:callre()*
Внутренний API роутера	• *vshard.router.bucket_discovery(bucket_id)*
Общедоступный API хранилища	• *vshard.storage.cfg(cfg, name)* • *vshard.storage.info()* • *vshard.storage.call(bucket_id, mode, function_name, {argument_list})* • *vshard.storage.sync(timeout)* • *vshard.storage.bucket_pin(bucket_id)* • *vshard.storage.bucket_unpin(bucket_id)* • *vshard.storage.bucket_ref(bucket_id, mode)* • *vshard.storage.bucket_refro()* • *vshard.storage.bucket_refrw()* • *vshard.storage.bucket_unref(bucket_id, mode)* • *vshard.storage.bucket_unrefro()* • *vshard.storage.bucket_unrefrw()* • *vshard.storage.find_garbage_bucket(bucket_index, control)* • *vshard.storage.rebalancer_disable()* • *vshard.storage.rebalancer_enable()* • *vshard.storage.is_locked()* • *vshard.storage.rebalancing_is_in_progress()*
5.3. Справочник по сторонним библиотекам	• *vshard.storage.buckets_info()* • *vshard.storage.buckets_count()* • *vshard.storage.sharded_spaces()*

Общедоступный API роутера

vshard.router.bootstrap()
Инициализация кластера и распределение всех сегментов по наборам реплик.

Параметры

- `timeout` – количество секунд ожидания до признания попытки инициализации неуспешной. Пересоздайте кластер в случае блокировки инициализации по истечении времени ожидания.
- `if_not_bootstrapped` – По умолчанию `false`, то есть «вызвать ошибку, если кластер уже был инициализирован». `True` значит «если кластер уже был инициализирован, то ничего не делать.»

Пример:

```
vshard.router.bootstrap({timeout = 4, if_not_bootstrapped = true})
```

Примечание: Чтобы определить, инициализирован ли кластер , vshard ищет по крайней мере один сегмент во всем кластере. Если кластер был инициализирован частично (например, из-за ошибки при первой инициализации),то он все равно будет считаться инициализированным при следующей попытке инициализации с флагом `if_not_bootstrapped`. Поэтому лучше избегать вызова `bootstrap()` несколько раз.

---

vshard.router.cfg(*cfg*)
Настройка базы данных и начало шардинга указанного `роутера`. См. *образец конфигурации*.

Параметры

- `cfg` – конфигурационная таблица

vshard.router.new(*name*, *cfg*)
Создание нового экземпляра роутера. `vshard` поддерживает работу нескольких роутеров в отдельном экземпляре Tarantool'а. Каждый роутер может подключаться к любому кластеру `vshard`, несколько роутеров могут подключаться к одному кластеру.

Роутер, созданный с помощью `vshard.router.new()`, работает так же, как и статичный роутер, но перед его методами указывается двоеточие (`vshard.router:имя_метода(...)`), а перед методами статичного роутера – точка (`vshard.router.имя_метода(...)`).

Статичный роутер можно получить при помощи метода `vshard.router.static()`, а затем использовать его как роутер, созданный с помощью метода `vshard.router.new()`.

Примечание: `box.cfg` используется всеми роутерами одного экземпляра.

Параметры

- `name` – имя экземпляра роутера, которое используется в качестве префикса в журналах роутера и должно быть уникальным в пределах экземпляра
- `cfg` – конфигурационная таблица. См. *образец конфигурации*.

Возвращается экземпляр роутера, если он создан; в противном случае, nil и ошибка

vshard.router.call(*bucket_id*, *mode*, *function_name*, *{argument_list}*, *{options}*)

Вызов функции по имени функции (function-name) на шарде, где хранится сегмент с указанным идентификатором (bucket_id). Для получения подробной информации о работе функции см. раздел *Обработка запросов*.

Параметры

- `bucket_id` – идентификатор сегмента
- `mode` – либо строка = „read“|“write“ (чтение|запись), либо ассоциативный массив с параметром mode =“read“|“write“ (чтение|запись) и/или prefer_replica=true|false (правда|ложь), и/или balance=true|false (правда|ложь).
- `function_name` – выполняемая функция
- `argument_list` – массив аргументов функции
- `options` –
  - `timeout` – время ожидания запроса в секундах. Если `роутер` не может определить шард с указанным идентификатором сегмента `bucket_id`, операция повторяется до истечения времени ожидания.
  - другие *net.box опции*, такие как `is_async`, `buffer`, `on_push` также поддерживаются.

У параметра режима mode есть две доступные формы: строка или ассоциативный массив. Примеры строки: `'read'` (чтение), `'write'` (запись). Примеры ассоциативного массива: `{mode='read'}`, `{mode='write'}`, `{mode='read', prefer_replica=true}`, `{mode='read', balance=true}`, `{mode='read', prefer_replica=true, balance=true}`.

Если указать значение `'write'` (запись), то целью будет мастер.

Если указать `prefer_replica=true`, то предпочитаемая цель – одна из реплик; если же доступной реплики нет, то целью будет мастер.

Удобно указать `prefer_replica=true` для ресурсозатратных функций во избежание замедления работы мастера.

Если задать `balance=true`, добавится балансировка нагрузки – запросы на чтение распределяются по всем узлам набора реплик по кругу, предпочтение отдается репликам, если также задано `prefer_replica=true`.

Возвращается Исходное возвращаемое значение выполняемой функции или `nil` и ошибка. Объект ошибки содержит атрибут типа, который равен `ShardingError` или одной из стандартных ошибок Tarantool'а(`ClientError`, `OutOfMemory`, `SocketError` и т.д.).

`ShardingError` возвращается в случае ошибок шардинга: отсутствует мастер, неверный идентификатор сегмента и т.д. Такая ошибка содержит код с одним из значений из Lua-таблицы `vshard.error.code.*`, необязательный атрибут сообщения с удобным для восприятия описанием ошибки и другие атрибуты, специфичные для данного кода ошибки.

Примеры:

Для вызова функции `customer_add` из `vshard/example` выполните команду:

```
vshard.router.call(100,
                   'write',
                   'customer_add',
                   {{customer_id = 2, bucket_id = 100, name = 'name2', accounts = {}}},
```

(continues on next page)

(продолжение с предыдущей страницы)

```
                    {timeout = 5})
-- or, the same thing but with a map for the second argument
vshard.router.call(100,
                    {mode='write'},
                    'customer_add',
                    {{customer_id = 2, bucket_id = 100, name = 'name2', accounts = {}}},
                    {timeout = 5})
```

vshard.router.callro(*bucket_id*, *function_name*, *{argument_list}*, *{options}*)

Вызов функции по имени функции (function-name) на шарде, где хранится сегмент с указанным идентификатором (bucket_id) в режиме только для чтения (аналогично вызову vshard.router.call в режиме mode="read"). Для получения подробной информации о работе функции см. раздел *Обработка запросов*.

Параметры

- `bucket_id` – идентификатор сегмента
- `function_name` – выполняемая функция
- `argument_list` – массив аргументов функции
- `options` –
  - `timeout` – время ожидания запроса в секундах. Если `роутер` не может определить шард с идентификатором сегмента, операция повторяется до истечения времени ожидания.
  - другие *net.box опции*, такие как `is_async`, `buffer`, `on_push` также поддерживаются.

Возвращается

Исходное возвращаемое значение выполняемой функции или `nil` и ошибка. Объект ошибки содержит атрибут типа, который равен `ShardingError` или одной из стандартных ошибок Tarantool'а(`ClientError`, `OutOfMemory`, `SocketError` и т.д.).

`ShardingError` возвращается в случае ошибок шардинга: набор реплик недоступен, отсутствует мастер, неверный идентификатор сегмента и т.д. Такая ошибка сб.одержит код с одним из значений из Lua-таблицы `vshard.error.code.*`, необязательный атрибут сообщения с удобным для восприятия описанием ошибки и другие атрибуты, специфичные для данного кода ошибки.

vshard.router.callrw(*bucket_id*, *function_name*, *{argument_list}*, *{options}*)

Вызов функции по имени функции (function-name) на шарде, где хранится сегмент с указанным идентификатором (bucket_id) в режиме чтения и записи (аналогично вызову vshard.router.call в режиме mode="write"). Для получения подробной информации о работе функции см. раздел *Обработка запросов*.

Параметры

- `bucket_id` – идентификатор сегмента
- `function_name` – выполняемая функция
- `argument_list` – массив аргументов функции
- `options` –
  - `timeout` – время ожидания запроса в секундах. Если `роутер` не может определить шард с идентификатором сегмента, операция повторяется до истечения времени ожидания.

– другие *net.box опции*, такие как `is_async`, `buffer`, `on_push` также поддерживаются.

Возвращается

Исходное возвращаемое значение выполняемой функции или `nil` и ошибка. Объект ошибки содержит атрибут типа, который равен `ShardingError` или одной из стандартных ошибок Tarantool'a(`ClientError`, `OutOfMemory`, `SocketError` и т.д.).

`ShardingError` возвращается в случае ошибок шардинга: набор реплик недоступен, отсутствует мастер, неверный идентификатор сегмента и т.д. Такая ошибка сб.одержит код с одним из значений из Lua-таблицы `vshard.error.code.*`, необязательный атрибут сообщения с удобным для восприятия описанием ошибки и другие атрибуты, специфичные для данного кода ошибки.

`vshard.router.callre`(*bucket_id*, *function_name*, *{argument_list}*, *{options}*)
Вызов функции по имени функции (function-name) на шарде, где хранится сегмент с указанным идентификатором (bucket_id) в режиме только для чтения (аналогично вызову `vshard.router.call` в режиме чтения `mode='read'`), когда предпочтение отдается реплике, а не мастеру (аналогично вызову `vshard.router.call` с параметром `prefer_replica = true`). Для получения подробной информации о работе функции см. раздел *Обработка запросов*.

Параметры

- `bucket_id` – идентификатор сегмента
- `function_name` – выполняемая функция
- `argument_list` – массив аргументов функции
- `options` –
  - `timeout` – время ожидания запроса в секундах. Если `роутер` не может определить шард с идентификатором сегмента, операция повторяется до истечения времени ожидания.
  - другие *net.box опции*, такие как `is_async`, `buffer`, `on_push` также поддерживаются.

Возвращается

Исходное возвращаемое значение выполняемой функции или `nil` и ошибка. Объект ошибки содержит атрибут типа, который равен `ShardingError` или одной из стандартных ошибок Tarantool'a(`ClientError`, `OutOfMemory`, `SocketError` и т.д.).

`ShardingError` возвращается в случае ошибок шардинга: набор реплик недоступен, отсутствует мастер, неверный идентификатор сегмента и т.д. Такая ошибка сб.одержит код с одним из значений из Lua-таблицы `vshard.error.code.*`, необязательный атрибут сообщения с удобным для восприятия описанием ошибки и другие атрибуты, специфичные для данного кода ошибки.

`vshard.router.callbro`(*bucket_id*, *function_name*, *{argument_list}*, *{options}*)
Эквивалент *vshard.router.call()* с параметром mode = `{mode='read', balance=true}`.

`vshard.router.callbre`(*bucket_id*, *function_name*, *{argument_list}*, *{options}*)
Эквивалент *vshard.router.call()* с параметром режима mode = `{mode='read', balance=true, prefer_replica=true}`.

`vshard.router.route`(*bucket_id*)
Возврат объекта набора реплик для сегмента с указанным значением идентификатора сегмента (bucket id).

Параметры

- `bucket_id` – идентификатор сегмента

Возвращается объект набора реплик

Пример:

```
replicaset = vshard.router.route(123)
```

vshard.router.routeall()

Возврат всех доступных объектов наборов реплик.

Возвращается ассоциативный массив следующего вида: `{UUID = replicaset}`

Тип возвращаемого значения ассоциативный массив объектов набора реплик

Пример:

```
function selectall()
    local resultset = {}
    shards, err = vshard.router.routeall()
    if err ~= nil then
        error(err)
    end
    for uid, replica in pairs(shards) do
        local set = replica:callro('box.space.*space-name*:select', {{}, {limit=10}},
↪{timeout=5})
        for _, item in ipairs(set) do
            table.insert(resultset, item)
        end
    end
    table.sort(resultset, function(a, b) return a[1] < b[1] end)
    return resultset
end
```

vshard.router.bucket_id(*key*)

Объявлено устаревшим. Записывает в журнал предупреждение при использовании, так как не согласуется с числами cdata.

В частности, возвращает 3 различных значения для обычных чисел Lua, таких как 123, для unsigned long long cdata (например `123ULL`, или `ffi.cast('unsigned long long',123)`), и для signed long long cdata (например `123LL`, или `ffi.cast('long long', 123)`). И это важно.

```
vshard.router.bucket_id(123)
vshard.router.bucket_id(123LL)
vshard.router.bucket_id(123ULL)
```

Для float и double cdata (`ffi.cast('float', number)`, `ffi.cast('double', number)`) эти функции возвращают разные значения даже для тех же чисел того же типа с плавающей точкой. Это связано с тем, что функция `tostring()` для числа cdata с плавающей точкой возвращает не число, а указатель на него. Разное при каждом вызове.

`vshard.router.bucket_id_strcrc32()` имеет такое же поведение, но не записывает предупреждение. Для случаев, когда такое поведение действительно необходимо.

vshard.router.bucket_id_strcrc32(*key*)

Вычисление идентификатора сегмента с помощью простой встроенной хеш-функции.

Параметры

- `key` – хеш-ключ. Это может быть любой Lua-объект (число, таблица, строка).

Возвращается идентификатор сегмента

Тип возвращаемого значения число

Пример:

```
tarantool> vshard.router.bucket_count()
---
- 3000
...

tarantool> vshard.router.bucket_id_strcrc32("18374927634039")
---
- 2032
...

tarantool> vshard.router.bucket_id_strcrc32(18374927634039)
---
- 2032
...

tarantool> vshard.router.bucket_id_strcrc32("test")
---
- 1216
...

tarantool> vshard.router.bucket_id_strcrc32("other")
---
- 2284
...
```

Примечание: Помните, что это небезопасно. См. *bucket_id()*

vshard.router.bucket_id_mpcrc32(*key*)

Эта функция безопаснее, чем bucket_id_strc32. Она берет CRC32 из кодированного значения MessagePack. То есть bucket id целых чисел не зависит от их типа Lua. В случае строкового ключа, он не кодирует его в MessagePack, а берет хэш прямо из строки.

Параметры

- `key` – хеш-ключ. Это может быть любой Lua-объект (число, таблица, строка).

Возвращается идентификатор сегмента

Тип возвращаемого значения число

Однако он все равно может возвращать разные значения для не одинакового типа с плавающей точкой. То есть, `ffi.cast('float', number)` может быть отражено в bucket id, не равном `ffi.cast('double', number)`. Это не может быть исправлено, так как значение с плавающей точкой, даже будучи приведенным к double, может иметь мусор в своей дробной части.

Ключи с плавающей точкой обычно не должны использоваться для вычисления идентификатора сегмента.

Будьте очень осторожны, если вы храните типы с плавающей точкой в спейсе. Когда данные возвращаются из спейса, они приводятся к Lua числам. А если это значение имело пустую дробную часть, то оно будет обработано как целое число функцией `bucket_id_mpcrc32()`. Поэтому в таких случаях необходимо выполнять явное приведение. Приведем пример проблемы:

```
tarantool> s = box.schema.create_space('test', {format = {{'id', 'double'}}}); _ = s:create_
↪index('pk')
---
```

(continues on next page)

(продолжение с предыдущей страницы)

```
...

tarantool> inserted = ffi.cast('double', 1)
---
...

-- Value is stored as double
tarantool> s:replace({inserted})
---
- [1]
...

-- But when returned to Lua, stored as Lua number, not cdata.
tarantool> returned = s:get({inserted}).id
---
...

tarantool> type(returned), returned
---
- number
- 1
...

tarantool> vshard.router.bucket_id_mpcrc32(inserted)
---
- 1411
...
tarantool> vshard.router.bucket_id_mpcrc32(returned)
---
- 1614
...
```

vshard.router.bucket_count()
Возврат общего количества сегментов, указанных в *vshard.router.cfg()*‘.

Возвращается общее количество сегментов

Тип возвращаемого значения число

```
tarantool> vshard.router.bucket_count()
---
- 10000
...
```

vshard.router.sync(*timeout*)
Ожидание синхронизации набора данных на репликах.

Параметры

- `timeout` – время ожидания в секундах

возвращает `true` (правда), если выполнена синхронизация набора данных; или же `nil` и ошибка `err` с объяснением причины невозможности синхронизации набора данных.

vshard.router.discovery_wakeup()
Принудительный запуск файбера обнаружения сегментов.

`vshard.router.discovery_set(`*mode*`)`

Запуск/выключение фонового файбера, используемого роутером для обнаружения сегментов.

Параметры

- `mode` – режим работы файбера обнаружения. Существует три режима: `on`, `off` и `once`

В режиме `on` (по умолчанию) файбер обнаружения работает в течение всего жизненного цикла роутера. Даже после того, как все сегменты были найдены, он продолжает проверять хранилища и загружать сегменты с некоторой большой периодичностью (DISCOVERY_IDLE_INTERVAL). Это полезно, если топология сегментов часто меняется, а их число небольшое. Роутер будет поддерживать свою таблицу маршрутов в актуальном состоянии даже тогда, когда никакие запросы не обрабатываются.

В режиме `off` обнаружение сегментов не производится.

В режиме `once` файбер обнаружения найдет локации всех сегментов, а затем ликвидируется. Это полезно для большого числа сегментов и для кластеров, в которых редко происходит балансировка.

Этот метод подойдет для включения/выключения обнаружения после того, как роутер уже запущен, но по умолчанию обнаружение включено. Возможно, вы захотите никогда не включать его даже на короткое время – тогда задайте значение опции `discovery_mode` при *конфигурации*. Она принимает те же значения, что и `vshard.router.discovery_set(`*mode*`)`.

Вы можете решить, что лучше отключить обнаружение или осуществить его в режиме `once`, если у вас много роутеров или очень много сегментов (сотни тысяч и более), и вы видите, что процесс обнаружения потребляет заметное количество ресурса CPU на роутерах и хранилищах. В этом случае, возможно, было бы разумно отключить обнаружение, когда в кластере нет балансировки. И включать его для новых роутеров, а также для всех роутеров, когда начинается балансировка.

`vshard.router.info()`

Возврат информации по каждому экземпляру.

Возвращается

Параметры набора реплик:

- UUID набора реплик
- параметры мастер-экземпляра
- параметры реплики

Параметры экземпляра:

- `uri` – URI экземпляра
- `uuid` – UUID экземпляра
- `status` – статус экземпляра: `available` (доступный), `unreachable` (недоступный), `missing` (отсутствующий)
- `network_timeout` – время ожидания запроса. Данное значение обновляется автоматически на каждом 10 выполненном запросе и на каждом 2 невыполненном запросе.

Параметры сегмента:

- `available_ro` – количество сегментов, известных роутеру и доступных для запросов чтения
- `available_rw` – количество сегментов, известных роутеру и доступных для запросов чтения и записи

- `unavailable` – количество сегментов, известных роутеру, но недоступных для любых запросов
- `unreachable`– количество сегментов, для которых роутер не знает соответствующие наборы реплик

Пример:

```
tarantool> vshard.router.info()
---
- replicasets:
    ac522f65-aa94-4134-9f64-51ee384f1a54:
      replica: &0
        network_timeout: 0.5
        status: available
        uri: storage@127.0.0.1:3303
        uuid: 1e02ae8a-afc0-4e91-ba34-843a356b8ed7
      uuid: ac522f65-aa94-4134-9f64-51ee384f1a54
      master: *0
    cbf06940-0790-498b-948d-042b62cf3d29:
      replica: &1
        network_timeout: 0.5
        status: available
        uri: storage@127.0.0.1:3301
        uuid: 8a274925-a26d-47fc-9e1b-af88ce939412
      uuid: cbf06940-0790-498b-948d-042b62cf3d29
      master: *1
  bucket:
    unreachable: 0
    available_ro: 0
    unknown: 0
    available_rw: 3000
  status: 0
  alerts: []
...
```

`vshard.router.buckets_info()`

Возврат информации по каждому сегменту. Поскольку массив сегментов может быть огромен, можно указать только необходимый ряд сегментов.

Параметры

- `offset` – начальное значение выборки сегментов
- `limit` – максимальное количество показываемых сегментов

Возвращается ассоциативный массив следующего вида: `{bucket_id = 'unknown'/ replicaset_uuid}`

```
tarantool> vshard.router.buckets_info()
---
- - uuid: aaaaaaaa-0000-4000-a000-000000000000
    status: available_rw
  - uuid: aaaaaaaa-0000-4000-a000-000000000000
    status: available_rw
  - uuid: aaaaaaaa-0000-4000-a000-000000000000
    status: available_rw
  - uuid: bbbbbbbb-0000-4000-a000-000000000000
    status: available_rw
  - uuid: bbbbbbbb-0000-4000-a000-000000000000
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    status: available_rw
  - uuid: bbbbbbbb-0000-4000-a000-000000000000
    status: available_rw
  - uuid: bbbbbbbb-0000-4000-a000-000000000000
    status: available_rw
...
```

object replicaset_object

replicaset_object:call(*function_name*, *{argument_list}*, *{options}*)

Вызов функции с указанными аргументами на ближайшем доступном мастере (расстояние определяется с помощью матрицы `replica.zone` и `cfg.weights`).

Примечание: Метод `replicaset_object:call` аналогичен `replicaset_object:callrw`.

Параметры

- `function_name` – выполняемая функция
- `argument_list` – массив аргументов функции
- `options` –
  - `timeout` – время ожидания запроса в секундах. Если `router` не может определить шард с идентификатором сегмента, операция повторяется до истечения времени ожидания.
  - другие *net.box опции*, такие как `is_async`, `buffer`, `on_push` также поддерживаются.

возвращает

- результат вызываемой функции при успехе
- nil, err иначе

replicaset_object:callrw(*function_name*, *{argument_list}*, *{options}*)

Вызов функции с указанными аргументами на ближайшем доступном мастере (расстояние определяется с помощью матрицы `replica.zone` и `cfg.weights`).

Примечание: Метод `replicaset_object:callrw` аналогичен `replicaset_object:call`.

Параметры

- `function_name` – выполняемая функция
- `argument_list` – массив аргументов функции
- `options` –
  - `timeout` – время ожидания запроса в секундах. Если `router` не может определить шард с идентификатором сегмента, операция повторяется до истечения времени ожидания.
  - другие *net.box опции*, такие как `is_async`, `buffer`, `on_push` также поддерживаются.

возвращает

- результат вызываемой функции при успехе
- nil, err иначе

```
tarantool> local bucket = 1; return vshard.router.callrw(
         >     bucket,
         >     'box.space.actors:insert',
         >     {{
         >         1, bucket, 'Renata Litvinova',
         >         {theatre="Moscow Art Theatre"}
         >     }},
         >     {timeout=5}
         > )
```

`replicaset_object:callro(`*function_name*, *{argument_list}*, *{options}*`)`

Вызов функции с указанными аргументами на ближайшей доступной реплике (расстояние определяется с помощью матрицы `replica.zone` и `cfg.weights`). С помощью `replicaset_object:callro()` рекомендуется вызывать исключительно функции, доступные только для чтения. поскольку такие функции можно выполнять не только на мастере, но и на репликах.

Параметры

- `function_name` – выполняемая функция
- `argument_list` – массив аргументов функции
- `options` –
  - `timeout` – время ожидания запроса в секундах. Если `роутер` не может определить шард с идентификатором сегмента, операция повторяется до истечения времени ожидания.
  - другие *net.box опции*, такие как `is_async`, `buffer`, `on_push` также поддерживаются.

возвращает

- результат вызываемой функции при успехе
- nil, err иначе

`replicaset:callre(`*function_name*, *{argument_list}*, *{options}*`)`

Вызов функции с указанными аргументами на ближайшей доступной реплике (расстояние определяется с помощью матрицы `replica.zone` и `cfg.weights`), предпочтение отдается реплике, а не мастеру (аналогично вызову vshard.router.call с параметром prefer_replica = true). С помощью `replicaset_object:callre()` рекомендуется вызывать исключительно функции, доступные только для чтения, поскольку такие функции можно выполнять не только на мастере, но и на репликах.

Параметры

- `function_name` – выполняемая функция
- `argument_list` – массив аргументов функции
- `options` –
  - `timeout` – время ожидания запроса в секундах. Если `роутер` не может определить шард с идентификатором сегмента, операция повторяется до истечения времени ожидания.

– другие *net.box опции*, такие как `is_async`, `buffer`, `on_push` также поддерживаются.

возвращает

- результат вызываемой функции при успехе
- nil, err иначе

Внутренний API роутера

`vshard.router.bucket_discovery(`*bucket_id*`)`
Поиск сегмента по всему кластеру. Если сегмент не обнаружен, скорее всего, он не существует. Также сегмент также может быть перемещен во время балансировки и в данный момент находится в статусе получения RECEIVING.

Параметры

- `bucket_id` – идентификатор сегмента

Общедоступный API хранилища

`vshard.storage.cfg(`*cfg*, *name*`)`
Конфигурация базы данных и начало шардинга на указанном экземпляре хранилища.

Параметры

- `cfg` – конфигурация хранилища
- `instance_uuid` – UUID экземпляра

`vshard.storage.info()`
Возврат информации по экземпляру хранилища в следующем формате:

```
tarantool> vshard.storage.info()
---
- buckets:
    2995:
      status: active
      id: 2995
    2997:
      status: active
      id: 2997
    2999:
      status: active
      id: 2999
  replicasets:
    2dd0a343-624e-4d3a-861d-f45efc571cd3:
      uuid: 2dd0a343-624e-4d3a-861d-f45efc571cd3
      master:
        state: active
        uri: storage:storage@127.0.0.1:3301
        uuid: 2ec29309-17b6-43df-ab07-b528e1243a79
    c7ad642f-2cd8-4a8c-bb4e-4999ac70bba1:
      uuid: c7ad642f-2cd8-4a8c-bb4e-4999ac70bba1
      master:
        state: active
        uri: storage:storage@127.0.0.1:3303
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        uuid: 810d85ef-4ce4-4066-9896-3c352fec9e64
...
```

vshard.storage.call(*bucket_id*, *mode*, *function_name*, *{argument_list}*)
Вызов указанной функции на текущем экземпляре хранилища.

Параметры

- `bucket_id` – идентификатор сегмента
- `mode` – тип функции: „read“ или „write“ (чтение или запись)
- `function_name` – выполняемая функция
- `argument_list` – массив аргументов функции

Возвращается

Исходное возвращаемое значение выполняемой функции или `nil` и ошибка.

vshard.storage.sync(*timeout*)
Ожидание синхронизации набора данных на репликах.

Параметры

- `timeout` – время ожидания в секундах

возвращает `true` (правда), если выполнена синхронизация набора данных; или же `nil` и ошибка `err` с объяснением причины невозможности синхронизации набора данных.

vshard.storage.bucket_pin(*bucket_id*)
Закрепление сегмента в наборе реплик. Закрепленный сегмент нельзя перемещать, даже если это нарушает баланс в кластере.

Параметры

- `bucket_id` – идентификатор сегмента

возвращает `true` (правда), если выполнено закрепление сегмента; или же `nil` и ошибка `err` с объяснением причины невозможности закрепления сегмента

vshard.storage.bucket_unpin(*bucket_id*)
Возврат закрепленного сегмента в активное состояние.

Параметры

- `bucket_id` – идентификатор сегмента

возвращает `true` (правда), если выполнено открепление сегмента; или же `nil` и ошибка `err` с объяснением причины невозможности открепления сегмента

vshard.storage.bucket_ref(*bucket_id*, *mode*)
Создание *ссылки* типа RO или RW.

Параметры

- `bucket_id` – идентификатор сегмента
- `mode` – „read“ или „write“ (чтение или запись)

возвращает `true` (правда), если выполнено создание ссылки; или же `nil` и ошибка `err` с объяснением причины невозможности создания ссылки

vshard.storage.bucket_refro()
Псевдоним для *vshard.storage.bucket_ref* в режиме только чтения.

vshard.storage.bucket_refrw()
Псевдоним для *vshard.storage.bucket_ref* в режиме чтения и записи.

vshard.storage.bucket_unref(*bucket_id*, *mode*)
Удаление *ссылки* RO/RW.

Параметры

- `bucket_id` – идентификатор сегмента
- `mode` – „read“ или „write“ (чтение или запись)

возвращает `true` (правда), если выполнено удаление ссылки; или же `nil` и ошибка `err` с объяснением причины невозможности удаления ссылки

vshard.storage.bucket_unrefro()
Псевдоним для *vshard.storage.bucket_unref* в режиме только чтения.

vshard.storage.bucket_unrefrw()
Псевдоним для *vshard.storage.bucket_unref* в режиме чтения и записи.

vshard.storage.find_garbage_bucket(*bucket_index*, *control*)
Поиск сегмента, который хранит данные в спейсе, но не указан в спейсе `_bucket`, или находится в статусе мусора (GARBAGE).

Параметры

- `bucket_index` – индекс спейса с частью идентификатора спейса
- `control` – контроллер сборщика мусора. Если увеличивается масштаб создания сегментов, поиск следует прервать.

возвращает идентификатор сегмента в статусе мусора, если таковой обнаружен; в противном случае, nil

vshard.storage.buckets_info()
Возврат информации по каждому сегменту, расположенному в хранилище. Например:

```
tarantool> vshard.storage.buckets_info(1)
---
- 1:
    status: active
    ref_rw: 1
    ref_ro: 1
    ro_lock: true
    rw_lock: true
    id: 1
```

vshard.storage.buckets_count()
Возврат количества сегментов, расположенных в хранилище.

vshard.storage.recovery_wakeup()
Немедленный запуск файбера восстановления, если такой есть.

vshard.storage.rebalancing_is_in_progress()
Возврат флага, указывающего на ход процесса балансировки. Результатом будет true (правда), если в данный момент узел применяет маршруты, полученные от узла балансировки в специальном файбере.

vshard.storage.is_locked()
: Возврат флага, указывающего на недоступность хранилища для балансировщика.

vshard.storage.rebalancer_disable()
: Отключение балансировки. Отключенный балансировщик находится в режиме ожидания до повторного запуска с помощью vshard.storage.rebalancer_enable().

vshard.storage.rebalancer_enable()
: Запуск балансировки.

vshard.storage.sharded_spaces()
: Отображение спейсов, которые доступны балансировщику и файберам сборщика мусора.

```
tarantool> vshard.storage.sharded_spaces()
---
- 513:
    engine: memtx
    before_replace: 'function: 0x010e50e738'
    field_count: 0
    id: 513
    on_replace: 'function: 0x010e50e700'
    temporary: false
    index:
      0: &0
        unique: true
        parts:
        - type: number
          fieldno: 1
          is_nullable: false
        id: 0
        type: TREE
        name: primary
        space_id: 513
      1: &1
        unique: false
        parts:
        - type: number
          fieldno: 2
          is_nullable: false
        id: 1
        type: TREE
        name: bucket_id
        space_id: 513
      primary: *0
      bucket_id: *1
    is_local: false
    enabled: true
    name: actors
    ck_constraint: []
...
```

Внутренний API хранилища

vshard.storage.bucket_recv(*bucket_id*, *from*, *data*)
: Получение сегмента по идентификатору сегмента (bucket id) из удаленного набора реплик.

Параметры

- `bucket_id` – идентификатор сегмента
- `from` – UUID исходного набора реплик
- `data` – данные, которые хранятся логически в сегменте, определенном по идентификатору сегмента (bucket_id), в том же формате, что и возвращаемое значение метода `bucket_collect() <storage_api-bucket_collect>`

`vshard.storage.bucket_stat(`*bucket_id*`)`

Возврат информации об идентификаторе сегмента (bucket id):

```
tarantool> vshard.storage.bucket_stat(1)
---
- 0
- status: active
  id: 1
...
```

Параметры

- `bucket_id` – идентификатор сегмента

`vshard.storage.bucket_delete_garbage(`*bucket_id*`)`

Принудительная сборка мусора для сегмента, найденного по идентификатору (bucket_id), если сегмент был перемещен в другой набор реплик.

Параметры

- `bucket_id` – идентификатор сегмента

`vshard.storage.bucket_collect(`*bucket_id*`)`

Сбор всех данных, которые хранятся логически в сегменте, найденном по идентификатору (bucket_id):

```
tarantool> vshard.storage.bucket_collect(1)
---
- 0
- - - 514
    - - [10, 1, 1, 100, 'Account 10']
      - [11, 1, 1, 100, 'Account 11']
      - [12, 1, 1, 100, 'Account 12']
      - [50, 5, 1, 100, 'Account 50']
      - [51, 5, 1, 100, 'Account 51']
      - [52, 5, 1, 100, 'Account 52']
  - - 513
    - - [1, 1, 'Customer 1']
      - [5, 1, 'Customer 5']
...
```

Параметры

- `bucket_id` – идентификатор сегмента

`vshard.storage.bucket_force_create(`*first_bucket_id*, *count*`)`

Принудительное создание сегментов (одного или нескольких) в текущем наборе реплик. Используется только для ручного аварийного восстановления или для начальной настройки.

Параметры

- `first_bucket_id` – идентификатор первого сегмента в диапазоне

- `count` – количество вставляемых сегментов (по умолчанию, 1)

`vshard.storage.bucket_force_drop(`*bucket_id*`)`
Удаление сегмента вручную для тестирования или в аварийной ситуации.

Параметры

- `bucket_id` – идентификатор сегмента

`vshard.storage.bucket_send(`*bucket_id*, *to*`)`
Отправка указанного сегмента из текущего набора реплик в удаленный набор реплик.

Параметры

- `bucket_id` – идентификатор сегмента
- `to` – UUID удаленного набора реплик

`vshard.storage.rebalancer_request_state()`
Проверка всех сегментов хост-хранилища в статусе отправки SENT или активном статусе ACTIVE, возврат количества активных сегментов.

возвращает количество сегментов в активном статусе, если таковые обнаружены; в противном случае, nil

`vshard.storage.buckets_discovery()`
Сбор массива идентификаторов активных сегментов для обнаружения.

5.3.5 Luatest

More about Luatest API see *below*.

Overview

Tool for testing tarantool applications. (Build Status).

Highlights:

- executable to run tests in directory or specific files,
- before/after suite hooks,
- before/after test group hooks,
- *output capturing*,
- *helpers* for testing tarantool applications,
- *luacov integration*.

Requirements

- Tarantool (it requires tarantool-specific `fio` module and `ffi` from LuaJIT).

Installation

```
tarantoolctl rocks install luatest

.rocks/bin/luatest --help # list available options
```

Usage

Define tests.

```
-- test/feature_test.lua
local t = require('luatest')
local g = t.group('feature')
-- Default name is inferred from caller filename when possible.
-- For `test/a/b/c_d_test.lua` it will be `a.b.c_d`.
-- So `local g = t.group()` works the same way.

-- Tests. All properties with name staring with `test` are treated as test cases.
g.test_example_1 = function() ... end
g.test_example_n = function() ... end

-- Define suite hooks
t.before_suite(function() ... end)
t.before_suite(function() ... end)

-- Hooks to run once for tests group
g.before_all(function() ... end)
g.after_all(function() ... end)

-- Hooks to run for each test in group
g.before_each(function() ... end)
g.after_each(function() ... end)

-- test/other_test.lua
local t = require('luatest')
local g = t.group('other')
-- ...
g.test_example_2 = function() ... end
g.test_example_m = function() ... end
```

Run them.

```
luatest --help                        # list available options
luatest                               # run all in ./test directory
luatest test/feature_test.lua         # run test by file
luatest test/integration              # run all within directory
luatest feature other.test_example_2  # run by group or test name
```

Luatest automatically requires `test/helper.lua` file if it's present. You can configure luatest or run any bootstrap code there.

See the getting-started example in cartridge-cli repo.

Tests order

Use the `--shuffle` option to tell luatest how to order the tests. The available ordering schemes are `group`, `all` and `none`.

`group` shuffles tests within the groups.

`all` randomizes execution order across all available tests. Be careful: `before_all/after_all` hooks run always when test group is changed, so it may run multiple time.

`none` is the default, which executes examples within the group in the order they are defined (eventually they are ordered by functions line numbers).

With `group` and `all` you can also specify a `seed` to reproduce specific order.

```
--shuffle none
--shuffle group
--shuffle all --seed 123
--shuffle all:123 # same as above
```

To change default order use:

```
-- test/helper.lua
local t = require('luatest')
t.configure({shuffle = 'group'})
```

List of luatest functions

Assertions	
assert (value[, message])	Check that value is truth
assert_almost_equals (actual, expected, margin[, message])	Check that two floats are
assert_covers (actual, expected[, message])	Checks that actual map
assert_equals (actual, expected[, message[, deep_analysis]])	Check that two values ar
assert_error (fn, ...)	Check that calling fn rais
assert_error_msg_contains (expected_partial, fn, ...)	
assert_error_msg_content_equals (expected, fn, ...)	Strips location info from
assert_error_msg_equals (expected, fn, ...)	Checks full error: locatio
assert_error_msg_matches (pattern, fn, ...)	
assert_eval_to_false (value[, message])	Alias for assert_not.
assert_eval_to_true (value[, message])	Alias for assert.
assert_items_include (actual, expected[, message])	Checks that actual inclu
assert_is (actual, expected[, message])	Check that values are th
assert_is_not (actual, expected[, message])	Check that values are no
assert_items_equals (actual, expected[, message])	Checks equality of tables
assert_nan (value[, message])	
assert_not (value[, message])	Check that value is falsy.
assert_not_almost_equals (actual, expected, margin[, message])	Check that two floats are
assert_not_covers (actual, expected[, message])	Checks that map does no
assert_not_equals (actual, expected[, message])	Check that two values ar
assert_not_nan (value[, message])	
assert_not_str_contains (actual, expected[, is_pattern[, message]])	Case-sensitive strings co
assert_not_str_icontains (value, expected[, message])	Case-insensitive strings c
assert_str_contains (value, expected[, is_pattern[, message]])	Case-sensitive strings co
assert_str_icontains (value, expected[, message])	Case-insensitive strings c
assert_str_matches (value, pattern[, start=1[, final=value:len() [, message]]])	Verify a full match for th
assert_type (value, expected_type[, message])	Check value's type.
Flow control	
fail (message)	Stops a test due to a fail
fail_if (condition, message)	Stops a test due to a fail
skip (message)	Skip a running test.
skip_if (condition, message)	Skip a running test if co
success ()	Stops a test with a succe
success_if (condition)	Stops a test with a succe
Suite and groups	

Про

Таблица 6 – продолжение с предыдущей страницы

after_suite (fn)	Add after suite hook.
before_suite (fn)	Add before suite hook.
group (name)	Create group of tests.

Capturing output

By default runner captures all stdout/stderr output and shows it only for failed tests. Capturing can be disabled with `-c` flag.

Test helpers

There are helpers to run tarantool applications and perform basic interaction with it. If application follows configuration conventions it is possible to use options to configure server instance and helpers at the same time. For example `http_port` is used to perform http request in tests and passed in `TARANTOOL_HTTP_PORT` to server process.

```
local server = luatest.Server:new({
    command = '/path/to/executable.lua',
    -- arguments for process
    args = {'--no-bugs', '--fast'},
    -- additional envars to pass to process
    env = {SOME_FIELD = 'value'},
    -- passed as TARANTOOL_WORKDIR
    workdir = '/path/to/test/workdir',
    -- passed as TARANTOOL_HTTP_PORT, used in http_request
    http_port = 8080,
    -- passed as TARANTOOL_LISTEN, used in connect_net_box
    net_box_port = 3030,
    -- passed to net_box.connect in connect_net_box
    net_box_credentials = {user = 'username', password = 'secret'},
})
server:start()
-- Wait until server is ready to accept connections.
-- This may vary from app to app: for one server:connect_net_box() is enough,
-- for another more complex checks are required.
luatest.helpers.retrying({}, function() server:http_request('get', '/ping') end)

-- http requests
server:http_request('get', '/path')
server:http_request('post', '/path', {body = 'text'})
server:http_request('post', '/path', {json = {field = value}, http = {
    -- http client options
    headers = {Authorization = 'Basic ' .. credentials},
    timeout = 1,
}})

-- This method throws error when response status is outside of then range 200..299.
-- To change this behaviour, path `raise = false`:
t.assert_equals(server:http_request('get', '/not_found', {raise = false}).status, 404)
t.assert_error(function() server:http_request('get', '/not_found') end)

-- using net_box
server:connect_net_box()
server.net_box:eval('return do_something(...)', {arg1, arg2})
```

(continues on next page)

(продолжение с предыдущей страницы)

```
server:stop()
```

`luatest.Process:start(path, args, env)` provides low-level interface to run any other application.

There are several small helpers for common actions:

```
luatest.helpers.uuid('ab', 2, 1) == 'abababab-0002-0000-0000-000000000001'

luatest.helpers.retrying({timeout = 1, delay = 0.1}, failing_function, arg1, arg2)
-- wait until server is up
luatest.helpers.retrying({}, function() server:http_request('get', '/status') end)
```

luacov integration

- Install luacov with `tarantoolctl rocks install luacov`
- Configure it with `.luacov` file
- Clean old reports `rm -f luacov.*.out*`
- Run luatest with `--coverage` option
- Generate report with `.rocks/bin/luacov .`
- Show summary with `grep -A999 '^Summary' luacov.report.out`

When running integration tests with coverage collector enabled, luatest automatically starts new tarantool instances with luacov enabled. So coverage is collected from all the instances. However this has some limitations:

- It works only for instances started with `Server` helper.
- Process command should be executable lua file or tarantool with script argument.
- Instance must be stopped with `server:stop()`, because this is the point where stats are saved.
- Don't save stats concurrently to prevent corruption.

Development

- Check out the repo.
- Prepare makefile with `cmake .`.
- Install dependencies with `make bootstrap`.
- Run it with `make lint` before committing changes.
- Run tests with `bin/luatest`.

Contributing

Bug reports and pull requests are welcome on at https://github.com/tarantool/luatest.

License

MIT

API

Module *luatest*

Functions

Methods

after_suite (fn)

Add after suite hook.

Parameters:

- *fn:* (**func**)

before_suite (fn)

Add before suite hook.

Parameters:

- *fn:* (**func**)

configure ([options={}])

Update default options. See *luatest.runner:run* for the list of available options.

Parameters:

- *options:* (**tab**) list of options to update (default $(def))

Returns:

options after update

group ([name])

Create group of tests.

Parameters:

- *name:* (string) (optional)

Returns:

Group object

See also:

- luatest.group

Fields

Server

Class to manage tarantool instances.

See also:

- luatest.server

helpers

Helpers.

See also:

- luatest.helpers

assertions Functions

EPS

EPS is meant to help with Lua's floating point math in simple corner cases like almost_equals(1.1-0.1, 1), which may not work as-is (e.g. on numbers with rational binary representation) if the user doesn't provide some explicit error margin.

The default margin used by almost_equals() in such cases is EPS; and since Lua may be compiled with different numeric precisions (single vs. double), we try to select a useful default for it dynamically. Note: If the initial value is not acceptable, it can be changed by the user to better suit specific needs.

See also: https://en.wikipedia.org/wiki/Machine_epsilon

almost_equals (actual, expected, margin)

Parameters:

- *actual:* (**number**)
- *expected:* (**number**)
- *margin:* (**number**)

assert (value[, message[, ...]])

Check that value is truthy.

Parameters:

- *value:*
- *message:* (string) (optional)
- *...:* (optional)

Returns:

input values

assert_almost_equals (actual, expected, margin[, message])

Check that two floats are close by margin.

Parameters:

- *actual:* (**number**)
- *expected:* (**number**)
- *margin:* (**number**)
- *message:* (string) (optional)

assert_covers (actual, expected[, message])

Checks that actual map includes expected one.

Parameters:

- *actual:* (**tab**)
- *expected:* (**tab**)
- *message:* (string) (optional)

assert_equals (actual, expected[, message[, deep_analysis]])

Check that two values are equal. Tables are compared by value.

Parameters:

- *actual:*
- *expected:*
- *message:* (string) (optional)
- *deep_analysis:* (**bool**) print diff. (optional)

assert_error (fn, ...)

Check that calling fn raises an error.

Parameters:

- *fn:* (**func**)
- *...:* arguments for function

assert_error_msg_contains (expected_partial, fn, ...)

Parameters:

- *expected_partial:* (string)
- *fn:* (**func**)
- *...:* arguments for function

assert_error_msg_content_equals (expected, fn, ...)

Strips location info from message text.

Parameters:

- *expected:* (string)
- *fn:* (**func**)
- *...:* arguments for function

assert_error_msg_equals (expected, fn, ...)

Checks full error: location and text.

Parameters:

- *expected:* (string)
- *fn:* (**func**)
- *...:* arguments for function

assert_error_msg_matches (pattern, fn, ...)

Parameters:

- *pattern:* (string)
- *fn:* (**func**)
- *...:* arguments for function

assert_eval_to_false (value[, message])

Alias for *assert_not*.

Parameters:

- *value:*
- *message:* (string) (optional)

assert_eval_to_true (value[, message])

Alias for *assert*.

Parameters:

- *value:*
- *message:* (string) (optional)

assert_is (actual, expected[, message])

Check that values are the same.

Parameters:

- *actual:*
- *expected:*
- *message:* (string) (optional)

assert_is_not (actual, expected[, message])

Check that values are not the same.

Parameters:

- *actual:*
- *expected:*
- *message:* (string) (optional)

assert_items_equals (actual, expected[, message])

Checks equality of tables regardless of the order of elements.

Parameters:

- *actual:*
- *expected:*
- *message:* (string) (optional)

assert_items_include (actual, expected[, message])

Checks that actual includes all items of expected.

Parameters:

- *actual:*
- *expected:*
- *message:* (string) (optional)

assert_nan (value[, message])

Parameters:

- *value:*
- *message:* (string) (optional)

assert_not (value[, message])

Check that value is falsy.

Parameters:

- *value:*
- *message:* (string) (optional)

assert_not_almost_equals (actual, expected, margin[, message])

Check that two floats are not close by margin.

Parameters:

- *actual:* (**number**)
- *expected:* (**number**)
- *margin:* (**number**)
- *message:* (string) (optional)

assert_not_covers (actual, expected[, message])

Checks that map does not contain the other one.

Parameters:

- *actual:* (**tab**)
- *expected:* (**tab**)
- *message:* (string) (optional)

assert_not_equals (actual, expected[, message])

Check that two values are not equal. Tables are compared by value.

Parameters:

- *actual:*
- *expected:*
- *message:* (string) (optional)

assert_not_nan (value[, message])

Parameters:

- *value:*
- *message:* (string) (optional)

assert_not_str_contains (actual, expected[, is_pattern[, message]])

Case-sensitive strings comparison.

Parameters:

- *actual:* (string)
- *expected:* (string)
- *is_pattern:* (**bool**) (optional)
- *message:* (string) (optional)

assert_not_str_icontains (value, expected[, message])

Case-insensitive strings comparison.

Parameters:

- *value:* (string)
- *expected:* (string)
- *message:* (string) (optional)

assert_str_contains (value, expected[, is_pattern[, message]])

Case-sensitive strings comparison.

Parameters:

- *value:* (string)
- *expected:* (string)
- *is_pattern:* (**bool**) (optional)
- *message:* (string) (optional)

assert_str_icontains (value, expected[, message])

Case-insensitive strings comparison.

Parameters:

- *value:* (string)
- *expected:* (string)
- *message:* (string) (optional)

assert_str_matches (value, pattern[, start=1[, final=value:len()[, message]]])

Verify a full match for the string.

Parameters:

- *value:* (string)

- *pattern:* (string)
- *start:* (**int**) (default $(def))
- *final:* (**int**) (default $(def))
- *message:* (string) (optional)

assert_type (value, expected_type[, message[, level]])

Check value's type.

Parameters:

- *value:* (string)
- *expected_type:* (string)
- *message:* (string) (optional)
- *level:* (**int**) (optional)

fail (message)

Stops a test due to a failure.

Parameters:

- *message:* (string)

fail_if (condition, message)

Stops a test due to a failure if condition is met.

Parameters:

- *condition:*
- *message:* (string)

skip (message)

Skip a running test.

Parameters:

- *message:* (string)

skip_if (condition, message)

Skip a running test if condition is met.

Parameters:

- *condition:*
- *message:* (string)

success ()

Stops a test with a success.

success_if (condition)

Stops a test with a success if condition is met.

Parameters:

- *condition:*

Module *luatest.helpers*

Collection of test helpers.

Functions

Methods

retrying (config, fn, ...)

Keep calling fn until it returns without error. Throws last error if config.timeout is elapsed. Default options are taken from helpers.RETRYING_TIMEOUT and helpers.RETRYING_DELAY.

```
helpers.retrying({}, fn, arg1, arg2)
helpers.retrying({timeout = 2, delay = 0.5}, fn, arg1, arg2)
```

Parameters:

- *config:*
  - *timeout:* (**number**)
  - *delay:* (**number**)
- *fn:* (**func**)
- *...:* args

uuid (a, ...)

Generates uuids from its 5 parts. Strings are repeated and numbers are padded to match required part length. If number of arguments is less than 5 then first and last arguments are used for corresponding parts, missing parts are set to 0.

```
'aaaaaaaa-0000-0000-0000-000000000000' == uuid('a')
'abababab-0000-0000-0000-000000000001' == uuid('ab', 1)
'00000001-0002-0000-0000-000000000003' == uuid(1, 2, 3)
'11111111-2222-0000-0000-333333333333' == uuid('1', '2', '3')
'12121212-3434-5656-7878-909090909090' == uuid('12', '34', '56', '78', '90')
```

Parameters:

- *a:* first part
- *. . . :* parts

Class *luatest.group*

Tests group.

To add new example add function at key starting with `test` .

Group hooks run always when test group is changed. So it may run multiple times when `--shuffle` option is used.

Instance methods

Group.mt.after_all (fn)

Add callback to run once after all tests in the group.

Parameters:

- *fn:*

Group.mt.after_each (fn)

Add callback to run after each test in the group.

Parameters:

- *fn:*

Group.mt.before_all (fn)

Add callback to run once before all tests in the group.

Parameters:

- *fn:*

Group.mt.before_each (fn)

Add callback to run before each test in the group.

Parameters:

- *fn:*

Group.mt:initialize ([name])

Parameters:

- *name:* (string) Default name is inferred from caller filename when possible.For `test/a/b/c_d_test.lua` it will be `a.b.c_d` . (optional)

Returns:

Group instance

Class *luatest.http_response*

Class to provide helper methods for HTTP responses

Instance getter methods

HTTPResponse.getters

For backward compatibility this methods should be accessed as object's fields (eg., `response.json.id`). They are not assigned to object's fields on initialization to be evaluated lazily and to be able to throw errors.

HTTPResponse.getters:json ()

Parse json from body.

Usage:

```
response.json.id
```

HTTPResponse.mt:is_successful ()

Check that status code is 2xx.

Class *luatest.runner*

Class to run test suite.

Functions

Methods

Runner.is_test_name (s)

Check that string matches the name of a test method. Default rule is that is starts with „test“

Parameters:

- *s:*

Runner.run ([args=_G.args[, options]])

Main entrypoint to run test suite.

Parameters:

- *args:* (**tab**) List of CLI arguments (default $(def))
- *options:*
  - *verbosity:* (**int**) (optional)
  - *fail_fast:* (**bool**) (default $(def))
  - *output_file_name:* (string) Filename for JUnit report (optional)
  - *exe_repeat:* (**int**) Times to repeat each test (optional)
  - *tests_pattern:* (**tab**) Patterns to filter tests (optional)
  - *tests_names:* (**tab**) List of test names or groups to run (optional)
  - *paths:* (**tab**) List of directories to load tests from. (default $(def))
  - *load_tests:* (**func**) Function to load tests. Called once for every item in `paths` . (optional)
  - *shuffle:* (string) Shuffle method (none, all, group) (default $(def))
  - *seed:* (**int**) Random seed for shuffle (optional)
  - *output:* (string) Output formatter (text, tap, junit, nil) (default $(def))

Runner.split_test_method_name (someName)

Split `some.group.name.method` into `some.group.name` and `method` . Returns `nil, input` if input value does not have a dot.

Parameters:

- *someName:*

Runner:expand_group (group)

Exrtact all test methods from group.

Parameters:

- *group:*

Class *luatest.server*

Class to run tarantool instance.

Functions

Methods

Server:build_env ()

Generates environment to run process with. The result is merged into os.environ().

Returns:

map

Server:connect_net_box ()

Establish `net.box` connection. It's available in `net_box` field.

Server:http_request (method, path[, options])

Perform HTTP request.

Parameters:

- *method:* (string)
- *path:* (string)
- *options:*
  - *body:* (string) request body (optional)
  - *json:* data to encode as JSON into request body (optional)
  - *http:* (**tab**) other options for HTTP-client (optional)
  - *raise:* (**bool**) raise error when status is not in 200..299. Default to true. (optional)

Returns:

response object from HTTP client with helper methods.

Raises:

HTTPRequest error when response status is not 200.

See also:

- luatest.http_response

Server:new (object)

Build server object.

Parameters:

- *object:*
  - *command:* (string) Command to start server process.
  - *workdir:* (string) Value to be passed in `TARANTOOL_WORKDIR` .
  - *chdir:* (string) Path to cwd before running a process. (optional)
  - *env:* (**tab**) Table to pass as env variables to process. (optional)
  - *args:* (**tab**) Args to run command with. (optional)

- *http_port:* (**int**) Value to be passed in `TARANTOOL_HTTP_PORT` and used to perform HTTP requests. (optional)
- *net_box_port:* (**int**) Value to be passed in `TARANTOOL_LISTEN` and used for net_box connection. (optional)
- *net_box_credentials:* (**tab**) Override default net_box credentials. (optional)
- *alias:* (string) Instance alias. Used to prefix output. (optional)

Returns:

input object.

Server:start ()

Start server process.

Server:stop ()

Stop server process.

Changelog

Unreleased

0.5.2

- Throw parser error when .json is accessed on response with invalid body.
- Set *Content-Type: application/json* for *:http_request(..., {json = ...})* requests.

0.5.1

- Assertions pretty-prints non-string extra messages (useful for custom errors as tables).
- String values in errors are printed as valid Lua strings (with *%q* formatter).
- Add *TARANTOOL_DIR* to rockspec build.variables
- Replace *–error* and *–failure* options with *–fail-fast*.
- Fix stripping luatest trace from backtrace.
- Fix luarocks 3 test engine installation.

0.5.0

- *assert_is* treats *box.NULL* and *nil* as different values.
- Add luacov integration.
- Fix *assert_items_equals* for repeated values. Add support for *tuple* items.
- Add *assert_items_include* matcher.

- *assert_equals* uses same comparison rules for nested values.
- Fix generated group names when running files within specific directory.

0.4.0

- Fix not working *–exclude*, *–pattern* options
- Fix error messages for **_covers* matchers
- Raise error when *group()* is called with existing group name.
- Allow dot in group name.
- Prevent using / in group name.
- Decide group name from filename for *group()* call without args.
- *assert* returns input values.
- *assert[_not]_equals* works for Tarantool's box.tuple.
- Print tables in lua-compatible way in errors.
- Fix performance issue with large errors messages.
- Unify hooks definition: group hooks are defined via function calls.
- Keep running other groups when group hook failed.
- Prefix and colorize captured output.
- Fix numeric assertions for cdata values.

0.3.0

- Make –shuffle option accept *group*, *all*, *none* values
- Replace *raw* option for *Server:http_request* with *raise*.
- Remove not documented methods inherited from luaunit.
- Colorize report.

0.2.2

- Fix issue with crashes in capture.
- Do not raise error for 2xx responses in Server:http_request

0.2.1

- Don't run suite hooks when suite is not going to be run.
- Gracefully shutdown even when luanit calls *os.exit*.
- Show failed tests summary.
- Capture works with large outputs.

0.2.0

- GC'ed processes are killed automatically.
- Print captured output when suite/group hook fails.
- Rename Server:console to Server:net_box.
- Use real time instead of CPU time for duration.
- LDoc comments.
- Make assertions box.NULL aware.
- Luarocks 3 tests engine.
- *assert_covers* matcher.

0.1.1

- Fix exit code on failure.

0.1.0

- Initial implementation.

5.4 Справочник по настройке

В данном справочнике рассматриваются все опции и параметры, которые можно использовать в командной строке или в *файле инициализации*.

Tarantool можно запустить путем ввода одной из следующих команд:

```
$ tarantool
```

```
$ tarantool options
```

```
$ tarantool lua-initialization-file [ arguments ]
```

5.4.1 Опции командной строки

`-h, --help`
: Вывод аннотированного списка всех доступных опций и выход.

`-V, --version`
: Вывод названия и версии продукта, например:

```
$ ./tarantool --version
Tarantool 1.7.0-1216-g73f7154
Target: Linux-x86_64-Debug
...
```

В данном примере:

“Tarantool” – это название многократно используемого асинхронного сетевого фреймворка.

Версия из 3 чисел создается по стандартной схеме `<мажорная>-<минорная>-<патч-версия>`, где `<мажорная>` версия изменяется редко, `<минорная>` последовательно увеличивается с каждым новым выпущенным стабильным релизом и указывает на возможные несовместимые изменения, а `<патч-версия>` означает количество версий с исправленными ошибками с момента выхода стабильного релиза. Еще не вышедшие версии могут также содержать номер коммита и коммит SHA1, чтобы показать, насколько данная сборка отходит от последнего релиза.

“Target” – это платформа, на которой собран Tarantool. Некоторые платформенно-зависимые детали могут следовать за этой строкой.

Примечание: При выставлении номера версии Tarantool’а применяется git describe, и этот номер версии можно в любое время использовать для проверки соответствующего исходного кода в репозитории git.

5.4.2 Унифицированный идентификатор ресурса (URI)

Некоторые конфигурационные параметры и некоторые функции зависимы от URI (унифицированного идентификатора ресурса). Формат URI-строки похож на общий синтаксис URI-схемы. Он может содержать следующие данные (указаны по порядку): имя пользователя для входа в систему, пароль, имя хоста или IP-адрес хоста и номер порта. Обязательным параметром является только номер порта. Пароль является обязательным, только если указано имя пользователя – за исключением случаев, когда пользователем будет „guest“. Формально URI-синтаксис представляет собой `[хост:]порт` или `[имя-пользователя:пароль@]хост:порт`. Если хост не указан, то предполагается хост „0.0.0.0“ или „[::]“, что означает любой IPv4-адрес или IPv6-адрес на локальной машине соответственно. Если не указать имя-пользователя:пароль, предполагается, что пользователем будет „guest“. Некоторые примеры:

Фрагмент URI	Пример
порт	3301
хост:порт	127.0.0.1:3301
имя-пользователя:пароль@хост:порт	notguest:sesame@mail.ru:3301

В определенных обстоятельствах можно использовать доменный сокет Unix, когда ожидается URI, например, `unix/:/tmp/unix_domain_socket.sock` или просто `/tmp/unix_domain_socket.sock`.

Метод разбора URI проиллюстрирован в справочнике по *модулю uri*.

5.4.3 Файл инициализации

Если команда запуска Tarantool’а включает в себя файл инициализации, то Tarantool запустится посредством вызова Lua-программы из этого файла, который обычно называется «`script.lua`». В Lua-программу можно добавить дополнительные аргументы из командной строки или функции операционной системы, такие как `getenv()`. Lua-программа практически всегда запускается посредством вызова `box.cfg()`, если будет использоваться сервер базы данных или же необходимо открыть порты. Например, предположим, что файл `script.lua` содержит строки:

```
#!/usr/bin/env tarantool
box.cfg{
   listen              = os.getenv("LISTEN_URI"),
```

(continues on next page)

(продолжение с предыдущей страницы)

```
   memtx_memory        = 100000,
   pid_file            = "tarantool.pid",
    wal_max_size        = 2500
}
print('Starting ', arg[1])
```

и предположим, что переменная окружения LISTEN_URI содержит значение 3301, а также предположим, что в командной строке `~/tarantool/src/tarantool script.lua ARG`. Тогда вывод на экране может выглядеть следующим образом:

```
$ export LISTEN_URI=3301
$ ~/tarantool/src/tarantool script.lua ARG
... main/101/script.lua C> version 1.7.0-1216-g73f7154
... main/101/script.lua C> log level 5
... main/101/script.lua I> mapping 107374184 bytes for a shared arena..... main/101/script.lua I>␣
↪recovery start
... main/101/script.lua I> recovering from './00000000000000000000.snap'... main/101/script.lua I>␣
↪primary: bound to 0.0.0.0:3301
... main/102/leave_local_hot_standby I> ready to accept requests
Starting  ARG
... main C> entering the event loop
```

Если необходимо начать интерактивную сессию на том же терминале по окончании инициализации, можно использовать *console.start()*.

5.4.4 Конфигурационные параметры

Конфигурационные параметры выглядят так:

`box.cfg{[`*`ключ = значение [, ключ = значение ...]`*`]}`

Поскольку в `box.cfg` может быть множество конфигурационных параметров, а некоторые параметры (такие как адреса директорий) являются полупостоянными, лучше всего хранить `box.cfg` в Lua-файле. Как правило, такой Lua-файл представляет собой файл инициализации, который указан в командной строке Tarantool'а.

Большинство конфигурационных параметров предназначены для распределения ресурсов, открытия портом и указания поведения базы данных. Все параметры необязательны. Некоторые параметры динамичны, то есть могут изменяться во время исполнения кода посредством повторного вызова `box.cfg{}`.

Чтобы увидеть все ненулевые параметры, выполните `box.cfg` (без круглых скобок). Чтобы увидеть определенный параметр, например, адрес для прослушивания, выполните команду `box.cfg.listen`.

В последующих разделах описаны все параметры для основных возможностей, для хранения, для записи в бинарный журнал и создания снимков, для репликации, для работы по сети, для журналирования и для обратной связи.

Базовые параметры

- *background*
- *custom_proc_title*
- *listen*
- *memtx_dir*

- *pid_file*
- *read_only*
- *vinyl_dir*
- *vinyl_timeout*
- *username*
- *wal_dir*
- *work_dir*
- *worker_pool_threads*
- *strip_core*

`background`
: Для версий от 1.6.2. и выше. Запуск сервера в виде фоновой задачи. Чтобы это сработало, параметры *log* и *pid_file* должны быть не равны нулю.

  Тип: логический

  По умолчанию: false (ложь)

  Динамический: нет

`custom_proc_title`
: Для версий от 1.6.7. и выше. Добавление заданной строки к названию процесса сервера (что показано в столбце COMMAND для команд `ps -ef` и `top -c`.

  Например, как правило, `ps -ef` показывает процесс Tarantool-сервера так:

```
$ ps -ef | grep tarantool
1000     14939 14188  1 10:53 pts/2    00:00:13 tarantool <running>
```

  Но если указан конфигурационный параметр `custom_proc_title='sessions'`, вывод выглядит так:

```
$ ps -ef | grep tarantool
1000     14939 14188  1 10:53 pts/2    00:00:16 tarantool <running>: sessions
```

  Тип: строка

  По умолчанию: null

  Динамический: да

`listen`
: Для версий от 1.6.4. и выше. Номер порта для чтения/записи данных или строка *URI* (унифицированный идентификатор ресурса). Значение, используемое по умолчанию, отсутствует, поэтому его **обязательно указать**, если подключение выполняется с удаленных клиентов, которые не используют *"порт администрирования"*. Подключения, выполняемые с помощью `listen = URI`, называются соединения по бинарному порту или бинарному протоколу.

  Как правило, используется значение 3301.

Примечание: Реплика также привязана на этот порт и принимает соединения, но эти соединения служат только для чтения до тех пор, пока реплика не станет мастером.

Тип: целое число или строка

По умолчанию: null

Динамический: да

`memtx_dir`

Для версий от 1.7.4. и выше. Директория, где memtx хранит файлы снимков (.snap). Может относиться к *work_dir*. Если не указан, по умолчанию `work_dir`. См. также *wal_dir*.

Тип: строка

По умолчанию: «.»

Динамический: нет

`pid_file`

Для версий от 1.4.9. и выше. Хранение идентификатора процесса в данном файле. Может относиться к *work_dir*. Как правило, используется значение "`tarantool.pid`".

Тип: строка

По умолчанию: null

Динамический: нет

`read_only`

Для версий от 1.7.1. и выше. Чтобы ввести экземпляр сервера в режим только для чтения, выполните команду `box.cfg{read_only=true...}`. После этого не будут выполняться любые запросы по изменению персистентных данных с ошибкой `ER_READONLY`. Режим только для чтения следует использовать в *репликации* типа мастер-реплика. Режим только для чтения не влияет на запросы по изменению данных в спейсах, которые считаются *временными*. Хотя режим только для чтения не позволяет серверу делать записи в *WAL-файлы*, запись диагностической информации в *модуле log* все равно осуществляется.

Тип: логический

По умолчанию: false (ложь)

Динамический: да

Установка `read_only == true` по-разному влияет на спейсы в зависимости от опций, использованных во время *box.schema.space.create*, как описано в таблице:

Характеристика	Можно создать?	Допускает запись?	Реплицируется?	Сохраняется?
(по умолчанию)	нет	нет	да	да
temporary	нет	да	нет	нет
is_local	нет	да	нет	да

`vinyl_dir`
: Для версий от 1.7.1. и выше. Директория, где хранятся файлы или поддиректории vinyl'а. Может относиться к *work_dir*. Если не указан, по умолчанию `work_dir`.

  Тип: строка

  По умолчанию: «.»

  Динамический: нет

`vinyl_timeout`
: Для версий от 1.7.5. и выше. В движке базы данных vinyl есть планировщик, который осуществляет слияние. Когда vinyl'у не хватает доступной памяти, планировщик не сможет поддерживать скорость слияния в соответствии со входящими запросами обновления. В такой ситуации время ожидания обработки запроса может истечь после `vinyl_timeout` секунд. Это происходит редко, поскольку обычно vinyl управляет загрузкой при операциях вставки, когда не хватает скорости для слияния. Слияние можно запустить автоматически с помощью *index_object:compact()*.

  Тип: число с плавающей запятой

  По умолчанию: 60

  Динамический: да

`username`
: Для версий от 1.4.9. и выше. Имя пользователя в UNIX, на которое переключается система после запуска.

  Тип: строка

  По умолчанию: null

  Динамический: нет

`wal_dir`
: Для версий от 1.6.2. и выше. Директория, где хранятся файлы журнала упреждающей записи (.xlog). Может относиться к *work_dir*. Иногда в `wal_dir` и *memtx_dir* указываются разные значения, чтобы WAL-файлы и файлы снимков хранились на разных дисках. Если не указан, по умолчанию `work_dir`.

  Тип: строка

  По умолчанию: «.»

  Динамический: нет

`work_dir`
: Для версий от 1.4.9. и выше. Директория, где хранятся рабочие файлы базы данных. Экземпляр сервера переключается на `work_dir` с помощью *chdir(2)* после запуска. Может относиться к текущей директории. Если не указан, по умолчанию = текущей директории. Другие параметры директории могут относиться к `work_dir`, например:

```
box.cfg{
    work_dir = '/home/user/A',
    wal_dir = 'B',
    memtx_dir = 'C'
}
```

поместит xlog-файлы в `/home/user/A/B`, файлы снимков в `/home/user/A/C`, а все остальные файлы или поддиректории в `/home/user/A`.

Тип: строка

По умолчанию: null

Динамический: нет

`worker_pool_threads`
: Для версий от 1.7.5. и выше. Максимальное количество потоков, используемых во время исполнения определенных внутренних процессов (сейчас *socket.getaddrinfo()* и *coio_call()*).

Тип: целое число

По умолчанию: 4

Динамический: да

`strip_core`
: Для версий от 2.2.2. и выше. Указывает, должны ли файлы coredump включать в себя память, выделенную для кортежей. (Эти файлы могут занимать много места, если Tarantool работает под высокой нагрузкой). Если установлено `true`, то память, выделенная для кортежей, НЕ включается в файлы coredump. В более старых версиях Tarantool по умолчанию стояло `false`.

Тип: логический

По умолчанию: true

Динамический: нет

Настройка хранения

- *memtx_memory*
- *memtx_max_tuple_size*
- *memtx_min_tuple_size*
- *vinyl_bloom_fpr*
- *vinyl_cache*
- *vinyl_max_tuple_size*

- *vinyl_memory*
- *vinyl_page_size*
- *vinyl_range_size*
- *vinyl_run_count_per_level*
- *vinyl_run_size_ratio*
- *vinyl_read_threads*
- *vinyl_write_threads*

`memtx_memory`
: Для версий от 1.7.4. и выше. Количество памяти, которое Tarantool выделяет для фактического хранения кортежей. При достижении предельного значения запросы вставки *INSERT* или обновления *UPDATE* выполняться не будут, выдавая ошибку `ER_MEMORY_ISSUE`. Сервер не выходит за установленный предел памяти `memtx_memory` при распределении кортежей, но есть дополнительная память, которая используется для хранения индексов и информации о подключении. В зависимости от рабочей конфигурации и загрузки, Tarantool может потреблять на 20% больше предела `memtx_memory`.

  Тип: число с плавающей запятой

  По умолчанию: 256 * 1024 * 1024 = 268435456 байтов

  Динамический: **да**, но нельзя уменьшить

`memtx_max_tuple_size`
: Для версий от 1.7.4. и выше. Размер наибольшего блока выделения памяти для движка базы данных memtx. Его можно увеличить, если есть необходимость в хранении больших кортежей. См. также *vinyl_max_tuple_size*.

  Тип: целое число

  По умолчанию: 1024 * 1024 = 1048576 байтов

  Динамический: нет

`memtx_min_tuple_size`
: Для версий от 1.7.4. и выше. Размер наименьшего блока выделения памяти . Его можно уменьшить, если кортежи очень малого размера. Значение должно быть от 8 до 1 048 280 включительно.

  Тип: целое число

  По умолчанию: 16 байтов

  Динамический: нет

`vinyl_bloom_fpr`
: Для версий от 1.7.4. и выше. Доля ложноположительного срабатывания фильтра Блума – подходящая вероятность того, что фильтр Блума выдаст ошибочный результат. Настройка `vinyl_bloom_fpr` – это значение, которое используется по умолчанию для одного из параметров в таблице *Параметры space_object:create_index()*.

Тип: число с плавающей запятой

По умолчанию = 0.05

Динамический: нет

`vinyl_cache`

Для версий от 1.7.4. и выше. Размер кэша для движка базы данных vinyl. Размер кэша можно изменить динамически.

Тип: целое число

По умолчанию = 128 * 1024 * 1024 = 134217728 байтов

Динамический: **да**

`vinyl_max_tuple_size`

Для версий от 1.7.5. и выше. Размер наибольшего блока выделения памяти для движка базы данных vinyl. Его можно увеличить, если есть необходимость в хранении больших кортежей. См. также *memtx_max_tuple_size*.

Тип: целое число

По умолчанию: 1024 * 1024 = 1048576 байтов

Динамический: нет

`vinyl_memory`

Для версий от 1.7.4. и выше. Максимальное количество байтов оперативной памяти, которые использует vinyl.

Тип: целое число

По умолчанию = 128 * 1024 * 1024 = 134217728 байтов

Динамический: **да**, но нельзя уменьшить

`vinyl_page_size`

Для версий от 1.7.4. и выше. Размер страницы в байтах. Страница представляет собой блок чтения и записи для операций на диске vinyl. Настройка `vinyl_page_size` – это значение, которое используется по умолчанию для одного из параметров в таблице *Параметры space_object:create_index()*.

Тип: целое число

По умолчанию = 8 * 1024 = 8192 байтов

Динамический: нет

`vinyl_range_size`

Для версий от 1.7.4. и выше. Максимальный размер диапазона для индекса vinyl'а. Максимальный размер диапазона влияет на принятие решения о *разделении* диапазона.

Если `vinyl_range_size` содержит не нулевое значение nil и не 0, это значение используется в качестве значения по умолчанию для параметра `range_size` в таблице *Параметры space_object:create_index()*.

Если `vinyl_range_size` содержит нулевое значение nil или 0, а параметр `range_size` не задан при создании индекса, то Tarantool сам задает это значение позднее в результате оценки производительности. Чтобы узнать текущее значение, используйте *index_object:stat().range_size*.

До версии Tarantool'а 1.10.2, значение `vinyl_range_size` по умолчанию было 1073741824.

Тип: целое число

По умолчанию = нулевое значение nil байтов

Динамический: нет

`vinyl_run_count_per_level`

Для версий от 1.7.4. и выше. Максимальное количество забегов на уровень журнально-структурированного дерева со слиянием в vinyl'е. Настройка `vinyl_run_count_per_level` – это значение, которое используется по умолчанию для одного из параметров в таблице *Параметры space_object:create_index()*.

Тип: целое число

По умолчанию = 2

Динамический: нет

`vinyl_run_size_ratio`

Для версий от 1.7.4. и выше. Отношение размеров различных уровней журнально-структурированного дерева со слиянием. Настройка `vinyl_run_size_ratio` – это значение, которое используется по умолчанию для одного из параметров в таблице *Параметры space_object:create_index()*.

Тип: число с плавающей запятой

По умолчанию = 3.5

Динамический: нет

`vinyl_read_threads`

Для версий от 1.8.2. и выше. Максимальное количество потоков чтения, которые vinyl может использовать в одновременных операциях, такие как ввод-вывод и компрессия.

Тип: целое число

По умолчанию = 1

Динамический: нет

`vinyl_write_threads`

Для версий от 1.8.2. и выше. Максимальное количество потоков записи, которые vinyl может использовать в одновременных операциях, такие как ввод-вывод и компрессия.

Тип: целое число
По умолчанию = 2
Динамический: нет

Демон создания контрольных точек

- *checkpoint_count*
- *checkpoint_interval*
- *checkpoint_wal_threshold*

Демон создания контрольных точек – это постоянно работающий файбер. Периодически он может создавать *файлы снимка (.snap)*, а затем может удалять старые файлы снимка.

Настройки конфигурации *checkpoint_interval* и *checkpoint_count* определяют длительность интервалов и количество снимков, которое должно присутствовать до начала удалений.

Сборщик мусора Tarantool'а

Демон создания контрольных точек может запустить сборщик мусора Tarantool'а, который удаляет старые файлы. Такой сборщик мусора не отличается от сборщика мусора в Lua, который предназначен для Lua-объектов, и от сборщика мусора, который специализируется на *обработке блоков шарда*.

Если демон создания контрольных точек удаляет старый файл снимка, сборщик мусора Tarantool'а также удалит любые файлы *журнала упреждающей записи (.xlog)* старше файла снимка, содержащие информацию, которая присутствует в файле снимка. Он также удаляет устаревшие файлы `.run` в vinyl'е.

Демон создания контрольных точек и сборщик мусора Tarantool'а **не удалят** файл, если:

- идет **резервное копирование**, и файл еще не был скопирован (см. *«Резервное копирование»*), или
- идет **репликация**, и файл еще не был передан на реплику (см. *«Архитектуру механизма репликации»*),
- реплика подключается, или
- реплика отстает. Ход выполнения на каждой реплике отслеживается. Если реплика далеко не актуальна, сервер останавливается, чтобы она могла обновиться. Если администратор делает вывод, что реплика окончательно недоступна, необходимо перезагрузить сервер или же (предпочтительно) *удалить реплику из кластера*.

`checkpoint_interval`
: Для версий от 1.7.4. и выше. Промежуток времени между действиями демона создания контрольных точек в секундах. Если значение параметра `checkpoint_interval` больше нуля, и выполняется изменение базы данных, то демон создания контрольных точек будет вызывать *box.snapshot* каждые `checkpoint_interval` секунд, каждый раз создавая новый файл снимка. Если значение параметра `checkpoint_interval` равно нулю, то демон создания контрольных точек отключен.

  Пример:

```
box.cfg{checkpoint_interval=60}
```

  приведет к созданию нового снимка базы данных демоном создания контрольных точек каждую минуту, если наблюдается активность в базе данных.

  Тип: целое число

По умолчанию: 3600 (один час)

Динамический: да

`checkpoint_count`

Для версий от 1.7.4. и выше. Максимальное количество снимков, которые могут находиться в директории *memtx_dir* до того, как демон создания контрольных точек будет удалять старые снимки. Если значение `checkpoint_count` равно нулю, то демон создания контрольных точек не удаляет старые снимки. Например:

```
box.cfg{
    checkpoint_interval = 3600,
    checkpoint_count  = 10
}
```

заставит демон создания контрольных точек создавать снимок каждый час до тех пор, пока не будет создано десять снимков. Затем самый старый снимок удаляется (а также любые связанные с ним WAL-файлы) после создания нового снимка.

Следует помнить, что как упоминалось выше, снимки не удаляются, если выполняется репликация, и файл еще не был передан на реплику. Таким образом, параметр `checkpoint_count` бесполезен, если какая-то реплика неактивна.

Тип: целое число

По умолчанию: 2

Динамический: да

`checkpoint_wal_threshold`

Для версий от 2.1.2. и выше. Порог общего размера в байтах всех файлов WAL, созданных с момента последней контрольной точки. После превышения настроенного порога поток WAL уведомляет демона контрольной точки о том, что он должен создать новую контрольную точку и удалить старые файлы WAL.

Этот параметр позволяет администраторам справиться с проблемой, которая может возникнуть при вычислении объема дискового пространства для раздела, содержащего файлы WAL.

Например, предположим, что *checkpoint_interval* = 2 и *checkpoint_count* = 5 и в среднем за каждый интервал Tarantool записывает 1 Гб. Тогда можно будет вычислить, что необходимо (2*5*1) 10 Гб. Но это вычисление было бы неправильным, если бы вместо записи 1 ГБ в течение одного интервала между контрольными точками, Tarantool столкнулся с необычным всплеском и попытался записать 11 ГБ, что привело бы к ошибке операционной системы ENOSPC («нет места»). Установив значение `checkpoint_wal_threshold` на меньшее, скажем, 9 Гб, администратор может предотвратить ошибку.

Тип: целое число

По умолчанию: 10^18 (большое число, так что предела как-будто нет)

Динамический: да

Записи в бинарный журнал и создание снимков

- *force_recovery*,

- *wal_max_size*,
- *snap_io_rate_limit*,
- *wal_mode*,
- *wal_dir_rescan_delay*

`force_recovery`
: Для версий от 1.7.4. и выше. Если значение `force_recovery` равно true (правда), Tarantool пытается продолжать работу при обнаружении ошибки во время чтения *файла снимка* (при запуске экземпляра сервера) или *файла журнала упреждающей записи* (при запуске экземпляра сервера или применении обновлений к реплике): пропускает нерабочие записи, считывает максимальное количество данных и позволяет завершить процесс предупреждением. Пользователи могут предотвратить повторное появление ошибки, записав данные в базу и выполнив *box.snapshot()*.

В остальных случаях Tarantool прерывает восстановление на ошибке чтения.

Тип: логический

По умолчанию: false (ложь)

Динамический: нет

`wal_max_size`
: Для версий от 1.7.4. и выше. Максимальное количество байтов в отдельном журнале упреждающей записи. Если в результате запроса файл .xlog будет больше, чем указано в параметре `wal_max_size`, Tarantool создает другой WAL-файл – то же самое происходит, когда достигнуто количество строк в журнале, указанное в *rows_per_wal*.

Тип: целое число

По умолчанию: 268435456 (256 * 1024 * 1024) байтов

Динамический: нет

`snap_io_rate_limit`
: Для версий от 1.4.9. и выше. Уменьшение загрузки *box.snapshot* при выполнении операций вставки, обновления и удаления (INSERT/UPDATE/DELETE) путем установки предела скорости записи на диск – количества мегабайт в секунду. Того же эффекта можно достичь, разделив директории *wal_dir* и *memtx_dir* и перенося снимки на отдельный диск. Такой предел также ограничивает результат *box.stat.vinyl().regulator* относительно скорости записи дампов в файлы формата .run и .index.

Тип: число с плавающей запятой

По умолчанию: null

Динамический: **да**

`wal_mode`
: Для версий от 1.6.2. и выше. Определение синхронизации работы файбера с журналом упреждающей записи:

  - `none`: журнал упреждающей записи не поддерживается;

- `write`: *файберы* ожидают записи данных в журнал упреждающей записи (не *fsync(2)*);
- `fsync`: файберы ожидают данные, синхронизация *fsync(2)* следует за каждой операцией записи *write(2)*;

Тип: строка
По умолчанию: «write»
Динамический: нет

`wal_dir_rescan_delay`
Для версий от 1.6.2. и выше. Количество секунд между периодическим сканирование директории WAL-файла при проверке изменений в WAL-файле для целей *репликации* или *горячего резервирования*.

Тип: число с плавающей запятой
По умолчанию: 2
Динамический: нет

Горячее резервирование

`hot_standby`
Для версий от 1.7.4. и выше. Запуск сервера в режиме **горячего резервирования**.

Горячее резервирование – это функция, которая обеспечивает простое восстановление после отказа без *репликации*.

Предполагается, что есть два экземпляра сервера, использующих одну и ту же конфигурацию. Первый из них станет «основным» экземпляром. Тот, который запускается вторым, станет «резервным» экземпляром.

Чтобы создать резервный экземпляр, запустите второй экземпляр Tarantool-сервера на том же компьютере с теми же настройками конфигурации *box.cfg* – включая одинаковые директории и ненулевые URI – и с дополнительной настройкой конфигурации `hot_standby = true`. В ближайшее время вы увидите уведомление, которое заканчивается словами `I> Entering hot standby mode` (вход в режим горячего резервирования). Всё в порядке – это означает, что резервный экземпляр готов взять работу на себя, если основной экземпляр прекратит работу.

Резервный экземпляр начнет инициализацию и попытается заблокировать *wal_dir*, но не сможет, поскольку директория `wal_dir` заблокирована основным экземпляром. Поэтому резервный экземпляр входит в цикл, выполняя чтение журнала упреждающей записи, в который записывает данные основной экземпляр (поэтому два экземпляра всегда синхронизированы), и пытаясь произвести блокировку. Если основной экземпляр по какой-либо причине прекращает работу, блокировка снимается. В таком случае резервный экземпляр сможет заблокировать директорию на себя, подключится по адресу для *прослушивания* и станет основным экземпляром. В ближайшее время вы увидите уведомление, которое заканчивается словами `I> ready to accept requests` (готов принимать запросы).

Таким образом, если основной экземпляр прекращает работу, время простоя отсутствует.

Функция горячего резервирования не работает:

- если *wal_dir_rescan_delay = большое число* (в Mac OS и FreeBSD); на этих платформах цикл запрограммирован на повторение каждые `wal_dir_rescan_delay` секунд.

- если *wal_mode = „none“*; будет работать только при `wal_mode = 'write'` или `wal_mode = 'fsync'`.
- со спейсами, созданными на движке vinyl *engine = „vinyl“*; работает с движком memtx `engine = 'memtx'`.

Тип: логический
По умолчанию: false (ложь)
Динамический: нет

Репликация

- *replication*
- *replication_connect_timeout*
- *replication_connect_quorum*
- *replication_skip_conflict*
- *replication_sync_lag*
- *replication_sync_timeout*
- *replication_timeout*
- *replicaset_uuid*
- *instance_uuid*

`replication`

Для версий от 1.7.4. и выше. Если `replication` не содержит пустую строку, экземпляр считается *репликой*. Реплика попытается подключиться к мастеру, указанному в параметре `replication` по *URI* (унифицированному идентификатору ресурса), например:

konstantin:secret_password@tarantool.org:3301

Если в наборе реплик более одного источника репликации, укажите массив URI, например (замените „uri“ и „uri2“ в данном примере на рабочие URI):

`box.cfg{ replication = { „uri1“, „uri2“ } }`

Если один из URI «свой» – то есть один URI принадлежит экземпляру, где выполняется `box.cfg{}` – он не принимается во внимание. Таким образом, можно использовать одну и ту же настройку параметра `replication` на нескольких экземплярах сервера, как показано в *этих примерах*.

По умолчанию, пользователем считается „guest“.

Реплика в режиме только для чтения не принимает запросы по изменению данных по порту для *прослушивания*.

Параметр `replication` является динамическим, то есть для входа в режим мастера необходимо просто присвоить параметру `replication` пустую строку и выполнить следующее:

`box.cfg{ replication = новое-значение }`

Тип: строка
По умолчанию: null
Динамический: **да**

`replication_connect_timeout`

Для версий от 1.9.0. и выше. Количество секунд, в течение которых реплика ожидает попытки подключения к мастеру в кластере. Для получения подробной информации, см. *статус orphan*.

This parameter is different from *replication_timeout*, which a master uses to disconnect a replica when the master receives no acknowledgments of heartbeat messages.

Тип: число с плавающей запятой

По умолчанию: 30

Динамический: **да**

`replication_connect_quorum`

Для версий от 1.9.0. и выше. По умолчанию, реплика попытается подключиться ко всем мастерам или не запустится. (По умолчанию, рекомендуется, чтобы у всех реплик был одинаковый UUID набора реплик).

Однако, если указать `replication_connect_quorum = N`, где N означает число больше или равное нулю, это будет означать, что реплике нужно подключиться к N количеству мастеров.

Данный параметр используется во время настройки и *обновления конфигурации*. При настройке `replication_connect_quorum = 0` Tarantool не требует немедленного переподключения в случае восстановления. Для получения подробной информации, см. *статус orphan*.

Пример:

```
box.cfg{replication_connect_quorum=2}
```

Тип: целое число

По умолчанию: null

Динамический: **да**

`replication_skip_conflict`

Для версий от 1.10.1. и выше. По умолчанию, если реплика добавляет уникальный ключ, который уже добавила другая реплика, репликация *останавливается* с ошибкой = ER_TUPLE_FOUND.

Однако если указать `replication_skip_conflict = true`, пользователи могут задать пропуск таких ошибок. Так, вместо сохранения сломанной транзакции в xlog, там будет записано `NOP` (No operation).

Пример:

```
box.cfg{replication_skip_conflict=true}
```

Тип: логический

По умолчанию: false (ложь)

Динамический: **да**

Примечание: `replication_skip_conflict = true` рекомендуется использовать только для ручного восстановления репликации.

`replication_sync_lag`

Для версий от 1.9.0. и выше. Максимально допустимое *отставание* для реплики. Если реплика *синхронизируется* (то есть получает обновления от мастера), она может обновиться не полностью. Количество секунд, когда реплика находится позади мастера, называется «отставание» (lag). Синхронизация считается завершенной, когда отставание реплики меньше или равно `replication_sync_lag`.

Если пользователь задает значение `replication_sync_lag`, равное nil или 365 * 100 * 86400 (TIMEOUT_INFINITY), то отставание не имеет значения – реплика всегда будет синхронизирована. Кроме того, отставание не учитывается (считается бесконечным), если мастер работает на версии Tarantool'а старше 1.7.7, которая не отправляет *сообщения контрольного сигнала*.

Этот параметр не учитывается во время настройки. Для получения подробной информации, см. *статус orphan*.

Тип: число с плавающей запятой

По умолчанию: 10

Динамический: **да**

`replication_sync_timeout`

Для версий от 1.10.2. и выше. Количество секунд, в течение которых реплика ожидает попытки синхронизации с мастером в кластере или *кворумом* мастеров после подключения или во время *обновления конфигурации*, что может никогда не произойти, если значение `replication_sync_lag` меньше сетевой задержки, или реплика не может поддерживать темп обновлений мастера. По истечении времени `replication_sync_timeout` реплика получает *статус orphan*.

Тип: число с плавающей запятой

По умолчанию: 300

Динамический: **да**

`replication_timeout`

Для версий от 1.8.2. и выше. Если у мастера нет обновлений для реплик, он отправляет сообщения контрольного сигнала каждые `replication_timeout` секунд, а каждая реплика возвращает сообщение подтверждения.

И мастер, и реплики запрограммированы разорвать соединение при отсутствии сообщений в течение четырех промежутков времени, указанного в параметре `replication_timeout`. После разрыва соединения реплика пытается снова подключиться к мастеру.

См. дополнительную информацию в разделе *Мониторинг набора реплик*.

Тип: целое число

По умолчанию: 1

Динамический: **да**

`replicaset_uuid`

Для версий от 1.9.0. и выше. Как описано в разделе *«Архитектура механизма репликации»*, каждый набор реплик идентифицируется по Универсальному уникальному идентификатору (UUID), который называется **UUID набора реплик**, и каждый экземпляр идентифицируется по **UUID экземпляра**.

Как правило, достаточно позволить системе сгенерировать и форматировать строки, содержащие UUID, которые будут храниться постоянно.

Однако, некоторые администраторы предпочитают сохранять конфигурацию Tarantool'а в центральном репозитории, например, Apache ZooKeeper. Они могут самостоятельно присвоить значения экземплярам (*instance_uuid*) и набору реплик (`replicaset_uuid`) при первом запуске.

Общие правила:

- Значения должны быть действительно уникальными; они не должны одновременно принадлежать другим экземплярам или наборам реплик в той же инфраструктуре.
- Значения должны использоваться постоянно, неизменно с первого запуска (первоначальные значения хранятся в *файлах снимков* и проверяются при каждом перезапуске системы).
- Значения должны соответствовать требованиям RFC 4122. Нулевой UUID не допускается.

Формат UUID включает в себя шестнадцать октетов, представленных в виде 32 шестнадцатеричных чисел (с основанием 16) в пяти группах, разделенных дефисами в форме `8-4-4-4-12` – 36 символов (32 буквенно-цифровых символа и четыре дефиса).

Пример:

```
box.cfg{replicaset_uuid='7b853d13-508b-4b8e-82e6-806f088ea6e9'}
```

Тип: строка

По умолчанию: null

Динамический: нет

`instance_uuid`

Для версий от 1.9.0. и выше. Для целей администрирования репликации можно самостоятельно присвоить универсально уникальные идентификаторы экземпляру (`instance_uuid`) и набору реплик (`replicaset_uuid`) вместо использования сгенерированных системой значений.

Для получения подробной информации см. описание параметра *replicaset_uuid*.

Пример:

```
box.cfg{instance_uuid='037fec43-18a9-4e12-a684-a42b716fcd02'}
```

Тип: строка

По умолчанию: null

Динамический: нет

Работа с сетями

- *io_collect_interval*,

- *net_msg_max*
- *readahead*,

`io_collect_interval`
: Для версий от 1.4.9. и выше. Экземпляр уходит в режим ожидания на io_collect_interval секунд между итерациями событийного цикла. Это можно использовать для снижения загрузки процессора в системах с большим количество клиентских соединений, но нечастыми запросами (например, каждое соединение передает лишь небольшое количество запросов в секунду).

  Тип: число с плавающей запятой

  По умолчанию: null

  Динамический: **да**

`net_msg_max`
: Для версий от 1.10.1. и выше. Для обработки сообщений Tarantool выделяет файберы. Чтобы не допустить перегрузки файберов, которая влияет на всю систему, Tarantool ограничивает число сообщений, которые могут обрабатывать файберы, чтобы блокировать некоторые отложенные запросы.

  В мощных системах увеличьте значение `net_msg_max`, и планировщик немедленно приступит к обработке отложенных запросов.

  В более слабых системах уменьшите значение `net_msg_max`, чтобы снизить загрузку, хотя это и займет некоторое время, поскольку планировщик будет ожидать завершения уже запущенных запросов.

  По достижении значения `net_msg_max` Tarantool приостанавливает обработку входящих пакетов до тех пор, пока не обработает ранее полученные сообщения. Это не ограничение количества файберов, которые обрабатывают сетевые сообщения, напрямую, а скорее общесистемное ограничение ширины полосы канала. В свою очередь, это вызывает ограничение количества входящих сетевых сообщений, которые обрабатывает *поток обработки транзакций*, таким образом косвенно воздействуя на количество файберов, которые обрабатывают сетевые сообщения. (Количество файберов меньше количества сообщений, поскольку сообщения можно освободить сразу после доставки, а входящие запросы могут ждать обработки в течение некоторого времени после доставки.)

  Для стандартных систем подойдет значение, используемое по умолчанию (768).

  Тип: целое число

  По умолчанию: 768

  Динамический: **да**

`readahead`
: Для версий от 1.6.2. и выше. Размер буфера опережающего считывания, связанный с клиентским соединением. Чем больше буфер, тем больше памяти потребляет активное соединение и тем больше запросов можно считать из буфера операционной системы за отдельный системный вызов. Общее правило состоит в том, чтобы убедиться, что буфер может содержать как минимум несколько десятков соединений. Таким образом, если размер стандартного кортежа в запросе значительный, например, несколько килобайтов или даже мегабайтов, следует увеличить размер буфера опережающего считывания. Если не используется пакетная обработка запросов, будет целесообразно оставить значение, используемое по умолчанию.

Тип: целое число
По умолчанию: 16320
Динамический: **да**

Запись в журнал

- *log_level*
- *log*
- *log_nonblock*
- *too_long_threshold*
- *log_format*

log_level
: Для версий от 1.6.2. и выше. Уровень детализации записей *журнала*. Есть 7 уровней:

  - 1 – SYSERROR
  - 2 – ERROR
  - 3 – CRITICAL
  - 4 – WARNING
  - 5 – INFO
  - 6 – VERBOSE
  - 7 – DEBUG

  Задав значение параметра log_level, можно включить запись в журнал всех событий заданного уровня или ниже. По умолчанию, Tarantool выводит записи в стандартный поток сообщений об ошибках, но это можно изменить с помощью конфигурационного параметра *log*.

  Тип: целое число
  По умолчанию: 5
  Динамический: **да**

  Внимание: до версии Tarantool'а 1.7.5 было только 6 уровней, из них шестым был уровень DEBUG. Начиная с версии Tarantool'а 1.7.5 VERBOSE становится уровнем 6, а DEBUG – уровнем 7. VERBOSE представляет собой новый уровень для мониторинга повторяющихся событий, которые бы привели к слишком большому количеству записей журнала при использовании уровня INFO.

log
: Для версий от 1.7.4. и выше. По умолчанию, Tarantool выводит записи в стандартный поток сообщений об ошибках (stderr). Если задан параметр log, Tarantool отправит записи журнала в файл, в конвейер или в системный журнал syslog.

  Пример настройки для отправки журнала в файл:

```
box.cfg{log = 'tarantool.log'}
-- или
box.cfg{log = 'file:tarantool.log'}
```

Откроется файл `tarantool.log` для вывода в директории сервера, используемой по умолчанию. Если в строке `log` нет префикса или есть префикс «file:», то строка считается путем к файлу.

Пример настройки для отправки журнала в конвейер:

```
box.cfg{log = '| cronolog tarantool.log'}
-- или
box.cfg{log = 'pipe: cronolog tarantool.log'}'
```

Запустится программа cronolog при запуске сервера, которая будет отправлять все сообщения журнала на стандартный вывод (`stdin`) в cronolog. Если строка `log` начинается с „|“ или содержит префикс «pipe:», то строка считается Unix-конвейером.

Пример настройки для отправки журнала в системный журнал syslog:

```
box.cfg{log = 'syslog:identity=tarantool'}
-- или
box.cfg{log = 'syslog:facility=user'}
-- или
box.cfg{log = 'syslog:identity=tarantool,facility=user'}
-- или
box.cfg{log = 'syslog:server=unix:/dev/log'}
```

Если строка `log` начинается с «syslog:», это считается сообщением для программы syslogd, которая, как правило, работает в фоне на любой Unix-платформе. Настройка может быть: „syslog:“, „syslog:facility=...“, „syslog:identity=...“, „syslog:server=...“, или их комбинация.

Настройка `syslog:identity` представляет собой произвольную строку, которая размещается в начале всех сообщений. По умолчанию: tarantool.

В настоящий момент настройка `syslog:facility` не учитывается, но будет использоваться в дальнейшем. Ее значением должно быть одно из ключевых слов syslog, которые сообщают программе syslogd, куда отправлять сообщение. Возможные значения: auth, authpriv, cron, daemon, ftp, kern, lpr, mail, news, security, syslog, user, uucp, local0, local1, local2, local3, local4, local5, local6, local7. По умолчанию: user.

Настройка `syslog:server` – это указатель для сервера syslog. Это может быть путь к сокету Unix, который начинается с «unix:», или же номер IPv4-порта. Значение по умолчанию для сокета: dev/log (в Linux) или /var/run/syslog (в Mac OS). Значение по умолчанию для порта: 514, UDP-порт.

При записи в файл Tarantool повторно открывает журнал при сигнале SIGHUP. Если журнал является программой, его PID сохраняется в переменной *log.logger_pid*. Необходимо отправить сигнал для ротации файлов журнала.

Тип: строка

По умолчанию: null

Динамический: нет

`log_nonblock`

Для версий от 1.7.4. и выше. Если значение `log_nonblock` равно true (правда), Tarantool не блокирует дескриптор файла журнала, когда он не готов вести запись, а вместо этого сбрасывает сообщение. Если задан высокий уровень *log_level*,и много сообщений попадают в файл журнала, перевод `log_nonblock` в true может улучшить производительность ценой потери некоторых сообщений журнала.

Данный параметр сработает, только если вывод производится в системный журнал `syslog` или в конвейер. Недопустимо устанавливать `log_nonblock` в `true`, если вывод идет в файл.

Значение по умолчанию `log_nonblock` равно `nil`, что означает, что поведение блокировки соответствует типу логгера. Это изменение в поведении: в более ранних версиях сервера Tarantool значение по умолчанию было `true`.

Тип: логический

По умолчанию: nil

Динамический: нет

`too_long_threshold`

Для версий от 1.6.2. и выше. Если обработка запроса занимает дольше времени, чем заданное значение (в секундах), в журнал заносится соответствующее предупреждение. Сработает, только если в *log_level* задан уровень 4 (WARNING) или выше.

Тип: число с плавающей запятой

По умолчанию: 0.5

Динамический: **да**

`log_format`

Для версий от 1.7.6. и выше. Данные в журнал записываются в двух форматах:

- „plain“ (по умолчанию) или
- „json“ (более детально с JSON-метками).

Вот как будет выглядеть запись в журнале после выполнения `box.cfg{log_format='plain'}`:

```
2017-10-16 11:36:01.508 [18081] main/101/interactive I> set 'log_format' configuration option␣
↪to "plain"
```

Вот как будет выглядеть запись в журнале после выполнения `box.cfg{log_format='json'}`:

```
{"time": "2017-10-16T11:36:17.996-0600",
"level": "INFO",
"message": "set 'log_format' configuration option to \"json\"",
"pid": 18081,|
"cord_name": "main",
"fiber_id": 101,
"fiber_name": "interactive",
"file": "builtin\/box\/load_cfg.lua",
"line": 317}
```

В простом формате (`log_format='plain'`) запись содержит время, идентификатор процесса, имя файбера, идентификатор файбера *fiber_id*, имя файбера *fiber_name*, *уровень записи в журнал* и сообщение.

В JSON-формате (`log_format='json'`) запись содержит все вышеперечисленное с соответствующими метками, а также имя файла и номер строки Tarantool-источника.

Недопустимо устанавливать значение „json“ для `log_format`, если вывод идет «syslog:».

Тип: строка
По умолчанию: „plain“
Динамический: **да**

Пример записи в журнал

Данный пример проиллюстрирует ротацию файлов журнала, то есть что происходит, когда экземпляр сервера производит запись в журнал? а при архивировании используются сигналы.

Запустите две оболочки, терминал №1 и терминал №2.

На терминале №1 запустите интерактивную сессию Tarantool'a, затем укажите, что запись в журнал ведется в файл *Log_file*, а затем поместите сообщение «Log Line #1» в файл журнала:

```
box.cfg{log='Log_file'}
log = require('log')
log.info('Log Line #1')
```

На терминале №2 используйте команду `mv`, чтобы файл журнала назывался *Log_file.bak*. Результатом будет то, что следующее сообщение журнала пойдет в файл *Log_file.bak*.

```
mv Log_file Log_file.bak
```

На терминале №1 поместите сообщение «Log Line #2» в файл журнала.

```
log.info('Log Line #2')
```

На терминале №2 используйте команду `ps`, чтобы найти ID процесса экземпляра Tarantool'a.

```
ps -A | grep tarantool
```

На терминале №2 используйте команду `kill -HUP` для отправки сигнала SIGHUP на экземпляр Tarantool'a. Результат: Tarantool снова откроет *Log_file*, и следующее сообщение журнала пойдет в *Log_file*. (Тот же результат можно получить путем выполнения команды log.rotate() на экземпляре.)

`kill -HUP` *process_id*

На терминале №1 поместите сообщение «Log Line #3» в файл журнала.

```
log.info('Log Line #3')
```

На терминале №2 используйте команду `less` для просмотра файлов. *Log_file.bak* будет содержать следующие строки, но дата и время будут указаны в зависимости от времени выполнения примера:

```
2015-11-30 15:13:06.373 [27469] main/101/interactive I> Log Line #1`
2015-11-30 15:14:25.973 [27469] main/101/interactive I> Log Line #2`
```

а *Log_file* будет содержать

```
log file has been reopened
2015-11-30 15:15:32.629 [27469] main/101/interactive I> Log Line #3
```

Обратная связь

- *feedback_enabled*

- *feedback_host*
- *feedback_interval*

По умолчанию, демон Tarantool'а отправляет небольшой пакет каждый час на https://feedback.tarantool.io. Пакет содержит три значения из *box.info*: `box.info.version`, `box.info.uuid` и `box.info.cluster_uuid`. Изменив конфигурационные параметры обратной связи, пользователи могут настроить или отключить эту функцию.

`feedback_enabled`
: Для версий от 1.10.1. и выше. Отправлять обратную связь или нет.

  Если задано значение `true`, обратная связь будет отправлена, как описано выше. Если задано значение `false`, обратная связь не отправляется.

  Тип: логический

  По умолчанию: true

  Динамический: **да**

`feedback_host`
: Для версий от 1.10.1. и выше. Адрес, на который отправляется пакет. Как правило, получателем будет Tarantool, но можно указать любой URL.

  Тип: строка

  По умолчанию: „https://feedback.tarantool.io“

  Динамический: **да**

`feedback_interval`
: Для версий от 1.10.1. и выше. Количество секунд между отправками, обычно 3600 (1 час).

  Тип: число с плавающей запятой

  По умолчанию: 3600

  Динамический: **да**

Устаревшие параметры

Данные параметры объявлены устаревшими с версии Tarantool'а 1.7.4:

- *coredump*
- *logger*
- *logger_nonblock*
- *panic_on_snap_error*,
- *panic_on_wal_error*
- *replication_source*
- *slab_alloc_arena*
- *slab_alloc_factor*

- *slab_alloc_maximal*
- *slab_alloc_minimal*
- *snap_dir*
- *snapshot_count*
- *snapshot_period*
- *rows_per_wal*,

`coredump`
Устаревший, не использовать.

Тип: логический
По умолчанию: false (ложь)
Динамический: нет

`logger`
Устаревший, заменен параметром *log*. Параметр был лишь переименован, его тип, значения и семантика остались прежними.

`logger_nonblock`
Устаревший, заменен параметром *log_nonblock*. Параметр был лишь переименован, его тип, значения и семантика остались прежними.

`panic_on_snap_error`
Устаревший, заменен параметром *force_recovery*.

Если при чтении файла снимка произошла ошибка (при запуске экземпляра сервера), прервать выполнение.

Тип: логический
По умолчанию: true
Динамический: нет

`panic_on_wal_error`
Устаревший, заменен параметром *force_recovery*.

Тип: логический
По умолчанию: true
Динамический: да

`replication_source`
Устаревший, заменен параметром *replication*. Параметр был лишь переименован, его тип, значения и семантика остались прежними.

`slab_alloc_arena`
Устаревший, заменен параметром *memtx_memory*.

Количество памяти, которое Tarantool выделяет для фактического хранения кортежей, **в гигабайтах**. При достижении предельного значения запросы вставки INSERT или обновления UPDATE выполняться не будут, выдавая ошибку `ER_MEMORY_ISSUE`. Сервер не выходит за установленный предел памяти `memtx_memory` при распределении кортежей, но есть дополнительная память, которая используется для хранения индексов и информации о подключении. В зависимости от рабочей конфигурации и загрузки, Tarantool может потреблять на 20% больше установленного предела.

Тип: число с плавающей запятой

По умолчанию: 1.0

Динамический: нет

`slab_alloc_factor`
: **Устаревший**, не использовать.

  Множитель для вычисления размеров блоков памяти, в которых хранятся кортежи. Уменьшение значения может привести к уменьшению потерь памяти в зависимости от общего объема доступной памяти и распределения размеров элементов.

  Тип: число с плавающей запятой

  По умолчанию: 1.1

  Динамический: нет

`slab_alloc_maximal`
: **Устаревший**, заменен параметром *memtx_max_tuple_size*. Параметр был лишь переименован, его тип, значения и семантика остались прежними.

`slab_alloc_minimal`
: **Устаревший**, заменен параметром *memtx_min_tuple_size*. Параметр был лишь переименован, его тип, значения и семантика остались прежними.

`snap_dir`
: **Устаревший**, заменен параметром *memtx_dir*. Параметр был лишь переименован, его тип, значения и семантика остались прежними.

`snapshot_period`
: **Устаревший**, заменен параметром *checkpoint_interval*. Параметр был лишь переименован, его тип, значения и семантика остались прежними.

`snapshot_count`
: **Устаревший**, заменен параметром *checkpoint_count*. Параметр был лишь переименован, его тип, значения и семантика остались прежними.

`rows_per_wal`
: **Устаревший**, заменен параметром *wal_max_size*. Параметр не позволял четко ограничивать размер журналов WAL.

5.5 Справочник по C API

5.5.1 Модуль *box*

`box_function_ctx_t`
Непрозрачная структура, передаваемая в хранимую процедуру на языке C.

int `box_return_tuple`(*box_function_ctx_t *ctx*, *box_tuple_t *tuple*)
Возврат кортежа с помощью хранимой процедуры на языке C.

Для возвращаемого кортежа Tarantool проводит автоматический подсчет ссылок. Пример программы, которая использует `box_return_tuple()`: *write.c*.

Параметры

- `ctx` (*box_funtion_ctx_t**) – непрозрачная структура, передаваемая Tarantool'ом в хранимую процедуру на языке C
- `tuple` (`box_tuple_t*`) – возвращаемый кортеж

Результат -1 в случае ошибки (возможная нехватка памяти; проверьте *box_error_last()*)

Результат 0 в остальных случаях

uint32_t `box_space_id_by_name`(const char **name*, uint32_t *len*)
Поиск идентификатора спейса по имени.

Данная функция делает запрос выборки SELECT из системного спейса `_vspace`.

Параметры

- `char* name` (*const*) – имя спейса
- `len` (*uint32_t*) – длина имени `name`

Результат `BOX_ID_NIL` в случае ошибки или отсутствия (проверьте *box_error_last()*)

Результат space_id в остальных случаях

См. также *box_index_id_by_name*

uint32_t `box_index_id_by_name`(uint32_t *space_id*, const char **name*, uint32_t *len*)
Поиск идентификатора индекса по имени.

Данная функция делает запрос выборки SELECT из системного спейса `_vindex`.

Параметры

- `space_id` (*uint32_t*) – идентификатор спейса
- `char* name` (*const*) – имя индекса
- `len` (*uint32_t*) – длина имени `name`

Результат `BOX_ID_NIL` в случае ошибки или отсутствия (проверьте *box_error_last()*)

Результат space_id в остальных случаях

См. также *box_space_id_by_name*

int `box_insert`(uint32_t *space_id*, const char **tuple*, const char **tuple_end*, *box_tuple_t ***result*)
Выполнение запроса вставки или замены (INSERT/REPLACE).

Параметры

- `space_id` (*uint32_t*) – идентификатор спейса
- `char* tuple` (*const*) – закодированный кортеж в формате MsgPack-массива ([field1, field2, ...])
- `char* tuple_end` (*const*) – конец кортежа `tuple`
- `result` (`box_tuple_t**`) – аргумент вывода. Возвращаемый кортеж. Можно задать значение NULL для сброса результата

Результат -1 в случае ошибки (проверьте *box_error_last()*)

Результат 0 в остальных случаях

См. также *space_object.insert()*

int `box_replace`(uint32_t *space_id*, const char **tuple*, const char **tuple_end*, *box_tuple_t* ***result*)

Выполнение запроса замены (REPLACE).

Параметры

- `space_id` (*uint32_t*) – идентификатор спейса
- `char* tuple` (*const*) – закодированный кортеж в формате MsgPack-массива ([field1, field2, ...])
- `char* tuple_end` (*const*) – конец кортежа `tuple`
- `result` (`box_tuple_t**`) – аргумент вывода. Возвращаемый кортеж. Можно задать значение NULL для сброса результата

Результат -1 в случае ошибки (проверьте *box_error_last()*)

Результат 0 в остальных случаях

См. также *space_object.replace()*

int `box_delete`(uint32_t *space_id*, uint32_t *index_id*, const char **key*, const char **key_end*, *box_tuple_t* ***result*)

Выполнение запроса удаления (DELETE).

Параметры

- `space_id` (*uint32_t*) – идентификатор спейса
- `index_id` (*uint32_t*) – идентификатор индекса
- `char* key` (*const*) – закодированный ключ в формате MsgPack-массива ([field1, field2, ...])
- `char* key_end` (*const*) – конец ключа `key`
- `result` (`box_tuple_t**`) – аргумент вывода. Старый кортеж. Можно задать значение NULL для сброса результата

Результат -1 в случае ошибки (проверьте *box_error_last()*)

Результат 0 в остальных случаях

См. также *space_object.delete()*

int `box_update`(uint32_t *space_id*, uint32_t *index_id*, const char **key*, const char **key_end*, const char **ops*, const char **ops_end*, int *index_base*, *box_tuple_t* ***result*)

Выполнение запроса обновления (UPDATE).

Параметры

- `space_id` (*uint32_t*) – идентификатор спейса

- `index_id` (*uint32_t*) – идентификатор индекса
- `char* key` (*const*) – закодированный ключ в формате MsgPack-массива ([field1, field2, ...])
- `char* key_end` (*const*) – конец ключа `key`
- `char* ops` (*const*) – закодированные операции в формате MsgPack-массива, например `[[ '=', field_id, value ], ['!', 2, 'xxx']]`
- `char* ops_end` (*const*) – конец раздела операций `ops`
- `index_base` (*int*) – 0, если идентификаторы полей field_id с основанием 0, как в C, 1, если идентификаторы полей с основанием 1, как в Lua
- `result` (`box_tuple_t**`) – аргумент вывода. Старый кортеж. Можно задать значение NULL для сброса результата

Результат -1 в случае ошибки (проверьте *box_error_last()*)

Результат 0 в остальных случаях

См. также *space_object.update()*

int `box_upsert`(uint32_t *space_id*, uint32_t *index_id*, const char **tuple*, const char **tuple_end*, const char **ops*, const char **ops_end*, int *index_base*, *box_tuple_t* ***result*)

Выполнение запроса обновления и вставки (UPSERT).

Параметры

- `space_id` (*uint32_t*) – идентификатор спейса
- `index_id` (*uint32_t*) – идентификатор индекса
- `char* tuple` (*const*) – закодированный кортеж в формате MsgPack-массива ([field1, field2, ...])
- `char* tuple_end` (*const*) – конец кортежа `tuple`
- `char* ops` (*const*) – закодированные операции в формате MsgPack-массива, например `[[ '=', field_id, value ], ['!', 2, 'xxx']]`
- `char* ops_end` (*const*) – конец операций `ops`
- `index_base` (*int*) – 0, если идентификаторы полей field_id с основанием 0, как в C, 1, если идентификаторы полей с основанием 1, как в Lua
- `result` (`box_tuple_t**`) – аргумент вывода. Старый кортеж. Можно задать значение NULL для сброса результата

Результат -1 в случае ошибки (проверьте *box_error_last()*)

Результат 0 в остальных случаях

См. также *space_object.upsert()*

int `box_truncate`(uint32_t *space_id*)

Очистка спейса.

Параметры

- `space_id` (*uint32_t*) – идентификатор спейса

5.5.2 Модуль *clock*

double `clock_realtime`(void)
double `clock_monotonic`(void)
double `clock_process`(void)
double `clock_thread`(void)

uint64_t `clock_realtime64`(void)
uint64_t `clock_monotonic64`(void)
uint64_t `clock_process64`(void)
uint64_t `clock_thread64`(void)

5.5.3 Модуль *coio*

`enum COIO_EVENT`

`enumerator COIO_READ`
событие чтения READ

`enumerator COIO_WRITE`
событие записи WRITE

int `coio_wait`(int *fd*, int *event*, double *timeout*)
Ожидание события чтения или записи (READ / WRITE) на сокете (`fd`) с передачей управления.

Параметры

- `fd` (*int*) – дескриптор файла сокета без блокировки
- `event` (*int*) – запрашиваемые события. Комбинация битовых флагов `COIO_READ | COIO_WRITE`.
- `timeout` (*double*) – время ожидания в секундах.

Результат 0 - время ожидания

Результат >0 - возвращаемые события. Комбинация битовых флагов `TNT_IO_READ | TNT_IO_WRITE`.

ssize_t `coio_call`(ssize_t (**func*)(va_list), ...)
Создание новой задачи для eio с указанной функцией и аргументами. Передает управление и ожидает окончания задачи. Функция может использовать конфигурационный параметр *worker_pool_threads*.

Во избежание двойной проверки ошибок функция не выбрасывает исключения. В большинстве случаев также необходимо проверять возвращаемое значение вызванной функции и выполнить необходимые действия. Если функция определяет номер ошибки errno, этот номер ошибки сохраняется в течение вызова.

Результат -1 и `errno` = ENOMEM, если задача не была создана

Результат возврат функции (`errno` сохраняется).

Пример:

```
static ssize_t openfile_cb(va_list ap)
{
        const char* filename = va_arg(ap);
        int flags = va_arg(ap);
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        return open(filename, flags);
}

if (coio_call(openfile_cb, "/tmp/file", 0) == -1)
    // обработка ошибок.
...
```

int coio_getaddrinfo(const char **host*, const char **port*, const struct addrinfo **hints*, struct addrinfo ***res*, double *timeout*)

Вариант функции getaddrinfo(3), совместимый с файберами.

int coio_close(int *fd*)

Закрытие `fd` и пробуждение любого файбера, заблокированного в вызове *coio_wait()* на данном сокете `fd`.

Параметры

- `fd` (*int*) – дескриптор файла сокета без блокировки

Результат результат `close(fd)`, см. *close(2)*

5.5.4 Модуль *error*

enum box_error_code

enumerator ER_UNKNOWN

enumerator ER_ILLEGAL_PARAMS

enumerator ER_MEMORY_ISSUE

enumerator ER_TUPLE_FOUND

enumerator ER_TUPLE_NOT_FOUND

enumerator ER_UNSUPPORTED

enumerator ER_NONMASTER

enumerator ER_READONLY

enumerator ER_INJECTION

enumerator ER_CREATE_SPACE

enumerator ER_SPACE_EXISTS

enumerator ER_DROP_SPACE

enumerator ER_ALTER_SPACE

enumerator ER_INDEX_TYPE

enumerator ER_MODIFY_INDEX

enumerator ER_LAST_DROP

enumerator ER_TUPLE_FORMAT_LIMIT

enumerator ER_DROP_PRIMARY_KEY

enumerator ER_KEY_PART_TYPE

enumerator ER_EXACT_MATCH

enumerator ER_INVALID_MSGPACK

enumerator ER_PROC_RET

enumerator ER_TUPLE_NOT_ARRAY

enumerator ER_FIELD_TYPE

enumerator ER_FIELD_TYPE_MISMATCH

enumerator ER_SPLICE

enumerator ER_UPDATE_ARG_TYPE

enumerator ER_TUPLE_IS_TOO_LONG

enumerator ER_UNKNOWN_UPDATE_OP

enumerator ER_UPDATE_FIELD

enumerator ER_FIBER_STACK

enumerator ER_KEY_PART_COUNT

enumerator ER_PROC_LUA

enumerator ER_NO_SUCH_PROC

enumerator ER_NO_SUCH_TRIGGER

enumerator ER_NO_SUCH_INDEX

enumerator ER_NO_SUCH_SPACE

enumerator ER_NO_SUCH_FIELD

enumerator ER_EXACT_FIELD_COUNT

enumerator ER_INDEX_FIELD_COUNT

enumerator ER_WAL_IO

enumerator ER_MORE_THAN_ONE_TUPLE

enumerator ER_ACCESS_DENIED

enumerator ER_CREATE_USER

enumerator ER_DROP_USER

enumerator ER_NO_SUCH_USER

enumerator ER_USER_EXISTS

enumerator ER_PASSWORD_MISMATCH

enumerator ER_UNKNOWN_REQUEST_TYPE

enumerator ER_UNKNOWN_SCHEMA_OBJECT

enumerator ER_CREATE_FUNCTION

enumerator ER_NO_SUCH_FUNCTION

enumerator ER_FUNCTION_EXISTS

enumerator ER_FUNCTION_ACCESS_DENIED

enumerator ER_FUNCTION_MAX

```
enumerator ER_SPACE_ACCESS_DENIED
enumerator ER_USER_MAX
enumerator ER_NO_SUCH_ENGINE
enumerator ER_RELOAD_CFG
enumerator ER_CFG
enumerator ER_UNUSED60
enumerator ER_UNUSED61
enumerator ER_UNKNOWN_REPLICA
enumerator ER_REPLICASET_UUID_MISMATCH
enumerator ER_INVALID_UUID
enumerator ER_REPLICASET_UUID_IS_RO
enumerator ER_INSTANCE_UUID_MISMATCH
enumerator ER_REPLICA_ID_IS_RESERVED
enumerator ER_INVALID_ORDER
enumerator ER_MISSING_REQUEST_FIELD
enumerator ER_IDENTIFIER
enumerator ER_DROP_FUNCTION
enumerator ER_ITERATOR_TYPE
enumerator ER_REPLICA_MAX
enumerator ER_INVALID_XLOG
enumerator ER_INVALID_XLOG_NAME
enumerator ER_INVALID_XLOG_ORDER
enumerator ER_NO_CONNECTION
enumerator ER_TIMEOUT
enumerator ER_ACTIVE_TRANSACTION
enumerator ER_NO_ACTIVE_TRANSACTION
enumerator ER_CROSS_ENGINE_TRANSACTION
enumerator ER_NO_SUCH_ROLE
enumerator ER_ROLE_EXISTS
enumerator ER_CREATE_ROLE
enumerator ER_INDEX_EXISTS
enumerator ER_TUPLE_REF_OVERFLOW
enumerator ER_ROLE_LOOP
enumerator ER_GRANT
enumerator ER_PRIV_GRANTED
enumerator ER_ROLE_GRANTED
```

enumerator ER_PRIV_NOT_GRANTED

enumerator ER_ROLE_NOT_GRANTED

enumerator ER_MISSING_SNAPSHOT

enumerator ER_CANT_UPDATE_PRIMARY_KEY

enumerator ER_UPDATE_INTEGER_OVERFLOW

enumerator ER_GUEST_USER_PASSWORD

enumerator ER_TRANSACTION_CONFLICT

enumerator ER_UNSUPPORTED_ROLE_PRIV

enumerator ER_LOAD_FUNCTION

enumerator ER_FUNCTION_LANGUAGE

enumerator ER_RTREE_RECT

enumerator ER_PROC_C

enumerator ER_UNKNOWN_RTREE_INDEX_DISTANCE_TYPE

enumerator ER_PROTOCOL

enumerator ER_UPSERT_UNIQUE_SECONDARY_KEY

enumerator ER_WRONG_INDEX_RECORD

enumerator ER_WRONG_INDEX_PARTS

enumerator ER_WRONG_INDEX_OPTIONS

enumerator ER_WRONG_SCHEMA_VERSION

enumerator ER_MEMTX_MAX_TUPLE_SIZE

enumerator ER_WRONG_SPACE_OPTIONS

enumerator ER_UNSUPPORTED_INDEX_FEATURE

enumerator ER_VIEW_IS_RO

enumerator ER_UNUSED114

enumerator ER_SYSTEM

enumerator ER_LOADING

enumerator ER_CONNECTION_TO_SELF

enumerator ER_KEY_PART_IS_TOO_LONG

enumerator ER_COMPRESSION

enumerator ER_CHECKPOINT_IN_PROGRESS

enumerator ER_SUB_STMT_MAX

enumerator ER_COMMIT_IN_SUB_STMT

enumerator ER_ROLLBACK_IN_SUB_STMT

enumerator ER_DECOMPRESSION

enumerator ER_INVALID_XLOG_TYPE

enumerator ER_ALREADY_RUNNING

enumerator ER_INDEX_FIELD_COUNT_LIMIT

enumerator ER_LOCAL_INSTANCE_ID_IS_READ_ONLY

enumerator ER_BACKUP_IN_PROGRESS

enumerator ER_READ_VIEW_ABORTED

enumerator ER_INVALID_INDEX_FILE

enumerator ER_INVALID_RUN_FILE

enumerator ER_INVALID_VYLOG_FILE

enumerator ER_CHECKPOINT_ROLLBACK

enumerator ER_VY_QUOTA_TIMEOUT

enumerator ER_PARTIAL_KEY

enumerator ER_TRUNCATE_SYSTEM_SPACE

enumerator box_error_code_MAX

box_error_t
Ошибка – содержит информацию об ошибке.

const char * box_error_type(const *box_error_t* *error)
Возврат типа ошибки, например, «ClientError», «SocketError» и т.д.

Параметры

- error (box_error_t*) – ошибка

Результат ненулевая строка

uint32_t box_error_code(const *box_error_t* *error)
Возврат кода ошибки IPROTO

Параметры

- error (box_error_t*) – ошибка

Результат enum *box_error_code*

const char * box_error_message(const *box_error_t* *error)
Возврат сообщения ошибки

Параметры

- error (box_error_t*) – ошибка

Результат ненулевая строка

box_error_t * box_error_last(void)
Получение информации о последней ошибке вызова API.

Обработка ошибок в Tarantool'е больше всего похожа на errno в стандартной библиотеке языка C libc. Все вызовы API возвращают -1 или NULL в случае ошибки. Внутренний указатель на тип box_error_t задается функциями, чтобы указать, что пошло не так. Это значение показательно, если вызов API не прошел (вернулось -1 или NULL).

Выполненная функция в некоторых случаях также может затрагивать последнюю ошибку. Необязательно удалять последнюю ошибку перед вызовом API-функций. Возвращаемый объект применим только до следующего вызова **любой** API-функции.

Следует задать последнюю ошибку с помощью box_error_set() из хранимых процедур на языке C, если необходимо вернуть специальное сообщение об ошибке. Можно повторно сгенерировать

последнюю API-ошибку в клиент IPROTO, сохранив текущее значение и вернув -1 to Tarantool из хранимой процедуры.

Результат последняя ошибка

void box_error_clear(void)
Удаление последней ошибки.

int box_error_set(const char *file, unsigned line, uint32_t code, const char *format, ...)
Определение последней ошибки.

Параметры

- char* file (const) –
- line (unsigned) –
- code (uint32_t) – IPROTO error code
- char* format (const) –
- ... – аргументы формата

См. также IPROTO error code

box_error_raise(code, format, ...)
Обратно совместимые определения API.

5.5.5 Модуль fiber

struct fiber
Файбер – содержит информацию о файбере.

typedef int (*fiber_func)(va_list)
Функции для выполнения в файбере.

struct fiber *fiber_new(const char *name, fiber_func f)
Создание нового файбера.

Берет файбер из кэша файберов, если в нем что-то есть. Может не сработать, только если недостаточно памяти для структуры файбера или стека файбера.

Созданный файбер автоматически возвращается в кэш файберов, когда выполнена его основная функция.

Параметры

- char* name (const) – строка с именем файбера
- f (fiber_func) – функция для выполнения в файбере

См. также fiber_start()

struct fiber *fiber_new_ex(const char *name, const struct fiber_attr *fiber_attr, fiber_func f)
Создание нового файбера с заданными атрибутами.

Может не сработать, только если недостаточно памяти для структуры файбера или стека файбера.

Созданный файбер автоматически возвращается в кэш файберов, если у него размер стека по умолчанию, когда выполнена его основная функция.

Параметры

- char* name (const) – строка с именем файбера

- `struct fiber_attr* fiber_attr` (*const*) – контейнер с атрибутами файбера
- `f` (`fiber_func`) – функция для выполнения в файбере

См. также *fiber_ start()*

void `fiber_start`(struct *fiber* **callee*, ...)
Запуск созданного файбера.

Параметры

- `fiber* callee` (*struct*) – запускаемый файбер
- `...` – аргументы для запуска файбера

void `fiber_yield`(void)
Передача управления другому файберу и ожидание его пробуждения.

См. также *fiber_ wakeup()*

void `fiber_wakeup`(struct *fiber* **f*)
Прерывание синхронного ожидания файбера

Параметры

- `fiber* f` (*struct*) – пробуждаемый файбер

void `fiber_cancel`(struct *fiber* **f*)
Отмена файбера (установка флага `FIBER_IS_CANCELLED`)

Если на нужном файбере установлен флаг `FIBER_IS_CANCELLABLE`, он возобновит работу (возможно досрочно). Тогда текущий файбер передает управление до тех пор, пока нужный файбер не будет удален (или не возобновит работу с помощью *fiber_ wakeup()*).

Параметры

- `fiber* f` (*struct*) – отменяемый файбер

bool `fiber_set_cancellable`(bool *yesno*)
Возможность или невозможность пробуждения текущего файбера сразу после его отмены.

Параметры

- `fiber* f` (*struct*) – файбер
- `yesno` (*bool*) – назначаемый статус

Результат предыдущий статус

void `fiber_set_joinable`(struct *fiber* **fiber*, bool *yesno*)
Определение файбера как присоединяемого (по умолчанию `false`)

Параметры

- `fiber* f` (*struct*) – файбер
- `yesno` (*bool*) – назначаемый статус

void `fiber_join`(struct *fiber* **f*)
Ожидание удаления файбера, а затем передача статуса его выполнения вызывающему клиенту. Файбер не должен быть откреплённым.

Параметры

- `fiber* f` (*struct*) – пробуждаемый файбер

Ранее: установлен флаг `FIBER_IS_JOINABLE`.

См. также *fiber_set_joinable()*

void `fiber_sleep`(double *s*)
Перевод текущего файбера в режим ожидания как минимум на „s“ секунд.

Параметры

- `s` (*double*) – время ожидания

Примечание: это и есть точка отмены.

См. также *fiber_is_cancelled()*

bool `fiber_is_cancelled`(void)
Проверка отмены текущего файбера (это делается вручную).

double `fiber_time`(void)
Сообщение времени начала цикла в виде числа двойной точности.

uint64_t `fiber_time64`(void)
Сообщение времени начала цикла в виде 64-битного целого числа.

void `fiber_reschedule`(void)
Перенос файбера для завершения событийного цикла.

struct `slab_cache`

struct *slab_cache* *`cord_slab_cache`(void)
Возврат `slab_cache`, подходящего для использования с библиотекой `tarantool/small`

struct *fiber* *`fiber_self`(void)
Возврат текущего файбера.

struct `fiber_attr`

void `fiber_attr_new`(void)
Создание нового контейнера с атрибутами файбера и его инициализация с параметрами по умолчанию.

Можно использовать для создания множества файберов: смена владельца не произойдет.

void `fiber_attr_delete`(struct *fiber_attr* **fiber_attr*)
Удаление `fiber_attr` и освобождение всех выделенных ресурсов. Используется, когда есть файберы, созданные с данным атрибутом.

Параметры

- `fiber_attr* fiber_attribute` (*struct*) – контейнер с атрибутами файбера

int `fiber_attr_setstacksize`(struct *fiber_attr* **fiber_attr*, size_t *stack_size*)
Определение размера стека файбера в контейнере с атрибутами файбера.

Параметры

- `fiber_attr* fiber_attr` (*struct*) – контейнер с атрибутами файбера
- `stack_size` (*size_t*) – размер стека для новых файберов (в байтах)

Результат 0, если выполнено

Результат -1, если не выполнено (если размер стека `stack_size` меньше минимально допустимого размера стека файбера)

size_t `fiber_attr_getstacksize`(struct *fiber_attr* **fiber_attr*)
Получение размера стека файбера из контейнера с атрибутами файбера.

Параметры

- `fiber_attr* fiber_attr` (*struct*) – контейнер с атрибутами файбера или NULL, по умолчанию

Результат размер стека (в байтах)

struct `fiber_cond`

Условная переменная: примитив синхронизации, который позволяет файберам в среде *кооперативной многозадачности* Tarantool'а передавать управление до выполнения какого-либо предиката.

Условия работы файбера поддерживают две основные операции – «wait» (ожидание) и «signal» (сигнал), – где «wait» откладывает выполнение файбера (то есть передает управление) до тех пор, пока не будет вызван «signal».

В отличие от `pthread_cond`, `fiber_cond` не требует функции-обертки в виде мьютекса или защелки.

struct *fiber_cond* *`fiber_cond_new`(void)

Создание новой условной переменной.

void `fiber_cond_delete`(struct *fiber_cond* *cond*)

Удаление условной переменной.

Примечание: поведение не определено, если есть файберы, ожидающие условной переменной.

Параметры

- `fiber_cond* cond` (*struct*) – удаляемая условная переменная

`void fiber_cond_signal(struct fiber_cond *cond);`

Пробуждение **одного** (любого) файбера, ожидающего условной переменной.

Не делает ничего, если нет ожидающих файберов.

Параметры

- `fiber_cond* cond` (*struct*) – условная переменная

`void fiber_cond_broadcast(struct fiber_cond *cond);`

Пробуждение **всех** файберов, ожидающих условной переменной.

Не делает ничего, если нет ожидающих файберов.

Параметры

- `fiber_cond* cond` (*struct*) – условная переменная

int `fiber_cond_wait_timeout`(struct *fiber_cond* *cond*, double *timeout*)

Приостановление выполнения текущего файбера (т.е. передача управления) до вызова *fiber_cond_signal()*.

Как и `pthread_cond`, `fiber_cond` может отправлять ложные сигналы пробуждения с помощью вызова *fiber_wakeup()* или *fiber_cancel()*. Настоятельно рекомендуется заключать вызовы данной функции в цикл и проверять предикат и *fiber_is_cancelled()* при каждой итерации.

Параметры

- `fiber_cond* cond` (*struct*) – условная переменная
- `double timeout` (*struct*) – время ожидания в секундах

Результат 0 при вызове *fiber_cond_signal()* или ложном пробуждении

Результат -1 в случае ожидания, и задается код ошибки „TimedOut“ (истекло время ожидания)

int `fiber_cond_wait`(struct *fiber_cond* *cond)
Ускоренный метод для *fiber_cond_wait_timeout()*.

5.5.6 Модуль *index*

`box_iterator_t`
Итератор спейса

`enum iterator_type`
Управление итерацией кортежей в индексе. Различные типы индексов поддерживают различные типы итераторов. Например, можно начать итерацию с определенного значения (ключ запроса), а затем получить все кортежи, ключи которых больше или равны (= GE) заданному ключу.

Если тип итератора не поддерживается выбранным типом индекса, конструктор итератора прекратит работу с ошибкой ER_UNSUPPORTED. Чтобы индекс можно было выбрать для первичного ключа, он должен поддерживать типы ITER_EQ и ITER_GE.

Значение ключа запроса NULL соответствует первому или последнему ключу в индексе, в зависимости от направления итерации (первый ключ для типов GE и GT, последний ключ для типов LE и LT). Таким образом, для итерации по всем кортежам в индексе можно использовать типы итерации ITER_GE или ITER_LE с начальным ключом, который равен NULL. Для ITER_EQ ключ не должен равняться NULL.

`enumerator ITER_EQ`
ключ == x в порядке возрастания

`enumerator ITER_REQ`
ключ == x в порядке убывания

`enumerator ITER_ALL`
все кортежи

`enumerator ITER_LT`
ключ < x

`enumerator ITER_LE`
ключ <= x

`enumerator ITER_GE`
ключ >= x

`enumerator ITER_GT`
ключ > x

`enumerator ITER_BITS_ALL_SET`
все биты из x заданы в ключе

`enumerator ITER_BITS_ANY_SET`
задан хотя бы один бит из x

`enumerator ITER_BITS_ALL_NOT_SET`
ни один бит не задан

`enumerator ITER_OVERLAPS`
ключ пересекается с x

`enumerator ITER_NEIGHBOR`
кортежи в порядке возрастания расстояния из указанной точки

box_iterator_t *box_index_iterator(uint32_t *space_id*, uint32_t *index_id*, int *type*, const char *key, const char *key_end)

Выделение и инициализация итератора для space_id, index_id.

Возвращаемый итератор следует удалить с помощью *box_iterator_free*.

Параметры

- `space_id` (*uint32_t*) – идентификатор спейса
- `index_id` (*uint32_t*) – идентификатор индекса
- `type` (*int*) – *iterator_type*
- `char* key` (*const*) – кодировка ключа в формате MsgPack-массива ([part1, part2, ...])
- `char* key_end` (*const*) – часть закодированного ключа `key`

Результат NULL в случае ошибки (проверьте *box_error_last()*)

Результат итератор в остальных случаях

См. также *box_iterator_next*, *box_iterator_free*

int box_iterator_next(*box_iterator_t* *iterator, *box_tuple_t* **result)

Получение следующего пункта из итератора `iterator`.

Параметры

- `iterator` (`box_iterator_t*`) – итератор, возвращаемый *box_index_iterator*
- `result` (`box_tuple_t**`) – аргумент вывода. Результатом будет кортеж или NULL, если данных больше нет.

Результат -1 в случае ошибки (проверьте *box_error_last()*)

Результат 0 в случае выполнения. Отсутствие данных не является ошибкой.

void box_iterator_free(*box_iterator_t* *iterator)

Удаление и освобождение итератора.

Параметры

- `iterator` (`box_iterator_t*`) – итератор, возвращаемый *box_index_iterator*

int iterator_direction(enum iterator_type *type*)

Определение направления заданного типа итератора: -1 для REQ, LT, LE, и +1 для всех остальных.

ssize_t box_index_len(uint32_t *space_id*, uint32_t *index_id*)

Возврат номера элемента в индексе.

Параметры

- `space_id` (*uint32_t*) – идентификатор спейса
- `index_id` (*uint32_t*) – идентификатор индекса

Результат -1 в случае ошибки (проверьте *box_error_last()*)

Результат >= 0 в остальных случаях

ssize_t box_index_bsize(uint32_t *space_id*, uint32_t *index_id*)

Возврат количества байтов памяти, используемых индексом.

Параметры

- `space_id` (*uint32_t*) – идентификатор спейса

- `index_id` (*uint32_t*) – идентификатор индекса

Результат -1 в случае ошибки (проверьте *box_error_last()*)

Результат >= 0 в остальных случаях

int `box_index_random`(uint32_t *space_id*, uint32_t *index_id*, uint32_t *rnd*, *box_tuple_t* ***result*)

Возврат случайного кортежа из индекса (используется для статистического анализа).

Параметры

- `space_id` (*uint32_t*) – идентификатор спейса
- `index_id` (*uint32_t*) – идентификатор индекса
- `rnd` (*uint32_t*) – случайное начальное число
- `result` (`box_tuple_t**`) – аргумент вывода. Результатом будет кортеж или NULL, если в спейсе нет кортежей.

См. также *index_object.random*

int `box_index_get`(uint32_t *space_id*, uint32_t *index_id*, const char **key*, const char **key_end*, *box_tuple_t* ***result*)

Получение кортежа из индекса по ключу.

Следует отметить, что данная функция работает намного быстрее, чем *index_object.select* или *box_index_iterator* + *box_iterator_next*.

Параметры

- `space_id` (*uint32_t*) – идентификатор спейса
- `index_id` (*uint32_t*) – идентификатор индекса
- `char* key` (*const*) – кодировка ключа в формате MsgPack-массива ([part1, part2, ...])
- `char* key_end` (*const*) – часть закодированного ключа `key`
- `result` (`box_tuple_t**`) – аргумент вывода. Результатом будет кортеж или NULL, если в спейсе нет кортежей.

Результат -1 в случае ошибки (проверьте *box_error_last()*)

Результат 0, если выполнено

См. также `index_object.get()`

int `box_index_min`(uint32_t *space_id*, uint32_t *index_id*, const char **key*, const char **key_end*, *box_tuple_t* ***result*)

Возврат первого (минимального) кортежа, который соответствует заданному ключу.

Параметры

- `space_id` (*uint32_t*) – идентификатор спейса
- `index_id` (*uint32_t*) – идентификатор индекса
- `char* key` (*const*) – кодировка ключа в формате MsgPack-массива ([part1, part2, ...])
- `char* key_end` (*const*) – часть закодированного ключа `key`
- `result` (`box_tuple_t**`) – аргумент вывода. Результатом будет кортеж или NULL, если в спейсе нет кортежей.

Результат -1 в случае ошибки (проверьте *box_error_last()*)

Результат 0, если выполнено

См. также *index_object.min()*

int box_index_max(uint32_t *space_id*, uint32_t *index_id*, const char **key*, const char **key_end*, *box_tuple_t* ***result*)

Возврат последнего (максимального) кортежа, который соответствует заданному ключу.

Параметры

- space_id (*uint32_t*) – идентификатор спейса
- index_id (*uint32_t*) – идентификатор индекса
- char* key (*const*) – кодировка ключа в формате MsgPack-массива ([part1, part2, ...])
- char* key_end (*const*) – часть закодированного ключа key
- result (box_tuple_t**) – аргумент вывода. Результатом будет кортеж или NULL, если в спейсе нет кортежей.

Результат -1 в случае ошибки (проверьте *box_error_last()*)

Результат 0, если выполнено

См. также *index_object.max()*

ssize_t box_index_count(uint32_t *space_id*, uint32_t *index_id*, int *type*, const char **key*, const char **key_end*)

Подсчет количества кортежей, которые соответствуют заданному ключу.

Параметры

- space_id (*uint32_t*) – идентификатор спейса
- index_id (*uint32_t*) – идентификатор индекса
- type (*int*) – *iterator_type*
- char* key (*const*) – кодировка ключа в формате MsgPack-массива ([part1, part2, ...])
- char* key_end (*const*) – часть закодированного ключа key

Результат -1 в случае ошибки (проверьте *box_error_last()*)

Результат 0, если выполнено

См. также *index_object.count()*

const *box_key_def_t* *box_index_key_def(uint32_t *space_id*, uint32_t *index_id*)

Возврат *определения ключа* для индекса

Возвращаемый объект действителен до следующей передачи управления.

Параметры

- space_id (*uint32_t*) – идентификатор спейса
- index_id (*uint32_t*) – идентификатор индекса

Результат определение ключа, если выполнено

Результат NULL в случае ошибки

См. также ***box_tuple_compare()***, *box_tuple_format_new()*

5.5.7 Модуль *latch*

box_latch_t
: Блокировка среды кооперативной многозадачности

box_latch_t *box_latch_new(void)
: Выделение и инициализация новой защелки.

  Результат выделенная защелка

  Тип результата box_latch_t *

void box_latch_delete(*box_latch_t *latch*)
: Удаление и освобождение защелки.

  Параметры

  - latch (box_latch_t*) – удаляемая защелка

void box_latch_lock(*box_latch_t *latch*)
: Применение защелки. Бесконечно ожидает момента, когда текущий файбер может получить доступ к защелке.

  param box_latch_t* latch применяемая защелка

int box_latch_trylock(*box_latch_t *latch*)
: Попытка применить защелку. Возвращается незамедлительно, если защелка поставлена.

  Параметры

  - latch (box_latch_t*) – применяемая защелка

  Результат статус операции. 0 – успешно, 1 – защелка поставлена

  Тип результата целое число

void box_latch_unlock(*box_latch_t *latch*)
: Отмена защелки. Файбер, который вызывает данную функцию, должен иметь права на защелку.

  Параметры

  - latch (box_latch_t*) – отменяемая защелка

5.5.8 Модуль *lua/utils*

void *luaL_pushcdata(struct lua_State *L, uint32_t *ctypeid*)
: Занесение cdata заданного ctypeid в стек.

  CTypeID должен быть использован хотя бы один раз из FFI. Выделенная область памяти возвращается неинициализированной. Поддерживаются только числа и указатели.

  Параметры

  - L (*lua_State**) – Lua_State
  - ctypeid (*uint32_t*) – CTypeID из FFI для cdata

  Результат область памяти, ассоциированная с cdata

  См. также *luaL_checkcdata()*

void *luaL_checkcdata(struct lua_State *L, int *idx*, uint32_t **ctypeid*)
: Проверка, является ли аргумент функции idx cdata.

  Параметры

- `L` (*lua_State**) – Lua_State
- `idx` (*int*) – индекс стека
- `ctypeid` (*uint32_t**) – аргумент вывода. CTypeID из FFI для возвращаемого cdata

Результат область памяти, ассоциированная с cdata

См. также *luaL_pushcdata()*

void `luaL_setcdatagc`(struct lua_State **L*, int *idx*)
Определение функции-финализатора для cdata.

Аналог вызова *ffi.gc(obj, function)*. Функция-финализатор должна быть на вершине стека.

Параметры

- `L` (*lua_State**) – Lua_State
- `idx` (*int*) – индекс стека

uint32_t `luaL_ctypeid`(struct lua_State **L*, const char **ctypename*)
Возврат CTypeID (FFI) заданного типа CDATA.

Параметры

- `L` (*lua_State**) – Lua_State
- `char* ctypename` (*const*) – Имя типа в C в виде строки (например, «struct request» или «uint32_t»)

Результат CTypeID

См. также *luaL_pushcdata()*, *luaL_checkcdata()*

int `luaL_cdef`(struct lua_State **L*, const char **ctypename*)
Объявление символов для FFI.

Параметры

- `L` (*lua_State**) – Lua_State
- `char* ctypename` (*const*) – C-определения (например, «struct stat»)

Результат 0, если выполнено

Результат `LUA_ERRRUN`, `LUA_ERRMEM` или `LUA_ERRERR`, в противном случае.

См. также `ffi.cdef(def)`

void `luaL_pushuint64`(struct lua_State **L*, uint64_t *val*)
Принудительная передача uint64_t в стек.

Параметры

- `L` (*lua_State**) – Lua_State
- `val` (*uint64_t*) – передаваемое значение

void `luaL_pushint64`(struct lua_State **L*, int64_t *val*)
Принудительная передача int64_t в стек.

Параметры

- `L` (*lua_State**) – Lua_State
- `val` (*int64_t*) – передаваемое значение

uint64_t `luaL_checkuint64`(struct lua_State *L, int *idx*)
Проверка, является ли аргумент idx uint64 или конвертируемой строкой, и возврат этого числа.

выбрасывает ошибку, если аргумент нельзя конвертировать

uint64_t `luaL_checkint64`(struct lua_State *L, int *idx*)
Проверка, является ли аргумент idx int64 или конвертируемой строкой, и возврат этого числа.

выбрасывает ошибку, если аргумент нельзя конвертировать

uint64_t `luaL_touint64`(struct lua_State *L, int *idx*)
Проверка, является ли аргумент idx uint64 или конвертируемой строкой, и возврат этого числа.

Результат конвертированное число или 0, если аргумент нельзя конвертировать

int64_t `luaL_toint64`(struct lua_State *L, int *idx*)
Проверка, является ли аргумент idx int64 или конвертируемой строкой, и возврат этого числа.

Результат конвертированное число или 0, если аргумент нельзя конвертировать

void `luaT_pushtuple`(struct lua_State *L, *box_tuple_t* *tuple*)
Принудительная передача кортежа в стек.

Параметры

- `L` (*lua_State**) – Lua_State

выбрасывает ошибка при нехватке памяти

См. также *luaT_istuple*

box_tuple_t *`luaT_istuple`(struct lua_State *L, int *idx*)
Проверка, является ли `idx` кортежем.

Параметры

- `L` (*lua_State**) – Lua_State
- `idx` (*int*) – индекс стека

Результат не NULL, если `idx` – это кортеж

Результат NULL, если `idx` – это не кортеж

int `luaT_error`(lua_State *L)
Повторение последней ошибки в Tarantool'е в виде Lua-объекта.

См. также **lua_error()**, *box_error_last()*.

int `luaT_cpcall`(lua_State *L, lua_CFunction *func*, void *ud*)
Аналог lua_cpcall(), но с соответствующей поддержкой ошибок Tarantool'а.

lua_State *`luaT_state`(void)
Получение глобального состояния Lua, используемого Tarantool'ом.

5.5.9 Модуль *say* (запись в журнал)

`enum say_level`

`enumerator S_FATAL`
не используйте непосредственно данное значение

`enumerator S_SYSERROR`

`enumerator S_ERROR`

enumerator S_CRIT

enumerator S_WARN

enumerator S_INFO

enumerator S_VERBOSE

enumerator S_DEBUG

say(level, format, ...)
Форматирование и запись сообщения в файл журнала Tarantool'а.

Параметры

- level (*int*) – *log level*
- char* format (*const*) – строка в формате типа printf()
- ... – аргументы формата

См. также *printf(3)*, *say_level*

say_error(format, ...)
say_crit(format, ...)
say_warn(format, ...)
say_info(format, ...)
say_verbose(format, ...)
say_debug(format, ...)
say_syserror(format, ...)
Форматирование и запись сообщения в файл журнала Tarantool'а.

Параметры

- char* format (*const*) – строка в формате типа printf()
- ... – аргументы формата

См. также *printf(3)*, *say_level*

Пример:

```
say_info("Some useful information: %s", status);
```

5.5.10 Модуль *schema*

enum SCHEMA

enumerator BOX_SYSTEM_ID_MIN
Начало выделенного диапазона системных спейсов.

enumerator BOX_SCHEMA_ID
Идентификатор спейса _schema.

enumerator BOX_SPACE_ID
Идентификатор спейса _space.

enumerator BOX_VSPACE_ID
Идентификатор виртуального спейса _vspace.

enumerator BOX_INDEX_ID
Идентификатор спейса _index.

enumerator BOX_VINDEX_ID
: Идентификатор виртуального спейса _vindex.

enumerator BOX_FUNC_ID
: Идентификатор спейса _func.

enumerator BOX_VFUNC_ID
: Идентификатор виртуального спейса _vfunc.

enumerator BOX_USER_ID
: Идентификатор спейса _user.

enumerator BOX_VUSER_ID
: Идентификатор виртуального спейса _vuser.

enumerator BOX_PRIV_ID
: Идентификатор спейса _priv.

enumerator BOX_VPRIV_ID
: Идентификатор виртуального спейса _vpriv.

enumerator BOX_CLUSTER_ID
: Идентификатор спейса _cluster.

enumerator BOX_TRIGGER_ID
: Идентификатор спейса _trigger.

enumerator BOX_TRUNCATE_ID
: Идентификатор спейса _truncate.

enumerator BOX_SYSTEM_ID_MAX
: Окончание выделенного диапазона системных спейсов.

enumerator BOX_ID_NIL
: Нулевое значение NULL возвращается в случае ошибки.

5.5.11 Модуль *trivia/config*

API_EXPORT
: Внешний модификатор для всех доступных функций.

PACKAGE_VERSION_MAJOR
: Мажорная версия пакета – 1 в 2.0.5.

PACKAGE_VERSION_MINOR
: Минорная версия пакета – 0 в 2.0.5.

PACKAGE_VERSION_PATCH
: Патч-версия пакета – 5 в 2.0.5

PACKAGE_VERSION
: Строка с идентификатором версии: мажорная-минорная-патч-коммит-идентификатор, например, 2.0.5-75-gdd8e14ffb.

SYSCONF_DIR
: Директория для системной конфигурации (например, `/etc`)

INSTALL_PREFIX
: Префикс установки (например, `/usr`)

BUILD_TYPE
: Тип сборки, например, отладочная сборка или релиз.

`BUILD_INFO`
: Подпись типа сборки CMake, например, `Linux-x86_64-Debug`

`BUILD_OPTIONS`
: Командная строка для запуска CMake.

`COMPILER_INFO`
: Пути к компиляторам C и CXX.

`TARANTOOL_C_FLAGS`
: Флаги компиляции C, используемые для сборки Tarantool'а.

`TARANTOOL_CXX_FLAGS`
: Флаги компиляции CXX, используемые для сборки Tarantool'а.

`MODULE_LIBDIR`
: Путь для установки файлов модуля `*.lua`.

`MODULE_LUADIR`
: Путь для установки файлов модуля `*.so`/`*.dylib`

`MODULE_INCLUDEDIR`
: Путь к Lua (директория, где хранится этот файл).

`MODULE_LUAPATH`
: Постоянная, добавляемая к `package.path` в Lua для поиска файлов модуля `*.lua`.

`MODULE_LIBPATH`
: Постоянная, добавляемая к `package.cpath` в Lua для поиска файлов модуля `*.so`.

5.5.12 Модуль *tuple*

`box_tuple_format_t`

box_tuple_format_t `*box_tuple_format_default(void)`
: Формат кортежа.

  Каждому кортежу соответствует определенный формат (класс). По умолчанию, используется формат для создания кортежей, не привязанных к определенному спейсу.

`box_tuple_t`
: Кортеж

box_tuple_t `*box_tuple_new(`*box_tuple_format_t* `*format,` const char `*tuple,` const char `*tuple_end)`
: Выделение и инициализация нового кортежа из сырых данных MsgPack-массива.

  Параметры

  - `format` (`box_tuple_format_t*`) – формат кортежа. Используйте *box_tuple_format_default()* для создания кортежа независимо от спейса.
  - `char* tuple` (*const*) – данные кортежа в формате MsgPack-массива ([field1, field2, ...])
  - `char* tuple_end` (*const*) – конец данных `data`

  Результат NULL при нехватке памяти

  Результат в остальных случаях кортеж

  См. также *box.tuple.new()*

> **Предупреждение:** При работе с кортежами в обязанности разработчика входит выделение достаточного места, уделяя особое внимание записи данных с помощью таких msgpuck-функций, как `mp_encode_array()`.

int `box_tuple_ref`(*box_tuple_t *tuple*)
Увеличение значения счетчика количества ссылок на кортеж.

Для кортежей подсчитываются ссылки. Все функции, которые возвращают кортежи, обеспечивают внутренний подсчет ссылок для последнего возвращенного кортежа до следующего вызова API-функции, которая передает управление или возвращает другой кортеж.

Следует увеличивать значение счетчика количества ссылок перед длительной обработкой кортежей в коде. Сборщик мусора в Lua не будет удалять кортежи с ссылками, даже если другой файбер удалит их из спейса. После обработки уменьшите значение счетчика количества ссылок с помощью *box_tuple_unref()*, иначе кортеж будет допускать утечку.

Параметры

- `tuple` (`box_tuple_t*`) – кортеж

Результат -1 в случае ошибки

Результат 0 в остальных случаях

См. также *box_tuple_unref()*

void `box_tuple_unref`(*box_tuple_t *tuple*)
Увеличение значения счетчика количества ссылок на кортеж.

Параметры

- `tuple` (`box_tuple_t*`) – кортеж

Результат -1 в случае ошибки

Результат 0 в остальных случаях

См. также *box_tuple_ref()*

uint32_t `box_tuple_field_count`(const *box_tuple_t *tuple*)
Возврат количества полей в кортеже (размер MsgPack-массива).

Параметры

- `tuple` (`box_tuple_t*`) – кортеж

size_t `box_tuple_bsize`(const *box_tuple_t *tuple*)
Возврат количества байтов, используемых для хранения внутренних данных кортежа (MsgPack-массив).

Параметры

- `tuple` (`box_tuple_t*`) – кортеж

ssize_t `box_tuple_to_buf`(const *box_tuple_t *tuple*, char **buf*, size_t *size*)
Передача сырых MsgPack-данных в буфер памяти `buf` размера `size`.

Хранение полей кортежа в буфере памяти.

При успешном выполнении функция возвращает количество записанных байтов. Если размер буфера недостаточный, возвращается количество байтов, которое было бы записано, если бы было достаточно места.

Результат -1 в случае ошибки

Результат количество записанных байтов при успешном выполнении.

box_tuple_format_t *box_tuple_format(const *box_tuple_t* *tuple)
Возврат взаимосвязанного формата.

Параметры

- tuple (box_tuple_t*) – кортеж

Результат формат кортежа

const char *box_tuple_field(const *box_tuple_t* *tuple, uint32_t *field_id*)
Возврат поля кортежа в MsgPack-формате. Результатом будет указатель на сырые данные в формате MessagePack, которые можно расшифровать с помощью функций mp_decode. Пример можно увидеть в программе практикума *read.c*.

Буфер действует до следующего вызова функции box_tuple_*.

Параметры

- tuple (box_tuple_t*) – кортеж
- field_id (*uint32_t*) – индекс с основанием 0 в MsgPack-массиве.

Результат NULL, если i >= *box_tuple_field_count()*

Результат в остальных случаях msgpack

enum field_type

enumerator FIELD_TYPE_ANY

enumerator FIELD_TYPE_UNSIGNED

enumerator FIELD_TYPE_STRING

enumerator FIELD_TYPE_ARRAY

enumerator FIELD_TYPE_NUMBER

enumerator FIELD_TYPE_INTEGER

enumerator FIELD_TYPE_SCALAR

enumerator field_type_MAX

Допустимые типы данных для полей кортежа.

Нельзя использовать макросы STRS/ENUM для типов, поскольку есть несоответствие между именем enum (STRING) и литералом имени типа («STR»). STR уже используется в качестве типа в Objective-C.

typedef struct key_def box_key_def_t
Определение ключа

box_key_def_t *box_key_def_new(uint32_t *fields, uint32_t *types, uint32_t *part_count*)
Создание определения ключа с полям ключа с переданными типами по переданным позициям.

Можно использовать для создания формата кортежа и/или сопоставления кортежей.

Параметры

- fields (*uint32_t**) – массив с идентификаторами поля ключа
- types (*uint32_t*) – массив с *типами поля* ключа
- part_count (*uint32_t*) – количество полей ключа

Результат определение ключа, если выполнено

Результат NULL в случае ошибки

void box_key_def_delete(*box_key_def_t *key_def*)
Удаление определения ключа

Параметры

- key_def (box_key_def_t*) – удаляемое определение ключа

box_tuple_format_t *box_tuple_format_new(struct key_def *keys*, uint16_t *key_count*)
Возврат нового формата кортежа на основании переданных определений ключа

Параметры

- keys (*key_def*) – массив ключей, определенный для формата
- key_count (*uint16_t*) – количество ключей

Результат новый формат кортежа, если выполнено

Результат NULL в случае ошибки

void box_tuple_format_ref(*box_tuple_format_t *format*)
Увеличение значения подсчета ссылок на формат кортежа

Параметры

- tuple_format (box_tuple_format_t) – формат кортежа для ссылок

void box_tuple_format_unref(*box_tuple_format_t *format*)
Уменьшение значения подсчета ссылок на формат кортежа

Параметры

- tuple_format (box_tuple_format_t) – формат кортежа для уменьшения

int box_tuple_compare(const *box_tuple_t *tuple_a*, const *box_tuple_t *tuple_b*, const *box_key_def_t *key_def*)
Сопоставление кортежей, используя определение ключа

Параметры

- box_tuple_t* tuple_a (*const*) – первый кортеж
- box_tuple_t* tuple_b (*const*) – второй кортеж
- box_key_def_t* key_def (*const*) – определение ключа

Результат 0, если key_fields(tuple_a) == key_fields(tuple_b)

Результат <0, если key_fields(tuple_a) < key_fields(tuple_b)

Результат >0, если key_fields(tuple_a) > key_fields(tuple_b)

См. также enum *field_type*

int box_tuple_compare_with_key(const box_tuple_t *tuple, const char *key, const box_key_def_t *key_def)
Сопоставление кортежа с ключом, используя определение ключа

Параметры

- box_tuple_t* tuple (*const*) – кортеж
- char* key (*const*) – ключ с заголовком MessagePack-массива
- box_key_def_t* key_def (*const*) – определение ключа

Результат 0, если key_fields(tuple) == parts(key)

Результат <0, если `key_fields(tuple)` < `parts(key)`

Результат >0, если `key_fields(tuple)` > `parts(key)`

См. также enum *field_type*

`box_tuple_iterator_t`
Итератор кортежей

box_tuple_iterator_t *`box_tuple_iterator`(*box_tuple_t *tuple*)
Выделение и инициализация нового итератора кортежей. Итератор кортежей позволяет проводить итерацию по полям на корневом уровне MsgPack-массива.

Пример:

```
box_tuple_iterator_t* it = box_tuple_iterator(tuple);
if (it == NULL) {
    // обработка ошибок с помощью box_error_last()
}
const char* field;
while (field = box_tuple_next(it)) {
    // обработка сырых MsgPack-данных
}

// перемотка итератора на начальное положение
box_tuple_rewind(it)
assert(box_tuple_position(it) == 0);

// перемотка на три поля
field = box_tuple_seek(it, 3);
assert(box_tuple_position(it) == 4);

box_iterator_free(it);
```

void `box_tuple_iterator_free`(*box_tuple_iterator_t* *it*)
Удаление и освобождение итератора кортежей

uint32_t `box_tuple_position`(*box_tuple_iterator_t* *it*)
Возврат следующего положения с основанием 0 в итераторе. То есть функция возвращает идентификатор поля, который вернется при следующем вызове *box_tuple_next()*. Возвращается значение 0 после инициализации или перемотки и *box_tuple_field_count()* по окончании итерации.

Параметры

- `it` (`box_tuple_iterator_t*`) – итератор кортежей

Результат положение

void `box_tuple_rewind`(*box_tuple_iterator_t* *it*)
Перемотка итератора в начальное положение.

Параметры

- `it` (`box_tuple_iterator_t*`) – итератор кортежей

После: `box_tuple_position(it) == 0`

const char *`box_tuple_seek`(*box_tuple_iterator_t* *it*, uint32_t *field_no*)
Поиск итератора кортежей.

Результатом будет указатель на сырые MessagePack-данные, которые можно расшифровать с помощью функций `mp_decode`. Пример можно увидеть в программе практикума *read.c*. Возвра-

щаемый буфер действует до следующего вызова API `box_tuple_*` . Запрашиваемый номер поля `field_no` возвращается при следующем вызове `box_tuple_next(it)`.

Параметры

- `it` (`box_tuple_iterator_t*`) – итератор кортежей
- `field_no` (*uint32_t*) – номер поля – положение с основанием 0 в MsgPack-массиве

После:

- `box_tuple_position(it) == field_not`, если возвращается не NULL.
- `box_tuple_position(it) == box_tuple_field_count(tuple)`, если возвращается NULL.

const char *`box_tuple_next`(*box_tuple_iterator_t *it*)
Возврат следующего поля кортежа из итератора кортежей.

Результатом будет указатель на сырые MessagePack-данные, которые можно расшифровать с помощью функций mp_decode. Пример можно увидеть в программе практикума *read.c*. Возвращаемый буфер действует до следующего вызова API `box_tuple_*`.

Параметры

- `it` (`box_tuple_iterator_t*`) – итератор кортежей

Результат NULL, если полей больше нет

Результат в остальных случаях MsgPack

Ранее: *box_tuple_position()* – это идентификатор с основанием 0 возвращаемого поля.

После: `box_tuple_position(it) == box_tuple_field_count(tuple)`, если возвращается NULL.

box_tuple_t *`box_tuple_update`(const *box_tuple_t *tuple*, const char *expr*, const char *expr_end*)

box_tuple_t *`box_tuple_upsert`(const *box_tuple_t *tuple*, const char *expr*, const char *expr_end*)

5.5.13 Модуль *txn*

bool `box_txn`(void)
Возврат true (правда), если есть активная транзакция.

int `box_txn_begin`(void)
Начало транзакции в текущем файбере.

Транзакция привязана к вызывающему файберу, поэтому в одном файбере может быть только одна активная транзакция. См. также *box.begin()*.

Результат 0, если выполнено

Результат -1 в случае ошибки. Возможно, транзакция уже была запущена.

int `box_txn_commit`(void)
Коммит текущей транзакции. См. также *box.commit()*.

Результат 0, если выполнено

Результат -1 в случае ошибки. Возможен отказ записи на диск

void `box_txn_rollback`(void)
Откат текущей транзакции. См. также *box.rollback()*.

box_txn_savepoint_t * `savepoint`(void)
Возврат дескриптора контрольной точки.

void box_txn_rollback_to_savepoint(box_txn_savepoint_t *savepoint)
: Откат текущей транзакции до указанной контрольной точки.

void *box_txn_alloc(size_t *size*)
: Выделение памяти в пул памяти txn.

  Память автоматически освобождается при коммите или откате транзакции.

  Результат NULL при нехватке памяти

5.6 Детали реализации

5.6.1 Binary protocol

The binary protocol in Tarantool is a binary request/response protocol.

Система обозначений в схематическом представлении

```
0    X
+----+
|    | - X + 1 байт
+----+
 TYPE - тип MsgPack-значения (если это MsgPack-объект)

+====+
|    | - MsgPack-объект изменяемого размера
+====+
 TYPE - тип MsgPack-значения

+~~~~+
|    | - Массив или ассоциативный массив в формате MsgPack изменяемого размера
+~~~~+
 TYPE - тип MsgPack-значения
```

Типы MsgPack-данных:

- **MP_INT** - целое число
- **MP_MAP** - ассоциативный массив
- **MP_ARR** - массив
- **MP_STRING** - строка
- **MP_FIXSTR** - строка фиксированной длины
- **MP_OBJECT** - любой MsgPack-объект
- **MP_BIN** - бинарный формат MsgPack

Пакет приветствия

```
ПРИВЕТСТВИЕ TARANTOOL'A:

 0                                  63
 +-----------------------------------+
```

(continues on next page)

(продолжение с предыдущей страницы)

```
 |                                        |
 | Приветствие Tarantool'a (версия сервера)  |
 |               64 байта                 |
 +----------------------+-----------------+
 |                      |                 |
 | СОЛЬ в кодировке BASE64 |      NULL       |
 |      44 байта        |                 |
 +----------------------+-----------------+
 64                   107               127
```

Экземпляр сервера начинает диалог с отправки клиенту текста приветствия фиксированного размера (128 байтов). Приветствие всегда содержит две 64-байтные строки текста в формате ASCII, каждая строка заканчивается символом разрыва строки (\n). Первая строка описывает версию экземпляра и тип протокола. Вторая строка содержит случайную строку в кодировке base64 размером до 44 байтов для использования в пакете аутентификации и заканчивается на пробелы (до 23).

Унифицированная структура пакета

После того, как приветствие прочитано, протокол становится простым запросно-ответным протоколом и предоставляет полный доступ к функциям Tarantool'a, включая:

- мультиплексирование запросов, т.е. возможность асинхронной отправки множества запросов по одному соединению;
- формат ответа, который поддерживает запись в режиме без копирования (zero-copy).

Для структуризации и кодирования данных протокол использует формат данных msgpack.

The protocol uses maps that contain some integer constants as keys. These constants are defined in src/box/iproto_constants.h. We list common constants here:

```
-- user keys
<iproto_sync>          ::= 0x01
<iproto_schema_id>     ::= 0x05  /* also known as schema_version */
<iproto_space_id>      ::= 0x10
<iproto_index_id>      ::= 0x11
<iproto_limit>         ::= 0x12
<iproto_offset>        ::= 0x13
<iproto_iterator>      ::= 0x14
<iproto_key>           ::= 0x20
<iproto_tuple>         ::= 0x21
<iproto_function_name> ::= 0x22
<iproto_username>      ::= 0x23
<iproto_expr>          ::= 0x27 /* also known as expression */
<iproto_ops>           ::= 0x28
<iproto_data>          ::= 0x30
<iproto_error>         ::= 0x31
<iproto_sql_text>      ::= 0x40
<iproto_sql_bind>      ::= 0x41
<iproto_sql_info>      ::= 0x42
```

```
-- -- Value for <code> key in request can be:
-- User command codes
<iproto_select>         ::= 0x01
<iproto_insert>         ::= 0x02
<iproto_replace>        ::= 0x03
```

(continues on next page)

(продолжение с предыдущей страницы)

```
<iproto_update>         ::= 0x04
<iproto_delete>         ::= 0x05
<iproto_call_16>        ::= 0x06 /* as used in version 1.6 */
<iproto_auth>           ::= 0x07
<iproto_eval>           ::= 0x08
<iproto_upsert>         ::= 0x09
<iproto_call>           ::= 0x0a
<iproto_execute>        ::= 0x0b
<iproto_nop>            ::= 0x0c
<iproto_type_stat_max>  ::= 0x0d
-- Admin command codes
-- (including codes for replica-set initialization and master election)
<iproto_ping>         ::= 0x40
<iproto_join>         ::= 0x41 /* i.e. replication join */
<iproto_subscribe>    ::= 0x42
<iproto_request_vote> ::= 0x43

-- -- Value for <code> key in response can be:
<iproto_ok>           ::= 0x00
<iproto_type_error>   ::= 0x8XXX /* where XXX is a value in errcode.h */
```

И заголовок `<header>` и тело сообщения `<body>` представляют собой ассоциативные массивы в формате msgpack:

```
Запрос / ответ:

0        5
+--------+ +============+ +====================================+
| BODY + | |            | |                                    |
| HEADER | |   HEADER   | |               BODY                 |
|  SIZE  | |            | |                                    |
+--------+ +============+ +====================================+
  MP_INT       MP_MAP                    MP_MAP
```

```
УНИФИЦИРОВАННЫЙ ЗАГОЛОВОК:

+================+================+=====================+
|                |                |                     |
|   0x00: CODE   |   0x01: SYNC   |    0x05: SCHEMA_ID  |
| MP_INT: MP_INT | MP_INT: MP_INT |  MP_INT: MP_INT     |
|                |                |                     |
+================+================+=====================+
                         MP_MAP
```

They only differ in the allowed set of keys and values. The key defines the type of value that follows. In a request, the body map can be absent. Responses will contain it anyway even if it is a `PING`. `schema_id` may be absent in the request's header, meaning that there will be no version checking, but it must be present in the response. If `schema_id` is sent in the header, then it will be checked.

Аутентификация

Когда клиент подключается к экземпляру сервера, экземпляр отвечает 128-байтным текстовым сообщением приветствия. Часть приветствия представляет собой закодированное в формате base-64 значение соль для сессии (случайная строка), которое можно использовать для аутентификации. Длина расшифрованного значения соль (44 байта) выходит за пределы сообщения для аутентификации (первые

20 байтов). Остаток предназначается для будущих схем аутентификации.

```
ПОДГОТОВКА КОДИРОВАНИЯ:

    LEN(ENCODED_SALT) = 44;
    LEN(SCRAMBLE)     = 20;

подготовить кодирование 'chap-sha1':

    salt = base64_decode(encoded_salt);
    step_1 = sha1(password);
    step_2 = sha1(step_1);
    step_3 = sha1(salt, step_2);
    scramble = xor(step_1, step_3);
    return scramble;

ТЕЛО СООБЩЕНИЯ АВТОРИЗАЦИИ: CODE = 0x07

+==================+====================================+
|                  |        +-------------+-----------+ |
|  (KEY)           | (TUPLE)|  len == 9   | len == 20 | |
|   0x23:USERNAME  |   0x21:| "chap-sha1" |  SCRAMBLE | |
| MP_INT:MP_STRING | MP_INT:|  MP_STRING  |  MP_BIN   | |
|                  |        +-------------+-----------+ |
|                  |                    MP_ARRAY        |
+==================+====================================+
                       MP_MAP
```

<key> содержит имя пользователя. <tuple> должен представлять собой массив из 2 полей: механизм аутентификации (в данный момент поддерживается только механизм «chap-sha1») и пароль, закодированный в соответствии с указанным механизмом. Аутентификация в Tarantool'е необязательна: если аутентификация не проводится, то пользователем в сессии будет „guest“. Экземпляр отвечает на пакет аутентификации стандартным ответом с 0 кортежей.

Запросы

- SELECT: CODE - 0x01 Поиск кортежей, соответствующих шаблону поиска

```
ТЕЛО СООБЩЕНИЯ ВЫБОРКИ SELECT:

+==================+==================+==================+
|                  |                  |                  |
|   0x10: SPACE_ID |   0x11: INDEX_ID |   0x12: LIMIT    |
| MP_INT: MP_INT   | MP_INT: MP_INT   | MP_INT: MP_INT   |
|                  |                  |                  |
+==================+==================+==================+
|                  |                  |                  |
|   0x13: OFFSET   |   0x14: ITERATOR |   0x20: KEY      |
| MP_INT: MP_INT   | MP_INT: MP_INT   | MP_INT: MP_ARRAY |
|                  |                  |                  |
+==================+==================+==================+
                        MP_MAP
```

- INSERT: CODE - 0x02 Вставка кортежа в спейс, если нет кортежей с такими же уникальными ключами. Если есть, выдать ошибку *duplicate key* (повторяющееся значение ключа).
- REPLACE: CODE - 0x03 Вставка кортежа в спейс или замена существующего кортежа.

```
ТЕЛО СООБЩЕНИЯ ВСТАВКИ/ЗАМЕНЫ INSERT/REPLACE:

+==================+==================+
|                  |                  |
|   0x10: SPACE_ID |   0x21: TUPLE    |
| MP_INT: MP_INT   | MP_INT: MP_ARRAY |
|                  |                  |
+==================+==================+
                MP_MAP
```

- UPDATE: CODE - 0x04 Обновление кортежа

```
ТЕЛО СООБЩЕНИЯ ОБНОВЛЕНИЯ UPDATE:

+==================+======================+
|                  |                      |
|   0x10: SPACE_ID |   0x11: INDEX_ID     |
| MP_INT: MP_INT   | MP_INT: MP_INT       |
|                  |                      |
+==================+======================+
|                  |          +~~~~~~~~~~+ |
|                  |          |          | |
|                  | (TUPLE)  |    OP    | |
|   0x20: KEY      |    0x21: |          | |
| MP_INT: MP_ARRAY |  MP_INT: +~~~~~~~~~~+ |
|                  |            MP_ARRAY  |
+==================+======================+
                MP_MAP
```

```
OP:
    Работает только для целочисленных полей:
    * Сложение    OP = '+' . space[key][field_no] += argument
    * Вычитание OP = '-' . space[key][field_no] -= argument
    * Побитовое И OP = '&' . space[key][field_no] &= argument
    * Исключающее ИЛИ OP = '^' . space[key][field_no] ^= argument
    * Побитовое ИЛИ  OP = '|' . space[key][field_no] |= аргумент
    Работает для любых полей:
    * Удаление      OP = '#'
      удалить поля <argument>, начиная
      с поля <field_no> в спейсе с ключом space[<key>]

0           2
+-----------+----------+----------+
|           |          |          |
|    OP     | FIELD_NO | ARGUMENT |
| MP_FIXSTR |  MP_INT  |  MP_INT  |
|           |          |          |
+-----------+----------+----------+
              MP_ARRAY
```

Note that `FIELD_NO` is one based (starts from 1) unlike indices numbers which are usually zero based.

```
    * Вставка      OP = '!'
      вставить <argument> до поля <field_no>
    * Присвоение      OP = '='
      присвоить <argument> полю <field_no>.
      увеличит кортеж, если <field_no> == <max_field_no> + 1
```

(continues on next page)

(продолжение с предыдущей страницы)

```
0           2
+-----------+==========+===========+
|           |          |           |
|    OP     | FIELD_NO | ARGUMENT  |
| MP_FIXSTR |  MP_INT  | MP_OBJECT |
|           |          |           |
+-----------+==========+===========+
              MP_ARRAY

    Работает со строковыми полями:
    * Разделение      OP = ':'
      взять строку из space[key][field_no] и
      заменить <offset> байтов из положения <position> на <argument>
```

```
0           2
+-----------+==========+==========+========+==========+
|           |          |          |        |          |
|    ':'    | FIELD_NO | POSITION | OFFSET | ARGUMENT |
| MP_FIXSTR |  MP_INT  |  MP_INT  | MP_INT |  MP_STR  |
|           |          |          |        |          |
+-----------+==========+==========+========+==========+
                         MP_ARRAY
```

Указать аргумент типа, который отличается от ожидаемого типа, будет ошибкой.

- DELETE: CODE - 0x05 Удаление кортежа

```
ТЕЛО СООБЩЕНИЯ УДАЛЕНИЯ DELETE:

+==================+==================+==================+
|                  |                  |                  |
|   0x10: SPACE_ID |   0x11: INDEX_ID |   0x20: KEY      |
| MP_INT: MP_INT   | MP_INT: MP_INT   | MP_INT: MP_ARRAY |
|                  |                  |                  |
+==================+==================+==================+
                         MP_MAP
```

- CALL_16: CODE - 0x06 Вызов хранимой функции с возвратом массива кортежей. Объявлен устаревшим; рекомендуется использовать CALL (0x0a).

```
ТЕЛО СООБЩЕНИЯ CALL_16:

+======================+==================+
|                      |                  |
|   0x22: FUNCTION_NAME |   0x21: TUPLE    |
| MP_INT: MP_STRING    | MP_INT: MP_ARRAY |
|                      |                  |
+======================+==================+
                   MP_MAP
```

- EVAL: CODE - 0x08 Оценка Lua-выражения

```
ТЕЛО СООБЩЕНИЯ EVAL:

+======================+==================+
```

(continues on next page)

(продолжение с предыдущей страницы)

```
|                       |                   |
|   0x27: EXPRESSION    |   0x21: TUPLE     |
| MP_INT: MP_STRING     | MP_INT: MP_ARRAY  |
|                       |                   |
+=======================+===================+
                    MP_MAP
```

- UPSERT: CODE - 0x09 Обновление кортежа, если он уже существует, попытка вставить кортеж. Всегда используйте первичный индекс.

```
ТЕЛО СООБЩЕНИЯ ОБНОВЛЕНИЯ И ВСТАВКИ UPSERT:

+==================+==================+==========================+
|                  |                  |             +~~~~~~~~~~+ |
|                  |                  |             |          | |
|   0x10: SPACE_ID |   0x21: TUPLE    |       (OPS) |    OP    | |
| MP_INT: MP_INT   | MP_INT: MP_ARRAY |       0x28: |          | |
|                  |                  |     MP_INT: +~~~~~~~~~~+ |
|                  |                  |               MP_ARRAY   |
+==================+==================+==========================+
                                 MP_MAP

Структура операции аналогична структуре операции обновления UPDATE.
   0           2
+-----------+==========+==========+
|           |          |          |
|    OP     | FIELD_NO | ARGUMENT |
| MP_FIXSTR |  MP_INT  |  MP_INT  |
|           |          |          |
+-----------+==========+==========+
              MP_ARRAY

Поддерживаются следующие операции:

'+' - прибавление значения к числовому полю. Если поле не является числовым, оно
      сначала изменяется на 0. Если поле отсутствует, операция
      пропускается. В случае переполнения ошибки также не будет, значение
      просто переносится в стиле языка C. Диапазон целых чисел в формате MsgPack:
      от -2^63 до 2^64-1
'-' - как в предыдущей операции, но значение вычитается
'=' - присвоение значения полю. Если поле отсутствует,
      операция пропускается.
'!' - вставка поля. Можно вставить поле, если при этом не будут созданы
      промежутки с нулевым значением nil между полями. Например, можно добавить поле между
      существующими полями или последнее поле в кортеже.
'#' - удаление поля. Если поле отсутствует, операция пропускается.
      Нельзя с помощью операции обновления update изменить компонент первичного
      ключа (это проверяется перед выполнением операции upsert).
```

- CALL: CODE - 0x0a Аналог CALL_16, но как и операция EVAL, CALL возвращает список неконвертированных значений

```
ТЕЛО СООБЩЕНИЯ CALL:

+======================+==================+
|                      |                  |
```

(continues on next page)

(продолжение с предыдущей страницы)

```
|   0x22: FUNCTION_NAME |   0x21: TUPLE    |
| MP_INT: MP_STRING     | MP_INT: MP_ARRAY |
|                       |                  |
+=======================+==================+
                    MP_MAP
```

Структура пакета ответа

Здесь мы продемонстрируем пакеты полностью:

```
OK:    LEN + HEADER + BODY

0      5                                          OPTIONAL
+------++================+================++====================+
|      ||                |                ||                    |
| BODY ||   0x00: 0x00   |   0x01: SYNC   ||   0x30: DATA      |
|HEADER|| MP_INT: MP_INT | MP_INT: MP_INT || MP_INT: MP_OBJECT |
| SIZE ||                |                ||                    |
+------++================+================++====================+
 MP_INT                MP_MAP                      MP_MAP
```

Предполагается, что набор кортежей в ответе `<data>` будет представлять собой msgpack-массив кортежей, поскольку команда EVAL возвращается произвольный MsgPack-массив *MP_ARRAY* с произвольными MsgPack-значениями.

```
ОШИБКА: LEN + HEADER + BODY

0      5
+------++================+================++====================+
|      ||                |                ||                    |
| BODY ||   0x00: 0x8XXX |   0x01: SYNC   ||   0x31: ERROR     |
|HEADER|| MP_INT: MP_INT | MP_INT: MP_INT || MP_INT: MP_STRING |
| SIZE ||                |                ||                    |
+------++================+================++====================+
 MP_INT                MP_MAP                      MP_MAP

Где 0xXXX -- это код ошибки ERRCODE.
```

Сообщение об ошибке будет включено в ответ только в случае ошибки; предполагается, что значение `<error>` будет msgpack-строкой.

Convenience macros which define hexadecimal constants for return codes can be found in src/box/errcode.h

Структура пакета при репликации

```
-- replication keys
<server_id>    ::= 0x02
<lsn>          ::= 0x03
<timestamp>    ::= 0x04
<server_uuid>  ::= 0x24
<cluster_uuid> ::= 0x25
<vclock>       ::= 0x26
```

```
-- коды для репликации
<join>         ::= 0x41
<subscribe>    ::= 0x42
```

```
JOIN:

Сначала необходимо отправить изначальный запрос JOIN
               HEADER                       BODY
+================+================++===================+
|                |                ||   SERVER_UUID     |
|   0x00: 0x41   |   0x01: SYNC   ||   0x24: UUID      |
| MP_INT: MP_INT | MP_INT: MP_INT || MP_INT: MP_STRING |
|                |                ||                   |
+================+================++===================+
               MP_MAP                      MP_MAP

Затем экземпляр, к которому мы подключаемся, отправит последний файл снимка SNAP,
просто создав количество запросов вставки INSERT (с дополнительным LSN и ServerID)
(не отвечайте). Затем он отправит MP_MAP из vclock и закроет сокет.

+================+================++============================+
|                |                ||        +~~~~~~~~~~~~~~~~~+ |
|                |                ||        |                 | |
|   0x00: 0x00   |   0x01: SYNC   ||   0x26:| SRV_ID: SRV_LSN | |
| MP_INT: MP_INT | MP_INT: MP_INT || MP_INT:| MP_INT: MP_INT  | |
|                |                ||        +~~~~~~~~~~~~~~~~~+ |
|                |                ||               MP_MAP       |
+================+================++============================+
               MP_MAP                      MP_MAP

SUBSCRIBE:

Далее необходимо отправить запрос SUBSCRIBE:

                             HEADER
+===================+===================+
|                   |                   |
|     0x00: 0x42    |    0x01: SYNC     |
|   MP_INT: MP_INT  |  MP_INT: MP_INT   |
|                   |                   |
+===================+===================+
|    SERVER_UUID    |    CLUSTER_UUID   |
|   0x24: UUID      |   0x25: UUID      |
| MP_INT: MP_STRING | MP_INT: MP_STRING |
|                   |                   |
+===================+===================+
                MP_MAP

      BODY
+================+
|                |
|   0x26: VCLOCK |
| MP_INT: MP_INT |
|                |
+================+
      MP_MAP
```

(continues on next page)

(продолжение с предыдущей страницы)

```
Затем следует обработать каждый запрос, который пришел от других мастеров.
Каждый запрос между мастерами получит дополнительный LSN и SERVER_ID.
```

XLOG / SNAP

Файлы форматов XLOG и SNAP выглядят практически одинаково. Заголовок выглядит следующим образом:

```
<type>\n                  SNAP\n или XLOG\n
<version>\n               в данный момент 0.13\n
Server: <server_uuid>\n   где UUID -- это 36-байтная строка
VClock: <vclock_map>\n    например, {1: 0}\n
\n
```

После файла заголовка идут кортежи с данными. Кортежи начинаются с маркера строки `0xd5ba0bab`, а после последнего кортежа может стоять маркер конца файла `0xd510aded`. Таким образом, между заголовком файла и маркером конца файла могут быть кортежи с данными в следующем виде:

```
0            3 4                                        17
+-------------+========+============+===========+=========+
|             |        |            |           |         |
| 0xd5ba0bab  | LENGTH | CRC32 PREV | CRC32 CUR | PADDING |
|             |        |            |           |         |
+-------------+========+============+===========+=========+
  MP_FIXEXT2    MP_INT     MP_INT       MP_INT      ---

+============+ +====================================+
|            | |                                    |
|   HEADER   | |                BODY                |
|            | |                                    |
+============+ +====================================+
    MP_MAP                    MP_MAP
```

См. пример в разделе *Файловые форматы*.

5.6.2 SQL protocol

Tarantool's SQL protocol regulates how to build SQL requests and parse responses using Tarantool's common binary protocol.

Special SQL keys:

```
<metadata>      ::= 0x32
<sql_text>      ::= 0x40
<sql_bind>      ::= 0x41
<sql_info>      ::= 0x42
```

Special SQL commands:

```
<execute> ::= 11
```

Request packet body

An SQL request has the type `EXECUTE=11`.

```
EXECUTE REQUEST BODY:
                          MAP
+====================+==============================+
|                    |                              |
|   0x40: SQL_TEXT   |     0x41: SQL_BIND           |
| MP_STR: SQL request | MP_ARRAY: array of parameters |
|                    |                              |
+====================+==============================+
```

- **SQL_TEXT** is a single non-empty SQL statement. For SQL syntax, see https://sqlite.org/lang.html
- **SQL_BIND** is an optional array of bindings (parameters). Each parameter value is a scalar: number, string, binary, null.

  A parameter can be *ordinal* or *named*. An ordinal parameter is encoded as a message pack scalar value (MP_UINT, INT, DOUBLE, FLOAT, STR, BIN, EXT, NIL). A named parameter is encoded as a map with one string key – its name. For bindings syntax, see https://sqlite.org/lang_expr.html#varparam

Examples:

- `[100, 'abc', NULL, -345.6] = MP_ARRAY[ MP_UINT, MP_STR, MP_NIL, MP_DOUBLE ]`
- `[1, 2, {'name': 300}] = MP_ARRAY[ MP_UINT, MP_UINT, MP_MAP{ MP_STR : MP_UINT } ]`

Response packet body

Body structure depends on the type of the SQL request.

If the SQL request is SELECT, the response contains:

- metadata for columns (metadata for a single column contains only the column's name and type) and
- result rows.

```
EXECUTE SELECT RESPONSE BODY:
                                MAP
+====================================+============================+
|                                    |                            |
|     0x32: METADATA                 |                            |
| MP_ARRAY: array of maps:           |                            |
|           +~~~~~~~~~~~~~~~~~~~~~~+ |                            |
|           | +~~~~~~~~~~~~~~~~~~+ | |     0x30: DATA             |
|           | |   0x00: FIELD_NAME | | | MP_ARRAY: array of tuples |
|           | | MP_STR: field name | | |                            |
|           | |   0x01: FIELD_TYPE | | |                            |
|           | | MP_STR: field type | | |                            |
|           | +~~~~~~~~~~~~~~~~~~+ | |                            |
|           |        MP_MAP        | |                            |
|           +~~~~~~~~~~~~~~~~~~~~~~+ |                            |
|                   MP_ARRAY         |                            |
|                                    |                            |
+====================================+============================+
```

Пример:

Request: `SELECT x, y FROM test_space;`

Response:

```
BODY = {
    METADATA = [
    { FIELD_NAME: 'X', FIELD_TYPE: 'TEXT'}, { FIELD_NAME: 'Y', FIELD_TYPE: 'INTEGER'},
    DATA = [ ['a', 1], ['c', 2], ['e', 5], ... ]
}
```

If the SQL request is not SELECT, the response body contains only SQL_INFO. Usually SQL_INFO is a map with only one key – SQL_INFO_ROW_COUNT (0) – which is the number of changed rows. For example, if the request is `INSERT INTO test VALUES (1), (2), (3)`, the response body contains an SQL_INFO map with SQL_INFO_ROW_COUNT = 3. SQL_INFO_ROW_COUNT can be 0 for statements that do not change rows, such as CREATE TABLE.

The SQL_INFO map may contain a second key – SQL_INFO_AUTO_INCREMENT_IDS (1) – which is the new primary-key value for an INSERT in a table defined with PRIMARY KEY AUTOINCREMENT. In this case the MP_MAP will have two keys, and one of the two keys will be 0x01: SQL_INFO_AUTO_INCREMENT_IDS, which is an MP_UINT number.

```
EXECUTE NOT-SELECT RESPONSE BODY:

+=========================================================+
|                                                         |
|   0x42: SQL_INFO                                        |
| MP_MAP: usually 1 key   +~~~~~~~~~~~~~~~~~~~~~~~~~~~~+  |
|                         |                            |  |
|                         |    0x00: SQL_INFO_ROW_COUNT |  |
|                         | MP_UINT: changed row count  |  |
|                         |                            |  |
|                         +~~~~~~~~~~~~~~~~~~~~~~~~~~~~+  |
|                                                         |
+=========================================================+
```

5.6.3 File formats

Персистентность данных и формат WAL-файла

Чтобы поддерживать персистентность данных, Tarantool записывает каждый запрос изменения данных (insert, update, delete, replace, upsert) в файл журнала упреждающей записи (WAL-файл) в директорию *wal_dir*. Новый WAL-файл создается для количества записей, определенного в параметре *rows_per_wal*, или для количества байтов, указанного в *wal_max_size* . Каждому запросу на изменение данных присваивается постоянно возрастающее 64-битное число, представляющее собой регистрационный номер в журнале (LSN). Название WAL-файла состоит из LSN первой записи в файле плюс расширение `.xlog`.

Apart from a log sequence number and the data change request (formatted as in *Tarantool's binary protocol*), each WAL record contains a header, some metadata, and then the data formatted according to msgpack rules. For example, this is what the WAL file looks like after the first INSERT request («s:insert({1})») for the sandbox database created in our *«Getting started» exercises*. On the left are the hexadecimal bytes that you would see with:

```
$ hexdump 00000000000000000000.xlog
```

а справа – комментарии.

```
Шестнадцатеричный дамп WAL-файла       Комментарий
--------------------       -------
58 4c 4f 47 0a             "XLOG\n"
30 2e 31 33 0a             "0.13\n" = version
53 65 72 76 65 72 3a 20    "Server: "
38 62 66 32 32 33 65 30 2d [Server UUID]\n
36 39 31 34 2d 34 62 35 35
2d 39 34 64 32 2d 64 32 62
36 64 30 39 62 30 31 39 36
0a
56 43 6c 6f 63 6b 3a 20    "Vclock: "
7b 7d                      "{}" = vclock value, initially blank
...                        (not shown = tuples for system spaces)
d5 ba 0b ab                Magic row marker always = 0xab0bbad5
19                         Length, not including length of header, = 25 bytes
00                           Record header: previous crc32
ce 8c 3e d6 70               Record header: current crc32
a7 cc 73 7f 00 00 66 39      Record header: padding
84                         msgpack code meaning "Map of 4 elements" follows
00 02                         element#1: tag=request type, value=0x02=IPROTO_INSERT
02 01                         element#2: tag=server id, value=0x01
03 04                         element#3: tag=lsn, value=0x04
04 cb 41 d4 e2 2f 62 fd d5 d4 element#4: tag=timestamp, value=an 8-byte "Float64"
82                         msgpack code meaning "map of 2 elements" follows
10 cd 02 00                   element#1: tag=space id, value=512, big byte first
21 91 01                      element#2: tag=tuple, value=1-element fixed array={1}
```

Tarantool обрабатывает запросы атомарно: изменение либо принимается и записывается в WAL-файл, или полностью исключается. Проясним, как этом работает, используя в качестве примера REPLACE-запрос:

1. Экземпляр сервера пытается найти оригинальный кортеж по первичному ключу. Если кортеж найден, ссылка на него сохраняется для дальнейшего использования.
2. Происходит проверка нового кортежа. Например, если в нем нет проиндексированного поля, или же тип проиндексированного поля не совпадает с типом в определении индекса, изменение прерывается.
3. Новый кортеж заменяет старый кортеж во всех существующих индексах.
4. A message is sent to the WAL writer running in a separate thread, requesting that the change be recorded in the WAL. The instance switches to work on the next request until the write is acknowledged.
5. При успешном выполнении на клиент отправляется подтверждение. В случае ошибки начинается процедура отката. Во время процедуры отката поток обработки транзакций откатывается все изменения в базу данных, которые произошли после первого невыполненного изменения, от последнего с первому, вплоть до первого невыполненного изменения. Все запросы, которые подверглись откату, прерываются с ошибкой `ER_WAL_IO`. Новые изменения не применяются во время отката. По окончании процедуры отката сервер повторно запускает конвейер обработки операций.

Одно из преимуществ описанного алгоритма заключается в том, что достигается полная обработка запроса по конвейеру даже для запросов с одинаковым значением первичного ключа. В результате производительность базы данных не падает, даже если все запросы относятся к одному ключу в одном спейсе.

Поток обработки транзакций взаимодействует с потоком записи в журнал упреждающей записи с помощью асинхронного (однако надежного) обмена сообщениями. Поток обработки транзакций, который не блокируется при задачах записи в журнал, продолжает быстро обрабатывать запрос даже при большом объеме дискового ввода-вывода. Ответ на запрос отправляется по готовности, даже если ранее на том же соединении были незавершенные запросы. В частности, на производительность выборки не влияет загрузка диска, даже если SELECT-запросы передаются вместе с запросами UPDATE и DELETE.

При записи в WAL можно применять различные режимы долговечности, что определяет конфигурационная переменная *wal_mode*. Можно полностью отключить журнал упреждающей записи, присвоив *wal_mode* значение *none*. Даже без журнала упреждающей записи возможно сделать персистентную копию всего набора данных с помощью запроса *box.snapshot()*.

Файл в формате .xlog всегда содержит изменения на основании первичного ключа. Даже если клиент запрашивает обновление или удаление по вторичному ключу, запись в файле в формате .xlog будет содержать первичный ключ.

Формат файла снимка

Формат файла снимка .snap практически такой же, что и формат WAL-файла .xlog. Тем не менее, заголовок снимка отличается: он содержит глобально уникальный идентификатор экземпляра и положения файла снимка в истории относительно более ранних файлов снимка. Кроме того, отличается содержание: .xlog-файл может содержать записи о любых запросах изменения данных (вставка, обновление, обновление и вставка и удаление), а .snap-файл может содержать лишь записи о вставках в спейсы memtx'а.

В первую очередь записи в .snap-файле упорядочены по идентификатору спейса. Таким образом, записи в системные спейсы – такие как `_schema`, `_space`, `_index`, `_func`, `_priv` и `_cluster` – будут находиться в начале .snap-файла до записей в другие спейсы, созданные пользователями.

Во вторую очередь записи в .snap-файле упорядочены по первичному ключу.

5.6.4 Процесс восстановления

Процесс восстановления начинается, когда box.cfg{} впервые используется после запуска экземпляра Tarantool-сервера.

Процесс восстановления должен восстановить базы данных на момент последнего отключения экземпляра. Для этого можно использовать последний файл снимка и любые WAL-файлы, которые были записаны после создания снимка. Ситуацию осложняет фактор того, что в Tarantool'е используются два движка – данные memtx'а должны быть реконструированы полностью из снимка и WAL-файлов, тогда как данные vinyl'а будут находиться на диске, но может потребоваться их обновление на время создания контрольной точки. (При создании снимка Tarantool передает движку vinyl команду создания контрольной точки, а операция создания снимка откатывается в случае какой-либо ошибки, поэтому контрольная точка vinyl'а будет настолько же актуальной, как и файл снимка.)

Шаг 1 Выполнить чтение конфигурационных параметров из запроса `box.cfg{}`. Параметры, которые могут повлиять на восстановление: *work_dir*, *wal_dir*, *memtx_dir*, *vinyl_dir* и *force_recovery*.

Шаг 2 Найти последний файл снимка. Использовать данные для реконструкции in-memory баз данных. Передать команду vinyl'у о восстановлении до последней контрольной точки.

На самом деле, есть два варианта реконструкции баз данных memtx'а в зависимости от того, выполняется ли стандартная процедура.

Если выполняется стандартная процедура (`force_recovery = false`), memtx может выполнить чтение данных из снимка с отключенными индексами. Сначала все кортежи считываются в память. Затем происходит массовая загрузка первичных ключей с учетом того, что данные уже отсортированы по первичному ключу в каждом спейсе.

Если выполняется нестандартная процедура принудительного восстановления (`force_recovery = true`), Tarantool проводит дополнительную проверку. Сначала индексы активны, и кортежи добавляются по одному. Это означает, что будут выявлены любые нарушения ограничений уникальности ключей, и все повторяющиеся значения пропускаются. Как правило, не будет нарушений ограничений или повторяющихся значений, поэтому такие проверки проводятся только в случае ошибки.

Шаг 3 Найти WAL-файл, который был создан во время создания файла снимка или позже. Выполнить чтение записей журнала до тех пор, пока LSN записи в журнале не будет больше LSN снимка или больше LSN контрольной точки в vinyl'е. Это и будет начальной точкой для процесса восстановления, которая соответствует текущему состоянию движков.

Шаг 4 Повторить записи журнала с начальной точки до конца WAL. Движок пропускает команду повторения, если данные старше контрольной точки движка.

Шаг 5 Повторно создать все вторичные индексы для движка memtx.

5.6.5 Server startup with replication

In addition to the recovery process described above, the server must take additional steps and precautions if *replication* is enabled.

Once again the startup procedure is initiated by the `box.cfg{}` request. One of the `box.cfg` parameters may be *replication* that specifies replication source(-s). We will refer to this replica, which is starting up due to `box.cfg`, as the «local» replica to distinguish it from the other replicas in a replica set, which we will refer to as «distant» replicas.

#1. If there is no snapshot .snap file and the "replication" parameter is empty: then the local replica assumes it is an unreplicated «standalone» instance, or is the first replica of a new replica set. It will generate new UUIDs for itself and for the replica set. The replica UUID is stored in the `_cluster` space; the replica set UUID is stored in the `_schema` space. Since a snapshot contains all the data in all the spaces, that means the local replica's snapshot will contain the replica UUID and the replica set UUID. Therefore, when the local replica restarts on later occasions, it will be able to recover these UUIDs when it reads the .snap file.

#2. If there is no snapshot .snap file and the "replication" parameter is not empty and the "_cluster" space contains no other replica UUIDs: then the local replica assumes it is not a standalone instance, but is not yet part of a replica set. It must now join the replica set. It will send its replica UUID to the first distant replica which is listed in `replication` and which will act as a master. This is called the «join request». When a distant replica receives a join request, it will send back:

(1) the distant replica's replica set UUID,

(2) the contents of the distant replica's .snap file. When the local replica receives this information, it puts the replica set UUID in its `_schema` space, puts the distant replica's UUID and connection information in its `_cluster` space, and makes a snapshot containing all the data sent by the distant replica. Then, if the local replica has data in its WAL .xlog files, it sends that data to the distant replica. The distant replica will receive this and update its own copy of the data, and add the local replica's UUID to its `_cluster` space.

#3. If there is no snapshot .snap file and the "replication" parameter is not empty and the "_cluster" space contains other replica UUIDs: then the local replica assumes it is not a standalone instance, and is already part of a replica set. It will send its replica UUID and replica set UUID to all the distant replicas which

are listed in `replication`. This is called the «on-connect handshake». When a distant replica receives an on-connect handshake:

(1) the distant replica compares its own copy of the replica set UUID to the one in the on-connect handshake. If there is no match, then the handshake fails and the local replica will display an error.

(2) the distant replica looks for a record of the connecting instance in its `_cluster` space. If there is none, then the handshake fails. Otherwise the handshake is successful. The distant replica will read any new information from its own .snap and .xlog files, and send the new requests to the local replica.

In the end ... the local replica knows what replica set it belongs to, the distant replica knows that the local replica is a member of the replica set, and both replicas have the same database contents.

#4. If there is a snapshot file and replication source is not empty: first the local replica goes through the recovery process described in the previous section, using its own .snap and .xlog files. Then it sends a «subscribe» request to all the other replicas of the replica set. The subscribe request contains the server vector clock. The vector clock has a collection of pairs „server id, lsn“ for every replica in the `_cluster` system space. Each distant replica, upon receiving a subscribe request, will read its .xlog files“ requests and send them to the local replica if (lsn of .xlog file request) is greater than (lsn of the vector clock in the subscribe request). After all the other replicas of the replica set have responded to the local replica's subscribe request, the replica startup is complete.

The following temporary limitations apply for versions 1.7 and 2.1:

- The URIs in the `replication` parameter should all be in the same order on all replicas. This is not mandatory but is an aid to consistency.
- The maximum number of entries in the `_cluster` space is 32. Tuples for out-of-date replicas are not automatically re-used, so if this 32-replica limit is reached, users may have to reorganize the `_cluster` space manually.

5.7 Interactive console

The «interactive console» is Tarantool's basic «command-line interface» for entering requests and seeing results. It is what users see when they start the server without an *instance file*, or start *tarantoolctl* with `enter`. It is often called the Lua console to distinguish it from the administrative console, but in fact it can handle both Lua and SQL input. The majority of examples in this manual show what users see with the interactive console, including the prompt (which can be «tarantool> «), the instruction (which can be a Lua request or an SQL statement), and the response (which can be a display in either YAML format or Lua format).

```
-- Typical interactive console example with Lua input and YAML output
tarantool> box.info().replication
---
- 1:
    id: 1
    uuid: a5d22f66-2d28-4a35-b78f-5bf73baf6c8a
    lsn: 0
...
```

The **input language** can be changed to SQL with `\set language sql` or changed to Lua (the default) with `\set language lua`.

The **delimiter** can be changed to any character with `set delimiter <character>`. The default is nothing, which means input does not need to end with a delimiter. But a common recommendation is to say `set delimiter ;` especially if input is SQL.

The **output format** can be changed to Lua with `\set output lua` or changed to YAML (the default) with `\set output yaml`.

Ordinarily. output from the console has YAML format. That means that there is a line for document-start `"---"`, and each item begins on a separate line starting with `"- "`, and each sub-item in a nested structure is indented, and there is a line for document-end `"..."`.

Optionally, output from the console can have Lua format. That means that there are no lines for document-start or document-end, and items are not on separate lines (they are only separated by commas), and each sub-item in a nested structure is placed inside «`{}`» braces. So, when input is a Lua object description, output will equal input.

YAML is good for readability. Lua is good for re-using results as requests. A third format, MsgPack, is good for database storage. Currently the default output format is YAML but it may be Lua in a future version, and it may be Lua if the last *set_default_output* call was `console.set_default_output('lua')`.

Type	Lua input	Lua output	YAML output	MsgPack storage
scalar	`1`	`1`	`--- - 1 ...`	`\x01`
scalar sequence	`1,2,3`	`1,2,3`	`--- - 1 - 2 - 3 ...`	`\x01 \x02 \x03`
2-element table	`{1,2}`	`{1,2}`	`--- - - 1 - - 2 ...`	`0x92 0x01 0x02`
map	`{key=1}`	`{key=1}`	`--- - key: 1 ...`	`\x81 \xa3 \x6b \x65 \x79 \x01`

5.8 Утилита *tarantoolctl*

`tarantoolctl` представляет собой утилиту для администрирования *экземпляров*, *файлов контрольной точки* и *модулей* в Tarantool'е. Утилита поставляется и устанавливается как часть дистрибутива Tarantool'а.

См. также примеры использования `tarantoolctl` в разделе *Администрирование серверной части*.

5.8.1 Формат команд

```
tarantoolctl COMMAND NAME [URI] [FILE] [OPTIONS..]
```

где:

- `COMMAND` – это одна из следующих команд, описанных ниже: `start`, `stop`, `status`, `restart`, `logrotate`, `check`, `enter`, `eval`, `connect`, `cat`, `play`, `rocks`.
- `NAME` – это имя *файла экземпляра* или *модуля*.
- `FILE` – это путь к какому-либо файлу (.lua, .xlog или .snap).
- `URI` – это URI некого экземпляра Tarantool'а.
- `OPTIONS` – это параметры, которые принимают команды `tarantoolctl`.

5.8.2 Команды для управления экземплярами Tarantool'а

`tarantoolctl start NAME` Запуск экземпляра Tarantool'а.

> Кроме того, данная команда задает значение переменной окружения TARANTOOLCTL = „true“ (правда), чтобы отметить, что экземпляр был запущен с помощью `tarantoolctl`.

Примечание: `tarantoolctl` работает для экземпляров, где не вызвана функция `box.cfg{}` или вызов `box.cfg{}` отложен.

Например, это можно использовать для управления экземплярами, которые получают конфигурацию из внешнего сервера. Для таких экземпляров `tarantoolctl start` goes to background when `box.cfg{}` is called, so it will wait until options for `box.cfg` are received. However this is not the case for daemon management systems like `systemd`, as they handle backgrounding on their side.

`tarantoolctl stop NAME` Остановка экземпляра Tarantool'a.

`tarantoolctl status NAME` Отображение статуса экземпляра (работает/остановлен). Если есть PID-файл и активный управляющий сокет, возвращается код `0`. В остальных случаях возвращается не `0`.

Сообщает о типичных проблемах стандартного вывода ошибок (например, PID-файл есть, а управляющий сокет отсутствует).

`tarantoolctl restart NAME` Остановка и запуск экземпляра Tarantool'a.

Кроме того, данная команда задает значение переменной окружения TARANTOOL_RESTARTED = „true“ (правда), чтобы отметить, что экземпляр был перезапущен с помощью `tarantoolctl`.

`tarantoolctl logrotate NAME` Ротация файлов журнала работающего Tarantool-экземпляра. Работает только в том случае, если в файле экземпляра задан параметр записи журнала в файл. Отправка записей в конвейер или системный журнал syslog не имеет значения в данном случае.

`tarantoolctl check NAME` Проверка файла экземпляра на ошибки синтаксиса.

`tarantoolctl enter NAME [--language=language]` Enter an instance's interactive Lua or SQL console.

Supported option:

- `--language=language` to set *interactive console* language. Can be either `Lua` or `SQL`.

`tarantoolctl eval NAME FILE` Выполнение локального Lua-файла на работающем экземпляре Tarantool'a.

`tarantoolctl connect URI` Подключение к экземпляру Tarantool'a по порту административной консоли. Поддерживаются TCP и Unix сокеты.

5.8.3 Команды для управления файлами контрольной точки

`tarantoolctl cat FILE.. [--space=space_no ..] [--show-system] [--from=from_lsn] [--to=to_lsn] [--replica`
Стандартный вывод содержимого .snap-файла или .xlog-файла.

`tarantoolctl play URI FILE.. [--space=space_no ..] [--show-system] [--from=from_lsn] [--to=to_lsn] [--r`
Передача содержимого .snap-файла или .xlog-файла на другой экземпляр Tarantool'a.

Поддерживаемые опции:

- `--space=space_no` для фильтрации вывода по номеру спейса. Можно передавать несколько раз.
- `--show-system` для отображения содержимого системных спейсов.
- `--from=from_lsn` для отображения операций, начиная с заданного LSN.
- `--to=to_lsn` для отображения операций, заканчивая заданным LSN.
- `--replica=replica_id` для фильтрации вывода по идентификатору реплики. Можно передавать несколько раз.

5.8.4 Команды для управления модулями Tarantool'а

`tarantoolctl rocks install NAME` Установка модуля в текущей директории.

`tarantoolctl rocks remove NAME` Удаление модуля.

`tarantoolctl rocks show NAME` Отображение информации об установленном модуле.

`tarantoolctl rocks search NAME` Поиск модулей по репозиторию.

`tarantoolctl rocks list` Вывод списка всех установленных модулей.

`tarantoolctl rocks pack {<rockspec> | <имя> [<версия>]}` Создание модуля путем компоновки исходных или бинарных файлов.

В качестве аргумента можно указать:

- файл в формате `.rockspec` для создания модуля, который содержит исходные файлы или
- имя установленного модуля (с версией, если их больше одной) для создания модуля, который содержит скомпилированные файлы.

`tarantoolctl rocks unpack {<rock_file> | <rockspec> | <имя> [версия]}` Распаковка содержимого модуля в новую директорию в текущей директории.

В качестве аргумента можно указать:

- исходные или бинарные файлы модуля,
- файлы `.rockspec` или
- имя модулей или файлов в формате `.rockspec` в удаленных репозиториях (с версией модуля, если их больше одной).

Поддерживаемые опции:

- `--server=имя_севрера` сначала проверить данный сервер, затем по списку.
- `--only-server=имя_сервера` проверить только данный сервер, остальные пропустить.

5.9 Рекомендации по Lua-синтаксису

В *функциях управления данными* Lua-синтаксис может различаться. Далее приводятся варианты таких различий на примере запросов `select()`. Аналогичные правила существуют и для остальных функций.

В каждом из приведенных примеров выполняются следующие действия: производится выборка по набору кортежей из спейса с именем „tester“, где значение поля, которое соответствует ключу в первичном индексе, равно 1. Также во всех примерах мы принимаем, что числовой идентификатор спейса „tester“ равен 512, но это верно только для нашей тестовой базы.

5.9.1 Способы ссылки на объект

Во-первых, есть три *способа ссылки на объект*:

```
-- #1 модуль.подмодуль.имя
tarantool> box.space.tester:select{1}
-- #2 заменить имя буквенной константой в квадратных скобках
tarantool> box.space['tester']:select{1}
-- #3 использовать переменную для всей ссылки на объект
```

(continues on next page)

(продолжение с предыдущей страницы)

```
tarantool> s = box.space.tester
tarantool> s:select{1}
```

Для примеров в документации, как правило, используется вариант синтаксиса №1, например «`box.space.`*`tester`*`:`». Но вы можете с тем же успехом пользоваться любым из трех описанных выше вариантов.

Также описания в руководстве используют синтаксис типа «`space_object:`» для ссылки на спейсы и «`index_object:`» для ссылки на индексы (например, `box.space.`*`tester`*`.index.`*`primary`*`:`).

5.9.2 Способы задания параметров

Затем есть семь *способов задания параметров*:

```
-- #1
tarantool> box.space.tester:select{1}
-- #2
tarantool> box.space.tester:select({1})
-- #3
tarantool> box.space.tester:select(1)
-- #4
tarantool> box.space.tester.select(box.space.tester,1)
-- #5
tarantool> box.space.tester:select({1},{iterator='EQ'})
-- #6
tarantool> variable = 1
tarantool> box.space.tester:select{variable}
-- #7
tarantool> variable = {1}
tarantool> box.space.tester:select(variable)
```

В Lua допускается пропуск круглых скобок `()` при вызове функции, если единственным аргументом является Lua-таблица, и иногда мы этим пользуемся в примерах. Вот почему `select{1}` аналогично `select({1})`. Литеральные значения, такие как `1` (скалярное значение) или `{1}` (значение Lua-таблицы), можно заменить именами переменных, как в примерах 6 и 7.

Хотя есть особые случаи, когда фигурные скобки можно опустить, рекомендуется использовать их, потому что они означают Lua-таблицу. В примерах и описаниях данного руководства применяется форма `{1}`. Однако это тоже вопрос предпочтений пользователя, и на практике применимы все варианты.

5.9.3 Правила именования объектов

Правила именования объектов базы данных не слишком ограничены: максимальная длина составляет 65000 байтов (не символов), допускается практически любой символ Юникода, включая пробелы, идеограммы и знаки пунктуации.

В таких случаях во избежание путаницы с операторами и разделителями в Lua ссылки на объекты должны иметь форму типа литерал в квадратных скобках (2) или форму переменной (3). Например:

```
tarantool> box.space['1*A']:select{1}
tarantool> s = box.space['1*A !@$%^&*()_+12345678901234567890']
tarantool> s:select{1}
```

Не разрешаются:

- символы, которые представляют собой неназначенные кодовые точки,
- разделители строки и абзаца,
- управляющие символы,
- символ замены (U+FFFD).

Не рекомендуются: символы, которые не отображаются.

Имена зависимы от регистра, поэтому „A“ и „a“ – это не одно и то же.

ГЛАВА 6

Практические задания

Эти практические задания предназначены для тех, кто хочет поглубже узнать про использование Tarantool'а.

Если вы еще не использовали Tarantool, пожалуйста, сначала ознакомьтесь с *Руководством для начинающих*.

6.1 Практические задания на Lua

Практические задания по использованию хранимых процедур на языке Lua в работе с Tarantool'ом:

- *Вставка 1 млн кортежей с помощью хранимой процедуры на языке Lua*,
- *Подсчет суммы по JSON-полям во всех кортежах*,
- *Индексированный поиск по шаблонам*.

6.1.1 Вставка 1 млн кортежей с помощью хранимой процедуры на языке Lua

Задание по данному практикуму: "Вставьте 1 миллион кортежей. В каждом кортеже должно быть поле, которое соответствует ключу в первичном индексе, в виде постоянно возрастающего числа, а также поле в виде буквенной строки со случайным значением из 10 символов."

Цель данного упражнения состоит в том, чтобы показать, как выглядят Lua-функции в Tarantool'е. Необходимо будет работать с математической библиотекой Lua, библиотекой для работы со строками интерпретатора Lua, Tarantool-библиотекой `box`, Tarantool-библиотекой `box.tuple`, циклами и конкатенацией. Инструкции легко будет выполнять даже тем, кто никогда не использовал раньше Lua или Tarantool. Единственное требование – знание того, как работают другие языки программирования, и изучение первых двух глав данного руководства. Но для лучшего понимания можно следовать по комментариям и ссылкам на руководство по Lua или другим пунктам в данном руководстве по Tarantool'у. А чтобы облегчить изучение, читайте инструкции параллельно с вводом операторов в Tarantool-клиент.

Настройка

Будем использовать Tarantool-песочницу, которую создавали для *упражнений раздела «Руководство для начинающих»*. Таким образом, у нас есть один спейс и числовой ключ первичного индекса, а также экземпляр Tarantool'а, который также выступает в виде клиента.

Разделитель

В более ранних версиях Tarantool'а многострочные функции обрамляются символами-разделителями. Сейчас в них нет необходимости, поэтому в данном практическом задании они использоваться не будут. Однако они все еще поддерживаются. Если вы хотите использовать разделители или используете более раннюю версию Tarantool'а, перед работой проверьте описание синтаксиса для *объявления разделителя*.

Создание функции, которая возвращает строку

Начнем с создания функции, которая возвращает заданную строку – "Hello world".

```
function string_function()
  return "hello world"
end
```

Слово «`function`» (функция) – ключевое слово в языке Lua. Рассмотрим подробно работу с языком Lua. Имя функции – string_function (строковая_функция). В функции есть один исполняемый оператор, `return "hello world"` (вернуть «hello world»). Строка «hello world» здесь заключена в двойные кавычки, хотя в Lua это не имеет значения, можно использовать одинарные кавычки. Слово «`end`» означает, что "это конец объявления Lua-функции." Чтобы проверить работу функции, можем выполнить команду

```
string_function()
```

Отправка `function-name()` (имя-функции) означает команду вызова Lua-функции. В результате возвращаемая функцией строка появится на экране.

Для получения подробной информации о строках в языке Lua, см. Главу 2.4 «Строки» в руководстве по языку Lua. Для получения подробной информации о функциях см. Главу 5 «Функции» в руководстве по языку Lua (chapter 5 «Functions»).

Теперь вывод на экране выглядит следующим образом:

```
tarantool> function string_function()
         >   return "hello world"
         > end
---
...
tarantool> string_function()
---
- hello world
...
tarantool>
```

Создание функции, которая вызывает другую функцию и определяет переменную

Теперь у нас есть функция `string_function`, и можно вызвать ее с помощью другой функции.

```
function main_function()
  local string_value
  string_value = string_function()
  return string_value
end
```

Сначала объявим переменную «string_value» (значение_строки). Слово «local» (локально) означает, что string_value появится только в main_function (основная_функция). Если бы мы не использовали «local», то string_value увидели бы даже пользователи других клиентов, которые подключились к данному экземпляру! Иногда это может быть очень полезно при взаимодействии клиентов, но не в нашем случае.

Затем определим значение для string_value, а именно, результат функции string_function(). Сейчас вызовем main_function(), чтобы проверить, что значение определено.

Для получения подробной информации о переменных в языке Lua, см. Главу 4.2 «Локальные переменные и блоки» в руководстве по языку Lua (chapter 4.2 «Local Variables and Blocks»).

Теперь вывод на экране выглядит следующим образом:

```
tarantool> function main_function()
         >   local string_value
         >   string_value = string_function()
         >   return string_value
         > end
---
...
tarantool> main_function()
---
- hello world
...
tarantool>
```

Изменение функции для возврата строки из одной случайной буквы

Сейчас стало понятно, как задавать переменную, поэтому можно изменить функцию string_function() так, чтобы вместо возврата заданной фразы «Hello world», она возвращала случайным образом выбранную букву от „A“ до „Z“.

```
function string_function()
  local random_number
  local random_string
  random_number = math.random(65, 90)
  random_string = string.char(random_number)
  return random_string
end
```

Нет необходимости стирать содержание старой функции string_function(), оно просто перезаписывается. Первый оператор вызывает функцию из математической библиотеки Lua, которая возвращает случайное число; параметры означают, что число должно быть целым от 65 до 90. Второй оператор вызывает функцию из библиотеки Lua для работы со строками, которая преобразует число в символ; параметр представляет собой кодовую точку символа. К счастью, в кодировке ASCII символу „A“ соответствует значение 65, а „Z“ – 90, так что в результате всегда получим букву от A до Z.

Для получения подробной информации о функциях математической библиотеки в языке Lua, см. Практическое задание по математической библиотеке для пользователей Lua (Math Library Tutorial). Для получения подробной информации о функциях библиотеки для работы со строками в языке Lua,

см. Практическое задание по библиотеке для работы со строками для пользователей Lua (String Library Tutorial).

И снова функцию `string_function()` можно вызвать из main_function(), которую можно вызвать с помощью `main_function()`.

Теперь вывод на экране выглядит следующим образом:

```
tarantool> function string_function()
         >   local random_number
         >   local random_string
         >   random_number = math.random(65, 90)
         >   random_string = string.char(random_number)
         >   return random_string
         > end
---
...
tarantool> main_function()
---
- C
...
tarantool>
```

. . . На самом деле, вывод не всегда будет именно таким, поскольку функция `math.random()` вызывает случайные числа. Но для наглядности случайные значения в строке не важны.

Изменение функции для возврата строки из десяти случайных букв

Сейчас стало понятно, как вызывать строки из одной случайной буквы, поэтому можно перейти к нашей цели – возврату строки из десяти букв с помощью конкатенации десяти строк из одной случайной буквы в цикле.

```
function string_function()
  local random_number
  local random_string
  random_string = ""
  for x = 1,10,1 do
    random_number = math.random(65, 90)
    random_string = random_string .. string.char(random_number)
  end
  return random_string
end
```

Слова «for x = 1,10,1» означают: “начать с x, равного 1, зацикливать до тех пор, пока x не будет равен 10, увеличивать x на 1 на каждом шаге цикла”. Символ «..» означает «конкатенацию», то есть добавление строки справа от знака «..» к строке слева от знака «..». Поскольку в начале определяется, что random_string (случайная_строка) представляет собой «» (пустую строку), в результате получим, что в random_string 10 случайных букв. И снова функцию `string_function()` можно вызвать из `main_function()`, которую можно вызвать с помощью `main_function()`.

Для получения подробной информации о циклах в языке Lua, см. Главу 4.3.4 «Числовой оператор for» в руководстве по языку Lua (chapter 4.3.4 «Numeric for»).

Теперь вывод на экране выглядит следующим образом:

```
tarantool> function string_function()
         >   local random_number
```

(continues on next page)

(продолжение с предыдущей страницы)

```
         >   local random_string
         >   random_string = ""
         >   for x = 1,10,1 do
         >     random_number = math.random(65, 90)
         >     random_string = random_string .. string.char(random_number)
         >   end
         >   return random_string
         > end
---
...
tarantool> main_function()
---
- 'ZUDJBHKEFM'
...
tarantool>
```

Составление кортежа из числа и строки

Сейчас стало понятно, как создать строку из 10 случайных букв, поэтому можно создать кортеж, который будет содержать число и строку из 10 случайных букв, с помощью функции в Tarantool-библиотеке Lua-функций.

```
function main_function()
  local string_value, t
  string_value = string_function()
  t = box.tuple.new({1, string_value})
  return t
end
```

После этого, «t» будет представлять собой значение нового кортежа с двумя полями. Первое поле является числовым: «1». Второе поле представляет собой случайную строку. И снова функцию `string_function()` можно вызвать из `main_function()`, которую можно вызвать с помощью `main_function()`.

Для получения подробной информации о кортежах в Tarantool'е, см. раздел *Вложенный модуль box.tuple* руководства по Tarantool'у.

Теперь вывод на экране выглядит следующим образом:

```
tarantool> function main_function()
         > local string_value, t
         > string_value = string_function()
         > t = box.tuple.new({1, string_value})
         > return t
         > end
---
...
tarantool> main_function()
---
- [1, 'PNPZPCOOKA']
...
tarantool>
```

Изменение основной функции main_function для вставки кортежа в базу данных

Сейчас стало понятно, как создавать кортеж, который содержит число и строку из десяти случайных букв, поэтому осталось только поместить этот кортеж в спейс tester. Следует отметить, что tester – это первый спейс, определенный в песочнице, поэтому он представляет собой таблицу в базе данных.

```
function main_function()
  local string_value, t
  string_value = string_function()
  t = box.tuple.new({1,string_value})
  box.space.tester:replace(t)
end
```

Здесь новая строка – `box.space.tester:replace(t)`. Имя содержит слово „tester“, потому что вставка будет осуществляться в спейс tester. Второй параметр представляет собой значение в кортеже. Для абсолютной точности мы могли ввести команду `box.space.tester:insert(t)`, а не `box.space.tester:replace(t)`, но слово «replace» (заменить) означает “вставить, даже если уже существует кортеж, у которого значение первичного ключа совпадает”, и это облегчит повтор упражнения, даже если песочница не пуста. После того, как это будет выполнено, спейс tester будет содержать кортеж с двумя полями. Первое поле будет 1. Второе поле будет представлять собой строку из десяти случайных букв. И снова функцию `string_function()` можно вызвать из `main_function()`, которую можно вызвать с помощью `main_function()`. Но функция `main_function()` не может полностью отразить ситуацию, поскольку она не возвращает t, она только размещает t в базе данных. Чтобы убедиться, что произошла вставка, используем SELECT-запрос.

```
main_function()
        box.space.tester:select{1}
```

Для получения подробной информации о вызовах insert и replace в Tarantool'е, см. разделы *Вложенный модуль box.space*, *space_object:insert()* и *space_object:replace()* руководства по Tarantool'у.

Теперь вывод на экране выглядит следующим образом:

```
tarantool> function main_function()
         >   local string_value, t
         >   string_value = string_function()
         >   t = box.tuple.new({1,string_value})
         >   box.space.tester:replace(t)
         > end
---
...
tarantool> main_function()
---
...
tarantool> box.space.tester:select{1}
---
- - [1, 'EUJYVEECIL']
...
tarantool>
```

Изменение основной функции main_function для вставки миллиона кортежей в базу данных

Сейчас стало понятно, как вставить кортеж в базу данных, поэтому несложно догадаться, как можно увеличить масштаб: вместо того, чтобы вставлять значение 1 для первичного ключа, вставьте значение переменной от 1 до миллиона в цикле. Поскольку уже рассматривалось, как заводить цикл, это будет несложно. Мы лишь добавим небольшой штрих – функцию распределения во времени.

```
function main_function()
  local string_value, t
  for i = 1,1000000,1 do
    string_value = string_function()
    t = box.tuple.new({i,string_value})
    box.space.tester:replace(t)
  end
end
start_time = os.clock()
main_function()
end_time = os.clock()
'insert done in ' .. end_time - start_time .. ' seconds'
```

Стандартная Lua-функция os.clock() вернет время ЦП в секундах с момента начала программы. Таким образом, выводя start_time = number of seconds (время_начала = число секунд) прямо перед вставкой, а затем выводя end_time = number of seconds (время_окончания = число секунд) сразу после вставки, можно рассчитать (время_окончания - время_начала) = затраченное время в секундах. Отобразим это значение путем ввода в запрос без операторов, что приведет к тому, что Tarantool отправит значение на клиент, который выведет это значение. (Ответ Lua на C-функцию `printf()`, а именно `print()`, также сработает.)

Для получения подробной информации о функции `os.clock()` см. Главу 22.1 «Дата и время» в руководстве по языку Lua (chapter 22.1 «Date and Time»). Для получения подробной информации о функции print() см. Главу 5 «Функции» в руководстве по языку Lua (chapter 5 «Functions»).

И поскольку наступает кульминация – повторно введем окончательные варианты всех необходимых запросов: запрос, который создает `string_function()`, запрос, который создает `main_function()`, и запрос, который вызывает `main_function()`.

```
function string_function()
          local random_number
          local random_string
          random_string = ""
          for x = 1,10,1 do
            random_number = math.random(65, 90)
            random_string = random_string .. string.char(random_number)
          end
          return random_string
        end

        function main_function()
          local string_value, t
          for i = 1,1000000,1 do
            string_value = string_function()
            t = box.tuple.new({i,string_value})
            box.space.tester:replace(t)
          end
        end
        start_time = os.clock()
        main_function()
        end_time = os.clock()
        'insert done in ' .. end_time - start_time .. ' seconds'
```

Теперь вывод на экране выглядит следующим образом:

```
tarantool> function string_function()
         >   local random_number
```

(continues on next page)

(продолжение с предыдущей страницы)

```
         >   local random_string
         >   random_string = ""
         >   for x = 1,10,1 do
         >     random_number = math.random(65, 90)
         >     random_string = random_string .. string.char(random_number)
         >   end
         >   return random_string
         > end
---
...
tarantool> function main_function()
         >   local string_value, t
         >   for i = 1,1000000,1 do
         >     string_value = string_function()
         >     t = box.tuple.new({i,string_value})
         >     box.space.tester:replace(t)
         >   end
         > end
---
...
tarantool> start_time = os.clock()
---
...
tarantool> main_function()
---
...
tarantool> end_time = os.clock()
---
...
tarantool> 'insert done in ' .. end_time - start_time .. ' seconds'
---
- insert done in 37.62 seconds
...
tarantool>
```

Итак, мы доказали, что возможности Lua-функций довольно многообразны (на самом деле, с помощью хранимых процедур на языке Lua в Tarantool'е можно сделать больше, чем с помощью хранимых процедур в некоторых SQL СУБД), и несложно комбинировать функции Lua-библиотек и функции Tarantool-библиотек.

Также мы показали, что вставка миллиона кортежей заняла 37 секунд. Хостом выступил ноутбук с ОС Linux. А изменив значение *wal_mode* на „none“ перед запуском теста, можно уменьшить затраченное время до 4 секунд.

6.1.2 Подсчет суммы по JSON-полям во всех кортежах

Задание по данному практикуму: “Предположим, что в каждом кортеже есть строка в формате JSON. В каждой строке есть числовое поле формата JSON. Для каждого кортежа необходимо найти значение числового поля и прибавить его к переменной „sum“ (сумма). В конце функция должна вернуть переменную „sum“.” Цель данного упражнения – получить опыт в прочтении и обработке кортежей одновременно.

```
1 json = require('json')
2 function sum_json_field(field_name)
3   local v, t, sum, field_value, is_valid_json, lua_table
```

(continues on next page)

(продолжение с предыдущей страницы)

```
  sum = 0
  for v, t in box.space.tester:pairs() do
    is_valid_json, lua_table = pcall(json.decode, t[2])
    if is_valid_json then
      field_value = lua_table[field_name]
      if type(field_value) == "number" then sum = sum + field_value end
    end
  end
  return sum
end
```

СТРОКА 3: ЗАЧЕМ НУЖЕН «LOCAL». Эта строка объявляет все переменные, которые будут использоваться в функции. На самом деле, нет необходимости в начале объявлять все переменные, а в длинной функции лучше объявить переменные прямо перед их использованием. Фактически объявлять переменные вообще необязательно, но необъявленная переменная будет «глобальной». Это представляется нежелательным для всех переменных, объявленных в строке 1, поскольку все они используются только в рамках функции.

СТРОКА 5: ЗАЧЕМ НУЖЕН «PAIRS()». Наша задача – пройти по всем строкам, что можно сделать двумя способами: с помощью *box.space.space_object:pairs()* или с помощью `variable = select(...)` с указанием `for i, n, 1 do` *`некая-функция`*`(variable[i]) end`. Для данного примера мы предпочли использовать `pairs()`.

СТРОКА 5: НАЧАЛО ОСНОВНОГО ЦИКЛА. Всё внутри цикла «`for`» будет повторяться до тех пор, пока не кончатся индекс-ключи. На полученный кортеж можно сослаться с помощью переменной `t`.

СТРОКА 6: ЗАЧЕМ НУЖЕН «PCALL». Если бы мы просто ввели `lua_table = json.decode(t[2]))`, то функция завершила бы работу с ошибкой, обнаружив любое несоответствие в JSON-строке, например отсутствие запятой. Заключив функцию в «`pcall`» (protected call – защищенный вызов), мы заявляем следующее: хотим перехватывать ошибки такого рода, поэтому в случае ошибки следует просто указать `is_valid_json = false`, и позднее мы решим, что с этим делать.

СТРОКА 6: ЗНАЧЕНИЕ. Функция *json.decode* означает декодирование JSON-строки, а параметр t[2] представляет собой ссылку на JSON-строку. Здесь есть заранее заданные значения, а мы предполагаем, что JSON-строка была вставлена во второе поле кортежа. Например, предположим, что кортеж выглядит следующим образом:

```
field[1]: 444
field[2]: '{"Hello": "world", "Quantity": 15}'
```

что означает, что первое поле кортежа, первичное поле, представляет собой число, а второе поле кортежа, JSON-строка, является строкой. Таким образом, значение оператора будет следующим: «декодировать `t[2]` (второе поле кортежа) как JSON-строку; если обнаружится ошибка, то указать `is_valid_json = false`; если ошибок нет, указать `is_valid_json = true` и `lua_table` = Lua-таблица, в которой находится декодированная строка».

СТРОКА 8. Наконец, мы готовы получить значение JSON-поля из Lua-таблицы, взятое из JSON-строки. Значение в field_name (имя_поля), которое является параметром всей функции, должно представлять собой JSON-поле. Например, в JSON-строке `'{"Hello": "world", "Quantity": 15}'` есть два JSON-поля: «Hello» и «Quantity». Если вся функция вызывается с помощью `sum_json_field("Quantity")`, тогда `field_value = lua_table[field_name]` (значение_поля = Lua_таблица[имя_поля]) по сути аналогично `field_value = lua_table["Quantity"]` или даже `field_value = lua_table.Quantity`. Итак, этими тремя способами можно ввести следующую команду: получить значение поля Quantity в Lua-таблице и поместить его в переменную `field_value`.

СТРОКА 9: ЗАЧЕМ НУЖЕН «IF». Предположим, что JSON-строка не содержит синтаксических

ошибок, но JSON-поле не является числовым или вовсе отсутствует. В таком случае выполнение функции прервется при попытке прибавить значение к сумме. Если сначала проверить, `type(field_value) == "number"` (тип(значение_поля) == «число»), можно избежать прерывания функции. Если вы уверены, что база данных в идеальном состоянии, этот шаг можно пропустить.

И функция готова. Пора протестировать ее. Начинаем с пустой базы данных так же, как с песочницы в *упражнения в «Руководстве для начинающих»*,

```
-- если спейс tester остался от предыдущего задания, удалите его
box.space.tester:drop()
box.schema.space.create('tester')
box.space.tester:create_index('primary', {parts = {1, 'unsigned'}})
```

затем добавим несколько кортежей, где первое поле является числовым, а второе поле представляет собой строку.

```
box.space.tester:insert{444, '{"Item": "widget", "Quantity": 15}'}
box.space.tester:insert{445, '{"Item": "widget", "Quantity": 7}'}
box.space.tester:insert{446, '{"Item": "golf club", "Quantity": "sunshine"}'}
box.space.tester:insert{447, '{"Item": "waffle iron", "Quantit": 3}'}
```

Для целей практики здесь допущены ошибки. В «golf club» и «waffle iron» поля Quantity не являются числовыми, поэтому будут игнорироваться. Таким образом, итоговая сумма для полей Quantity в JSON-строках должна быть следующей: 15 + 7 = 22.

Вызовите функцию с помощью `sum_json_field("Quantity")`.

```
tarantool> sum_json_field("Quantity")
---
- 22
...
```

Сработало. Для дополнительной отработки материала можно убрать заранее заданные значения, добавить проверку потенциально возможного арифметического переполнения при наличии больших значений некоторых полей, а также команду *передачи управления* при огромном количестве кортежей.

6.1.3 Индексированный поиск по шаблонам

Здесь приведена обобщенная функция, которая берет идентификатор поля и шаблон поиска, а затем возвращает все кортежи, которые подходят под критерии. * Поле должно быть первым полем в TREE-индексе. * Функция применяет шаблоны в языке Lua, что позволяет использовать «магические символы» в регулярных выражениях. * Начальные символы в шаблоне до самого первого магического символа будут использоваться в качестве ключа поиска по индексу. Каждый кортеж, обнаруженный по индексу, будет соответствовать всему шаблону. * В целях *кооперативной многозадачности* функция должна передавать управление через каждые 10 кортежей, если только нет причин отложить передачу управления. С помощью данной функции можно воспользоваться индексами Tarantool'а для ускорения и шаблонами на языке Lua для гибкости. Поддерживаются все возможности поиска LIKE в SQL – и многие другие.

Прочитайте следующий Lua-код, чтобы понять, как он работает. Комментарии, которые начинаются с «СМ. ПРИМЕЧАНИЕ ...» ссылаются на подробные объяснения, приведенные ниже.

```
function indexed_pattern_search(space_name, field_no, pattern)
  -- СМ. ПРИМЕЧАНИЕ №1 "ПОИСК НУЖНОГО ИНДЕКСА"
  if (box.space[space_name] == nil) then
    print("Error: Failed to find the specified space")
```

(continues on next page)

(продолжение с предыдущей страницы)

```
  return nil
end
local index_no = -1
for i=0,box.schema.INDEX_MAX,1 do
  if (box.space[space_name].index[i] == nil) then break end
  if (box.space[space_name].index[i].type == "TREE"
      and box.space[space_name].index[i].parts[1].fieldno == field_no
      and (box.space[space_name].index[i].parts[1].type == "scalar"
      or box.space[space_name].index[i].parts[1].type == "string")) then
    index_no = i
    break
  end
end
if (index_no == -1) then
  print("Error: Failed to find an appropriate index")
  return nil
end
-- СМ. ПРИМЕЧАНИЕ №2 "ПОЛУЧЕНИЕ КЛЮЧА ИНДЕКСНОГО ПОИСКА ИЗ ШАБЛОНА"
local index_search_key = ""
local index_search_key_length = 0
local last_character = ""
local c = ""
local c2 = ""
for i=1,string.len(pattern),1 do
  c = string.sub(pattern, i, i)
  if (last_character ~= "%") then
    if (c == '^' or c == "$" or c == "(" or c == ")" or c == "."
                 or c == "[" or c == "]" or c == "*" or c == "+"
                 or c == "-" or c == "?") then
      break
    end
    if (c == "%") then
      c2 = string.sub(pattern, i + 1, i + 1)
      if (string.match(c2, "%p") == nil) then break end
      index_search_key = index_search_key .. c2
    else
      index_search_key = index_search_key .. c
    end
  end
  last_character = c
end
index_search_key_length = string.len(index_search_key)
if (index_search_key_length < 3) then
  print("Error: index search key " .. index_search_key .. " is too short")
  return nil
end
-- СМ. ПРИМЕЧАНИЕ №3 "ВНЕШНИЙ ЦИКЛ: НАЧАЛО"
local result_set = {}
local number_of_tuples_in_result_set = 0
local previous_tuple_field = ""
while true do
  local number_of_tuples_since_last_yield = 0
  local is_time_for_a_yield = false
  -- СМ. ПРИМЕЧАНИЕ №4 "ВНУТРЕННИЙ ЦИКЛ: ИТЕРАТОР"
  for _,tuple in box.space[space_name].index[index_no]:
  pairs(index_search_key,{iterator = box.index.GE}) do
```

(continues on next page)

(продолжение с предыдущей страницы)

```
      -- СМ. ПРИМЕЧАНИЕ №5 "ВНУТРЕННИЙ ЦИКЛ: ПРЕРЫВАНИЕ, ЕСЛИ КЛЮЧ ИНДЕКСА СЛИШКОМ БОЛЬШОЙ"
      if (string.sub(tuple[field_no], 1, index_search_key_length)
      > index_search_key) then
        break
      end
      -- СМ. ПРИМЕЧАНИЕ №6 "ВНУТРЕННИЙ ЦИКЛ: ПРЕРЫВАНИЕ ПОСЛЕ КАЖДЫХ ДЕСЯТИ КОРТЕЖЕЙ -- ВОЗМОЖНО"
      number_of_tuples_since_last_yield = number_of_tuples_since_last_yield + 1
      if (number_of_tuples_since_last_yield >= 10
          and tuple[field_no] ~= previous_tuple_field) then
        index_search_key = tuple[field_no]
        is_time_for_a_yield = true
        break
        end
      previous_tuple_field = tuple[field_no]
      -- СМ. ПРИМЕЧАНИЕ №7 "ВНУТРЕННИЙ ЦИКЛ: ДОБАВЛЕНИЕ В РЕЗУЛЬТАТ, ЕСЛИ ШАБЛОН СОВПАДЕТ"
      if (string.match(tuple[field_no], pattern) ~= nil) then
        number_of_tuples_in_result_set = number_of_tuples_in_result_set + 1
        result_set[number_of_tuples_in_result_set] = tuple
      end
    end
    -- СМ. ПРИМЕЧАНИЕ №8 "ВНЕШНИЙ ЦИКЛ: ПРЕРЫВАНИЕ ИЛИ ПЕРЕДАЧА УПРАВЛЕНИЯ И ПРОДОЛЖЕНИЕ"
    if (is_time_for_a_yield ~= true) then
      break
    end
    require('fiber').yield()
  end
  return result_set
end
```

ПРИМЕЧАНИЕ №1 «ПОИСК НУЖНОГО ИНДЕКСА» Вызывающий клиент передал space_name (имя_спейса – строка) и field_no (номер_поля – число). Требования следующие: (a) тип индекса должен быть «TREE», поскольку для других типов индекса (HASH, BITSET, RTREE) поиск с *итератором=GE* не вернет строки, упорядоченные по строковому значению; (b) field_no должен представлять собой первую часть индекса; (c) поле должно содержать строки, потому что для других типов данных (как «unsigned») шаблоны поиска не применяются; Если индекс не удовлетворяет этим требованиям, выдать сообщение об ошибке и вернуть нулевое значение nil.

ПРИМЕЧАНИЕ №2 «ПОЛУЧЕНИЕ КЛЮЧА ИНДЕКСНОГО ПОИСКА ИЗ ШАБЛОНА» Вызывающий клиент передал шаблон (строку). Ключом поиска по индексу являются символы в шаблоне до первого магического символа. Магические символы в Lua: % ^ $ ( ) . [ ] * + - ?. Например, если задан шаблон «ABC.E», точка будет магическим символом, и ключом поиска по индексу будет «ABC». Однако есть затруднение ... Если символ «%» будет идти следом за знаком препинания, этот знак препинания экранируется, поэтому следует убрать «%» из ключа поиска по индексу. Например, если задан шаблон «AB%$E», знак доллара экранируется, поэтому ключом поиска по индексу будет «AB$E». Наконец, есть проверка длины ключа поиска по индексу – не менее трех символов, причем это число выбрано произвольно, и даже ноль здесь подойдет, но по короткому ключу поиск займет длительное время.

ПРИМЕЧАНИЕ №3 «ВНЕШНИЙ ЦИКЛ: НАЧАЛО» Назначение функции – вернуть результирующий набор данных, как вернул бы запрос *box.space...select <box_space-select>*. Мы внесем ее во внешний цикл, который включает в себя внутренний цикл. Назначение внешнего цикла – выполнять внутренний цикл и, при необходимости, *передачу управления*, пока поиск не будет завершен. Назначение внутреннего цикла – находить кортежи по индексу и включать их в результирующий набор данных, если они подходят под шаблон.

ПРИМЕЧАНИЕ №4 «ВНУТРЕННИЙ ЦИКЛ: ИТЕРАТОР» Цикл for здесь использует pairs(), см.

объяснение, что такое итераторы. Во внутреннем цикле будет локальная переменная под названием «tuple» (кортеж), которая содержит последний кортеж, обнаруженный в ходе поиска по индексу.

ПРИМЕЧАНИЕ №5 «ВНУТРЕННИЙ ЦИКЛ: ПРЕРЫВАНИЕ, ЕСЛИ КЛЮЧ ИНДЕКСА СЛИШКОМ БОЛЬШОЙ» Используется итератор GE (Greater or Equal - больше или равно), поэтому необходимо уточнить: если ключ поиска по индексу включает в себя N символов, то крайние N символов слева от найденного поля индекса не должны быть больше ключа поиска. Например, если ключом поиска является „ABC“, то „ABCDE“ потенциально подходит, а „ABD“ означает, что в дальнейшем совпадений не будет.

ПРИМЕЧАНИЕ №6 «ВНУТРЕННИЙ ЦИКЛ: ПРЕРЫВАНИЕ ПОСЛЕ КАЖДЫХ ДЕСЯТИ КОРТЕЖЕЙ – ВОЗМОЖНО» Эта часть кода предназначена для кооперативной многозадачности. Число 10 выбрано произвольно, и как правило, большее число также подойдет. Простое правило гласит: «после проверки 10 кортежей передать управление, а затем возобновить поиск (то есть снова выполнять внутренний цикл), начиная с последнего обнаруженного значения». Однако, если индекс не уникален, или в индексе более одного поля, можно получить дублирующиеся результаты, например, {«ABC»,1}, {«ABC», 2}, {«ABC», 3}» – и будет трудно решить, с какого кортежа «ABC» возобновлять поиск. Таким образом, если найденное поле индекса совпадает с предыдущим найденным полем индекса, цикл не прерывается.

ПРИМЕЧАНИЕ №7 «ВНУТРЕННИЙ ЦИКЛ: ДОБАВЛЕНИЕ В РЕЗУЛЬТАТ, ЕСЛИ ШАБЛОН СОВПАДЕТ» Сравнение найденного поля индекса с шаблоном. Например, предположим, что вызывающий клиент передает шаблон «ABC.E», и существует поле индекса, содержащее «ABCDE». В таком случае, начальный ключ поиска будет «ABC». Таким образом, кортеж, содержащий поле индекса с «ABCDE» будет обнаружен итератором, поскольку «ABCDE» > «ABC». В этом случае, string.match вернет значение, отличное от нулевого nil. В итоге, этот кортеж можно добавить в результирующий набор данных.

ПРИМЕЧАНИЕ №8 «ВНЕШНИЙ ЦИКЛ: ПРЕРЫВАНИЕ ИЛИ ПЕРЕДАЧА УПРАВЛЕНИЯ И ПРОДОЛЖЕНИЕ» Существуют три условия, которые вызовут прерывание из внутреннего цикла: (1) цикл for заканчивается закономерно, потому что отсутствуют ключи индекса, которые больше или равны ключу поиска по индексу, (2) ключ индекса слишком большой, как описано в ПРИМЕЧАНИИ №5, (3) пора передавать управление, как описано в ПРИМЕЧАНИИ №6. Если условие (1) или условие (2) соблюдается, другие действия не требуются, и внешний цикл также заканчивается. Только в том случае, если справедливо условие (3), внешний цикл должен передать управление, а затем продолжить выполнение. Если он продолжит выполнение, то внутренний цикл – поиск с итератором – будет выполняться снова с новым значением для ключа поиска по индексу.

ПРИМЕР:

Запустите Tarantool, скопируйте и вставьте код для функции `indexed_pattern_search()` и попробуйте выполнить следующее:

```
box.space.t:drop()
box.schema.space.create('t')
box.space.t:create_index('primary',{})
box.space.t:create_index('secondary',{unique=false,parts={2,'string',3,'string'}})
box.space.t:insert{1,'A','a'}
box.space.t:insert{2,'AB',''}
box.space.t:insert{3,'ABC','a'}
box.space.t:insert{4,'ABCD',''}
box.space.t:insert{5,'ABCDE','a'}
box.space.t:insert{6,'ABCDE',''}
box.space.t:insert{7,'ABCDEF','a'}
box.space.t:insert{8,'ABCDF',''}
indexed_pattern_search("t", 2, "ABC.E.")
```

Получим следующий результат:

```
tarantool> indexed_pattern_search("t", 2, "ABC.E.")
---
- - [7, 'ABCDEF', 'a']
...
```

6.2 Практическое задание на C

Ниже приводится практическое занятие на языке C: *Хранимые процедуры на языке C*.

6.2.1 Хранимые процедуры на языке C

Tarantool может вызывать код на языке C с помощью *модулей*, *ffi* или хранимых процедур на C. В данном практическом задании рассматривается только третий метод, хранимые процедуры на языке C. На самом деле, программы всегда представляют собой функции на языке C, но исторически сложилось так, что широко используется фраза «хранимая процедура».

Данное практическое задание могут выполнить те, у кого есть пакет программ для разработки Tarantool'а и компилятор языка программирования C. Оно состоит из пяти задач:

(1) *easy.c* – выводит «hello world»;

(2) *harder.c* – декодирует переданное значение параметра;

(3) *hardest.c* – использует API для языка C для вставки в базу данных;

(4) *read.c* – использует API для языка C для выборки из базы данных;

(5) *write.c* – использует API для языка C для замены в базе данных.

По окончании задания, вы увидите описанные здесь результаты и сможете самостоятельно написать хранимые процедуры.

Подготовка

Проверьте наличие следующих элементов на компьютере:

- Tarantool 2.1
- Компилятор GCC, подойдет любая современная версия
- `module.h` и включенные в него файлы
- `msgpuck.h`
- `libmsgpuck.a` (только для некоторых последних версий msgpuck)

Файл `module.h` есть в системе, если Tarantool был установлен из исходных файлов. В противном случае, следует установить пакет Tarantool'а «developer». Например, на Ubuntu введите команду:

```
$ sudo apt-get install tarantool-dev
```

или на Fedora введите команду:

```
$ dnf -y install tarantool-devel
```

The `msgpuck.h` file will exist if Tarantool was installed from source. Otherwise the «msgpuck» package must be installed from https://github.com/tarantool/msgpuck.

Чтобы компилятор C увидел файлы `module.h` и `msgpuck.h`, путь к ним следует сохранить в переменной. Например, если адрес файла `module.h` – `/usr/local/include/tarantool/module.h`, а адрес файла `msgpuck.h` – `/usr/local/include/msgpuck/msgpuck.h`, введите команду:

```
$ export CPATH=/usr/local/include/tarantool:/usr/local/include/msgpuck
```

Статическая библиотека `libmsgpuck.a` нужна для версий msgpuck старше февраля 2017 года. Только в том случае, если встречаются проблемы соединения при использовании операторов GCC в примерах данного практического задания, в пути следует указывать `libmsgpuck.a` (`libmsgpuck.a` создан из исходных файлов загрузки msgpuck и Tarantool, поэтому его легко найти). Например, вместо «`gcc -shared -o harder.so -fPIC harder.c`» во втором примере ниже, необходимо ввести «`gcc -shared -o harder.so -fPIC harder.c libmsgpuck.a`».

Tarantool выполняет запросы в качестве *клиента*. Запустите Tarantool и введите эти запросы.

```
box.cfg{listen=3306}
box.schema.space.create('capi_test')
box.space.capi_test:create_index('primary')
net_box = require('net.box')
capi_connection = net_box:new(3306)
```

Проще говоря: создайте спейс под названием `capi_test`, и выполните соединение с одноименным `capi_connection`.

Не закрывайте клиент. Он понадобится для последующих запросов.

easy.c

Запустите еще один терминал. Измените директорию (`cd`), чтобы она совпадала с директорией, где запущен клиент.

Создайте файл. Назовите его `easy.c`. Запишите в него следующие шесть строк.

```
#include "module.h"
int easy(box_function_ctx_t *ctx, const char *args, const char *args_end)
{
  printf("hello world\n");
  return 0;
}
int easy2(box_function_ctx_t *ctx, const char *args, const char *args_end)
{
  printf("hello world -- easy2\n");
  return 0;
}
```

Скомпилируйте программу, что создаст файл библиотеки под названием `easy.so`:

```
$ gcc -shared -o easy.so -fPIC easy.c
```

Теперь вернитесь в клиент и выполните следующие запросы:

```
box.schema.func.create('easy', {language = 'C'})
box.schema.user.grant('guest', 'execute', 'function', 'easy')
capi_connection:call('easy')
```

Если эти запросы вам незнакомы, перечитайте описание *box.schema.func.create()*, *box.schema.user.grant()* и *conn:call()*.

Важна функция `capi_connection:call('easy')`.

Во-первых, она ищет функцию easy, что должно быть легко, потому что по умолчанию Tarantool ищет в текущей директории файл под названием `easy.so`.

Во-вторых, она вызывает функцию easy. Поскольку функция `easy()` в `easy.c` начинается с `printf("hello world\n")`, слова «hello world» появятся на экране.

В-третьих, она проверяет, что вызов прошел успешно. Поскольку функция `easy()` в `easy.c` оканчивается на `return 0`, сообщение об ошибке отсутствует, и запрос выполнен.

Результат должен выглядеть следующим образом:

```
tarantool> capi_connection:call('easy')
hello world
---
- []
...
```

Теперь вызовем другую функцию в easy.c – `easy2()`. Она практически совпадает с функцией `easy()`, но есть небольшое отличие: если имя файла не совпадет с именем функции, нужно будет указать *имя-файла*.*имя-функции*.

```
box.schema.func.create('easy.easy2', {language = 'C'})
box.schema.user.grant('guest', 'execute', 'function', 'easy.easy2')
capi_connection:call('easy.easy2')
```

. . . и на этот раз результатом будет: «hello world – easy2».

Вывод: вызвать C-функцию легко.

harder.c

Вернитесь в терминал, где была создана программа `easy.c`.

Создайте файл. Назовите его `harder.c`. Запишите в него следующие 17 строк:

```
#include "module.h"
#include "msgpuck.h"
int harder(box_function_ctx_t *ctx, const char *args, const char *args_end)
{
  uint32_t arg_count = mp_decode_array(&args);
  printf("arg_count = %d\n", arg_count);
  uint32_t field_count = mp_decode_array(&args);
  printf("field_count = %d\n", field_count);
  uint32_t val;
  int i;
  for (i = 0; i < field_count; ++i)
  {
    val = mp_decode_uint(&args);
    printf("val=%d.\n", val);
  }
  return 0;
}
```

Скомпилируйте программу, что создаст файл библиотеки под названием `harder.so`:

```
$ gcc -shared -o harder.so -fPIC harder.c
```

Теперь вернитесь в клиент и выполните следующие запросы:

```
box.schema.func.create('harder', {language = 'C'})
box.schema.user.grant('guest', 'execute', 'function', 'harder')
passable_table = {}
table.insert(passable_table, 1)
table.insert(passable_table, 2)
table.insert(passable_table, 3)
capi_connection:call('harder', passable_table)
```

На этот раз вызов передает Lua-таблицу (`passable_table`) в функцию `harder()`. Функция“harder()“ увидит это, как указано в параметре `char *args`.

At this point the `harder()` function will start using functions defined in msgpuck.h. The routines that begin with «mp» are msgpuck functions that handle data formatted according to the MsgPack specification. Passes and returns are always done with this format so one must become acquainted with msgpuck to become proficient with the C API.

Однако, пока достаточно понимать, что функция `mp_decode_array()` возвращает количество элементов в массиве, а функция `mp_decode_uint` возвращает целое число без знака из `args`. Есть также побочный эффект: по окончании декодирования `args` изменился и теперь указывает на следующий элемент.

Таким образом, первой будет отображена строка «arg_count = 1», поскольку был передан только один элемент: `passable_table`. Второй будет отображена строка «field_count = 3», потому что в таблице находятся три элемента. Следующие три строки будут «1», «2» и «3», потому что это значения элементов в таблице.

Теперь вывод на экране выглядит следующим образом:

```
tarantool> capi_connection:call('harder', passable_table)
arg_count = 1
field_count = 3
val=1.
val=2.
val=3.
---
- []
...
```

Вывод: на первый взгляд, декодирование значений параметров, переданных в C-функцию непросто, но существуют документированные процедуры для этих целей, и их не так много.

hardest.c

Вернитесь в терминал, где были созданы программы `easy.c` и `harder.c`.

Создайте файл. Назовите его `hardest.c`. Запишите в него следующие 13 строк:

```
#include "module.h"
#include "msgpuck.h"
int hardest(box_function_ctx_t *ctx, const char *args, const char *args_end)
{
  uint32_t space_id = box_space_id_by_name("capi_test", strlen("capi_test"));
  char tuple[1024]; /* Must be big enough for mp_encode results */
  char *tuple_pointer = tuple;
  tuple_pointer = mp_encode_array(tuple_pointer, 2);
  tuple_pointer = mp_encode_uint(tuple_pointer, 10000);
  tuple_pointer = mp_encode_str(tuple_pointer, "String 2", 8);
  int n = box_insert(space_id, tuple, tuple_pointer, NULL);
  return n;
}
```

Скомпилируйте программу, что создаст файл библиотеки под названием `hardest.so`:

```
$ gcc -shared -o hardest.so -fPIC hardest.c
```

Теперь вернитесь в клиент и выполните следующие запросы:

```
box.schema.func.create('hardest', {language = "C"})
box.schema.user.grant('guest', 'execute', 'function', 'hardest')
box.schema.user.grant('guest', 'read,write', 'space', 'capi_test')
capi_connection:call('hardest')
```

На этот раз C-функция выполняет три действия:

(1) найдет числовой идентификатор спейса `capi_test` путем вызова `box_space_id_by_name()`;

(2) форматирует кортеж, используя другие функции `msgpuck.h`;

(3) вставит кортеж с помощью `box_insert()`.

Предупреждение: `char tuple[1024];` используется здесь просто в качестве быстрого способа ввода команды «выделить байтов с запасом». В серьезных программах разработчику следует обратить внимание на то, чтобы выделить достаточно места, которое будут использовать процедуры `mp_encode`.

Затем всё еще в клиенте выполните следующий запрос:

```
box.space.capi_test:select()
```

Результат должен выглядеть следующим образом:

```
tarantool> box.space.capi_test:select()
---
- - [10000, 'String 2']
...
```

Это доказывает, что функция `hardest()` была успешно выполнена, но откуда взялись *box_space_id_by_name()* и *box_insert()*? Ответ: *API для языка C*.

read.c

Вернитесь в терминал, где были созданы программы `easy.c`, `harder.c` и `hardest.c`.

Создайте файл. Назовите его `read.c`. Запишите в него следующие 43 строки:

```
#include "module.h"
#include <msgpuck.h>
int read(box_function_ctx_t *ctx, const char *args, const char *args_end)
{
  char tuple_buf[1024];       /* здесь будет храниться тапл в сыром MsgPack-формате */
  uint32_t space_id = box_space_id_by_name("capi_test", strlen("capi_test"));
  uint32_t index_id = 0;      /* номер первого индекса спейса */
  uint32_t key = 10000;       /* значение ключа, используемое box_insert() */
  mp_encode_array(tuple_buf, 0); /* clear */
  box_tuple_format_t *fmt = box_tuple_format_default();
  box_tuple_t *tuple = box_tuple_new(fmt, tuple_buf, tuple_buf+512);
  assert(tuple != NULL);
  char key_buf[16];           /* передаем key_buf = закодированный ключ = 1000 */
  char *key_end = key_buf;
```

(continues on next page)

(продолжение с предыдущей страницы)

```
  key_end = mp_encode_array(key_end, 1);
  key_end = mp_encode_uint(key_end, key);
  assert(key_end < key_buf + sizeof(key_buf));
  /* Получить кортеж. У нас нет box_select(), но есть вот это. */
  int r = box_index_get(space_id, index_id, key_buf, key_end, &tuple);
  assert(r == 0);
  assert(tuple != NULL);
  /* Получить каждое поле кортежа + показать полученное значение */
  int field_no;             /* номер первого поля = 0 */
  for (field_no = 0; field_no < 2; ++field_no)
  {
    const char *field = box_tuple_field(tuple, field_no);
    assert(field != NULL);
    assert(mp_typeof(*field) == MP_STR || mp_typeof(*field) == MP_UINT);
    if (mp_typeof(*field) == MP_UINT)
    {
      uint32_t uint_value = mp_decode_uint(&field);
      printf("uint value=%u.\n", uint_value);
    }
    else /* если (mp_typeof(*field) == MP_STR) */
    {
      const char *str_value;
      uint32_t str_value_length;
      str_value = mp_decode_str(&field, &str_value_length);
      printf("string value=%.*s.\n", str_value_length, str_value);
    }
  }
  return 0;
}
```

Скомпилируйте программу, что создаст файл библиотеки под названием `read.so`:

```
$ gcc -shared -o read.so -fPIC read.c
```

Теперь вернитесь в клиент и выполните следующие запросы:

```
box.schema.func.create('read', {language = "C"})
box.schema.user.grant('guest', 'execute', 'function', 'read')
box.schema.user.grant('guest', 'read,write', 'space', 'capi_test')
capi_connection:call('read')
```

На этот раз C-функция выполняет четыре действия:

(1) снова найдет числовой идентификатор спейса `capi_test` путем вызова `box_space_id_by_name()`;

(2) форматирует ключ поиска = 10 000, используя другие функции `msgpuck.h`;

(3) получает кортеж с помощью `box_index_get()`;

(4) проходит по полям каждого кортежа с помощью `box_tuple_get()`. а затем декодирует каждое поле в зависимости от его типа. В данном случае, поскольку мы получаем кортеж, который сами вставили с помощью `hardest.c`, мы знаем заранее, что его тип будет MP_UINT или MP_STR. Однако, весьма часто здесь употребляется оператор выбора case с одной опцией для каждого возможного типа.

В результате вызова `capi_connection:call('read')` должны получить:

```
tarantool> capi_connection:call('read')
uint value=10000.
string value=String 2.
---
- []
...
```

Это доказывает, что функция `read()` была успешно выполнена. И снова важные функции, которые начинаются с *box* – *box_index_get()* и *box_tuple_field()* – пришли из *API для языка C*.

write.c

Вернитесь в терминал, где были созданы программы `easy.c`, `harder.c`, `hardest.c` и `read.c`.

Создайте файл. Назовите его `write.c`. Запишите в него следующие 24 строки:

```
#include "module.h"
#include <msgpuck.h>
int write(box_function_ctx_t *ctx, const char *args, const char *args_end)
{
  static const char *space = "capi_test";
  char tuple_buf[1024]; /* Должен быть достаточно большим, чтобы вместить результат mp_encode */
  uint32_t space_id = box_space_id_by_name(space, strlen(space));
  if (space_id == BOX_ID_NIL) {
    return box_error_set(__FILE__, __LINE__, ER_PROC_C,
    "Can't find space %s", "capi_test");
  }
  char *tuple_end = tuple_buf;
  tuple_end = mp_encode_array(tuple_end, 2);
  tuple_end = mp_encode_uint(tuple_end, 1);
  tuple_end = mp_encode_uint(tuple_end, 22);
  box_txn_begin();
  if (box_replace(space_id, tuple_buf, tuple_end, NULL) != 0)
    return -1;
  box_txn_commit();
  fiber_sleep(0.001);
  struct tuple *tuple = box_tuple_new(box_tuple_format_default(),
                                      tuple_buf, tuple_end);
  return box_return_tuple(ctx, tuple);
}
```

Скомпилируйте программу, что создаст файл библиотеки под названием `write.so`:

```
$ gcc -shared -o write.so -fPIC write.c
```

Теперь вернитесь в клиент и выполните следующие запросы:

```
box.schema.func.create('write', {language = "C"})
box.schema.user.grant('guest', 'execute', 'function', 'write')
box.schema.user.grant('guest', 'read,write', 'space', 'capi_test')
capi_connection:call('write')
```

На этот раз C-функция выполняет шесть действий:

(1) снова найдет числовой идентификатор спейса `capi_test` путем вызова `box_space_id_by_name()`;

(2) создает новый кортеж;

(3) начинает транзакцию;

(4) заменяет кортеж в `box.space.capi_test`

(5) заканчивает транзакцию;

(6) последняя строка заменяет цикл `read.c` – вместо получения и вывода каждого поля, использует функцию `box_return_tuple(...)` для возврата всего кортежа вызывающему клиенту, чтобы вывести его на экран.

В результате вызова `capi_connection:call('write')` должны получить:

```
tarantool> capi_connection:call('write')
---
- [[1, 22]]
...
```

Это доказывает, что функция `write()` была успешно выполнена. И снова важные функции, которые начинаются с *box* – *box_txn_begin()*, *box_txn_commit()* и *box_return_tuple()* – пришли из *API для языка C*.

Вывод: длинное описание всего API для языка C необходимо в силу весомых причин. Все функции можно вызвать из C-функций, которые вызываются из Lua. Таким образом, хранимые процедуры на языке C получают полный доступ к базе данных.

Очистка данных

- Удалите все кортежи с функцией с помощью *box.schema.func.drop*.
- Удалите спейс `capi_test` с помощью *box.schema.capi_test:drop()*.
- Удалите файлы с разрешением `.c` и `.so`, созданные для данного практического задания.

Пример из набора тестов

Скачайте исходный код Tarantool'а. Откройте поддиректорию `test/box`. Проверьте наличие файла под названием `tuple_bench.test.lua` и еще одного файла под названием `tuple_bench.c`. Изучите Lua-файл на предмет вызова функции в C-файле с использованием методов, описанных в данном практическом задании.

Вывод: некоторые тесты из стандартного набора используют хранимые процедуры на языке C, а они должны работать, поскольку мы не можем выпустить Tarantool, если он не прошел тестирование.

6.3 SQL tutorial

This tutorial is a demonstration of the SQL feature introduced in Tarantool 2.x series. There are two ways to go through this tutorial:

- read what we say the results are and take our word for it, or
- copy and paste each section and see everything work with Tarantool 2.1.

You will encounter all the functionality that you'd encounter in an «SQL-101» course.

6.3.1 Starting up with a first table and SELECTs

Initialize

Requests will be done using Tarantool as a *client*. Start Tarantool and, optionally, enter the Tarantool configuration request, for example:

```
tarantool> box.cfg{}
```

Before Tarantool 2.0 you needed to say `box.cfg{...}` prior to performing any database operations. Now you can start working with the database outright. Tarantool initiates the database module and applies *default settings*.

set

A feature of the client console program is that you can switch languages and specify the end-of-statement delimiter.

Here we say: default language is SQL and statements end with semicolons.

```
tarantool> \set language sql
tarantool> \set delimiter ;
```

CREATE, INSERT, UPDATE, SELECT

Start with simple SQL statements just to be sure they're there.

```
CREATE TABLE table1 (column1 INTEGER PRIMARY KEY, column2 VARCHAR(100));
INSERT INTO table1 VALUES (1, 'A');
UPDATE table1 SET column2 = 'B';
SELECT * FROM table1 WHERE column1 = 1;
```

The result of the `SELECT` statement will look like this:

```
tarantool> SELECT * FROM table1 WHERE column1 = 1;
---
- - [1, 'B']
...
```

Reality check: actually the result will include include initial fields called «metadata», the names and data types of each column. For all SELECT examples we show only the result rows without showing the metadata.

CREATE TABLE

Here is `CREATE TABLE` with more details:

- There are multiple columns, with different data types.
- There is a `PRIMARY KEY` (unique and not-null) for two of the columns.

```
CREATE TABLE table2 (column1 INTEGER,
                     column2 VARCHAR(100),
                     column3 SCALAR,
                     column4 FLOAT,
                     PRIMARY KEY (column1, column2));
```

The result will be: «`rowcount: 1`» (no error).

INSERT

Try to put 5 rows in the table:

- The INTEGER and FLOAT columns get numbers.
- The VARCHAR and SCALAR columns get strings (the SCALAR strings are expressed as hexadecimals).

```
INSERT INTO table2 VALUES (1, 'AB', X'4142', 5.5);
INSERT INTO table2 VALUES (1, 'CD', X'2020', 1E4);
INSERT INTO table2 VALUES (1, 'AB', X'A5', -5.5);
INSERT INTO table2 VALUES (2, 'AB', X'2020', 12.34567);
INSERT INTO table2 VALUES (-1000, '', X'', 0.0);
```

Получим следующий результат:

- The third `INSERT` will fail because of a primary-key violation (`1, 'AB'` is a duplication).
- The other four `INSERT` statements will succeed.

SELECT with ORDER BY clause

Retrieve the 4 rows in the table, in descending order by `column2`, then (where the `column2` values are the same) in ascending order by column4.

«*» is short for «all columns».

```
SELECT * FROM table2 ORDER BY column2 DESC, column4 ASC;
```

Получим следующий результат:

```
- - [1, 'CD', '  ', 10000]
  - [1, 'AB', 'AB', 5.5]
  - [2, 'AB', '  ', 12.34567]
  - [-1000, '', '', 0]
```

SELECT with WHERE clauses

Retrieve some of what you inserted:

- The first statement uses the `LIKE` comparison operator which is asking for «first character must be „A“, the next characters can be anything.»
- The second statement uses logical operators and parentheses, so the ANDed expressions must be true, or the ORed expression must be true. Notice the columns don't have to be indexed.

```
SELECT column1, column2, column1 * column4 FROM table2 WHERE column2
LIKE 'A%';
SELECT column1, column2, column3, column4 FROM table2
    WHERE (column1 < 2 AND column4 < 10)
    OR column3 = X'2020';
```

The results will be:

```
- - [1, 'AB', 5.5]
  - [2, 'AB', 24.69134]
```

and

```
- - [-1000, '', '', 0]
  - [1, 'AB', 'AB', 5.5]
  - [1, 'CD', '  ', 10000]
  - [2, 'AB', '  ', 12.34567]
```

SELECT with GROUP BY and aggregating

Retrieve with grouping.

The rows which have the same values for `column2` are grouped and are aggregated – summed, counted, averaged – for `column4`.

```
SELECT column2, SUM(column4), COUNT(column4), AVG(column4)
FROM table2
GROUP BY column2;
```

Получим следующий результат:

```
- - ['', 0, 1, 0]
  - ['AB', 17.84567, 2, 8.922835]
  - ['CD', 10000, 1, 10000]
```

6.3.2 Complications and complex SELECTs

NULLs

Insert more rows, containing NULL values.

NULL is not the same as Lua nil; it commonly is used in SQL for unknown or not-applicable.

```
INSERT INTO table2 VALUES (1, NULL, X'4142', 5.5);
INSERT INTO table2 VALUES (0, '!!@', NULL, NULL);
INSERT INTO table2 VALUES (0, '!!!', X'00', NULL);
```

Получим следующий результат:

- The first `INSERT` will fail because NULL is not permitted for a column that was defined with a `PRIMARY KEY` clause.
- The other `INSERT` statements will succeed.

Indexes

Make a new index on column4.

There already is an index for the primary key. Indexes are useful for making queries faster. In this case, the index also acts as a constraint, because it prevents two rows from having the same values in `column4`. However, it is not an error that `column4` has multiple occurrences of NULLs.

```
CREATE UNIQUE INDEX i ON table2 (column4);
```

The result will be: «`rowcount: 1`» (no error).

Create a subset table

Make a table which will have some of the columns of `table2`, and some of the rows of `table2`.

You can do this by combining `INSERT` with `SELECT`. Then select everything in the resultant subset table.

```
CREATE TABLE table3 (column1 INTEGER, column2 VARCHAR(100), PRIMARY KEY
(column2));
INSERT INTO table3 SELECT column1, column2 FROM table2 WHERE column1 <> 2;
SELECT * FROM table3;
```

Получим следующий результат:

```
- - [-1000, '']
  - [0, '!!!']
  - [0, '!!@']
  - [1, 'AB']
  - [1, 'CD']
```

SELECT with a subquery

A subquery is a query within a query.

Here we find all the rows in `table2` whose `(column1, column2)` values are not in `table3`.

```
SELECT * FROM table2 WHERE (column1, column2) NOT IN (SELECT column1,
column2 FROM table3);
```

The result is, unsurprisingly, the single row which we deliberately excluded when we inserted the rows in the `INSERT ... SELECT` statement:

```
- - [2, 'AB', '  ', 12.34567]
```

SELECT with a join

A join is a combination of two tables. There is more than one way to do them in Tarantool: «Cartesian joins», «left outer joins», etc.

Here we're just showing the most typical case, where column values from one table match column values from another table.

```
SELECT * FROM table2, table3
    WHERE table2.column1 = table3.column1 AND table2.column2 = table3.column2
    ORDER BY table2.column4;
```

Получим следующий результат:

```
- - [0, '!!!', "\0", null, 0, '!!!']
  - [0, '!!@', null, null, 0, '!!@']
  - [-1000, '', '', 0, -1000, '']
  - [1, 'AB', 'AB', 5.5, 1, 'AB']
  - [1, 'CD', ' ', 10000, 1, 'CD']
```

6.3.3 Constraints affecting updates

CREATE TABLE, with a CHECK clause

First we make a table which includes a «constraint» that there must not be any rows containing 13 in `column2`. Then we try to insert such a row.

```
CREATE TABLE table4 (column1 INTEGER PRIMARY KEY, column2 INTEGER, CHECK
(column2 <> 13));
INSERT INTO table4 VALUES (12, 13);
```

Result: the insert fails, as it should, with the message «`error: 'CHECK constraint failed: TABLE4'`».

CREATE TABLE, with a FOREIGN KEY clause

First we make a table which includes a «constraint» that there must not be any rows containing values that do not appear in `table2`.

When we made `table2`, we specified that its «primary key» columns were `(column1, column2)`.

```
CREATE TABLE table5 (column1 INTEGER, column2 VARCHAR(100),
    PRIMARY KEY (column1),
    FOREIGN KEY (column1, column2) REFERENCES table2 (column1, column2));
INSERT INTO table5 VALUES (2,'AB');
INSERT INTO table5 VALUES (3,'AB');
```

Result:

- The first `INSERT` statement succeeds because `table3` contains a row with `[2, 'AB', ' ', 12.34567]`.
- The second INSERT statement, correctly, fails with the message «`error: FOREIGN KEY constraint failed`».

UPDATE

Due to earlier INSERT statements, these values are in `table2 column4`: `{0, NULL, NULL, 5.5, 10000, 12.34567}`. We will add 5 to every one of them except the one with 0. (Adding 5 to NULL will result in NULL, as SQL arithmetic requires.) Then we'll use `SELECT` to see what happened to `column4`.

```
UPDATE table2 SET column4 = column4 + 5 WHERE column4 <> 0;
SELECT column4 FROM table2 ORDER BY column4;
```

The result is: `{NULL, NULL, 0, 10.5, 17.34567, 10005}`.

DELETE

Due to earlier `INSERT` statements, there are now 6 rows in `table2`:

```
- - [-1000, '', '', 0]
  - [0, '!!!', "\0", null]
  - [0, '!!@', null, null]
  - [1, 'AB', 'AB', 10.5]
  - [1, 'CD', '  ', 10005]
  - [2, 'AB', '  ', 17.34567]
```

We will try to delete the last and first of these rows.

```
DELETE FROM table2 WHERE column1 = 2;
DELETE FROM table2 WHERE column1 = -1000;
SELECT COUNT(column1) FROM table2;
```

Получим следующий результат:

- The first DELETE statement causes an error message because (remember?) there's a foreign-key constraint.
- The second DELETE statement succeeds.
- The SELECT statement shows that there are now only 5 rows remaining.

ALTER TABLE, with a FOREIGN KEY clause

Now we want to make another «constraint», that there must not be any rows in table1 containing values that do not appear in table5. We couldn't do this when we created table1 because at that time table5 did not exist. But we can add constraints to existing tables with the ALTER TABLE statement.

```
ALTER TABLE table1 ADD CONSTRAINT c
    FOREIGN KEY (column1) REFERENCES table5 (column1);
DELETE FROM table1;
ALTER TABLE table1 ADD CONSTRAINT c
    FOREIGN KEY (column1) REFERENCES table5 (column1);
```

Result: the ALTER TABLE statement fails the first time because there is a row in table1, and ADD CONSTRAINT requires that the table be empty. But after we delete that row, the ALTER TABLE statement succeeds the second time. Thus we have set up a chain of references, from table1 to table5 and from table5 to table2.

Triggers

The idea of a trigger is: if a change (INSERT or UPDATE or DELETE) happens, then a further action – perhaps another INSERT or UPDATE or DELETE – will happen.

There are many variants, the one we'll illustrate here is: just after doing an update in table3, do an update in table2. We will specify this as FOR EACH ROW, so (since there are 5 rows in table3) the trigger will be activated 5 times.

```
SELECT column4 FROM table2 WHERE column1 = 2;
CREATE TRIGGER tr AFTER UPDATE ON table3 FOR EACH ROW
BEGIN UPDATE table2 SET column4 = column4 + 1 WHERE column1 = 2; END;
UPDATE table3 SET column2 = column2;
SELECT column4 FROM table2 WHERE column1 = 2;
```

Result:

- The first SELECT shows that the original value of column4 in table2 where column1 = 2 was: 17.34567.
- The second SELECT returns:

```
- - [22.34567]
```

6.3.4 Operators and functions

String operations

You can manipulate string data (usually defined with CHAR or VARCHAR data types) in many ways.

We'll illustrate here:

- the || operator for concatenation and
- the SUBSTR function for extraction.

```
SELECT column2, column2 || column2, SUBSTR(column2, 2, 1) FROM table2;
```

Получим следующий результат:

```
- - ['!!!', '!!!!!!!', '!']
  - ['!!@', '!!@!!@', '!']
  - ['AB', 'ABAB', 'B']
  - ['CD', 'CDCD', 'D']
  - ['AB', 'ABAB', 'B']
```

Number operations

You can also manipulate number data (usually defined with INTEGER or FLOAT data types) in many ways.

We'll illustrate here:

- the << operator for shift left and
- the % operator for modulo.

```
SELECT column1, column1 << 1, column1 << 2, column1 % 2 FROM table2;
```

Получим следующий результат:

```
- - [0, 0, 0, 0]
  - [0, 0, 0, 0]
  - [1, 2, 4, 1]
  - [1, 2, 4, 1]
  - [2, 4, 8, 0]
```

Ranges and limits

Tarantool can handle:

- integers anywhere in the 4-byte integer range,
- approximate-numerics anywhere in the 8-byte IEEE floating point range,
- any Unicode characters, with UTF-8 encoding and a choice of collations.

Here we will insert some such values in a new table, and see what happens when we select them, with arithmetic on a number column and ordering by a string column.

```
CREATE TABLE t6 (column1 INTEGER, column2 VARCHAR(10), column4 FLOAT,
PRIMARY KEY (column1));
INSERT INTO t6 VALUES (-1234567890, 'АБВГД', 123456.123456);
INSERT INTO t6 VALUES (+1234567890, 'GD', 1e30);
INSERT INTO t6 VALUES (10, 'FADEW?', 0.000001);
INSERT INTO t6 VALUES (5, 'ABCDEFG', NULL);
SELECT column1 + 1, column2, column4 * 2 FROM t6 ORDER BY column2;
```

The result is:

```
- - [6, 'ABCDEFG', null]
  - [11, 'FADEW?', 2e-06]
  - [1234567891, 'GD', 2e+30]
  - [-1234567889, 'АБВГД', 246912.246912]
```

Views

A view, or «viewed table», is virtual, that is, its rows aren't physically in the database, their values are calculated from other tables.

Here we'll create a view `v3` based on `table3`, then we select from it.

```
CREATE VIEW v3 AS SELECT SUBSTR(column2,1,2), column4 FROM t6 WHERE
column4 >= 0;
SELECT * FROM v3;
```

The result is:

```
- - ['АБ', 123456.123456]
  - ['FA', 1e-06]
  - ['GD', 1e+30]
```

Common table expressions

By putting `WITH` + `SELECT` in front of a `SELECT`, we can make a sort of temporary view that lasts for the duration of the statement.

Here we'll select from the sort of temporary view.

```
WITH cte AS (
             SELECT SUBSTR(column2,1,2), column4 FROM t6 WHERE column4
             >= 0)
SELECT * FROM cte;
```

Result: the same as the result we got with `CREATE VIEW` earlier:

```
- - ['АБ', 123456.123456]
  - ['FA', 1e-06]
  - ['GD', 1e+30]
```

VALUES

Tarantool can handle statements like `SELECT 55;` (select without `FROM`) like some other popular DBMSs. But it also handles the more standard statement `VALUES (expression [, expression ...]);`.

Here we'll use both styles.

```
SELECT 55 * 55, 'The rain in Spain';
VALUES (55 * 55, 'The rain in Spain');
```

The result of either statement will be:

```
- - [3025, 'The rain in Spain']
```

Metadata

What database objects have we created? We can find out about:

- tables with `SELECT * FROM "_space";`
- indexes with `SELECT * FROM "_index";`
- triggers with `SELECT * FROM "_trigger";` (These names will be familiar to old Tarantool users because we're actually selecting from NoSQL «system spaces».)

Here we will select from `_space`.

```
SELECT "id", "name", "owner", "engine" FROM "_space" WHERE "name"='TABLE3';
```

The result is (we know we will get a row because we created `table3` earlier):

```
- - [517, 'table3', 1, 'memtx']
```

6.3.5 Calling from a host language to make a big table

box.execute()

Now we will change the settings so that the console accepts statements written in Lua instead of statements written in SQL. (More ways to switch languages will exist in Tarantool clients in our next version.)

This doesn't mean we have left the SQL world though, because we can invoke SQL statements using a Lua function: `box.execute(string)`.

Here we'll switch languages, and ask to select again what's in `table3`. These statements must be entered separately.

```
tarantool> \set language lua
tarantool> box.execute([[SELECT * FROM table3;]]);
```

Showing both the statements and the results:

```
tarantool> \set language lua
---
...
tarantool> box.execute([[SELECT * FROM table3;]]);
---
- - [-1000, '']
  - [0, '!!!']
  - [0, '!!@']
  - [1, 'AB']
  - [1, 'CD']
...
```

Create a million-row table

We've illustrated a lot of SQL, but does it scale? To answer that, let's make a bigger table.

For this we are going to use Lua. We will not explain the Lua, because that's in the Lua section of the Tarantool manual. Just copy-and-paste these instructions and wait for about a minute.

```
box.execute("CREATE TABLE tester (s1 INT PRIMARY KEY, s2 VARCHAR(10))");

function string_function()
   local random_number
   local random_string
   random_string = ""
   for x = 1,10,1 do
     random_number = math.random(65, 90)
     random_string = random_string .. string.char(random_number)
   end
   return random_string
end;

function main_function()
   local string_value, t, sql_statement
   for i = 1,1000000,1 do
     string_value = string_function()
     sql_statement = "INSERT INTO tester VALUES (" .. i .. ",'" .. string_value .. "')"
     box.execute(sql_statement)
   end
end;
start_time = os.clock();
main_function();
end_time = os.clock();
'insert done in ' .. end_time - start_time .. ' seconds';
```

The result is: you now have a table with a million rows, with a message saying «`insert done in 88.570578 seconds`».

Select from a million-row table

Now that we have something a bit larger to play with, let's see how long it takes to SELECT.

The first query we'll do will automatically go via an index, because `s1` is the primary key.

The second query we'll do will not go via an index, because for `s2` we didn't say `CREATE INDEX xxxx ON tester (s2);`.

```
box.execute([[SELECT * FROM tester WHERE s1 = 73446;]]);
box.execute([[SELECT * FROM tester WHERE s2 LIKE 'QFML%';]]);
```

The result is:

- the first statement will finish instantaneously,
- the second statement will be noticeably slower but still a fraction of a second.

Cleanup and exit

We're done. We've shown that Tarantool 2.1 has a very reasonable subset of SQL, and it works.

The rest of these commands will simply destroy all the database objects that were created so that you can do the demonstration again. These statements must be entered separately.

```
tarantool> \set language sql
tarantool> DROP TABLE tester;
tarantool> DROP TABLE table1;
tarantool> DROP VIEW v3;
tarantool> DROP TRIGGER tr;
tarantool> DROP TABLE table5;
tarantool> DROP TABLE table4;
tarantool> DROP TABLE table3;
tarantool> DROP TABLE table2;
tarantool> DROP TABLE t6;
tarantool> \set language lua
tarantool> os.exit();
```

6.4 Улучшаем работу MySQL с помощью Tarantool

MySQL-репликация — это одна из важнейших функций Tarantool'а. Она позволяет ускорить работу с существующей базой данных MySQL, а также значительно расширить базу за счет горизонтального масштабирования. Даже если вы не заинтересованы в расширении, сама замена существующих реплик на Tarantool может сэкономить вам деньги, потому что Tarantool более эффективно работает с ядром, чем MySQL. Отзыв компании, которая широко использует репликацию в Tarantool, можно прочитать здесь.

Примечания:

- если вы столкнетесь с какими-либо проблемами в отношении основ работы с Tarantool'ом, то можете обратиться к *руководству для начинающих* или *описанию модели данных*.
- приведенные ниже инструкции относятся к **CentOS 7.5** и **MySQL 5.7**. Также предполагается, что у вас установлен модуль systemd и вы работаете с существующей MySQL-установкой.
- во время выполнения практикума вам может пригодиться журнал для устранения неполадок `replicatord.log`, который находится в `/var/log`. Также обратите внимание на журнал экземпляра `example.log` в `/var/log/tarantool`.

Итак, начнем.

1. Для начала установим необходимые пакеты для CentOS:

   ```
   yum -y install git ncurses-devel cmake gcc-c++ boost boost-devel wget unzip nano bzip2 mysql-
   ↪devel mysql-lib
   ```

2. Затем клонируем пакет репликации Tarantool-MySQL из GitHub:

   ```
   git clone https://github.com/tarantool/mysql-tarantool-replication.git
   ```

3. Сейчас мы можем собрать репликатор с помощью cmake:

   ```
   cd mysql-tarantool-replication
   git submodule update --init --recursive
   cmake .
   make
   ```

4. Репликатор будет работать в виде демона systemd под названием replicatord, поэтому давайте отредактируем его служебный файл systemd, а именно `replicatord.service`, в репозитории

```
nano replicatord.service
```

Измените следующую строку:

```
ExecStart=/usr/local/sbin/replicatord -c /usr/local/etc/replicatord.cfg
```

Замените расширение `.cfg` на `.yml`:

```
ExecStart=/usr/local/sbin/replicatord -c /usr/local/etc/replicatord.yml
```

5. Затем скопируем некоторые файлы из репозитория replicatord в другие места:

```
cp replicatord /usr/local/sbin/replicatord
cp replicatord.service /etc/systemd/system
```

6. Сейчас откроем консоль MySQL и создадим пример базы данных (в зависимости от текущих настроек, разумеется, вы можете не быть пользователем root):

```
mysql -u root -p
CREATE DATABASE menagerie;
QUIT
```

7. Далее получим образец данных из MySQL, которые поместим в корневую директорию, а затем установим из терминала:

```
cd
wget http://downloads.mysql.com/docs/menagerie-db.zip
unzip menagerie-db.zip
cd menagerie-db
mysql -u root -p menagerie < cr_pet_tbl.sql
mysql -u root -p menagerie < load_pet_tbl.sql
mysql menagerie -u root -p < ins_puff_rec.sql
mysql menagerie -u root -p < cr_event_tbl.sql
```

8. Откроем MySQL-консоль и обработаем данные для использования с репликатором Tarantool'а (добавляем идентификатор, меняем имя поля во избежание конфликта и сокращаем количество полей; обратите внимание, что для реальных данных этот шаг потребует большого количества настроек):

```
mysql -u root -p
USE menagerie;
ALTER TABLE pet ADD id INT PRIMARY KEY AUTO_INCREMENT FIRST;
ALTER TABLE pet CHANGE COLUMN 'name' 'name2' VARCHAR(255);
ALTER TABLE pet DROP sex, DROP birth, DROP death;
QUIT
```

9. Сейчас образец данных готов, и нам необходимо отредактировать конфигурационный файл MySQL для использования с репликатором.

```
cd
nano /etc/my.cnf
```

Обратите внимание, что ваш `my.cnf` для MySQL может находиться в другом месте. Задайте:

```
[mysqld]
binlog_format = ROW
```

(continues on next page)

(продолжение с предыдущей страницы)

```
server_id = 1
log-bin = mysql-bin
interactive_timeout = 3600
wait_timeout = 3600
max_allowed_packet = 32M
socket = /var/lib/mysql/mysql.sock
bind-address = 127.0.0.1

[client]
socket = /var/lib/mysql/mysql.sock
```

10. После выхода из nano, перезапустим mysqld:

```
systemctl restart mysqld
```

11. Далее установим Tarantool и настроим спейсы для репликации. Перейдите на страницу загрузки и установите Tarantool, следуя инструкциям.

12. Сейчас напишем стандартную Tarantool-программу путем редактирования Lua-примера, который поставляется вместе с Tarantool'ом:

```
cd
nano /etc/tarantool/instances.available/example.lua
```

13. Полностью заменим содержимое файла следующим текстом:

```
box.cfg {
    listen = 3301;
    memtx_memory = 128 * 1024 * 1024; -- 128Mb
    memtx_min_tuple_size = 16;
    memtx_max_tuple_size = 128 * 1024 * 1024; -- 128Mb
    vinyl_memory = 128 * 1024 * 1024; -- 128Mb
    vinyl_cache = 128 * 1024 * 1024; -- 128Mb
    vinyl_max_tuple_size = 128 * 1024 * 1024; -- 128Mb
    vinyl_write_threads = 2;
    wal_mode = "none";
    wal_max_size = 256 * 1024 * 1024;
    checkpoint_interval = 60 * 60; -- one hour
    checkpoint_count = 6;
    force_recovery = true;

     -- 1 - SYSERROR
     -- 2 - ERROR
     -- 3 - CRITICAL
     -- 4 - WARNING
     -- 5 - INFO
     -- 6 - VERBOSE
     -- 7 - DEBUG
     log_level = 7;
     too_long_threshold = 0.5;
 }

box.schema.user.grant('guest','read,write,execute','universe')

local function bootstrap()

    if not box.space.mysqldaemon then
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        s = box.schema.space.create('mysqldaemon')
        s:create_index('primary',
        {type = 'tree', parts = {1, 'unsigned'}, if_not_exists = true})
    end

    if not box.space.mysqldata then
        t = box.schema.space.create('mysqldata')
        t:create_index('primary',
        {type = 'tree', parts = {1, 'unsigned'}, if_not_exists = true})
    end

end

bootstrap()
```

Чтобы понять, что здесь происходит, лучше всего обратиться к предыдущим статьям в серии Tarantool 101 или изучить наше *руководство для начинающих*.

14. Сейчас необходимо создать символическую ссылку из `instances.available` (доступные экземпляры) на директорию под названием `instances.enabled` (активные экземпляры – похоже на NGINX). В `/etc/tarantool` выполните следующую команду:

```
mkdir instances.enabled
ln -s /instances.available/example.lua instances.enabled
```

15. Далее мы можем запустить Lua-программу с помощью `tarantoolctl` (надстройки для systemd):

```
tarantoolctl start example.lua
```

16. Сейчас перейдем на наш экземпляр Tarantool'a, где можно проверить, что необходимые спейсы были успешно созданы:

```
tarantoolctl enter example.lua
```

```
tarantool> box.space._space:select()
```

В самом низу вы увидите спейсы «mysqldaemon» и «mysqldata». Затем выйдите с помощью «CTRL+C».

17. Сейчас, после настройки MySQL и Tarantool'a, можно перейти к конфигурации репликатора. Для начала поработаем с `replicator.yml` в основной директории `tarantool-mysql-replication`:

```
nano replicatord.yml
```

Полностью замените содержимое файла на следующее, убедившись, что вы добавили свой пароль для MySQL и указали правильного пользователя:

```
mysql:
    host: 127.0.0.1
    port: 3306
    user: root
    password:
    connect_retry: 15 # seconds

tarantool:
    host: 127.0.0.1:3301
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    binlog_pos_space: 512
    binlog_pos_key: 0
    connect_retry: 15 # seconds
    sync_retry: 1000 # milliseconds

mappings:
    - database: menagerie
      table: pet
      columns: [ id, name2, owner, species ]
      space: 513
      key_fields:  [ 0 ]
      # insert_call: function_name
      # update_call: function_name
      # delete_call: function_name
```

18. Сейчас необходимо скопировать replicatord.yml в место, где systemd будет искать его:

```
cp replicatord.yml /usr/local/etc/replicatord.yml
```

19. Теперь запустим репликатор:

```
systemctl start replicatord
```

Сейчас мы можем перейти на экземпляр Tarantool'а и выполнить выборку из спейса "mysqldata". Увидим реплицируемые данные из MySQL:

```
tarantoolctl enter example.lua
```

```
tarantool> box.space.mysqldata:select()
---
- - [1, 'Fluffy', 'Harold', 'cat']
  - [2, 'Claws', 'Gwen', 'cat']
  - [3, 'Buffy', 'Harold', 'dog']
  - [4, 'Fang', 'Benny', 'dog']
  - [5, 'Bowser', 'Diane', 'dog']
  - [6, 'Chirpy', 'Gwen', 'bird']
  - [7, 'Whistler', 'Gwen', 'bird']
  - [8, 'Slim', 'Benny', 'snake']
  - [9, 'Puffball', 'Diane', 'hamster']
```

20. Наконец, внесем запись в MySQL, а затем вернемся в Tarantool, чтобы убедиться, что она реплицирована. Итак, сначала выйдем из экземпляра Tarantool'а с помощью `CTRL-C`, а затем введем:

```
mysql -u root -p
USE menagerie;
INSERT INTO pet(name2, owner, species) VALUES ('Spot', 'Brad', 'dog');
QUIT
```

Вернувшись в терминал, введите:

```
tarantoolctl enter example.lua
```

```
tarantool> box.space.mysqldata:select()
```

Вы увидите реплицируемые данные в Tarantool'е!

6.5 Практические задания по *libslave*

`libslave` представляет собой библиотеку C++ для считывания изменений данных, внесенных с помощью MySQL, а также – опционально – для записи их в базу данных Tarantool'a. Она выступает в качестве ведомого в схеме репликации. Сервер MySQL записывает информацию об изменении данных в бинарный журнал и передает ее на любой клиент, который запрашивает: «Хочу увидеть всю информацию, начиная с этого файла и этой записи, безостановочно». Таким образом, библиотека `libslave`, прежде всего, используется для создания реплик базы данных Tarantool'a (намного быстрее, чем используя традиционный ведомый сервер MySQL) и для отслеживания изменений данных, чтобы они были пригодны для поиска.

Здесь мы не будем подробно рассматривать библиотеку – информация есть в документации по API. Мы лишь дадим упражнение: минимальная программа с использованием библиотеки.

Примечание: Используйте тестовый сервер. Не используйте боевой сервер.

ШАГ 1: Убедитесь в наличии следующего:

- последняя версия Linux (например, Ubuntu версии 14.04 не подойдет),
- сервер MySQL версии 5.6 или 5.7 (MariaDB не подойдет),
- пакет программ для разработки клиента MySQL. Например, на Ubuntu можно загрузить его с помощью следующей команды:

```
$ sudo apt-get install mysql-client-core-5.7
```

ШАГ 2: Установите `libslave`.

Рекомендуется источник по ссылке https://github.com/tarantool/libslave/. Загрузки включают в себя только исходный код.

```
$ sudo apt-get install libboost-all-dev
$ cd ~
$ git clone https://github.com/tarantool/libslave.git tarantool-libslave
$ cd tarantool-libslave
$ git submodule init
$ git submodule update
$ cmake .
$ make
```

Если система выдаст сообщение с ошибкой со словом «vector», отредактируйте `field.h`, добавив следующую строку:

```
#include <vector>
```

ШАГ 3: Запустите сервер MySQL. В командной строке добавьте соответствующие коммутаторы для выполнения репликации. Например:

```
$ mysqld --log-bin=mysql-bin --server-id=1
```

ШАГ 4: Для целей данного упражнения, предполагаем, что у вас есть:

- пользователь «root» с паролем «root» с правами,
- тестовая база данных «test» с тестовой таблицей под названием «test»,
- бинарный журнал под названием «mysql-bin»,

- сервер с идентификатором 1.

Значения заданы в программе, хотя программу, конечно, можно изменить – посмотреть настройки несложно.

ШАГ 5: Обратите внимание на программу:

```
#include <unistd.h>
#include <iostream>
#include <sstream>
#include "Slave.h"
#include "DefaultExtState.h"

slave::Slave* sl = NULL;

void callback(const slave::RecordSet& event) {
    slave::Position sBinlogPos = sl->getLastBinlogPos();
    switch (event.type_event) {
    case slave::RecordSet::Update: std::cout << "UPDATE" << "\n"; break;
    case slave::RecordSet::Delete: std::cout << "DELETE" << "\n"; break;
    case slave::RecordSet::Write:  std::cout << "INSERT" << "\n"; break;
    default: break;
    }
}

bool isStopping()
{
    return 0;
}

int main(int argc, char** argv)
{
    slave::MasterInfo masterinfo;
    slave::Position position("mysql-bin", 0);
    masterinfo.conn_options.mysql_host = "127.0.0.1";
    masterinfo.conn_options.mysql_port = 3306;
    masterinfo.conn_options.mysql_user = "root";
    masterinfo.conn_options.mysql_pass = "root";
    bool error = false;
    try {
        slave::DefaultExtState sDefExtState;
        slave::Slave slave(masterinfo, sDefExtState);
        sl = &slave;
        sDefExtState.setMasterPosition(position);
        slave.setCallback("test", "test", callback);
        slave.init();
        slave.createDatabaseStructure();
        try {
            slave.get_remote_binlog(isStopping);
        } catch (std::exception& ex) {
            std::cout << "Error reading: " << ex.what() << std::endl;
            error = true;
        }
    } catch (std::exception& ex) {
        std::cout << "Error initializing: " << ex.what() << std::endl;
        error = true;
    }
    return 0;
}
```

Всё лишнее почистили, чтобы можно было ясно увидеть, как это работает. В начале функции `main()` есть некоторые настройки, используемые для установки соединения – хост, порт, пользователь, пароль. Затем есть вызов инициализации с именем файла бинарного журнала = «mysql-bin». Обратите особое внимание на оператор `setCallback`, который передает имя базы данных = «test», имя таблицы = «test» и адрес функции обратного вызова = callback. Программа войдет в цикл и будет вызывать эту функцию обратного вызова. Посмотрите, как на ранних этапах программы функция обратного вызова выводит «UPDATE», «DELETE» или «INSERT» в зависимости от переданных данных.

ШАГ 5: Поместите программу в директорию `tarantool-libslave` и назовите ее `example.cpp`.

ШАГ 6: Выполните компиляцию и сборку:

```
$ g++ -I/tarantool-libslave/include example.cpp -o example libslave_a.a -ldl -lpthread
```

Примечание: Замените `tarantool-libslave/include` на полное имя директории.

Обратите внимание, что имя статической библиотеки – `libslave_a.a`, а не `libslave.a`.

ШАГ 7: Выполните:

```
$ ./example
```

Результат нет – программа в цикле ожидает, пока сервер MySQL запишет данные в бинарный журнал репликации.

ШАГ 8: Запустите клиентскую программу MySQL – подойдет любая клиентская программа. Введите следующие операторы:

```
USE test
INSERT INTO test VALUES ('A');
INSERT INTO test VALUES ('B');
DELETE FROM test;
```

Проверьте, что происходит в выводе программы `example.cpp` – отображается следующее:

```
INSERT
INSERT
DELETE
DELETE
```

Репликация является построчной, поэтому видим DELETE два раза – потому что есть две строки.

В результате выполнения упражнения видим:

- можно собрать библиотеку, а
- программы, которые используют библиотеку, могут получить доступ ко всему, что сохраняет сервер MySQL.

Более подробную информацию и примеры использования см. ниже:

- Загрузить нашу версию `libslave` можно по ссылке: https://github.com/tarantool/libslave
- Ответвление сделано из версии по ссылке (с другим файлом README): https://github.com/vozbu/libslave/wiki/API
- Статья How to speed up your MySQL with replication to in-memory database (на английском)
- Статья Репликация из MySQL в Tarantool

- Статья Асинхронная репликация без цензуры

ГЛАВА 7

Примечания к версиям

Примечания к версиям содержат краткое описание значимых изменений в следующих версиях Tarantool'a: *2.2.1*, *2.1.2*, *2.1.1*, *2.0.4*, *1.10.4*, *1.10.3*, *1.10.2*, *1.9.0*, *1.7.6*, *1.7.5*, *1.7.4*, *1.7.3*, *1.7.2*, *1.7.1*, *1.6.9*, *1.6.8*, and *1.6.6*.

Более мелкие изменения и исправления дефектов указаны в отчетах о выпущенных стабильных релизах (milestone = closed) на GitHub.

7.1 Version 2.x

Tarantool 2.x is backward compatible with Tarantool 1.10.x in binary data layout, client-server protocol and replication protocol. You can *upgrade* using the `box.schema.upgrade()` procedure.

Release 2.2.1

Release type: beta. Release date: 2019-08-02.

Announcement: https://github.com/tarantool/tarantool/releases/tag/2.2.1.

This is a *beta* version of the 2.2 series. The label «beta» means we have no critical issues and all planned features are there.

The goal of this release is to introduce new indexing features, extend SQL feature set, and improve integration with the core.

Изменения или добавления функциональности:

- (SQL) ALTER now allows to add a constraint:

```
CREATE TABLE t2 (id INT PRIMARY KEY);
ALTER TABLE t2 ADD CONSTRAINT ck CHECK(id  > 0);
```

- (SQL) CHECK constraints are validated during DML operations performed from the Lua land:

```
s = box.schema.space.create('withdata')
pk = s:create_index('pk')
s:format({{'idx', 'number'}})
s:create_check_constraint('le10', '"idx" < 10')
```

```
tarantool> s:insert({11})
---
- error: 'Check constraint failed ''le10'': "idx" < 10'
...
```

- (SQL) New *SQL types* introduced: VARBINARY, UNSIGNED, and *BOOLEAN*.
- (SQL) CREATE TABLE statement (and all other data definition statements) are now truly transactional.
- (SQL) SQL now uses Tarantool diagnostics API to set errors, so error reporting now provides an error code in addition to error message.
- (SQL) Multiple improvements to the type system to make it more consistent.
- (SQL) Added aliases for LENGTH() from ANSI SQL: CHAR_LENGTH() and CHARACTER_LENGTH().
- (SQL) It is possible to use HAVING without GROUP BY.
- (Server) New fixed point type (DECIMAL) introduced to Tarantool:

```
decimal = require('decimal')
tarantool> a = decimal.new('123.456789')
---
...
tarantool> decimal.precision(a)
---
- 9
...
tarantool> decimal.scale(a)
---
- 6
...
tarantool> decimal.round(a, 4)
---
- '123.4568'
...
```

- (Server) Multikey index support:

```
-- Multikey indexes (for memtx tree & vinyl);
-- cannot be primary; may be non-unique
s = box.schema.space.create('clients', {engine = 'vinyl'})
pk = s:create_index('pk')
phone_type = s:create_index('phone_type', {
    unique = false,
    parts = {{'[3][*].type', 'str'}}})

s:insert({1, 'James',
         {{type = 'home', number = '999'},
          {type = 'work', number = '777'}
         }})
s:insert({2, 'Bob',
```

(continues on next page)

(продолжение с предыдущей страницы)

```
          {{type = 'work', number = '888'}}})
s:insert({3, 'Alice', {{type = 'home', number = '333'}}})
```

```
tarantool> phone_type:select('work')
---
- - [1, 'James', [{'type': 'home', 'number': '999'},
                  {'type': 'work', 'number': '777'}]]
  - [2, 'Bob', [{'type': 'work', 'number': '888'}]]
...
```

- (Server) Now it is possible to make functions persistent:

```
box.schema.func.create('summarize',
                       {body = [[function(a,b) return a+b end]],
                       is_deterministic = true})
```

```
tarantool> box.func.summarize
- aggregate: none
  returns: any
  exports:
    lua: true
    sql: false
  id: 66
  is_sandboxed: false
  setuid: false
  is_multikey: false
  is_deterministic: true
  body: function(a,b) return a+b end
  name: summarize
  language: LUA

tarantool> box.func.summarize:call({1, 2})
---
- 3
...
```

- (Server) Functional indexes implemented:

```
-- Functional multikey indexes: define is_multikey = true
-- in function definition and return a table of keys from function
lua_code = [[function(tuple)
                local address = string.split(tuple[2])
                local ret = {}
                for _, v in pairs(address) do table.insert(ret, {utf8.upper(v)}) end
                return ret
             end]]
box.schema.func.create('addr_extractor', {body = lua_code,
                                          is_deterministic = true,
                                          is_sandboxed = true,
                                          opts = {is_multikey = true}})
s = box.schema.space.create('withdata')
pk = s:create_index('name', {parts = {1, 'string'}})
idx = s:create_index('addr', {unique = false, func = box.func.addr_extractor.id, parts = {{1,
↪'string', collation = 'unicode_ci'}}})

s:insert({"James", "SIS Building Lambeth London UK"})
```

(continues on next page)

(продолжение с предыдущей страницы)

```
s:insert({"Sherlock", "221B Baker St Marylebone London NW1 6XE UK"})
```

```
tarantool>  idx:select('Sis')
---
- - ['James', 'SIS Building Lambeth London UK']
...
```

- Partial core dumps, which are now on by default. It is now possible to avoid dumping tuples at all during core dump.
- Data definition statements, such as create or alter index, which do not yield, can now be used in a transaction. This in practice includes all statements except creating an index on a non-empty space, or changing a format on a non-empty space.
- It is now possible to set a sequence not only for the first part of the index:

```
s.index.pk:alter{sequence = {field = 2}}
```

- Allow to call `box.session.exists()` and `box.session.fd()` without any arguments.
- New function introduced to get an index key from a tuple:

```
s = box.schema.space.create('withdata')
pk = s:create_index('pk')
sk = s:create_index('sk', {parts = {
      {2, 'number', path = 'a'},
      {2, 'number', path = 'b'}}})
s:insert{1, {a = 1, b = 1}}
s:insert{2, {a = 1, b = 2}}
s:insert{3, {a = 3, b = 3}}
sk:select(2)

key_def_lib = require('key_def')
key_def = key_def_lib.new(pk.parts)
for _, tuple in sk:pairs({1}) do
    local key = key_def:extract_key(tuple)
    pk:delete(key)
end
s:select()
```

- (Engines) New protocol (called *SWIM*) implemented to keep a table of cluster members.
- (Engines) Removed yields from Vinyl DDL on commit triggers.
- (Engines) Improved performance of SELECT-s on memtx spaces. The drawback is that now every memtx-tree tuple consumes extra 8 bytes for a search hint.
- (Engines) Indexes of memtx spaces are now built in background fibers. This means that we do not block the event loop during index build anymore.
- Replication applier now can apply transactions which were concurrent on the master concurrently on replica. This dramatically improves replication peak performance, from ~50K writes per second to 200K writes per second and higher on a single instance.
- Transaction boundaries introduced to replication protocol. This means that Tarantool replication is now transaction-safe, and also reduces load on replica write ahead log in case the master uses a lot of multi-statement transactions.
- Tuple access by field name for `net.box`:

```
box.cfg{listen = 3302}
box.schema.user.grant('guest','read, write, execute', 'space')
box.schema.user.grant('guest', 'create', 'space')
box.schema.create_space("named", {format = {{name = "id"}}})
box.space.named:create_index('id', {parts = {{1, 'unsigned'}}})
box.space.named:insert({1})

require('net.box').connect('localhost', 3302).space.named:get(1).id
```

- Cluster id check is now the slave's responsibility.
- It is now possible to set the output format to Lua instead of YAML in the *interactive console*.
- Multiple new collations added. New collations follow this naming pattern:

```
unicode_<locale>_<strength>
```

  Three strengths are used:

  – Primary - «s1"
  – Secondary - «s2»
  – Tertiary - «s3»

  The following list contains so-called «stable» collations - the ones whose sort order doesn't depend on the ICU version:

```
unicode_am_s3
unicode_fi_s3
unicode_de__phonebook_s3
unicode_haw_s3
unicode_he_s3
unicode_hi_s3
unicode_is_s3
unicode_ja_s3
unicode_ko_s3
unicode_lt_s3
unicode_pl_s3
unicode_si_s3
unicode_es_s3
```

- New function `utime()` introduced to the `fio` module.
- *Merger* for tuples streams added.

Release 2.1.2

Release type: stable. Release date: 2019-04-05.

Announcement: https://github.com/tarantool/tarantool/releases/tag/2.1.2.

This is the first *stable* release in the 2.x series.

The goal of this release is to significantly extend SQL support and increase stability.

Изменения или добавления функциональности:

- (SQL) `box.sql.execute()` replaced with *box.execute()*. It now works just like `netbox.execute()`: returns result set metadata, row count, etc. E.g.:

```
box.execute("CREATE TABLE person(id INTEGER PRIMARY KEY, birth_year INT)")
---
- row_count: 1
...
box.execute("SELECT birth_year FROM person")
---
- metadata:
  - name: birth_year
    type: INTEGER
  rows:
  - [1983]
  - [1984]
...
```

- (SQL) Type system was *significantly refactored*.
- (SQL) There are cases in SQL when it is possible to do Tarantool's update operation for UPDATE statement, instead of doing delete + insert. However, there are cases where SQL semantics is too complex. E.g.:

```
CREATE TABLE file (id INT PRIMARY KEY, checksum INT);
INSERT INTO stock VALUES (1, 3),(2, 4),(3,5);
CREATE UNIQUE INDEX i ON file (checksum);
SELECT * FROM file;
-- [1, 3], [2, 4], [3, 5]
UPDATE OR REPLACE file SET checksum = checksum + 1;
SELECT * FROM stock;
-- [1, 4], [3, 6]
```

  I.e. [1, 3] tuple is updated as [1, 4] and have replaced tuple [2, 4]. This logic is implemented by preventive tuple deletion from all corresponding indexes in SQL.

- (SQL) Now SQL's integer type is stored as integer in space's format. It was stored as scalar before, which made comarisons slow.
- (SQL) It is now possible to define a constraint *within column definition*. E.g.:

```
CREATE TABLE person (id INT PRIMARY KEY, age INT, CHECK (age > 10));
```

- (SQL) Syntax for the pragma `pragma index_info` is now unified with `table_info`. E.g. to get information on index `age_index` of table `person` you can write:

```
pragma index_info(person.age_index);
```

- (Server) It is now possible to index a field specified using JSON. E.g.:

```
person = box.schema.create_space("person")
name_idx = person:create_index('name', {parts = {{'[2]fname', 'str'}, {'[2]sname', 'str'}}})
person:insert({1, {fname='James', sname='Bond'}, {town='London', country='GB', organization=
↪'MI6'}})
```

- (Server) In case of out of space event, Tarantool is now allowed to delete backup WAL files not needed for recovery from the last checkpoint.
- (Server) Add support for *tarantoolctl rocks pack / unpack* subcommands. The subcommands are used to create / deploy binary rock distributions.
- (Server) `string.rstrip` and `string.lstrip` should accept symbols to strip. Add optional „chars“ parameter for specifying the unwanted characters. E.g.:

```
local chars = "#\0"
str = "##Hello world!#"
print(string.strip(str, chars)) -- "Hello world!"
```

- (Server) *on_shutdown* trigger added. It may be set in a way similar to `space:on_replace` triggers:

```
box.ctl.on_shutdown(new_trigger, old_trigger)
```

- (Server) *on_schema_init* trigger added. It may be set before the first call to `box.cfg()` and is fired during `box.cfg()` before user data recovery start. To set the trigger, say:

```
box.ctl.on_schema_init(new_trig, old_trig)
```

- (Server) A new option for the snapshot daemon, *box.cfg.checkpoint_wal_threshold*, allows to limit the maximum disk size of maintained WALs. Once the configured threshold is exceeded, the WAL thread notifies the checkpoint daemon that it's time to make a new checkpoint and delete old WAL files.
- (Server) New types of *privileges* – to create, alter and drop space – were introduced. In order to create, drop or alter space or index, you should have a corresponding privilege. E.g.:

```
box.schema.user.create("optimizer", { password  = 'secret' })
box.schema.user.grant("optimizer", "alter", "space")
person = box.schema.space.create("person")
box.session.su("optimizer")
i = s:create_index("primary") -- success
s:insert{1} -- fail
s:select{} -- fail
s:drop() -- fail
```

  Notice the incompatible change: Tarantool 1.10 requires read/write/execute privileges on an object to allow create, drop or alter. These privileges are no longer sufficient in 2.1. To remedy the problem, Tarantool 2.1 automatically grants create/drop/alter privileges on an object if a user has read/write/execute privileges on it during schema upgrade. But old scripts may stop working if read/write/execute is granted **after** schema upgrade.

  Additionally, create/drop/alter privileges are already supported in 1.10, which also supports the old semantics of read/write/execute. You are encouraged to grant new privileges in 1.10 before upgrade and modify your scripts.

Release 2.1.1

Release type: beta. Release date: 2018-11-14.

Announcement: https://github.com/tarantool/tarantool/releases/tag/2.1.1.

This release resolves all major bugs since 2.0.4 alpha and extends Tarantool's SQL feature set.

Release 2.0.4

Release type: alpha. Release date: 2018-02-15.

Announcement: https://github.com/tarantool/tarantool/releases/tag/2.0.4.

This is a successor of the 1.8.x releases. It improves the overall stability of the SQL engine and has some new features.

Изменения или добавления функциональности:

- Added support for SQL collations by incorporating libICU character set and collation library.
- IPROTO interface was extended to support SQL queries.

- `net.box` subsystem was extended to support SQL queries.
- Enabled `ANALYZE` statement to produce correct results, necessary for efficient query plans.
- Enabled savepoints functionality. `SAVEPOINT` statement works w/o issues.
- Enabled `ALTER TABLE ... RENAME` statement.
- Improved rules for identifier names: now fully consistent with Lua frontend.
- Enabled support for triggers; trigger bodies now persist in Tarantool snapshots and survive server restart.
- Significant performance improvements.

7.2 Версия 1.10

Версия 1.10.4

Тип версии: стабильная (lts). Дата выхода: 2019-09-26. Тег: 1-10-4.

Сообщение: https://github.com/tarantool/tarantool/releases/tag/1.10.4.

Общие сведения

1.10.4 представляет собой очередную *стабильную (lts)* версию в серии 1.10. Пометка «стабильная» означает, что некоторые системы в течение определенного времени успешно отработали в производственной среде без известных сбоев, ненадежных результатов и прочих неисправностей.

Данная версия содержит около 50 исправлений по сравнению с версией 1.10.3.

Совместимость

Tarantool 1.10.x обратно совместим с Tarantool 1.9.x в том, что касается структуры бинарных данных, клиент-серверного протокола и протокола репликации. Чтобф воспользоваться новыми функциями серии 1.10.x, *обновите* версию 1.9 с помощью процедуры `box.schema.upgrade()`.

Изменения или добавления функциональности

- (Движки) Улучшена запись в журнал о событиях начала/окончания процесса создания дампа. При запуске создания дампа записывается объем памяти, для которого создается дамп, предполагаемая скорость создания дампа, ETA, а также последняя скорость записи. По окончании создания дампа записывается зарегистрированная скорость создания дампа, а также размер дампа и длительность.
- (Движки) Поиск ключа в потоке чтения. Если ключ не обнаружен в кэше кортежа, забираем его из файла забега. В этом случае чтение с диска и распаковка страницы выполняется потоком чтения, однако поиск ключа на выбранной странице все еще выполняется потоком TX. Поскольку страницы являются неизменяемыми, это может сделать поток чтения, что позволит нам сэкономить ценные циклы ЦП для TX. Проблема 4257.
- (Ядро) Улучшена работа *box.stat.net*. Проблема 4150.
- (Ядро) Добавлен статус `idle` (простаивание) для downstream в `box.info`. Когда передается строка, обновляется значение `last_row_time` с текущим временем. Когда идет вызов `box.info()`, значение `idle` задается как `current_time` (текущее время) - `last_row_time` (время передачи последней строки).
- (Репликация) Вывод поврежденных данных при ошибке декодирования. Улучшена запись строк в журнал. Вывод заголовка построчно, 16 байтов в строке, формат вывода соответствует выводу `xxd`:

```
[001] 2019-04-05 18:22:46.679 [11859] iproto V> Got a corrupted row:
[001] 2019-04-05 18:22:46.679 [11859] iproto V> 00000000: A3 02 D6 5A E4 D9 E7 68 A1 53 8D 53␣
↪60 5F 20 3F
[001] 2019-04-05 18:22:46.679 [11859] iproto V> 00000010: D8 E2 D6 E2 A3 02 D6 5A E4 D9 E7 68␣
↪A1 53 8D 53
```

- (Lua) Добавлен тип операций в *параметры триггера*. Например, теперь функция с триггером может выглядеть следующим образом:

```
function before_replace_trig(old, new, space_name, op_type)
    if op_type == 'INSERT' then
        return old
    else
        return new
    end
end
```

  Проблема 4099.

- (Lua) Добавлены `debug.sourcefile()` и `debug.sourcedir()` (а также ускоренные методы `debug.__file__` и `debug.__dir__`) для обнаружения местонахождения текущего исходного файла на Lua. Часть проблемы 4193.
- (HTTP-клиент) Добавлена опция `max_total_connections` в дополнение к `max_connections`, что позволяет более тонко настраивать кэш соединения `libcurl`. Общее число соединений больше не ограничено постоянным значением по умолчанию, а используется значение по умолчанию из“libcurl“, что масштабирует предел в зависимости от количества обработчиков. Проблема 3945.

Исправленные ошибки

- (Vinyl) Исправлен отказ в *vy_tx_handle_deferred_delete*. Проблема 4294.
- (Vinyl) Не очищать удаленные забеги из vylog при слиянии. Отдельные элементы из проблемы 4218.
- (Vinyl) Не управлять загрузкой DDL. Проблема 4238.
- (Vinyl) Исправить потерю при коммите отложенных предложений DELETE. Проблема 4248.
- (Vinyl) Исправить допустимость неопределенного значения при восстановлении предложения из дампа. Проблема 4222.
- (Vinyl) Сбросить уровень дампа после обновления предела загрузки памяти. Проблема 3864.
- (Vinyl) Применять пессимистический прогноз скорости записи при установленном уровне дампа. Проблема 4166.
- (Vinyl) Исправить сбой при удалении спейса во время чтения из него через space.get. Проблема 4109.
- (Vinyl) Исправить сбой во время создания индекса. Проблема 4152.
- (Vinyl) Не сжимать забеги L1. Проблема 2389.
- (Vinyl) Считать операторы, пропущенные при чтении.
- (Vinyl) Принять во внимание поиск первичного ключа при учете задержек.
- (Vinyl) Исправить зависание `vy_range_update_compaction_priority`.
- (Vinyl) Освобождать область при фиксации vylog вместо сброса и очищать после выделения заместителя оператора.
- (Vinyl) Еще увеличить ограничение на количество открытых файлов в файле `systemd`.

- (Vinyl) Увеличить минимальный размер диапазона до 128 Мбайт
- (Memtx) Отменить поток создания контрольных точек на выходе. Проблема 4170.
- (Ядро) Исправить отказ для обновления с пустым кортежем. Проблема 4041.
- (Ядро) Исправить использование освобожденной памяти в `space_truncate`. Проблема 4093.
- (Ядро) Исправить ошибку при изменении индекса с последовательностью. Проблема 4214.
- (Ядро) Выявить новый случай неправильного json-пути. Проблема 4419.
- (Ядро) Исправить аутентификацию с пустым паролем. Проблема 4327.
- (Ядро) Исправить размер массива `txn::sub_stmt_begin`.
- (Ядро) Учитывать `index.pairs` в `box.stat.SELECT()`.
- (Репликация) Запретить настройку мастеров только для чтения. Проблема 4321.
- (Репликация) Входить в режим одиночного сервера при ручном изменении настройки репликации. Проблема 4424.
- (Репликация) Задать значение `now` для `last_row_time` в `relay_new` и `relay_start`. Проблема 4431.
- (Репликация) Остановить передачу данных при ошибке подписки. Проблема 4399.
- (Репликация) Запустить средство отслеживания `coio` перед join/subscribe. Проблема 4110.
- (Репликация) Разрешить изменение идентификатора экземпляра во время присоединения. Проблема 4107.
- (Репликация) Исправить логику сборки мусора.
- (Репликация) Вернуть проверку границ пакета для iproto.
- (Репликация) Не прерывать репликацию при ER_UNKNOWN_REPLICA.
- (Репликация) Уменьшить воздействие фрагментации буфера ввода при большом `cfg.readahead`.
- (Репликация) Исправить обновление с 1.7 (не распознает тип запроса IPROTO_VOTE).
- (Репликация) Исправить утечку памяти в call / eval в случае отсутствия коммита транзакции. Проблема 4388.
- (Lua) Исправить регистрацию ошибок `fio.mktree()`. Проблема 4044.
- (Lua) Исправить ошибку сегментации в `ffi.C_say()` без имени файла. Проблема 4336.
- (Lua) Исправить ошибку сегментации в `json.encode()` на рекурсивную таблицу. Проблема 4366.
- (Lua) Исправить зависание `pwd.getpwall()` и `pwd.getgrall()` на CentOS 6 и FreeBSD 12. Проблемы 4447, 4428.
- (Lua) Исправить ошибку сегментации во время инициализации cipher from `crypto` module. Проблема 4223.
- (HTTP-клиент) Уменьшить потребление стека во время ожидания результата определения DNS. Проблема 4179.
- (HTTP-клиент) Увеличить максимальный размер заголовка до 8 КиБ. Проблема 3959.
- (HTTP-клиент) Сильнее проверять опцию «headers». Проблемы 4281, 3679.
- (HTTP-клиент) Использовать `libcurl` в комплекте, а не системный по умолчанию. Проблемы 4318, 4180, 4288, 4389, 4397.

- (HTTP-клиент) Закрывает несколько известных проблем, которые были исправлены в последних версиях `libcurl`, включая ошибки сегментации, зависания, утечки памяти и проблемы производительности.
- (LuaJIT) Исправить переполнение массива снимка. Часть проблемы 4171.
- (LuaJIT) Исправить повторное сцепление псевдо-восстановленных строковых ключей. Часть проблемы 4171.
- (LuaJIT) Исправить ошибки алгоритма свертывания. Проблема 4376.
- (LuaJIT) Исправить *debug.getinfo(1,“>S“)*. Проблема 3833.
- (LuaJIT) Исправить записи *string.find*. Проблема 4476.
- (LuaJIT) Исправлена ошибка нетонущих 64-битных указателей.
- (Разное) Еще увеличить предел количества открытых файлов в файле `systemd`.
- (Разное) Выдавать ошибку в `tarantoolctl` при отсутствии вызова `box.cfg()`. Проблема 3953.
- (Разное) Поддерживать NOTIFY_SOCKET из `systemd` на OS X. Проблема 4436.
- (Разное) Исправить `coio_getaddrinfo()` при передаче времени ожидания 0 (влияет на `connect_timeout` в `netbox`). Проблема 4209.
- (Разное) Исправить `coio_do_copyfile()` для выполнения усечения места назначения (влияет на `fio.copyfile()`). Проблема 4181.
- (Разное) Сделать подсказки в `coio_getaddrinfo()` необязательными.
- (Разное) Проверять аргумент размера `msgpack.decode()`. Проблема 4224.
- (Разное) Исправить привязку к статической библиотеке `openssl`. Проблема 4437.

Устаревшие функции

- (Ядро) `wal_max_size` заменяет устаревший `rows_per_wal`. Часть проблемы 3762.

Версия 1.10.3

Тип версии: стабильная (lts). Дата выхода: 2019-04-01. Тег: 1-10-3.

Сообщение: https://github.com/tarantool/tarantool/releases/tag/1.10.3.

Общие сведения

1.10.3 представляет собой очередную *стабильную (lts)* версию в серии 1.10. Пометка «стабильная» означает, что некоторые системы в течение определенного времени успешно отработали в производственной среде без известных сбоев, ненадежных результатов и прочих неисправностей.

Данная версия содержит 69 исправлений по сравнению с версией 1.10.2.

Совместимость

Tarantool 1.10.x обратно совместим с Tarantool 1.9.x в том, что касается структуры бинарных данных, клиент-серверного протокола и протокола репликации. Чтобф воспользоваться новыми функциями серии 1.10.x, *обновите* версию 1.9 с помощью процедуры `box.schema.upgrade()`.

Изменения или добавления функциональности

- (Движки) Слияние индексов в vinyl’е носит случайный характер. Проблема 3944.
- (Движки) Регулировка потока tx, если слияние не успевает за созданием дампов. Проблема 3721.
- (Движки) Отмена run_count_per_level для последнего уровня. Проблема 3657.
- (Сервер) Отчет о количестве активных соединений iproto. Проблема 3905.

- (Репликация) Удаление мертвой реплики, когда не хватает свободного места на диске. Проблема 3397.
- (Репликация) Отчет о состоянии присоединения в журнале реплики. Проблема 3165.
- (Lua) Отображение статуса снимка в box.info.gc(). Проблема 3935.
- (Lua) Отображение имен Lua-функций в обратной трассировке fiber.info(). Проблема 3538.
- (Lua) Проверка наличия открытой транзакции. Проблема 3518.

Исправленные ошибки

- (Движки) Сбой Tarantool'а при гонке потоков DML и DDL. Проблема 3420.
- (Движки) Ошибка восстановления при прерывании работы DDL. Проблема 4066.
- (Движки) Коммиты Tarantool'а в режиме только для чтения. Проблема 4016.
- (Движки) Сбой итератора vinyl'а при использовании DDL. Проблема 4000.
- (Движки) Vinyl не завершает работу, пока не закончится создание дампа или слияние. Проблема 3949.
- (Движки) После повторного создания вторичного индекса не видно данных. Проблема 3903.
- (Движки) Незагруженность box.info.memory().tx. Проблема 3897.
- (Движки) Vinyl замедляет скорость при интенсивных случайных вставках. Проблема 3603.
- (Сервер) Новая версия libcurl вызывает переполнение стека файбера. Проблема 3569.
- (Сервер) SIGHUP вызывает завершение работы Tarantool'а. Проблема 4063.
- (Сервер) checkpoint_daemon.lua:49: неправильный аргумент №2 для „format“. Проблема 4030.
- (Сервер) fiber:name() показывает только часть имени. Проблема 4011.
- (Сервер) Второе переключение режима горячего резервирования hot standby может не сработать. Проблема 3967.
- (Сервер) Обновление box.cfg.readahead не влияет на текущие соединения. Проблема 3958.
- (Сервер) fiber.join() остается заблокирован в статусе „suspended“, если файбер был отменен. Проблема 3948.
- (Сервер) Tarantool может завершить работу с ошибкой при отправке ненужных данных в бинарный сокет. Проблема 3900.
- (Сервер) Хранимая процедура для создания push-сообщений не прерывается при отключении клиента. Проблема 3859.
- (Сервер) Tarantool завершил работу с ошибкой в lj_vm_return. Проблема 3840.
- (Сервер) Файбер, выполняющий box.cfg(), может обрабатывать сообщения из iproto. Проблема 3779.
- (Сервер) Возможная регрессия на nosqlbench. Проблема 3747.
- (Сервер) Утверждение после неправильного создания индекса. Проблема 3744.
- (Сервер) Сбой Tarantool'а при запуске novshard (lj_gc_step). Проблема 3725.
- (Сервер) Репликация не запускается повторно на box.cfg, если конфигурация не изменилась. Проблема 3711.
- (Репликация) Время работы наложения (applier) сокращается при чтении кортежей большого размера. Проблема 4042.

- (Репликация) Сбой присоединения реплики Vinyl. Проблема 3968.
- (Репликация) Ошибка во время репликации. Проблема 3910.
- (Репликация) Статус downstream не отображается в replication.info, если канал не сломан. Проблема 3904.
- (Репликация) Сбой репликации: несовпадение контрольной суммы tx. Проблема 3993.
- (Репликация) Повторная настройка не производится, если на мастере есть строки из реплики. Проблема 3740.
- (Репликация) После перезапуска состояние кортежей откатывается на дорепликационное состояние. Проблема 3722.
- (Репликация) Добавление vclock для более безопасного переключения в режим горячего резервирования hot standby. Проблема 3002.
- (Репликация) Строка из мастера исчезает при сбое записи в журнал упреждающей записи. Проблема 2283.
- (Lua) Сбой преобразования space:frommap():tomap(). Проблема 4045.
- (Lua) Неинформативное сообщение при попытке прочитать отрицательное значение счетчика байтов из сокета. Проблема 3979.
- (Lua) space:frommap вызывает ошибку несовпадения кортежей («tuple field does not match...») даже для нулевого поля. Проблема 3883.
- (Lua) Завершение работы Tarantool'а с ошибкой на net.box.call после нормальной работы с внутренним файбером vshard. Проблема 3751.
- (Lua) Использование динамической памяти в lbox_error. Проблема 1955.
- (Разное) http.client не подтверждает „connection: keep-alive“. Проблема 3955.
- (Разное) Сломан wait_connected в net.box. Проблема 3856.
- (Разное) Сборка Mac завершается с ошибкой в Mojave. Проблема 3797.
- (Разное) Ошибка сборки FreeBSD: отсутствует поддержка SSL. Проблема 3750.
- (Разное) „http.client“ выдает неправильную (?) причину. Проблема 3681.
- (Разное) Http client молча изменяет заголовки, когда значение – не «строка» и не «число». Проблема 3679.
- (Разное) yaml.encode использует многострочный формат для „false“ и „true“. Проблема 3662.
- (Разное) yaml.encode неправильно кодирует „null“. Проблема 3583.
- (Разное) Пустое сообщение объекта ошибки. Проблема 3604.
- (Разное) Журнал переполняется предупреждениями. Проблема 2218.

Устаревшие функции

- Опция `console=true` для *net.box.new()* объявлена устаревшей.

Версия 1.10.2

Тип версии: стабильная (lts). Дата выхода: 2018-10-13. Тег: 1-10-2.

Сообщение: https://github.com/tarantool/tarantool/releases/tag/1.10.2.

Данная сборка представляет собой первую *стабильную (lts)* версию в серии 1.10. Кроме того, Tarantool 1.10.2 представляет собой мажорную версию, версия Tarantool 1.9.2 объявлена устаревшей. Это обновление содержит 95 исправлений по сравнению с версией 1.9.2.

Tarantool 1.10.x обратно совместим с Tarantool 1.9.x в том, что касается структуры бинарных данных, клиент-серверного протокола и протокола репликации. *Обновление* можно произвести с помощью процедуры `box.schema.upgrade()`.

Цель данного релиза – значительно повысить стабильность `vinyl'a` и реализовать автоматическую повторную настройку набора реплик в Tarantool'е.

Изменения или добавления функциональности:

- (Движки) поддержка изменения ALTER непустых спейсов в vinyl'е. Проблема 1653.
- (Движки) кортежи, которые хранятся в кэше vinyl'а, не учитываются в индексах того же спейса. Проблема 3478.
- (Движки) хранение стека операций обновления и вставки UPSERT в `vy_read_iterator`. Проблема 1833.
- (Движки) `box.ctl.reset_stat()`, функция сброса статистики в vinyl'е. Проблема 3198.
- (Сервер) *настройка места назначения syslog*. Проблема 3487.
- (Сервер) допустимость неопределенного значения разного вида в индексах и форматах. Проблема 3430.
- (Сервер) возможность осуществлять *резервное копирование любой контрольной точки*, а не только последней. Проблема *3410 (Сервер) допустимость :ref:'резервного копирования любой контрольной точки <reference_lua-box_backup-backup_start>*, а не только последней. Проблема 3410.
- (Сервер) метод, чтобы определить был ли запуск или перезапуск процесса Tarantool'а осуществлен с помощью `tarantoolctl` (переменные окружения *TARANTOOLCTL и TARANTOOL_RESTARTED*). Проблемы 3384, 3215.
- (Сервер) конфигурационный параметр *net_msg_max* ограничивает число выделенных файберов. Проблема 3320.
- (Репликация) отображение статуса соединения, если последующий сервер отключается от предыдущего (*box.info.replication.downstream.status* `= disconnected`). Проблема 3365.
- (Репликация) *спейсы с локальной репликацией* Проблема 3443.
- (Репликация) *replication_skip_conflict*, новый параметр в `box.cfg{}` для пропуска конфликтов строк при репликации. Проблема 3270.
- (Репликация) удаление старых снимков, которые не нужны репликами. Проблема 3444.
- (Репликация) запись в журнал попытки повторного коммита. Проблема 3105.
- (Lua) новая функция *fiber.join()*. Проблема 1397.
- (Lua) новая опция `names_only` для *tuple:tomap()*. Проблема 3280.
- (Lua) поддержка специализированных серверов для модулей (опции `server` и `only-server` для команды *tarantoolctl rocks*). Проблема 2640.
- (Lua) передача триггеров `on_commit`/`on_rollback` в Lua. Проблема 857.
- (Lua) новая функция *box.is_in_txn()* для проверки наличия открытой транзакции. Проблема 3518.
- (Lua) доступ к полю кортежа по JSON-пути (по *номеру*, *имени* и *пути*). Проблема'1285 <https://github.com/tarantool/tarantool/issues/1285>'_.
- (Lua) новая функция *space:frommap()*. Проблема 3282.

- (Lua) новый модуль *utf8*, который имплементирует привязки libicu для использования в Lua. Проблемы 3290, 3385.

7.3 Версия 1.9

Версия 1.9.0

Тип версии: стабильная. Дата выхода: 2018-02-26. Тег: 1.9.0-4-g195d446.

Сообщение: https://github.com/tarantool/tarantool/releases/tag/1.9.0.

Эта версия следует за стабильной версией 1.7.6. Цель данной версии – повысить стабилизацию vinyl'а и репликации типа мастер-мастер, для чего предусмотрено значительное количество новых функций. Следуйте инструкциям по загрузке по ссылке https://tarantool.io/en/download/download.html для установки пакета для вашей операционной системы.

Изменения или добавления функциональности:

- (Безопасность) появилась возможность *блокировки и разблокировки* пользователей. Проблема 2898.
- (Безопасность) новая функция *box.session.euid()* возвращает действующего пользователя. Действующий пользователь может отличаться от авторизованного пользователя при использовании функций `setuid` или `box.session.su`. Проблема 2994.
- (Безопасность) новая роль суперпользователя *super*. Чтобы отключить управление доступом, следует назначить пользователю guest роль „super“. Проблема 3022.
- (Безопасность) триггер *on_auth* срабатывает, когда аутентификация пройдена, а также, когда аутентификация не пройдена. Проблема 3039.
- (Репликация/восстановление) новый алгоритм конфигурации репликации: если экземпляр не подключается к количеству узлов, указанному в replication_quorum, за количество секунд, указанное в *replication_connect_timeout*, сервер начинает работу, но в качестве *одиночного*, то есть в режиме только для чтения, пока реплики не подключатся друг к другу. Проблемы 3151 и 2958.
- (Репликация/восстановление) после включения репликации при запуске сервер не начинает обработку запросов на запись до *синхронизации* со всеми подключенными узлами.
- (Репликация/восстановление) появилась возможность явным образом задать *UUID экземпляра* и *UUID набора реплик* в качестве конфигурационных параметров. Проблема 2967.
- (Репликация/восстановление) *box.once()* больше не прекращает работу на реплике в режиме только для чтения, а переходит в режим ожидания. Проблема 2537.
- (Репликация/восстановление) *force_recovery* может пропускать поврежденный xlog-файл. Проблема 3076.
- (Репликация/восстановление) улучшен мониторинг репликации: *box.info.replication* показывает IP-адрес:порт узла в сети и правильную задержку репликации для неактивных узлов. Проблема 2753 и 2689.
- (Сервер приложений) новые триггеры до события (*before*) можно использовать для разрешения конфликтов при репликации типа мастер-мастер. Проблема 2993.
- (Сервер приложений) *http client* правильно разбирает файлы cookie и поддерживает пути http+unix://. Проблемы 3040 и 2801.
- (Сервер приложений) в модуле `fio` появилась поддержка `file_exists()`, `rename()` работает в разных файловых системах, `read()` без аргументов выполняет чтение всего файла. Проблемы 2924, 2751 и 2925.

- (Сервер приложений) ошибки в модуле `fio` соответствуют стандартам вызова функции в Tarantool'е и всегда возвращают сообщение об ошибке вместе с флагом ошибки.
- (Сервер приложений) модуль `digest` поддерживает алгоритм хеширования паролей pbkdf2, который используется в приложениях, совместимых с PCI/DSS. Проблема 2874.
- (Сервер приложений) *box.info.memory()* обеспечивает общий обзор использования памяти сервера: работа по сети, Lua, транзакции и индексы. Проблема 934.
- (База данных) появилась возможность *добавить отсутствующие поля кортежа* в индекс, что используется при добавлении индекса вместе с эволюцией схемы базы данных. Проблема 2988.
- (База данных) множество улучшений поддержки типов полей при создании или *изменении* спейсов и индексов. Проблемы 2893, 3011 и 3008.
- (База данных) появилась возможность включения опции *is_nullable* для поля, даже если спейс не является пустым, с мгновенным применением изменений. Проблема 2973.
- (База данных) улучшены многие аспекты *журналирования*: отдельные сообщения (проблемы 1972, 2743, 2900), увеличение количества записей при необходимости (проблемы 3096, 2871).
- (Движок базы данных Vinyl) появилась возможность сделать *уникальный* индекс в vinyl'е неуникальным без повторного создания индекса. Проблема 2449.
- (Движок базы данных Vinyl) улучшена производительность операций обновления UPDATE, замены REPLACE и восстановления при наличии вторичных ключей. Проблемы 2289, 2875 и 3154.
- (Движок базы данных Vinyl) *space:len()* и *space:bsize()* работают с vinyl'ом (хотя и неточно). Проблема 3056.
- (Движок базы данных Vinyl) улучшена скорость восстановления при наличии вторичных ключей. Проблема 2099.
- (Сборки) Поддержка Alpine Linux. Проблема 3067.

7.4 Version 1.8

Release 1.8.1

Release type: alpha. Release date: 2017-05-17. Tag: 1.8.1.

Announcement: https://groups.google.com/forum/#!msg/tarantool-ru/XYaoqJpc544/mSvKrYwNAgAJ.

This is an alpha release which delivers support for a substantial subset of the ISO/IEC 9075:2011 SQL standard, including joins, subqueries and views. SQL is a major feature of the 1.8 release series, in which we plan to add support for ODBC and JDBC connectors, SQL triggers, prepared statements, security and roles, and generally ensure SQL is a first class query language in Tarantool.

Изменения или добавления функциональности:

- A new function `box.sql.execute()` (later changed to `box.execute` in Tarantool 2.1) was added to query Tarantool databases using SQL statements, e.g.:

```
tarantool> box.sql.execute([[SELECT * FROM _schema]]);
```

- SQL and Lua are fully interoperable.
- New meta-commands introduced to Tarantool's console.

  You can now set input language to either SQL or Lua, e.g.:

```
tarantool> \set language sql
tarantool> SELECT * FROM _schema;
tarantool> \set language lua
tarantool> print("Hello, world!")
```

- Most SQL statements are supported:
  - CREATE/DROP TABLE/INDEX/VIEW

    ```
    tarantool> CREATE TABLE table1 (column1 INTEGER PRIMARY KEY, column2 VARCHAR(100));
    ```

  - INSERT/UPDATE/DELETE statements e.g.:

    ```
    tarantool> INSERT INTO table1 VALUES (1, 'A');
    ...
    tarantool> UPDATE table1 SET column2 = 'B';
    ```

  - SELECT statements, including complex complicated variants which include multiple JOINs, nested SELECTs etc. e.g.:

    ```
    tarantool> SELECT sum(column1) FROM table1 WHERE column2 LIKE '_B' GROUP BY column2;
    ```

  - WITH statements e.g.

    ```
    tarantool> WITH cte AS ( SELECT SUBSTR(column2,1,2), column1 FROM table1 WHERE column1 >
    ↪= 0) SELECT * FROM cte;
    ```

  - SQL schema is persistent, so it is able to survive `snapshot()`/`restore()` sequence.
  - SQL features are described in a *tutorial*.

7.5 Версия 1.7

Версия 1.7.6

Тип версии: стабильная. Дата выхода: 2017-11-07. Тег: 1.7.6-0-g7b2945d6c.

Объявление о выходе: https://groups.google.com/forum/#!topic/tarantool/hzc7O2YDZUc.

Данная сборка представляет собой очередную стабильную версию в серии 1.7. Это обновление содержит более 75 исправлений по сравнению с версией 1.7.5.

Что нового в Tarantool 1.7.6?

- В дополнение к *откату* транзакции, появился откат на определенную точку в пределах транзакции – поддержка *точки сохранения*.
- Появился новый объектный тип: *последовательности*. Устаревший вариант, *автоматическое увеличение*, объявлен устаревшим.
- В строковых индексах появилась *сортировка*.

Добавлены новые опции:

- *net_box* (время ожидания),
- функции *string*,
- *форматы* для спейса (имена и типы полей, задаваемые пользователем),

- *base64* (опция `urlsafe`), а также
- *создание* индекса (сортировка, *is-nullable* (возможность допустить неопределенное значение), имена полей).

Несовместимые изменения:

- Расширенная структура `box.space._index` поддерживает функции *is_nullable* и *collation* (сортировка). Все новые индексы, созданные по столбцам со свойствами `is_nullable` или `collation` получат новый формат определения. Обновите клиентские библиотеки, если планируете использовать новые возможности. Проблема 2802
- *fiber_name()* теперь выдает ошибку вместо усечения длинных имен файберов. Мы обнаружили, что некоторые Lua-модули, такие как *expirationd*, используют `fiber.name()` для определения фоновых задач. Если же имя усечено, они упускают файбер из вида. Обновление позволит обнаружить ошибки, вызванные усечением имени файбера `fiber.name()`. Используйте `fiber.name(name, { truncate = true })` для моделирования старого поведения системы. Проблема 2622
- *space:format()* проверяется в DML-операциях. Раньше `space:format()` использовался только в клиентских библиотеках, но с версии Tarantool 1.7.6 типы полей в `space:format()` проверяются на стороне сервера при каждой DML-операции, и имена полей могут использоваться в индексах и Lua-коде. Если `space:format()` использовался нестандартно, обновите структуру и имена типов в соответствии с официальной документацией по форматам спейса.

Изменения или добавления функциональности:

- Гибридная модель данных без схемы + со схемой. Раньше версии Tarantool позволяли хранить произвольный набор документов в формате MessagePack в спейсах. Начиная с версии Tarantool 1.7.6, можно использовать *space:format()* для определения условий и ограничений схемы для кортежей в спейсах. Определенные типы полей автоматически проверяются при каждой DML-операции, а определенные имена полей могут использоваться вместо номеров полей в Lua-коде. Добавлена новая функция *tuple:tomap()* для конвертации кортежа в Lua-словарь пар ключ-значение.
- Поддержка сортировки и Юникода. По умолчанию, когда Tarantool сопоставляет строки, он берет во внимание только числовое значение каждого байта в строке. Чтобы задействовать такое распределение, как в телефонных справочниках и словарях, в Tarantool'е версии 1.7.6 впервые поддерживается сортировка по Таблице сортировки символов Юникода по умолчанию (Default Unicode Collation Element Table (DUCET)) и в соответствии с правилами, описанными в Техническом стандарте Юникода №10 – Алгоритм сортировки по Юникоду (Unicode® Technical Standard #10 Unicode Collation Algorithm (UTS #10 UCA)). См. *сортировку*.
- Значения NULL в уникальных и неуникальных индексах. По умолчанию, все поля в Tarantool'е «НЕ NULL». Начиная с версии Tarantool 1.7.6, можно использовать опцию `is_nullable` (возможность допустить неопределенное значение) в *space:format()* или *в определении части индекса*, чтобы разрешить хранение значения NULL в индексах. Tarantool частично реализует троичную логику из стандарта SQL и позволяет хранить несколько значений NULL в уникальных индексах. Проблема 1557.
- Последовательности и внедрение автоматического увеличения *auto_increment()*. В версии Tarantool 1.7.6 впервые реализованы *генераторы порядковых номеров* (как CREATE SEQUENCE – создание последовательности – в SQL). Эта функция используется для внедрения нового персистентного автоматического увеличения в спейсах. Проблема 389.
- Vinyl: появляется блокировка разрывов в менеджере транзакций Vinyl'а. Новый блокирующий механизм в менеджере Vinyl TX снижает количество конфликтов в транзакциях. Проблема 2671.
- net.box: триггеры *on_connect* и *on_disconnect* (по подключению/отключению). Проблема 2858.

- Структурированная запись в журнал в *формате JSON*. Проблема 2795.
- (Lua) Lua: *string.strip()* Проблема 2785.
- (Lua) добавлен API *base64_urlsafe_encode()* для модуля `digest`. Проблема 2777.
- Запись конфликтов в ключах в журнал в рамках репликации мастер-мастер. Проблема 2779.
- Возможность отключить обратную трассировку в *fiber.info()*. Проблема 2878.
- Реализована возможность создания сторонних библиотек `tarantoolctl rocks make *.spec`. Проблема 2846.
- Новая функция загрузчика, используемого по умолчанию, позволяет искать модули `.rocks` в родительской иерархии. Проблема 2676.
- Поддержка опций `SOL_TCP` в *socket:setsockopt()*. Проблема 598.
- Частичное моделирование LuaSocket поверх Tarantool Socket. Проблема 2727.

Инструменты разработчика:

- Интеграция с IntelliJ IDEA с поддержкой отладки. Появилась возможность использовать IntelliJ IDEA в качестве IDE для разработки и отладки Lua-приложений для Tarantool'а. См. *Использование IDE*.
- Интеграция с удаленным Lua-отладчиком MobDebug. Проблема 2728.
- Настройка `/usr/bin/tarantool` в качестве альтернативного Lua-интерпретатора для Debian/Ubuntu. Проблема 2730.

Новые сторонние библиотеки:

- smtp.client – поддержка SMTP по `libcurl`.

Версия 1.7.5

Тип версии: стабильная. Дата выхода: 2017-08-22. Тег: 1.7.5.

Объявление о выходе: `https://github.com/tarantool/doc/issues/289`.

Данная сборка представляет собой стабильную версию в серии 1.7. Это обновление содержит более 160 исправлений по сравнению с версией 1.7.4.

Изменения или добавления функциональности:

- (Vinyl) новый режим принудительного восставовления *force_recovery* для восстановления поврежденных файлов на диске. Используйте `box.cfg{force_recovery=true}` для восстановления файлов с данными, поврежденными в результате проблем с оборудованием или отключения электроэнергии. Проблема 2253.
- (Vinyl) параметры индекса можно менять на лету без необходимости пересборки. Появилась возможность динамически изменять параметры *page_size*, *run_size_ratio*, *run_count_per_level* и *bloom_fpr* с помощью *index:alter()*. Изменения вступают в силу только для вновь созданных файлов. Проблема 2109.
- (Vinyl) улучшен вывод *box.info.vinyl()* и `index:info()`. Проблема 1662.
- (Vinyl) появляется опция *box.cfg.vinyl_timeout* для управления загрузкой на основе квот. Проблема 2014.
- Memtx: стабильные итераторы *index:pairs()* для TREE-индекса. TREE-итераторы автоматически восстанавливаются в правильном положении после изменений индекса. Проблема 1796.
- (Memtx) *предсказуемый порядок* для неуникальных TREE-индексов. Неуникальные TREE-индексы сохраняют порядок сортировки для дублирующихся записей. Проблема 2476.

- (Memtx+Vinyl) динамическая настройка *максимального размера кортежа*. Впервые конфигурационные параметры `box.cfg.memtx_max_tuple_size` и `box.cfg.vinyl_max_tuple_size` можно изменять на лету без необходимости перезагрузки сервера. Проблема 2667.
- (Memtx+Vinyl) новая реализация. *Усечение* спейса больше не вызывает повторное создание всех индексов. Проблема 618.
- *Максимальная длина* всех идентификаторов расширена с 32 до 65 тысяч символов. Имена спейса, пользователя и функции больше не ограничены 32 символами. Проблема 944.
- Сообщения *контрольного сигнала* для репликации. Репликационный клиент теперь выборочно отправляет подтверждение обработки записей и автоматически переподключается в случае замедления. Также в рамках этого изменения *box.info.replication[replica_id].vclock* будет отображать определенный vclock удаленной реплики. Проблема 2484.
- Отслеживание удаленных реплик во время обслуживания WAL. Мастер репликации будет автоматически сохранять xlog-файлы, необходимые для удаленных реплик. Проблема 748.
- Enabled *box.tuple.new()* to work without `box.cfg()`. Issue 2047.
- Надстройка *box.atomic(fun, ...)* будет выполнять функции в транзакции. Проблема 818.
- Вспомогательная функция *box.session.type()* будет определять тип сессии. Проблема 2642.
- Горячая *перезагрузка кода* для хранимых процедур на языке C. Используйте `box.schema.func.reload('modulename.function')` для перезагрузки библиотек общего пользования на лету. Проблема 910.
- API для Lua: *string.hex()* и `str:hex()`. Проблема 2522.
- Менеджер пакетов на основе LuaRocks. Используйте `tarantoolctl rocks install MODULENAME` для установки Lua-модуля MODULENAME (имя модуля) из `https://rocks.tarantool.org/`. Проблема 2067.
- Опции командной строки в Lua 5.1. Бинарный протокол Tarantool'а поддерживает опции командной строки: „-i“, „-e“, „-m“ и „-l“. Проблема 1265.
- Экспериментальный режим GC64 для LuaJIT. Режим GC64 позволяет работать со спейсами с полным адресом на 64-битных хостах. Включить настройку можно с помощью `-DLUAJIT_ENABLE_GC64=ON compile-time`. Проблема 2643.
- Регистратор журнала syslog поддерживает неблокирующий режим. *box.cfg{log_nonblock=true}* также работает для регистратора syslog. Проблема 2466.
- Добавлен уровень *записи в журнал* VERBOSE выше INFO. Проблема 2467.
- Tarantool автоматически делает снимки каждый час. Установите *box.cfg{checkpoint_interval=0}*, чтобы восстановить поведение предыдущих версий. Проблема 2496.
- Увеличена точность для процентного соотношения, приведенного с помощью *box.slab.info()*. Проблема 2082.
- Трассировка стека будет содержать имена символов на всех поддерживаемых платформах. В предыдущих версиях Tarantool не отображал значимые имена функций в *fiber.info()* на платформах не-x86. Проблема 2103.
- Появилась возможность создания файбера с заданным размером стека из API для языка C. Проблема 2438.
- В API для языка C добавлена функция `ipc_cond`. Проблема 1451.

Новые сторонние библиотеки:

- *http.client* (встроенная) - HTTP-клиент на основе libcurl с поддержкой SSL/TLS. Проблема 2083.

- *iconv* (встроенная) - привязки для iconv. Проблема 2587.
- authman - API для регистрации пользователя и входа в систему с использованием email и социальных сетей.
- document - хранит вложенные документы в Tarantool'е.
- synchronized - критические секции для Lua.

Версия 1.7.4

Тип версии: предварительная версия. Дата выхода: 2017-05-12. Тег версии: 1.7.4.

Объявление о выходе: https://github.com/tarantool/tarantool/releases/tag/1.7.4 или https://groups.google.com/forum/#!topic/tarantool/3x88ATX9YbY

Данная сборка представляет собой предварительную версию перед выпуском нового релиза в серии 1.7. Движок vinyl, ключевой компонент 1.7.x, обладает полностью реализованной заявленной функциональностью.

Несовместимые изменения

- Для поддержки vinyl были внесены следующие изменения в параметры `box.cfg()`:
  - переименование `snap_dir` в `memtx_dir`
  - переименование `slab_alloc_arena` (гигабайты) в `memtx_memory` (байты), значение, используемое по умолчанию, изменилось с 1 Гб на 256 МБ
  - переименование `slab_alloc_minimal` в `memtx_min_tuple_size`
  - переименование `slab_alloc_maximal` в `memtx_max_tuple_size`
  - `slab_alloc_factor` больше не используется, не применимо в 1.7.x
  - переименование `snapshot_count` в `checkpoint_count`
  - переименование `snapshot_period` в `checkpoint_interval`
  - переименование `logger` в `log`
  - переименование `logger_nonblock` в `log_nonblock`
  - переименование `logger_level` в `log_level`
  - переименование `replication_source` в `replication`
  - `panic_on_snap_error = true` и `panic_on_wal_error = true` заменены `force_recovery = false`

  В версиях Tarantool'а до 1.8 можно использовать устаревшие параметры как для начальной, так и для рабочей конфигурации, но в таком случае система запишет сообщение предупреждения в журнал сервера. Проблемы 1927 и 2042.
- Режим hot standby (горячее резервирование) по умолчанию будет отключен. Tarantool автоматически находит еще один запущенный экземпляр в той же директории `wal_dir` и откажется запускаться. Используйте `box.cfg {hot_standby = true}` для включения режима hot standby. Проблема 775.
- Операция UPSERT по вторичному ключу запрещена во избежание неопределенности семантики. Проблема 2226.
- В формат `box.info` и `box.info.replication` для отображения информации о подключениях к upstream и downstream внесены следующие изменения (Проблема 723):
  - Добавление `box.info.replication[instance_id].downstream.vclock` для отображения последней строки, отправленной на удаленную реплику.

  - Добавление `box.info.replication[instance_id].id`.
  - Добавление `box.info.replication[instance_id].lsn`.
  - Перемещение `box.info.replication[instance_id].{vclock,status,error}` в `box.info.replication[instance_id].upstream.{vclock,status,error}`.
  - Включение всех зарегистрированных реплик из `box.space._cluster` в вывод `box.info.replication`.
  - Переименование `box.info.server.id` в `box.info.id`
  - Переименование `box.info.server.lsn` в `box.info.lsn`
  - Переименование `box.info.server.uuid` в `box.info.uuid`
  - Переименование `box.info.cluster.signature` в `box.info.signature`
  - Возврат значения *nil* вместо *-1* функциями `box.info.id` и `box.info.lsn` во время начальной настройки кластера.
- `net.box`: добавление запрошенные параметров во все запросы:
  - изменение `conn.call(func_name, arg1, arg2,...)` на `conn.call(func_name, {arg1, arg2, ...}, opts)`
  - изменение `conn.eval(func_name, arg1, arg2,...)` на `conn.eval(func_name, {arg1, arg2, ...}, opts)`
- Все запросы поддерживают параметры `timeout = <seconds>``(время задержки в секундах), ``buffer = <ibuf>` (буфер).
- Добавление опции `connect_timeout` в `netbox.connect()`.
- `netbox:timeout()` и `conn:timeout()` объявлены устаревшими. Используйте `netbox.connect(host, port, { call_16 = true })`, чтобы получить поведение как в 1.6.x. Проблема 2195.
- Конфигурация systemd будет поддерживать `Type=Notify` / `sd_notify()`. `systemctl start tarantool@ЭКЗЕМПЛЯР` будет ожидать, пока Tarantool не запустится и не восстановится из xlog-файлов. Статус восстановления передается в `systemctl status tarantool@ЭКЗЕМПЛЯР`. Проблема 1923.
- Модуль `log` не будет присоединять ко всем сообщениям полный путь к бинарному файлу при использовании без `box.cfg()`. Проблема 1876.
- Переименование `require('log').logger_pid()` в `require('log').pid()`. Проблема 2917.
- Удаленные определения и функции, совместимые с Lua 5.0 (Проблема 2396):
  - `luaL_Reg` заменяет удаленный `luaL_reg`
  - `lua_objlen(L, i)` заменяет удаленный `luaL_getn(L, i)`
  - Удаление `luaL_setn(L, i, j)` (пустая операция)
  - `luaL_ref(L, lock)` заменяет удаленный `lua_ref(L, lock)`
  - `lua_rawgeti(L, LUA_REGISTRYINDEX, (ref))` заменяет удаленный `lua_getref(L,ref)`
  - `luaL_unref(L, ref)` заменяет удаленный `lua_unref(L, ref)`.
  - `math.fmod()` заменяет удаленный `math.mod()`
  - `string.gmatch()` заменяет удаленный `string.gfind()`

Изменения или добавления функциональности:

- (Vinyl) многоуровневое слияние. Планировщик слияния будет группировать забеги одного диапазона в уровни, чтобы снизить «паразитную» запись во время слияния. Новая функция позволит Vinyl'у поддерживать сценарии 1:100+ оперативная память:диск. Проблема 1821.
- (Vinyl) Фильтры Блума для упорядоченных файлов. Фильтр Блума – это вероятностная структура данных, которую можно использовать для проверки наличия необходимого ключа в файле без считывания самого файла с диска. Фильтр Блума может выдавать ложноположительное срабатывание (элемента в множестве нет, но структура данных сообщает, что он есть), но не ложноотрицательное. Данная функция уменьшает объем поиска, необходимый для случайного просмотра, и ускоряет операции REPLACE/DELETE со вторичными ключами. Проблема 1919.
- (Vinyl) кэш на уровне ключей для поиска точек и запросов по диапазону. Движок базы данных Vinyl кэширует выбранные ключи и диапазоны ключей вместо страниц диска полностью, как в традиционных базах данных. Такой подход более эффективен, поскольку кэш не заполнен сырыми данными. Проблема 1692.
- (Vinyl) внедрение уровня общей памяти для in-memory индексов. Все in-memory индексы спейса будут хранить указатели на одни и те же кортежи, вместо закэшированных данных вторичного индекса. Данная функция значительно уменьшает объем необходимой памяти в случае вторичных ключей. Проблема 1908.
- (Vinyl) новая реализация передачи начального состояния JOIN-команды в протоколе репликации. Новый протокол репликации исправляет проблемы с согласованностью и вторичными ключами. Мы внедрили специальный вид просмотра по всей базе данных с небольшой нагрузкой, чтобы избежать неподтвержденного чтения в JOIN-процедуре. В традиционных базах данных на основе B-Tree такое не представляется возможным. Проблема 2001.
- (Vinyl) забеги по всему индексу. Удалены диапазоны из оперативной памяти и уровень LSM-дерева на диске. Проблема 2209.
- (Vinyl) объединение небольших диапазонов. Перед созданием дампа или слиянием диапазона рассмотрите возможность объединения его с соседними диапазонами. Проблема 1735.
- (Vinyl) внедрен многосторонний журнал для метаданных. Информация о всех Vinyl-файлах будет записываться в специальный `.vylog`-файл. Проблема 1967.
- (Vinyl) появились постоянные вторичные ключи. Проблема 2410.
- (Memtx+Vinyl) внедрен низкоуровневый API для Lua в целях создания согласованных резервных копий данных Memtx + Vinyl. Новая функциональность обеспечивает создание резервных копий всех спейсов с помощью функций `box.backup.start()/stop()`. *box.backup.start()* останавливает работу сборщика мусора Tarantool'а и возвращает список файлов для копирования. Затем эти файлы можно скопировать с помощью любого стороннего средства, например, cp, ln, tar, rsync и т.д. `box.backup.stop()` возобновляет работу сборщика мусора. Чтобы немедленно восстановить данные, скопируйте созданные резервные копии в новую директорию, а затем запустите новый экземпляр Tarantool'а. Нет необходимости в дополнительной подготовке, преобразовании или распаковывании. Проблема 1916.
- (Vinyl) добавлена статистика для фоновых рабочих процессов в `box.info.vinyl()`. Проблема 2005.
- (Memtx+Vinyl) уменьшен объем необходимой памяти для индексов с последовательными ключами, которые начинаются с первого поля. Такая оптимизация была необходима для вторичных ключей в Vinyl'е, но мы также оптимизировали Memtx. Проблема 2046.
- LuaJIT получил все изменения с последней версии 2.1.0b3 с нашими патчами (Проблема 2396):
  - Добавлен бэкенд для JIT-компилятора для архитектуры ARM64
  - Добавлен бэкенд и интерпретатор для JIT-компилятора для архитектуры MIPS64

  - Добавлены некоторые расширения для Lua 5.2 и Lua 5.3
  - Исправление нескольких ошибок
  - Удалены устаревшие функции Lua 5.0 (см. несовместимые изменения выше).
- Запущен новый умный алгоритм хеширования строк в LuaJIT, чтобы избежать замедления работы в случае множества коллизий. Разработали Юрий Соколов (@funny-falcon) и Ник Заварицкий (@mejedi). См. https://github.com/tarantool/luajit/pull/2.
- `box.snapshot()` теперь обновляет время mtime в файле снимка, если не было изменений в базе данных с момента последнего снимка. Проблема 2045.
- Внедрена функция `space:bsize()` для возврата объема памяти, занятого всеми кортежами спейса. Разработал Роман Токарев (@rtokarev). Проблема 2043.
- Новые функции Lua/C вынесены в общедоступный API:
  - `luaT_pushtuple`, `luaT_istuple` (проблема 1878)
  - `luaT_error`, `luaT_call`, `luaT_cpcall` (проблема 2291)
  - `luaT_state` (проблема 2416)
- Новые функции Box/C вынесены в общедоступный API: `box_key_def`, `box_tuple_format`, `tuple_compare()`, `tuple_compare_with_key()`. Проблема 2225.
- Можно осуществлять ротацию xlog-файлов на основе размера (`wal_max_size`), а также количества записанных строк (`rows_per_wal`). Проблема 173.
- Добавлены следующие API: `string.split()`, `string.startswith()`, `string.endswith()`, `string.ljust()`, `string.rjust()`, `string.center()`. Проблемы 2211, 2214, 2415.
- Добавлены функции `table.copy()` и `table.deepcopy()`. Проблема 2212.
- Добавлен модуль `pwd` для работы с пользователями и группами в UNIX. Проблема 2213.
- Удалены неуместные сообщения «client unix/: connected» из журналов. Используйте вместо них триггеры `box.session.on_connect()`/`on_disconnect()` (на подключение / отключение). Проблема 1938.

  Триггеры `box.session.on_connect()`/`on_disconnect()`/`on_auth()` также срабатывают для подключений административной консоли.
- tarantoolctl: следующие команды: `eval`, `enter`, `connect` – теперь поддерживают конвейеры UNIX. Проблема 672.
- tarantoolctl: более точные сообщения об ошибке; добавлена новая страница справочника. Проблема 1488.
- tarantoolctl: добавлен фильтр по `replica_id` для команд `cat` и `play`. Проблема 2301.
- tarantoolctl: Команды `start`, `stop` и `restart` перенаправляют на `systemctl start/stop/restart`, когда запущен systemd. Проблема 2254.
- net.box: по запросу добавлена опция `buffer = <buffer>` для хранения исходных ответов MessagePack в буфер C. Проблема 2195.
- net.box: добавлена опция `connect_timeout`. Проблема 2054.
- net.box: добавлена ловушка `on_schema_reload()`. Проблема 2021.
- net.box: `conn.schema_version` и `space.connection` дополнены API. Проблема 2412.
- log: `debug()`/`info()`/`warn()`/`error()` не выдают сбой при ошибках форматирования. Проблема 889.

- crypto: добавлена поддержка HMAC. Разработал Андрей Куликов (@amdei). Проблема 725.

Версия 1.7.3

Тип версии: бета. Дата выхода: 2016-12-24. Тег версии: 1.7.3-0-gf0c92aa.

Объявление о выходе: https://github.com/tarantool/tarantool/releases/tag/1.7.3

Данная сборка представляет собой вторую бета-версию в серии 1.7.

Несовместимые изменения:

- Удалена поврежденная Lua-функция `coredump()`. Используйте вместо нее `gdb -batch -ex "generate-core-file" -p $PID`. Проблема 1886.
- Структура диска Vinyl изменилась с версии 1.7.2: добавлен механизм компрессии ZStandard и улучшена производительность вторичных ключей. Используйте механизм репликации для обновления с бета-версии 1.7.2. Проблема 1656.

Изменения или добавления функциональности:

- Значительный прогресс в стабилизации движка базы данных Vinyl:
  - Исправлены большинство известных отказов системы и ошибок, выдающих плохие результаты.
  - Замена формата всех файлов с данными на XLOG/SNAP.
  - Использование механизма компрессии ZStandard для всех файлов с данными.
  - Сжатие операций UPSERT на лету и объединение горячих клавиш с помощью фонового файбера.
  - Значительное улучшение производительности index:pairs() и index:count().
  - Удаление ненужных конфликтов из транзакций.
  - Уровень In-memory по большей части заменен структурами данных memtx.
  - В большинстве случаев используются специализированные распределители ресурсов.
- Мы все еще активно работаем над Vinyl'ом и планируем добавить многоуровневое слияние и улучшить производительность в работе со вторичными ключами в версии 1.7.4. Это подразумевает изменение формата данных.
- Поддержка DML-запросов для триггеров space:on_replace(). Проблема 587.
- UPSERT можно использовать с пустым списком операций. Проблема 1854.
- Lua-функции будут управлять переменными окружения. Проблема 1718.
- Lua-библиотека будет считывать снимки Tarantool'а и xlog-файлы. Проблема 1782.
- Новые команды в `tarantoolctl`: `play` и"cat". Проблема 1861.
- Улучшена поддержка большого количества активных сетевых клиентов. Проблема #5#1892.
- Поддержка синтаксиса `space:pairs(key, iterator-type)`. Проблема 1875.
- Автоматическая настройка кластера будет работать и без авторизации. Проблема 1589.
- При репликации попытки повторного подключения к мастеру бесконечны. Проблема 1511.
- Временные спейсы будут работать с `box.cfg { read_only = true }`. Проблема 1378.
- Максимальная длина имени спейса увеличена до 64 байтов (ранее 32). Проблема 2008.

Версия 1.7.2

Тип версии: бета. Дата выхода: 2016-09-29. Тег версии: *1.7.2-1-g92ed6c4*.

Объявление о выходе: https://groups.google.com/forum/#!topic/tarantool-ru/qUYUesEhRQg

Данная сборка представляет собой версию в серии 1.7.

Несовместимые изменения:

- Команда нового бинарного протокола для вызова CALL больше не ограничивает функцию в возврате массива кортежей и позволяет возвращать произвольный результат в формате MsgPack/JSON, включая scalar (скалярные значения), nil (нулевые значения) и void (пусто). Старый метод CALL оставлен нетронутым для обратной совместимости. В следующей основной версии он будет удален. Все драйверы для языков программирования будут постепенно переведены на использование нового метода CALL. Проблема 1296.

Изменения или добавления функциональности:

- Разработка движка базы данных Vinyl, наконец, перешла в бета-стадию. В данной версии исправлены более 90 ошибок в Vinyl'е, в частности, удаление непредсказуемых скачков задержки отклика, все известные отказы системы и ошибки, выдающие плохие результаты или их отсутствие.
  - новая архитектура на основе кооперативной многозадачности для устранения скачков задержки отклика,
  - поддержка непоследовательных составных ключей,
  - поддержка вторичных ключей,
  - поддержка `auto_increment()`,
  - типы полей в индексах: number (число), integer (целое число), scalar (скаляр),
  - операции INSERT, REPLACE и UPDATE возвращают новый кортеж, как в memtx'е.
- Мы все еще активно работаем над Vinyl'ом и планируем добавить механизм компрессии `zstd` и новый распределитель ресурсов для Vinyl'а в версии 1.7.3. Это подразумевает изменение формата данных, который планируется внедрить до того, как версия 1.7 станет общедоступной.
- Автодополнение по Tab в интерактивной консоли, команды *require(„console“).connect()‘*, `tarantoolctl enter` и `tarantoolctl connect`. Проблемы 86 и 1790. Используйте клавишу TAB для автодополнения имен переменных, функций и метаметодов в Lua.
- Новая реализация `net.box` с улучшенной производительностью и решением проблем, когда сборщик мусора в Lua работает с недоступными соединениями. Проблемы 799, 800, 1138 и 1750.
- Появилась компрессия снимков memtx и xlog-файлов на лету с использованием быстрого алгоритма компрессии ZStandard. Компрессия настраивается автоматически для получения оптимального соотношения между использованием ЦП и пропускной способностью диска.
- `fiber.cond()` – новый механизм синхронизации для кооперативной многозадачности. Проблема 1731.
- Tarantool теперь можно устанавливать из универсальных Snappy-пакетов (http://snapcraft.io/) с помощью команды `snap install tarantool --channel=beta`.

Новые модули и пакеты:

- curl - неблокирующие привязки для libcurl
- prometheus - сборщик метрик Prometheus для Tarantool'а
- gis - полнофункциональное геопространственное расширение для Tarantool'а

- mqtt - клиент MQTT-протокола для Tarantool'а
- luaossl - самый полноценный OpenSSL-модуль во вселенной Lua

Устаревшие, удаленные и несовместимые функции:

- Имена типов полей `num` и `str` объявлены устаревшими, используйте вместо них `unsigned` и `string`. Проблема 1534.
- Удалены `space:inc()` и `space:dec()` (объявлены устаревшими в версии 1.6.x). Проблема 1289.
- Функция `fiber:cancel()` теперь является асинхронной и не ждет завершения работы файбера. Проблема 1732.
- Склонная к ошибкам функция `tostring()` была удалена из API `digest`. Проблема 1591.
- Поддержка SHA-0 (`digest.sha()`) прекращается по причине обновления OpenSSL.
- `net.box` будет использовать индексы, начинающиеся с 1, для `space.name.index[x].parts`. Проблемы 1729.
- Бинарный файл Tarantool'а будет динамически связываться с `libssl.so` во время компиляции вместо загрузки во время выполнения.
- Пакеты Debian и Ubuntu будут использовать встроенную конфигурацию `systemd` вместе с вышедшими из употребления скриптами `sysvinit`.

  В `systemd` появляется возможность управления несколькими экземплярами. Чтобы обновить, выполните следующие действия:

  1. Установите новые пакеты версии 1.7.2.
  2. Убедитесь в наличии файла `ИМЯ_ЭКЗЕМПЛЯРА.lua` в директории `/etc/tarantool/instace.enabled`.
  3. Остановите ЭКЗЕМПЛЯР с помощью `tarantoolctl stop ИМЯ_ЭКЗЕМПЛЯРА`.
  4. Запустите ЭКЗЕМПЛЯР с помощью `systemctl start tarantool@ИМЯ_ЭКЗЕМПЛЯРА`.
  5. Включите ЭКЗЕМПЛЯР во время загрузки системы с помощью `systemctl enable tarantool@ИМЯ_ЭКЗЕМПЛЯРА`.
  6. Введите команду `systemctl disable tarantool; update-rc.d tarantool remove`, чтобы отключить надстройки, совместимые с sysvinit.

  Для получения дополнительной информации см. комментарии к проблеме 1291 и главу *по администрированию серверной части*.
- Пакеты для Debian и Ubuntu запускают готовый к использованию экземпляр `example.lua` при чистой установке пакета. В экземпляре, используемом по умолчанию, предоставлены права на universe для пользователя `guest` и настроено прослушивание по «locahost:3313».
- Пакеты для Fedora 22 объявлены устаревшими (прекращение поддержки).

Версия 1.7.1

Тип версии: альфа. Дата выхода: 2016-07-11.

Объявление о выходе: https://groups.google.com/forum/#!topic/tarantool/KGYj3VKJKb8

Данная сборка представляет собой первую альфа-версию в серии 1.7. Основной функцией данной версии является новый движок базы данных под названием «vinyl». Vinyl представляет собой оптимизированный для записи движок базы данных, который позволяет сохранять объем сохраняемых данных, превышающий объем доступной памяти в 10-100 раз. Vinyl является продолжением движка Sophia из версии 1.6, а именно ответвлением и дальним родственником Sophia Дмитрия Симоненко. Новый Vinyl заменяет Sophia. Он реализован в виде журнально-структурированного дерева со слиянием

(log-structured merge tree – LSM-tree). Однако усовершенствование таких традиционных недостатков журнально-структурированных хранилищ, как низкая производительность при чтении и непредсказуемая задержка во времени при записи, стоит больших усилий. Отдельный индекс секционирован по диапазонам между многими структурами данных LSM, в каждой из который находятся собственные буферы оперативной памяти регулируемого размера. Секционирование по диапазонам позволяет осуществить слияние LSM-уровней, чтобы добиться большей детализации, а также отдать приоритет горячим диапазонам по отношению к холодным в том, что касается доступа к ресурсам, таким как оперативная память и ввод-вывод. Планировщик слияний предназначен для сведения времени задержки записи к минимуму, а также для поддержания производительности при чтении в приемлемых пределах. На сегодняшний день Vinyl поддерживает только первичные индексы. Индекс может состоять из 256 частей, как в MemTX'е, по сравнению с 8 в Sophia. Поддерживает чтение по компонентам ключа. Вскоре ожидается поддержка непоследовательных составных ключей, а также вторичных ключей. Наше намерение заключается в том, чтобы убрать любые ограничения, которые есть сейчас в Vinyl'е, чтобы сделать его полноценным компонентом Tarantool'а.

Изменения или добавления функциональности:

- Дисковый движок, который в более ранних версиях Tarantool'а назывался `sophia` или `phia`, заменен новым движком под названием `vinyl`.
- Добавлены новые типы индексируемых полей.
- Обновлена версия LuaJIT.
- Поддерживается автоматическая настройка набора реплик, что существенно упрощает настройку нового набора реплик.
- Функция `space_object:inc()` объявлена устаревшей.
- Функция `space_object:dec()` объявлена устаревшей.
- Добавлена функция `space_object:bsize()`.
- Удалена функция `box.coredump()`, аналог см. в главе *Создание дампов памяти*.
- Добавлена опция настройки `hot_standby` (горячий резерв).
- Исправленные или переименованные конфигурационные параметры:
  - `slab_alloc_arena` (в гигабайтах) в `memtx_memory` (в байтах),
  - `slab_alloc_minimal` в `memtx_min_tuple_size`,
  - `slab_alloc_maximal` в `memtx_max_tuple_size`,
  - `replication_source` в `replication`,
  - `snap_dir` в `memtx_dir`,
  - `logger` в `log`,
  - `logger_nonblock` в `log_nonblock`,
  - `snapshot_count` в `checkpoint_count`,
  - `snapshot_period` в `checkpoint_interval`,
  - `panic_on_wal_error` и `panic_on_snap_error` объединены в `force_recovery`.
- В версиях Tarantool'а до 1.8 можно использовать *устаревшие параметры* как для начальной, так и для рабочей конфигурации, но в таком случае Tarantool выдаст предупреждение. Также можно указывать как устаревшие, так и новые параметры при условии, что их значения согласованы. В противном случае, Tarantool выдаст ошибку.

- У кластера репликации появилась возможность автоматической настройки, что существенно упрощает настройку нового кластера.
- Новые индексируемые типы данных: INTEGER (целое число) и SCALAR (скаляр).
- Рефакторинг кода и улучшение производительности.
- LuaJIT обновлен до версии 2.1-beta116.

7.6 Версия 1.6

Версия 1.6.9

Тип версии: обновленная. Дата выхода: 2016-09-27. Тег версии: 1.6.9-4-gcc9ddd7.

С 15 февраля 2017 года вследствие проблемы № 2040 Удалить движок sophia из версии 1.6, движок базы данных под названием *sophia* отсутствует. В версии 1.7 его заменит движок базы данных *vinyl*.

Несовместимые изменения:

- Поддержка SHA-0 (`digest.sha()`) прекращается по причине обновления OpenSSL.
- Бинарный файл Tarantool'а будет динамически связываться с libssl.so во время компиляции вместо загрузки во время выполнения.
- Пакеты для Fedora 22 объявлены устаревшими (прекращение поддержки).

Изменения или добавления функциональности:

- Автодополнение по Tab в интерактивной консоли. Проблема 86
- Принимаются во внимание переменные окружения LUA_PATH и LUA_CPATH, как в PUC-RIO Lua. Проблема 1428
- Поиск по библиотекам `.dylib`, а также `.so` в OS X. Проблема 810.
- Новая опция `box.cfg { read_only = true }` для моделирования поведения главный-ведомый. Проблема 246
- Опция `if_not_exists = true` добавлена в box.schema.user.grant. Проблема 1683
- Функции `clock_realtime()`/`monotonic()` добавлены в общедоступный API для языка C. Проблема 1455
- Появляется `space:count(key, opts)` в качестве псевдонима для `space.index.primary:count(key, opts)`. Проблема 1391
- Обновление скрипта для 1.6.4 -> 1.6.8 -> 1.6.9. Проблема 1281
- Поддержка OpenSSL 1.1. Проблема 1722

Новые модули и пакеты:

- curl - неблокирующие привязки для libcurl
- prometheus - сборщик метрик Prometheus для Tarantool'а
- gis – полнофункциональное геопространственное расширение для Tarantool'а.
- mqtt – клиент MQTT-протокола для Tarantool'а
- luaossl - самый полноценный OpenSSL-модуль во вселенной Lua

Версия 1.6.8

Тип версии: обновленная. Дата выхода: 2016-02-25. Тег версии: 1.6.8-525-ga571ac0.

Несовместимые изменения:

- RPM-пакеты для CentOS 7 / RHEL 7 Fedora 22+ будут использовать встроенную конфигурацию systemd без устаревших скриптов sysvinit. В systemd появляется возможность управления несколькими экземплярами. Чтобы обновить, выполните следующие действия:

  1. Убедитесь в наличии файла `ИМЯ_ЭКЗЕМПЛЯРА.lua` в директории `/etc/tarantool/instace.available`.
  2. Остановите ЭКЗЕМПЛЯР с помощью `tarantoolctl stop ИМЯ_ЭКЗЕМПЛЯРА`.
  3. Запустите ЭКЗЕМПЛЯР с помощью `systemctl start tarantool@ИМЯ_ЭКЗЕМПЛЯРА`.
  4. Включите ЭКЗЕМПЛЯР во время загрузки системы с помощью `systemctl enable tarantool@ИМЯ_ЭКЗЕМПЛЯРА`.

  Директория `/etc/tarantool/instance.enabled` больше не используется для платформ, запускаемых по `systemd`.

  Для получения дополнительной информации см. главу *по администрированию серверной части*.

- Движок Sophia был обновлен до версии 2.1 для исправления ошибок upsert, нарушения целостности данных в памяти и других ошибок. Sophia версии 2.1 не поддерживает старый формат данных версии 1.1. Используйте репликацию в Tarantool'е для обновления. Проблема 1222
- Ubuntu Vivid, Fedora 20, Fedora 21 объявлены устаревшими по причине прекращения поддержки.
- i686-пакеты объявлены устаревшими. Используйте наши спецификации по RPM и DEB для сборки на своей инфраструктуре.
- Обновите `yum.repos.d` и/или apt `sources.list.d` в соответствии с инструкциями по ссылке `http://tarantool.org/download.html`

Изменения или добавления функциональности:

- Tarantool в версии 1.6.8 полностью поддерживает процессоры ARMv7 и ARMv8 (aarch64). Теперь можно будет использовать Tarantool на самых разных пользовательских устройствах от популярного Raspberry PI 2 и до плат размером с монету и безымянных мини-микро-нано-компьютеров. Проблема 1153. (На qemu также работает хорошо, но у нас нет оборудования, чтобы проверить.)
- Функции компаратора кортежей были оптимизированы, чтобы обеспечить повышение производительности на 30%, когда индексный ключ состоит из 2, 3 и более частей. Проблема 969.
- Изменения распределителя кортежей дают улучшение производительности еще на 15%. Проблема 1298
- Производительность передачи данных репликации была улучшена путем уменьшения объема данных в повторном сканировании. Проблема 11150
- В демоне создания снимков появилась произвольная задержка, что снижает возможность того, что несколько экземпляров будут делать снимки одновременно. Проблема 732.
- Движок базы данных Sophia был обновлен до версии 2.1:
  - изоляция сериализуемых снимков (SSI – Serializable Snapshot Isolation),
  - режим хранения в оперативной памяти,
  - режим хранения без кэша,
  - режим хранения в кэше с подключением к базе данных,

  - внедренный AMQ-фильтр,
  - режим LRU (удаление страниц, которые дольше всего не использовались),
  - отдельная компрессия горячих и холодных данных,
  - внедрение снимков для быстрого восстановления,
  - реорганизация и исправление ошибок в upsert,
  - новые метрики производительности.

  Обратите внимание на «Несовместимые изменения» выше.

- Возможно удаление серверов с ненулевым LSN из спейса `_cluster`. Проблема 1219.
- `net.box` теперь автоматически перезагружает схемы спейса и индексов. Проблема 1183.
- Максимальное количество индексов в спейсе было увеличено до 128. Проблема 1311.
- Новая встроенная конфигурацию `systemd` с поддержкой управления экземплярами и контролем демонов (только CentOS 7 и Fedora 22+). См. «Несовместимые изменения» выше. Проблема 1264.
- Пакет Tarantool'a принят в официальный репозиторий Fedora (https://apps.fedoraproject.org/packages/tarantool).
- Пакет Tarantool'a (OS X) принят в официальный репозиторий Homebrew (http://brewformulas.org/tarantool).
- Поддержка компилятора Clang добавлена в FreeBSD. Проблема 786.
- Добавлена поддержка библиотеки musl libc, используемой образами Alpine Linux и Docker. Проблема 1249.
- Добавлена поддержка GCC 6.0.
- Получили поддержку Ubuntu Wily, Xenial и Fedora 22, 23 и 24, для которых мы создаем официальные пакеты.
- box.info.cluster.uuid можно использовать для получения UUID кластера. Проблема 1117.
- Многочисленные исправления в документации, добавлена документация по пакетам `syslog`, `clock`, `fiber.storage`, встроенное практическое задание получило обновление.

Новые модули и пакеты:

- Tarantool перешел на новую облачную инфраструктуру на основе Docker. Новый инструмент интеграции разработки buildbot значительно уменьшает время передачи коммитов в пакеты. Официальные репозитории по ссылке http://tarantool.org теперь содержат последнюю версию сервера, модулей и коннекторов. См. http://github.com/tarantool/build
- Репозитории по ссылке http://tarantool.org/download.html were был перенесены в облачное хранилище http://packagecloud.io (при поддержке Amazon AWS). Благодарим packagecloud.io за поддержку свободного ПО!
- `memcached` – внедрение текстового и бинарного протокола memcached для Tarantool'a. Превращает Tarantool в memcached с доступом к базе данных с репликацией по схеме мастер-мастер. См. https://github.com/tarantool/memcached
- `migrate` – модуль Tarantool'a для миграции с версии 1.5 на версию 1.6. См. https://github.com/bigbes/migrate
- `cqueues` – асинхронный Lua-каркас для работы по сети с потоками и уведомлениями (разработал @daurnimator). Проблема 1204.

Версия 1.6.7

Тип версии: обновленная. Дата выхода: 2015-11-17.

Несовместимые изменения:

- Изменился синтаксис команды `upsert`, и из нее был удален дополнительный аргумент `key`. Первичный ключ для поиска всегда берется из кортежа, который является вторым аргументом в upsert. `upsert()` добавили довольно поздно в рабочем цикле, и в проекте была очевидная ошибка, которую нам пришлось исправлять. Извините.
- Функцию `fiber.channel.broadcast()` удалили, потому что ее никто не использовал, и она работала некорректно.
- Команда `reload` утилиты tarantoolctl переименована в“eval“.

Изменения или добавления функциональности:

- Опция `logger` допускает синтаксис для вывода в системный журнал syslog. Используйте синтаксис URI, чтобы определить место назначения журнала: в файл, в конвейер или syslog.
- `replication_source` принимает массив URI, так что в каждой реплике может быть до 30 узлов.
- RTREE-индекс принимает два типа функций `distance`: `euclid` и `manhattan`.
- `fio.abspath()` – новая функция в модуле `fio` для конвертации относительного пути в абсолютный.
- Название процесса теперь можно определить с помощью встроенного модуля `title`.
- В данной версии используется LuaJIT 2.1.

Новые сторонние библиотеки:

- `memcached` помогает Tarantool'у понимать бинарный протокол Memcached. Поддержка текстового протокола находится в процессе разработки и будет добавлена в отдельный модуль без изменений основных компонентов.

Версия 1.6.6

Тип версии: обновленная. Дата выхода: 2015-08-28.

Tarantool версии 1.6 больше не получает значимых новых функций, но продолжает поддерживаться. Разработчики сосредоточили свои усилия на версии 1.9.

Несовместимые изменения:

- Появляется новая схема системного спейса `_index` для размещения многомерных RTREE-индексов. Tarantool 1.6.6 нормально работает со старыми снимками и системными спейсами, но нельзя будет запустить Tarantool версии 1.6.5 с директорий, созданной в Tarantool'е версии 1.6.6, как нельзя будет ввести запрос в Tarantool 1.6.6 с net.box версии 1.6.5.
- Переименование `box.info.snapshot_pid` в `box.info.snapshot_in_progress`

Изменения или добавления функциональности:

- Потоковая архитектура для работы по сети. Сетевой ввод-вывод окончательно переведен на отдельный поток, что увеличит производительность отдельного экземпляра до 50%.
- Потоковая архитектура для создания контрольных точек. Tarantool больше не делает ответвлений для создания снимка, а использует отдельный поток, получая доступ к данным с помощью вида постоянного просмотра. Это помогает устранить скачки задержки отклика во время создания снимков.

- Хранимые процедуры на языках C/C++. Хранимые процедуры на языках C/C++ дают скорость (в 3-4 раза больше по сравнению с Lua-версией по нашим подсчетам), а также возможность неограниченного расширения. Поскольку процедуры C/C++ выполняются там же, где располагается база данных, они могут с легкостью повредить базу данных. См. *API для языка C*.
- Многомерный RTREE-индекс. RTREE-индекс теперь поддерживает большое количество измерений (до 32). Структура данных RTREE была оптимизирована так, чтобы действительно использовать R*-TREE. Мы работаем над дальнейшим улучшением индекса, в частности, над функцией конфигурации расстояния. См. https://github.com/tarantool/tarantool/wiki/R-tree-index-quick-start-and-usage
- Sophia 2.1.1 с поддержкой компрессии и составных первичных ключей. См. https://groups.google.com/forum/#!topic/sophia-database/GfcbEC7ksRg
- В бинарном протоколе и в хранимых функциях доступна новая команда `upsert`. Ключевое преимущество команды upsert в том, что она работает намного быстрее с хранилищами, оптимизированными для чтения (движок базы данных sophia), однако есть также некоторые оговорки. Для получения дополнительной информации см. проблему 905. И хотя преимущество производительности upsert наиболее очевидно с движком sophia, команда работает со всеми движками базы данных.
- Более точная информация диагностики памяти для файберов, кортежей и индексов. Используйте новую команду `box.slab.stats()` для получения подробной информация о кортежах/индексах, команда `fiber.info()` отобразит информацию о памяти, занятой файбером.
- Операции update и delete работают с использованием вторичного индекса, если индекс уникальный.
- Триггеры для аутентификации. Установите триггеры `box.session.on_auth` для отслеживания событий аутентификации. API для триггеров улучшили, чтобы он отображал все заданные триггеры, старые триггеры легко удалить.
- Разнообразные улучшения производительности встроенного модуля `net.box`.
- Оптимизация производительности BITSET-индекса.
- `panic_on_wal_error` представляет собой динамический параметр конфигурации.
- Поле iproto `sync` доступно в Lua как `session.sync()`.
- `box.once()` – новый метод для вызова кода однократно в течение срока жизни экземпляра и набора реплик. Используйте `once()` для настройки спейсов и пользователей, а также для обновления схемы в эксплуатационной среде.
- `box.error.last()` возвращает последнюю ошибку в сессии.

Новые сторонние библиотеки:

- Следующие модули LuaJIT 2.0 теперь являются встроенными: `jit.*`, `jit.dump`, `jit.util`, `jit.vmdef`. См. http://luajit.org/ext_jit.html
- `strict` – встроенный пакет, который запрещает использование необъявленных переменных в Lua. Работа ведется в таком режиме, когда Tarantool компилируется с отладкой. Чтобы включить/отключить этот режим, используйте `require('strict').on()`/`require('strict').off()` соответственно.
- `pg` и `mysql` – модули, доступные по ссылке http://rocks.tarantool.org – работают с MySQL и PostgreSQL из Tarantool'a.
- `gperftools` – модуль, доступный по ссылке http://rocks.tarantool.org – получает данные о производительности с помощью Google gperf из Tarantool'a.

- `csv` – встроенный модуль для разбора и загрузки данных в формате CSV (значения, разделенные запятыми).

Поддержка новой платформы:

- Fedora 22, Ubuntu Vivid

ГЛАВА 8

Руководство разработчика

8.1 Содействие в разработке

8.1.1 Сборка из исходных файлов

При загрузке исходных файлов и сборке Tarantool'а могут отличаться платформы и настройки, но в целом предпринимаются одинаковые действия.

1. Найдите средства и библиотеки, которые будут нужны для сборки и тестирования.

   Абсолютно необходимы следующие:

   - Программа для скачивания репозиториев исходного кода. Для всех платформ это будет `git`. Программа позволяет скачивать самый актуальный набор исходных файлов из репозитория Tarantool'а на GitHub.
   - Компилятор C/C++. Как правило, это `gcc` и `g++` версии 4.6 или более новой. На Mac OS X это `Clang` версии 3.2+.
   - Программа для управления процессом сборки. Для всех платформ это будет `CMake` версии 2.8+.
   - Средство автоматизации сборок. На всех платформах это"GNU Make".
   - библиотека ReadLine любой версии
   - библиотека ncurses любой версии
   - библиотека OpenSSL версии 1.0.1+
   - библиотека ICU последней версии
   - библиотека Autoconf любой версии
   - библиотека Automake любой версии
   - библиотека Libtool любой версии
   - библиотека Zlib-devel любой версии

- Python и его модули. Интерпретатор для Python не нужен для сборки самого Tarantool'a, если вы не планируете проводить тестирование из шага 5. Для всех платформ это будет `python` версии 2.7+ (но не 3.x). Необходимы следующие модули Python:
  - pyyaml версии 3.10
  - argparse версии 1.1
  - msgpack-python версии 0.4.6
  - gevent версии 1.1.2
  - six версии 1.8.0

Чтобы установить все необходимые зависимости, следуйте инструкциям для вашей ОС:

- Если вы используете Debian/Ubuntu, выполните команду:

```
$ apt install -y build-essential cmake make coreutils sed \
      autoconf automake libtool zlib1g-dev \
      libreadline-dev libncurses5-dev libssl-dev \
      libunwind-dev libicu-dev \
      python python-pip python-setuptools python-dev \
      python-msgpack python-yaml python-argparse python-six python-gevent
```

- Если вы используете RHEL/CentOS (версии 7 и ниже)/Fedora, выполните команду:

```
$ yum install -y gcc gcc-c++ cmake make coreutils sed \
      autoconf automake libtool zlib-devel \
      readline-devel ncurses-devel libyaml-devel openssl-devel \
      libunwind-devel libicu-devel \
      python python-pip python-setuptools python-devel \
      python-msgpack python-yaml python-argparse python-six python-gevent
```

- Если вы используете CentOS 8, выполните команды:

```
$ yum install epel-release
$ curl -s https://packagecloud.io/install/repositories/packpack/backports/script.rpm.sh␣
↪| sudo bash
$ yum install -y gcc gcc-c++ cmake make coreutils sed \
      autoconf automake libtool zlib-devel \
      readline-devel ncurses-devel openssl-devel \
      libunwind-devel libicu-devel \
      python2 python2-pip python2-setuptools python2-devel \
      python2-yaml python2-six
```

- Если вы используете Mac OS X (команды для OS X El Capitan):

  Если вы пользуетесь Homebrew в качестве менеджера пакетов, выполните команду:

```
$ brew install cmake autoconf binutils zlib \
       autoconf automake libtool \
       readline ncurses openssl libunwind-headers icu4c \
       && pip install python-daemon \
       msgpack-python pyyaml configargparse six gevent
```

  Примечание: Таким образом невозможно установить пакет zlib-devel.

  Либо загрузите стандартный пакет Xcode для разработки:

```
$ xcode-select --install
$ xcode-select -switch /Applications/Xcode.app/Contents/Developer
```

- Если вы используете FreeBSD (дальнейшие инструкции работают для FreeBSD 10.1+), выполните команду:

```
$ pkg install -y sudo git cmake gmake gcc coreutils \
      autoconf automake libtool \
      readline ncurses openssl libunwind icu \
      python27 py27-pip py27-setuptools py27-daemon \
      py27-msgpack py27-yaml py27-argparse py27-six py27-gevent
```

Если некоторые модули Python недоступны в репозитории, лучше всего произвести настройку модулей, скачав пакет в формате TAR и выполнив установку с помощью `python setup.py` следующим образом:

```
$ # На некоторых машинах может потребоваться такая начальная команда:
$ wget https://bootstrap.pypa.io/ez_setup.py -O - | sudo python

$ # Модуль Python для анализа YAML (pyYAML) для набора тестов:
$ # (Если wget не работает, проверьте на сайте http://pyyaml.org/wiki/PyYAML
$ # актуальность версии.)
$ cd ~
$ wget http://pyyaml.org/download/pyyaml/PyYAML-3.10.tar.gz
$ tar -xzf PyYAML-3.10.tar.gz
$ cd PyYAML-3.10
$ sudo python setup.py install
```

Наконец, используйте `pip` в Python, чтобы импортировать пакеты Python, которые могут быть неактуальны в репозиториях дистрибутивов. (В CentOS 7 будет необходимо сначала установить `pip` так: `sudo yum install epel-release`, а затем `sudo yum install python-pip`.)

```
$ pip install -r \
      https://raw.githubusercontent.com/tarantool/test-run/master/requirements.txt \
      --user
```

Это действие следует выполнить только один раз при первой загрузке.

2. Use `git` to download the latest Tarantool source code from the GitHub repository `tarantool/tarantool`, branch 2.2, to a local directory named `~/tarantool`, for example:

```
$ git clone --recursive https://github.com/tarantool/tarantool.git -b 2.2 ~/tarantool
```

В редких случаях вложенные модули необходимо снова обновить с помощью команды:

```
cd ~/tarantool
$ git submodule update --init --recursive
```

3. Используйте CMake, чтобы начать сборку.

```
$ cd ~/tarantool
$ make clean          # необязательно, добавлено на удачу
$ rm CMakeCache.txt   # необязательно, добавлено на удачу
$ cmake .             # начать с типом сборки = Debug (отладка)
```

На некоторых платформах может потребоваться указать версии C и C++, например:

```
$ CC=gcc-4.8 CXX=g++-4.8 cmake .
```

Чтобы указать тип сборки в CMake используется опция `-DCMAKE_BUILD_TYPE=type`, где *type* может быть:

- `Debug` – отладка, используется эксплуатационным персоналом на проекте
- `Release` – релиз, используется только при необходимости высокой производительности
- `RelWithDebInfo` – используется для сборки в эксплуатации, также предоставляет возможности отладки

Чтобы указать в CMake, что результат будет распределен, используется опция `-DENABLE_DIST=ON`. При наличии такой опции `make install` в дальнейшем установит файлы `tarantoolctl` в дополнение к файлам `tarantool`.

4. Используйте `make` для завершения сборки.

```
$ make
```

Примечание: В FreeBSD используйте вместо этого `gmake`.

При этом создается исполняемый файл „tarantool“ в директории `src/`.

Примечание: Если на данном шаге вы сталкиваетесь с ошибками `curl` или `OpenSSL`, попробуйте установить пакет `openssl111` версии `1.1.1d`.

Далее настоятельно рекомендуется выполнить команду `make install` для установки Tarantool'а в директорию `/usr/local` и поддержания порядка в системе. Однако, можно запустить исполняемый файл и без установки.

5. Проведите тестирование.

   Это необязательное действие. Разработчики Tarantool'а всегда проводят тестирование до публикации новых версий. Следует проводить тестирование, если внесены изменения в код. Итак, после загрузки в `~/tarantool` основные действия:

```
$ # создание поддиректории под названием `bin`
$ mkdir ~/tarantool/bin

$ # привязка Python к bin (могут потребовать права пользователя superuser)
$ ln /usr/bin/python ~/tarantool/bin/python

$ # переход в поддиректорию с тестами
$ cd ~/tarantool/test

$ # проведение тестирования с помощью Python
$ PATH=~/tarantool/bin:$PATH ./test-run.py
```

Вывод должен включать в себя обнадеживающие результаты, например:

```
=================================================================
TEST                                            RESULT
-----------------------------------------------------------------
box/bad_trigger.test.py                         [ pass ]
box/call.test.py                                [ pass ]
```

(continues on next page)

(продолжение с предыдущей страницы)

```
box/iproto.test.py                                  [ pass ]
box/xlog.test.py                                    [ pass ]
box/admin.test.lua                                  [ pass ]
box/auth_access.test.lua                            [ pass ]
... etc.
```

Во избежание путаницы очистите поддиректорию `bin`:

```
$ rm ~/tarantool/bin/python
$ rmdir ~/tarantool/bin
```

6. Создайте пакеты RPM и Debian.

   Это необязательное действие, которое следует выполнить только тем, кто хочет перераспределить Tarantool. Мы настоятельно рекомендуем использовать официальные пакеты с сайта tarantool.org. Однако, можно собрать пакеты RPM и Debian с помощью PackPack или путем использования средств `dpkg-buildpackage` или `rpmbuild`. Для получения более подробной информации обратитесь к документации по `dpkg` или `rpmbuild`.

7. Проверьте установку Tarantool'а:

   ```
   $ # если tarantool установлен локально после сборки
   $ tarantool
   $ # - ИЛИ -
   $ # если tarantool не установлен локально после сборки
   $ ./src/tarantool
   ```

   Tarantool запустится в интерактивном режиме.

См. также:

- Tarantool README.md

8.1.2 Управление версиями

Политика управления версия

A Tarantool release is identified by three digits, for example, 1.10.7:

- The first digit stands for a MAJOR release series that introduces some *major changes*. Up to now, there has been only one major release jump when we delivered the 2.x release series with the SQL support.
- The second digit stands for a MINOR release series that is used for introducing new *features*. *Backward incompatible changes* are possible between these release series.
- The third digit is for PATCH releases by which we reflect how stable the MINOR release series is:
  - `0` meaning **alpha**
  - `1` meaning **beta**
  - `2` and above meaning **stable**.

So, each MINOR release series goes through a development-maturity life cycle as follows:

1. **Alpha**. Once a quarter, we start off with a new alpha version, such as 2.3.0, 2.4.0, and so on. This is not what an alpha release usually means in the typical software release life cycle but rather the current

trunk version which is under heavy development and can be unstable. The current alpha version always lives in the master branch.

2. **Beta**. When all the features planned are implemented, we fork a new branch from the master branch and tag it as a new beta version. It contains 1 for the PATCH digit, e.g., 2.3.1, 2.4.1, and so on. This version cannot be called stable yet (feature freeze has just been done) although there're no known critical regressions in it since the last stable release.

3. **Stable**. Finally, after we see our beta version runs successfully in a production or development environment during another quarter while we fix incoming bugs, we declare this version stable. It is tagged with 2 for the PATCH digit, e.g., 2.3.2, 2.4.2, and so on.

   We support such version for 3 months while making another stable release by fixing all bugs found. We release it in a quarter. This last tag contains 3 for the PATCH digit, e.g., 2.3.3, 2.4.3, and so on. After the tag is set, no new changes are allowed to the release branch, and it is declared deprecated and superseded by a newer MINOR version.

   Stable versions don't receive any new features and only get backward compatible fixes.

Like Ubuntu, in terms of support, we distinguish between two kinds of stable release series:

- **LTS (Long Term Support)** is a release series that is supported for 3 years (community) and up to 5 years (paying customers). Current LTS release series is 1.10, and it receives only PATCH level releases.
- **Standard** is a release series that is supported only for a few months until the next release series enters the stable state.

Below is a diagram that illustrates the release sequence issuing described above by an example of some latest releases and release series:

```
1.10 series -- 1.10.4 -- 1.10.5 -- 1.10.6 -- 1.10.7
(LTS)

....

2.2 series --- 2.2.1 --- 2.2.2 --- 2.2.3 (end of support)
                 |
                 V
2.3 series ... 2.3.0 --- 2.3.1 --- 2.3.2 --- 2.3.3 (end of support)
                           |
                           V
2.4 series ............. 2.4.0 --- 2.4.1 --- 2.4.2
                                     |
                                     V
2.5 series ....................... 2.5.0 --- 2.5.1
                                               |
                                               V
2.6 series ................................ 2.6.0

-----------------|---------|---------|---------|------> (time)
                   1/4 yr.   1/4 yr.   1/4 yr.
```

Support means that we continue fixing bugs. We add bug fixes simultaneously into the following release series: LTS, last stable, beta, and alpha. If we look at the release diagram above, it means that the bug fixes are to be added into 1.10, 2.4, 2.5, and 2.6 release series.

To sum it up, once a quarter we release (see the release diagram above for reference):

- next LTS release, e.g., 1.10.7
- two stable releases, e.g., 2.3.3 and 2.4.2

- beta version of the next release series, e.g., 2.5.1.

In all supported releases, when we find and fix an outstanding CVE/vulnerability, we deliver a patch for that but do not tag a new PATCH level version. Users will be informed about such critical patches via the official Tarantool news channel (tarantool_news).

Мы также публикуем ночные сборки и используем четвертый слот в идентификаторе версии для обозначения номера ночной сборки.

Примечание: A release series may introduce backward incompatible changes in a sense that existing Lua, SQL or C code that are run on a current release series may not be run with the same effect on a future series. However, we don't exploit this rule and don't make incompatible changes without appropriate reason. We usually deliver information how mature a functionality is via release notes.

Please note that binary data layout is always compatible with the previous series as well as with the LTS series (an instance of `X.Y` version can be started on top of `X.(Y+1)` or `1.10.z` data); binary protocol is compatible too (client-server as well as replication protocol).

Release list

Below is the table containing all Tarantool releases starting from 1.10.0 up to the current latest versions (as of September 1, 2020). For each release series, releases are sorted out as alpha, beta, and stable ones.

Release series	Alpha	Beta	Stable
1.10 (LTS)	1.10.0	1.10.1	1.10.2 1.10.3 1.10.4 1.10.5 1.10.6 1.10.7
2.1	2.1.0	2.1.1	2.1.2 2.1.3
2.2	2.2.0	2.2.1	2.2.2 2.2.3
2.3	2.3.0	2.3.1	2.3.2 2.3.3
2.4	2.4.0	2.4.1	2.4.2
2.5	2.5.0	2.5.1	
2.6	2.6.0		

Как собрать минорную версию

```
$ git tag -a 2.4 -m "Next minor in 2.x series"
$ vim CMakeLists.txt # редактировать CPACK_PACKAGE_VERSION_PATCH
$ git push --tags
```

Тег, который делается на ветке git, можно забрать при слиянии или оставить на ветке. Метод «сохранить тег на ветке, на которой он был первоначально установлен», заключается в использовании `--no-fast-forward` при слиянии этой ветки.

С помощью `--no-ff` создается набор изменений при слиянии для пояснения полученных изменений, и только этот набор изменений при слиянии оказывается в ветке назначения. Этот метод можно использовать, когда есть две активные линии разработки, например, «стабильная» и «следующая», и необходимо иметь возможность помечать тегами линии независимо друг от друга.

Чтобы убедиться, что тег не окажется в ветке назначения, необходимо, чтобы коммит, к которому привязан тег, остался в исходной ветке. Это и происходит при отключенном «fast-forward» – создается коммит для слияния и добавляется в обе ветки.

Вот как это может выглядеть:

```
kostja@shmita:~/work/tarantool$ git checkout master
Already on 'master'
kostja@shmita:~/work/tarantool$ git tag -a 2.4 -m "Next development"
kostja@shmita:~/work/tarantool$ git describe
2.4
kostja@shmita:~/work/tarantool$ git checkout master-stable
Switched to branch 'master-stable'
kostja@shmita:~/work/tarantool$ git tag -a 2.3 -m "Next stable"
kostja@shmita:~/work/tarantool$ git describe
2.3
kostja@shmita:~/work/tarantool$ git checkout master
Switched to branch 'master'
kostja@shmita:~/work/tarantool$ git describe
2.4
kostja@shmita:~/work/tarantool$ git merge --no-ff master-stable
Auto-merging CMakeLists.txt
Merge made by recursive.
 CMakeLists.txt |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
kostja@shmita:~/work/tarantool$ git describe
2.4.0-0-g0a98576
```

Кроме того, следует помнить:

1. Обновляйте все задачи. Обновляйте журнал изменений ChangeLog на основании вывода `git log`.

   Журнал изменений ChangeLog должен включать в себя только пункты, указанные в задачах на GitHub. Если что-то значительное не указано, значит, что-то пошло не так при планировании версии, и ее выход следует отложить до выяснения причин.

2. Нажимайте „Release milestone“ (создать промежуточную версию). Создавайте промежуточные версии для следующей минорной версии. Указывайте драйверу на дефекты и проекты для новой промежуточной версии.

8.2 Рекомендации

8.2.1 Рекомендации для разработчиков

Как работать над дефектами

На любой дефект, даже незначительный, если он изменяет доступное пользователю поведение сервера, необходимо составить отчет об ошибке. Сообщите о дефекте по ссылке http://github.com/tarantool/tarantool/issues.

Когда вы сообщаете об ошибке, постарайтесь сразу же приступить к тестовому сценарию. Установите текущую контрольную точку для исправления ошибки и укажите серию. Назначьте задачу на себя. Укажите статус «In progress» (выполняется). Как только патч готов, укажите статус ошибки «In review» (на рассмотрении) и отправьте версию с исправленными ошибками на рассмотрение.

После успешного рассмотрения кода опубликуйте патч и укажите статус «Closed» (закрыт).

Патчи для исправления ошибок должны содержать ссылку на соответствующую страницу дефекта Launchpad или хотя бы идентификатор дефекта. Каждому патча должен соответствовать отдельный тест, если только это не слишком трудно сделать в текущем окружении, и в этом случае следует предупредить тестировщиков.

Когда ваш патч доходит до главной ветки проекта, нужно сделать следующее:

- перевести статус ошибки в „fix committed“ (исправлено),
- удалить отдельную ветку.

Как писать сообщение о коммите

Любой коммит следует описать в полезном сообщении. Следуйте нижеприведенным рекомендациям при коммитах в любой репозиторий Tarantool'a на GitHub.

1. Отделяйте тему от тела сообщения пустой строкой.
2. Постарайтесь ограничить тему сообщения примерно **50 символами**.
3. Начните тему сообщения с прописной буквы, если ей не предшествует префикс с именем подсистемы и точка с запятой:
   - memtx:
   - vinyl:
   - xlog:
   - replication:
   - recovery:
   - iproto:
   - net.box:
   - lua:
   - sql:
4. Не заканчивайте тему сообщения точкой.
5. Не пишите «gh-xx», «closes #xxx» в строке темы.
6. В теме сообщения используйте повелительное наклонение. Правильно оформленная тема Git-коммита должна корректно дополнять следующее предложение: «Если применить, коммит */здесь тема сообщения/*».
7. Уместите тело сообщения в примерно **72 символа**.
8. Используйте тело сообщения, чтобы объяснить, **что и почему**, а не как.
9. Привяжите задачи на GitHub в последних строках (см. как).
10. Используйте настоящие имя и адрес электронной почты. Членам проектной команды Tarantool'a рекомендуется указывать почту на **@tarantool.org**, но это необязательно.

Шаблон:

```
Кратко сформулируйте изменения в пределах 50 символов.

При необходимости, более подробные объяснения.
Уместите детали в примерно 72 символов.
Иногда первая строка считается темой
коммита, а остальной текст -- телом сообщения.
Критически важна пустая строка, которая отделяет тему от тела сообщения
(если только тело не отсутствует совсем); различные средства, такие как `log`,
`shortlog` и `rebase` могут их перепутать, если нет разделения.
```

(continues on next page)

(продолжение с предыдущей страницы)

```
Объясните проблему, которую решает данный коммит. Уделите внимание тому, почему
вы вносите эти изменения, а не как (это объясняется в коде).
Есть ли побочные эффекты или другие неочевидные последствия применения этих
изменений? Здесь можно объяснить их.

Следующие абзацы идут после пустых строк.

- Можно также использовать элементы в списке.

- Как правило, в качестве маркера применяется дефис или звездочка, которой предшествует
  пробел, а между строками вставляются пустые строки, но в данном случае
  условные обозначения могут разниться.

Исправляет: #123
Закрывает: #456
Необходим для: #859
См. также: #343, #789
```

Некоторые реальные примеры:

- tarantool/tarantool@2993a75
- tarantool/tarantool@ccacba2
- tarantool/tarantool@386df3d
- tarantool/tarantool@076a842

Основано на [1] и [2].

Как отправить патч на рассмотрение

Мы не принимаем запросы на включение в проект на GitHub. Вместо этого все патчи следует отправлять в виде обычного текстового сообщения по адресу tarantool-patches@dev.tarantool.org. Подпишитесь на рассылку https://lists.tarantool.org/mailman/listinfo/tarantool-patches, чтобы убедиться, что ваши сообщения добавляются в архив.

1. **Подготовка патча**

После коммита патча в локальный репозиторий git вы можете отправить его на рассмотрение.

Чтобы подготовить сообщение, воспользуйтесь командой `git format-patch`:

```
$ git format-patch -1
```

В результате последний коммит в локальном репозитории git будет отформатирован в виде обычного текстового сообщения в файл в текущей директории. Название файла будет выглядеть так: `0001-тема-коммита.patch`. Чтобы указать другую директорию, используйте опцию `-o`:

```
$ git format-patch -1 -o ~/patches-to-send
```

После форматирования патча его можно просмотреть и отредактировать в вашем любимом текстовом редакторе (всё-таки это файл с обычным текстом!) Мы настоятельно рекомендуем добавить следующее:

- ссылка на ветку, где можно найти этот патч на GitHub, а также
- ссылку на проблему на GitHub, которую решает ваш патч.

Если патч всего один, журнал изменений должен идти сразу после `---` в теле сообщения (тогда `git am` проигнорирует его).

Если же вы хотите отправить сразу несколько патчей (например, это важная функция, для которой нужны несколько предварительных патчей), каждый из них следует отправлять в отдельном сообщении в ответ на сопроводительное письмо. Чтобы соответствующим образом отформатировать серию патчей, передайте следующие опции в `git format-patch`:

```
$ git format-patch --cover-letter --thread=shallow HEAD~2
```

где:

- `--cover-letter` заставит `git format-patch` сгенерировать сопроводительное письмо;
- `--thread=shallow` отметит каждое сообщение с отформатированными патчами, которые следует отправить в ответ на сопроводительное письмо;
- `HEAD~2` (мы используем вместо `-1`) заставит `git format-patch` форматировать последние два патча в локальной ветке git, а не один. Чтобы форматировать три патча, используйте `HEAD~3`, и так далее.

После успешного выполнения этой команды все ваши патчи будут отформатированы в виде отдельных сообщений в текущей директории (или в директории, указанной с помощью опции `-o`):

```
0000-cover-letter.patch
0001-first-commit.patch
0002-second-commit.patch
...
```

В теме и теле сопроводительного письма будут рекламные аннотации. Вам нужно их отредактировать перед отправкой (опять же, это обычный текст). Просьба указать следующее:

- короткое описание в теме сообщения;
- несколько слов о каждом патче в теле сообщения.

Кроме того, не забудьте добавить ссылки на проблему на GitHub и на ветку, где можно найти серию патчей. В таком случае нет необходимости указывать ссылки или дополнительную информацию в каждом отдельном письме, поскольку всё необходимое уже будет в сопроводительном письме.

Примечание: Чтобы не указывать опции `--cover-letter` и `--thread=shallow`, можно добавить в gitconfig следующие строки:

```
[format]
    thread = shallow
    coverLetter = auto
```

---

2. **Отправка патча**

После форматирования патчей их можно отправлять по электронной почте. Конечно, можно воспользоваться и любимым почтовым клиентом, но гораздо проще отправить их с помощью `git send-email`. Перед использованием команды ее необходимо настроить.

Если используется учетная запись GMail, добавьте следующий код в `.gitconfig`:

```
[sendemail]
    smtpencryption = tls
    smtpserver = smtp.gmail.com
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    smtpserverport = 587
    smtpuser = your.name@gmail.com
    smtppass = topsecret
```

Для пользователей mail.ru настройки будут слегка отличаться:

```
[sendemail]
    smtpencryption = ssl
    smtpserver = smtp.mail.ru
    smtpserverport = 465
    smtpuser = your.name@mail.ru
    smtppass = topsecret
```

Если ваша учетная запись электронной почты находится на другом ресурсе, уточните SMTP-настройки у поставщика услуг.

После настройки используйте следующую команду для отправки патчей:

```
$ git send-email --to tarantool-patches@dev.tarantool.org 00*
```

(подстановочный символ `00*` будет распространяться на список патчей, сгенерированных в предыдущем шаге.)

Если вы бы хотели, чтобы определенный человек рассматривал ваш патч, добавьте его в список получателей, передав `--to` или `--cc` для каждого получателя.

Примечание: Неплохо проверить, что `git send-email` будет работать должным образом, не отправив ничего на весь мир. Для этого воспользуйтесь опцией `--dry-run`.

---

3. **Процесс рассмотрения**

После отправки патчей вы ожидаете их рассмотрения. Редактор отправит свои комментарии в ответ на сообщение с патчем, который нуждается в доработке, по его мнению.

Получив электронное письмо с примечаниями, вы внимательно читаете его и отвечаете, согласны вы или нет. Обратите внимание, что мы используем стиль ответа с чередованием (он же «встроенный ответ») в сообщениях электронной почты.

Достигнув соглашения, вы отправляете доработанный патч в ответ на последнее сообщение в обсуждении. Чтобы отправить патч, вы можете либо вложить простой diff (созданный с помощью `git diff` или `git format-patch`) в сообщение электронной почте и отправить его с помощью вашего любимого почтового клиента, либо использовать опцию `--in-reply-to` команды `git send-email`.

Если вы считаете, что общий набор изменений достаточно велик, чтобы отправить всю серию заново и перезапустить процесс рассмотрения в рамках нового обсуждения, вы снова генерируете сообщения с патчами с помощью `git format-patch`, на этот раз добавив v2 (затем v3, v4 и так далее) в тему и журнал изменений в тело сообщения. Чтобы соответствующим образом изменить тему сообщения, используйте опцию `--subject-prefix` в команде `git format-patch`:

```
$ git format-patch -1 --subject-prefix='PATCH v2'
```

Чтобы добавить журнал изменений, откройте созданное сообщение с помощью любимого текстового редактора и отредактируйте тело сообщения. Если патч всего один, журнал изменений должен идти сразу после `---` в теле сообщения (тогда `git am` проигнорирует его). Если патчей несколько, журнал изменений следует добавить в сопроводительное письмо. Хороший пример журнала изменений:

```
Changes in v3:
  - Fixed comments as per review by Alex
  - Added more tests
 Changes in v2:
  - Fixed a crash if the user passes invalid options
  - Fixed a memory leak at exit
```

Также правильно будет добавить ссылку на предыдущую версию набора патчей (гиперссылку или идентификатор сообщения).

Примечание:

- Не спорьте с редактором без веских аргументов в свою поддержку.
- Не принимайте любые слова редактора без доказательств. Редакторы – тоже люди, которые могут ошибаться.
- Не ждите, что редактор скажет вам, как что делать. Это не их работа. Редактор может предложить пути решения проблемы, но вообще говоря, это ваша обязанность.
- Не забывайте обновлять удаленную ветку git каждый раз, когда отправляете новую версию патча.
- Соблюдайте вышеуказанные рекомендации. Если вы не будете их соблюдать, ваши патчи могут быть молча проигнорированы.

8.2.2 Рекомендации по написанию документации

Данные рекомендации обновляются по запросу, охватывая только те проблемы, которые вызывают вопросы у авторов документации. На данный момент мы не стремимся разработать исчерпывающее руководство по написанию документации для проекта Tarantool.

Вопросы по разметке

Перенос текста

Строка ограничена 80 символами для обычного текста и никак не ограничена для любых других конструкций, когда обтекание влияет на читаемость ReST и / или HTML-вывод. Кроме того, нет смысла переносить текст в строках короче 80 символов, если у вас для этого нет веских оснований.

Ограничение в 80 символов исходит из разрешения экрана ISO/ANSI 80x24, и маловероятно, что читатели/писатели будут использовать 80-символьные консоли. Тем не менее, такое ограничение по-прежнему является стандартом во многих рекомендациях по программированию (включая Tarantool). Что касается писателей, то благодаря ограничению размера страницы окно с текстом может быть довольно узким, оставляя больше места для других приложений в широкоэкранном окружении.

Форматирование фрагментов кода

Для фрагментов коды мы обычно используем директиву `code-block` с соответствующей подсветкой синтаксиса языка. Чаще всего используем следующее:

- `.. code-block:: tarantoolsession`
- `.. code-block:: console`

- `.. code-block:: lua`

Например (фрагмент Lua-кода):

```
for page in paged_iter("X", 10) do
  print("New Page. Number Of Tuples = " .. #page)
  for i=1,#page,1 do print(page[i]) end
end
```

В редких случаях при необходимости подсветить отдельные части фрагмента кода, когда директивы `code-block` недостаточно, мы используем директиву `codenormal` построчно вместе с явным форматированием вывода (как указано в `doc/sphinx/_static/sphinx_design.css`).

Примеры:

- Синтаксис функции (объект-заполнитель *имя-спейса* отображается курсивом):

  box.space.имя-спейса:create_index(„index-name“)

- Сессия tdb (ввод информации пользователем выделяется жирным шрифтом, приглашение на ввод команды – синим, вывод – зеленым):

  ```
  $ tarantool example.lua
  (TDB)  Tarantool debugger v.0.0.3. Type h for help
  example.lua
  (TDB)  [example.lua]
  (TDB)  3: i = 1
  ```

Внимание: Каждая запись с явным форматированием вывода (`codenormal`, `codebold` и т.п.) часто вызывает трудности при переводе документации на другие языки. Постарайтесь избегать специального форматирования, если только без него никак НЕЛЬЗЯ обойтись.

Использование разделенных ссылок

Избегайте разделения ссылки и определения цели (ref), например:

```
Это абзац, который содержит `ссылку`_.

 .. ссылка: http://example.com/
```

Используйте неразделенные ссылки:

```
Это абзац, который содержит `ссылку <http://example.com/>`_.
```

Внимание: Каждая разделенная ссылка часто вызывает трудности при переводе документации на другие языки. Постарайтесь избегать разделенных ссылок, если только без них никак НЕЛЬЗЯ обойтись (например, в таблицах).

Создание меток для локальных ссылок

Мы стараемся не использовать автоматически сгенерированные sphinx ссылки для большинства объектов. Вместо них мы добавляем собственные метки для ссылок на любое место в документации.

Соглашение об именовании заключается в следующем:

- Набор символов: от a до z, от 0 до 9, дефис, подчеркивание.

- Формат: `путь дефис имя файла дефис тег`

  Пример: `_c_api-box_index-iterator_type` где: `c_api` – имя директории, `box_index` – имя файла (без «.rst»), а `iterator_type` – тег.

Имя файла используется для того, чтобы понять, куда указывает «ref». И если имя файла имеет смысл, это гораздо понятнее.

Имени файла без пути достаточно, когда оно уникально в пределах `doc/sphinx`. Поэтому для файла `fiber.rst` достаточно будет «fiber», а не «reference-fiber». Тогда как для «index.rst» (а у нас множество файлов «index.rst» в разных директориях) необходимо указать путь до имени файла, например, «reference-index».

Используйте дефис «-«, чтобы разграничить путь и имя файла. В исходном коде документации мы пользуемся только символами подчеркивания «_» при указании пути и имени файла, оставляя дефисы «-» для разграничения в локальных ссылках.

Тег может содержать любую значимую информацию. Единственная рекомендация дается для элементов синтаксиса Tarantool'a, где предпочтительно использовать следующий синтаксис в тегах: `имя_объекта_или_модуля дефис имя_элемента`. Например, `box_space-drop`.

Добавление комментариев

Иногда могут потребоваться комментарии в файле ReST. Чтобы sphinx не учитывал этот текст во время обработки, используйте следующую запись в каждой строке в качестве маркера комментария («.. //»):

```
.. // здесь комментарий
```

Начальные символы «.. //» не пересекаются с другими символами разметки ReST, и их легко обнаружить как визуально, так и с помощью grep. В поиске grep нет символов, которые нужно избегать, просто выполните примерно следующее:

```
$ grep ".. //" doc/sphinx/dev_guide/*.rst
```

Тем не менее, эти комментарии не сработают должным образом во вложенной документации (например, если оставить комментарий в модуле -> объекте -> методе, sphinx игнорирует комментарий и всё вложенное содержимое, который следует в описании метода).

Вопросы по стилю и языку

Британский или американский вариант английского

В английской версии документации мы придерживаемся американского варианта английского языка.

Экземпляр или сервер

Ссылаясь на экземпляр Tarantool-сервера, мы говорим «экземпляр», а не «сервер». Это обеспечивает однородность терминологии в руководстве и именами в окружении Tarantool'a (например, `/etc/tarantool/instances.enabled` – активные экземпляры).

Неправильно: «С помощью репликации несколько *серверов* Tarantool'a могут работать на копиях одинаковых баз данных.»

Правильно: «С помощью репликации несколько *экземпляров* Tarantool'а могут работать на копиях одинаковых баз данных.»

Примеры и шаблоны

Модуль и функция

Ниже приводится пример документирования модуля (`my_fiber`) и функции (`my_fiber.create`).

`my_fiber.create(`*function*[, *function-arguments*]`)`
Создание и запуск `my_fiber`. Происходит создание объекта, который незамедлительно начинает работу.

Параметры

- `function` – функция, которая будет связана с `my_fiber`
- `function-arguments` – что передается в функцию

возвращает созданный объект `my_fiber`

тип возвращаемого значения пользовательские данные

Пример:

```
tarantool> my_fiber = require('my_fiber')
---
...
tarantool> function function_name()
         >   my_fiber.sleep(1000)
         > end
---
...
tarantool> my_fiber_object = my_fiber.create(function_name)
---
...
```

Модуль, класс и метод

Ниже приводится пример документирования модуля (`my_box.index`), класса (`my_index_object`) и функции (`my_index_object.rename`).

`object my_index_object`

`my_index_object:rename(`*index-name*`)`
Переименование индекса.

Параметры

- `index_object` – ссылка на объект
- `index_name` – новое имя для индекса (тип = строка)

возвращает nil

Возможные ошибки: index_object не существует.

Пример:

```
tarantool> box.space.space55.index.primary:rename('secondary')
---
...
```

Факторы сложности: Размер индекса, тип индекса, количество кортежей, к которым получен доступ.

8.2.3 Руководство по написанию кода на C

Стиль программирования проекта основан на версии стиля программирования ядра Linux.

Последнюю версию стиля программирования Linux можно найти по ссылке: http://www.kernel.org/doc/Documentation/CodingStyle

Мы придерживаемся версии от 13 июля 2007 года, которая приводится ниже в документе.

Здесь мы приводим дополнительные рекомендации, которые либо специфичны для Tarantool'a, либо отличаются от рекомендаций по программированию ядра Linux.

A. Следующие главы не применимы, поскольку они специфичны для среды программирования ядра Linux: 10 «Конфигурационные файлы Kconfig», 11 «Структуры данных», 13 «Вывод сообщений ядра», 14 «Выделение памяти» и 17 «Не изобретайте макросы снова».

B. Остальные главы документа «Стиль программирования ядра Linux» изменяются следующим образом:

Общие рекомендации

We use Git for revision control. The latest development is happening in the default branch (currently `master`). Our git repository is hosted on GitHub, and can be checked out with `git clone git://github.com/tarantool/tarantool.git` (anonymous read-only access).

Если у вас есть вопросы о внутреннем устройстве Tarantool'a, разместите их в списке вопросов к обсуждению для разработчиков: https://groups.google.com/forum/#!forum/tarantool. Однако, предупреждаем: Launchpad молча удаляет сообщения от тех, кто не является подписчиком, поэтому обязательно подпишитесь на список перед публикацией. Кроме того, несколько инженеров всегда находятся на канале #tarantool в irc.freenode.net.

Стиль комментирования кода

Используйте формат комментирования Doxygen, разновидность Javadoc, то есть *@tag* вместо *\tag*. Основные используемые теги: @param, @retval, @return, @see, @note и @todo.

Каждая функция, за исключением, пожалуй, очень короткой и очевидной, должна быть прокомментирована. Пример комментария функции может выглядеть следующим образом:

```
/** Запись всех данных в дескриптор.
 *
 * Эта функция аналогична 'write' во всём кроме того, что она обеспечивает
 * запись всех данных в файл, если не возникает ошибка,
 * которую нельзя игнорировать.
 *
 * @retval 0  Выполнено
 *
 * @retval  1  Ошибка (не EINTR)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
 * /
static int
write_all(int fd, void \*data, size_t len);
```

Доступные структуры и важные элементы структуры также должны быть прокомментированы.

Файлы заголовка

Используйте защиту заголовка. Поместите защиту заголовка в первую строку заголовка до авторского права или объявления. Для защиты заголовка используйте имя в верхнем регистре. Выводите имя защиты заголовка из имени файла и добавьте _INCLUDED, чтобы получить имя макроса. Например, core/log_io.h -> CORE_LOG_IO_H_INCLUDE. В файле `.c` (реализация) следует включить соответствующий заголовок с объявлением перед всеми другими заголовками, чтобы убедиться, что заголовок является автономным. Заголовок «header.h» является автономным, если компилируется без ошибок:

```
#include "header.h"
```

Выделение памяти

Предпочтительно использовать предоставляемые распределители slab'ов (salloc) и пулов (palloc) вместо malloc()/free() для любых операций выделения памяти большого объема. Многократное использование malloc()/free() может привести к фрагментации памяти, чего следует избегать.

Всегда освобождайте всю выделенную память, даже выделенную при запуске. Мы стремимся к тому, чтобы valgrind не находил утечек памяти, и в большинстве случаев так же легко освободить выделенную память по free(), как и записать подавление valgrind. Освобождение всей выделенной памяти также помогает динамическому балансированию нагрузки: предполагается, что подключаемый модуль может динамически загружаться и выгружаться несколько раз, перезагрузка не должна приводить к утечке памяти.

Именование функций

Our convention is to use:

- `new`/`delete` для функций, которые выделяют + инициализируют и удаляют + освобождают объект,
- `create`/`destroy` для функций, которые инициализируют/удаляют объект, но не занимаются управлением памятью,
- `init`/`free` для функций, которые инициализируют/удаляют библиотеки и подсистемы.

Прочее

Допускаются расширения GNU C99. Можно смешивать операторы и объявления в выражениях.

Не слишком актуальный список всех расширений семейства языка C можно найти по ссылке: http://gcc.gnu.org/onlinedocs/gcc-4.3.5/gcc/C-Extensions.html

Стиль программирования ядра Linux

В данном коротком документе описывается предпочтительный стиль программирования для ядра Linux. Стиль программирования – это личное дело каждого, и я не буду никому _навязывать_ свои убеждения, но поскольку это касается всего, что я должен поддерживать, я бы предпочел, чтобы эти правила использовали повсеместно. Пожалуйста, хотя бы рассмотрите описываемые здесь пункты.

Для начала я предлагаю вам распечатать копию стандартов написания кода GNU и НЕ читать их. Сожгите их в качестве весьма символического жеста.

В любом случае, поехали:

Глава 1: Отступы

Табуляция составляет 8 символов, то есть отступы будут также в 8 символов. Появляются отступнические движения, которые призывают делать отступы в 4 (или даже 2!) символа, а это сродни попытке округлить число Пи до 3.

Обоснование: Основная идея отступов состоит в том, чтобы показать, где начинается и заканчивается логический блок кода. Особенно если вы смотрите на один и тот же код в течение 20 часов, трудно не заметить пользу больших отступов.

Некоторые могут возразить, что отступ в 8 символов делает код слишком широким, особенно на 80-знаковой строке терминала. Ответ: Если вам понадобилось более трех уровней отступа, вы что-то делаете неправильно, и вам следует переписать этот участок.

Короче говоря, отступы в 8 символов облегчают чтение кода, да еще и предупреждают, когда вы слишком глубоко встраиваете свои функции. Прислушайтесь к этому.

Лучше всего упростить несколько уровней отступов в операторе switch, выравнивая «switch» и его вспомогательные метки «case» в одном столбце вместо использования двойных отступов для меток «case», например:

```
switch (suffix) {
case 'G':
case 'g':
    mem <<= 30;
    break;
case 'M':
case 'm':
    mem <<= 20;
    break;
case 'K':
case 'k':
    mem <<= 10;
    /* fall through */
default:
    break;
}
```

Не размещайте несколько операторов на одной строке, если вам нечего скрывать:

```
if (condition) do_this;
  do_something_everytime;
```

И не размещайте несколько операторов присваивания на одной строке. Стиль программирования ядра чрезвычайно прост. Избегайте сложных выражений.

За пределами комментариев, документации и Kconfig, пробелы никогда не используются для отступов, и приведенный выше пример намеренно нарушен.

Найдите достойный редактор и не оставляйте пробелы в конце строки.

Глава 2: Разрыв длинных строк

Смысл стиля программирования заключается в читаемости и удобстве сопровождения с использованием общедоступных средств.

Длина строк ограничена 80 символами, и этому следует уделить особое внимание. Для комментариев установлен тот же лимит в 80 символов.

Операторы длиной более 80 символов будут разбиты на логические части. Последующие части значительно короче основной и смещены вправо. То же относится к заголовкам функций с длинным списком аргументов. Длинные строки также разбиваются на более короткие строки. Единственным исключением может быть случай, если превышение ограничений повысит читаемость и не скроет необходимую информацию.

```
void fun(int a, int b, int c)
{
    if (condition)
        printk(KERN_WARNING "Warning this is a long printk with "
                        "3 parameters a: %u b: %u "
                        "c: %u \n", a, b, c);
    else
        next_statement;
}
```

Глава 3: Фигурные скобки и пробелы

Другой проблемой, которая всегда возникает в программировании на C, является размещение фигурных скобок. В отличие от отступов, есть несколько технических обоснований, чтобы выбрать один способ, а не другой, но предпочтительно, как нам показали великие Керниган и Ричи, поместить открывающую скобку в конце строки, а закрывающую в начале новой строки:

```
if (x is true) {
    we do y
}
```

Это применимо ко всем блокам операторов без функций (if, switch, for, while, do), например:

```
switch (action) {
case KOBJ_ADD:
    return "add";
case KOBJ_REMOVE:
    return "remove";
case KOBJ_CHANGE:
    return "change";
default:
    return NULL;
}
```

И только в особенных случаях, а именно для функций, открывающая скобка размещается в начале следующей строки:

```
int function(int x)
{
    body of function;
}
```

Отступники по всему миру утверждали, что такая несогласованность ... ну ... несогласованна, но все здравомыслящие люди знают: (а) K&R _правы_, (б) K&R правы. Кроме того, функции в любом случае будут особенными (в C их нельзя вложить).

Обратите внимание, что за закрывающей скобкой на отдельной строке ничего нет, _кроме_ тех случаев, когда за ней следует продолжение того же оператора, то есть «while» в do-операторе или «else» в if-операторе, например:

```
do {
    body of do-loop;
} while (condition);
```

и

```
if (x == y) {
    ..
} else if (x > y) {
    ...
} else {
    ....
}
```

Обоснование: K&R.

Кроме того, обратите внимание, что такое расположение скобок также сводит к минимуму количество пустых (или почти пустых) строк без потери читаемости. Таким образом, поскольку новые строки на экране – это не возобновляемый ресурс (вспомним о 25-строчных экранах терминала), у вас будет больше пустых строк для комментариев.

Не используйте лишние фигурные скобки, если нужен всего один оператор.

```
if (condition)
    action();
```

Это не применимо, если одна ветка условного оператора – это отдельный оператор. Используйте фигурные скобки в обеих ветках.

```
if (condition) {
    do_this();
    do_that();
} else {
    otherwise();
}
```

Глава 3.1: Пробелы

Стиль программирования ядра Linux в том, что касается пробелов, зависит (в основном) от использования функции или ключевого слова. Используйте пробел после (большинства) ключевых слов. Значимые исключения: sizeof, typeof, alignof и __attribute__, которые похожи на функции (и обычно используются с круглыми скобками в Linux, хотя они и не требуются, как в объявлении «sizeof info» после «struct fileinfo info;»).

Поэтому добавляйте пробел после следующих ключевых слов: if, switch, case, for, do, while, но не для sizeof, typeof, alignof или __attribute__. Пример:

```
s = sizeof(struct file);
```

Не добавляйте пробелы вокруг (внутри) выражений в круглых скобках. Этот пример **неправильный**:

```
s = sizeof( struct file );
```

Объявляя данных типа указателя или функцию, которая возвращает тип указателя, предпочтительно использовать „*“ рядом с именем данных или именем функции, а не рядом с именем типа. Примеры:

```
char *linux_banner;
unsigned long long memparse(char *ptr, char **retptr);
char *match_strdup(substring_t *s);
```

Добавляйте по одному пробелу вокруг (с каждой стороны) большинства знаков двухместных и трехместных операций, например, любое из следующих:

= + - < > * / % | & ^ <= >= == != ? :

но не добавляйте пробелы после знаков одноместных операций:

& * + - ~ ! sizeof typeof alignof __attribute__ defined

не нужны пробелы перед знаками одноместных операций увеличения или уменьшения постфикса:

++ --

не нужны пробелы после знаков одноместных операций увеличения или уменьшения префикса:

++ --

и не нужны пробелы вокруг знаков элементов структуры „.“ и «->».

Не оставляйте пробелы на концах строк. Некоторые редакторы с «умным» отступом вставляют пробелы в начале новых строк, поэтому вы можете сразу ввести следующую строку кода. Однако некоторые такие редакторы не удаляют пробелы, если вы не пишете там код, например, если вы оставите пустую строку. В результате имеем строки с пробелами в конце.

Git предупредит, если патчи содержат пробелы в конце строк, и может по желанию удалить пробелы за вас; однако, в серии патчей, это может привести к тому, что последующие патчи в серии не применятся, поскольку изменены контекстные строки.

Глава 4: Именование

C – это спартанский язык, и именование должно быть спартанским. В отличие от разработчиков на Modula-2 и Pascal, разработчики на языке C не используют забавные имена, такие как ThisVariableIsATemporaryCounter. Разработчик на языке C назвал бы такую переменную «tmp», что намного легче написать и не менее сложно понять.

ОДНАКО, хотя на имена со смешанным регистром смотрят неодобрительно, обязательным требованием будут описательные имена глобальных переменных. Назвать глобальную функцию «foo» – это оскорбление.

У ГЛОБАЛЬНЫХ переменных (которые надо использовать, только если без них НЕЛЬЗЯ обойтись) должны быть описательные имена, равно как и у глобальных функций. Если у вас есть функция, которая подсчитывает количество активных пользователей, нужно назвать ее «count_active_users()» или как-то похоже, _НЕ_ следует называть ее «cntusr()».

Кодирование типа функции в названии (так называемая венгерская нотация) – это признак плохого тона, поскольку компилятор в любом случае знает типы и может их проверять, и это только путает программиста. Неудивительно, что MicroSoft делает глючные программы.

Имена ЛОКАЛЬНЫХ переменных должны быть короткими и точными. Если у вас есть счетчик случайных целых чисел, его следует называть «i». Назвать его «loop_counter» будет непродуктивно, если нет никаких шансов, что его перепутают. Аналогично «tmp» может быть практически любым типом переменной, которая используется для хранения временного значения.

Если вы боитесь перепутать имена своих локальных переменных, у вас другая проблема, которая называется синдромом дисбаланса гормона роста функций. См. Главу 6 (Функции).

Глава 5: Директива Typedef

Не используйте что-то вроде «vps_t».

Будет _ошибкой_ использовать typedef для определения структур и указателей. Если вы видите

```
vps_t a;
```

в исходном коде, что это означает?

И наоборот, если говорится

```
struct virtual_container *a;
```

можно действительно понять, что такое «a».

Многие думают, что typedef «способствует читаемости». Это не так. Эту директиву нужно использовать для:

(a) непрозрачных объектов (где typedef активно используется для _сокрытия_ объекта).

Пример: «pte_t» и другие непрозрачные объекты, доступ к которым можно получить с помощью соответствующих функций доступа.

ВНИМАНИЕ! Непрозрачность и функции доступа сами по себе не слишком хороши. Причина, по которой мы используем их для pte_t и т.п., состоит в том, что на самом деле там _нет_ никакой информации для скачивания.

(b) Чисто целочисленные типы, где абстракция _помогает_ избежать путаницы, «int» это или «long».

u8/u16/u32 – вполне нормальные typedef, хотя они больше подходят для категории (d).

ВНИМАНИЕ! Опять же – для этого должна быть _причина_. Если что-то представляет собой «unsigned long», должна быть причина для

```
typedef unsigned long myflags_t;
```

но если есть четкая причина, почему при определенных обстоятельствах может быть «unsigned int», а в других случаях может быть «unsigned long», то на здоровье, используйте typedef.

(c) когда вы используете разрыв, чтобы буквально создать _новый_ тип для проверки типов.

(d) Новые типы, идентичные стандартным типам C99, в определенных исключительных обстоятельствах.

Хотя глазам и мозгу требуется лишь короткое время, чтобы привыкнуть к стандартным типам, например, „uint32_t“, некоторые в любом случае возражают против их использования.

Таким образом, допускаются специфичные для Linux типы „u8/u16/u32/u64“ и их эквиваленты, идентичные стандартным типам, хотя они и не обязательны новом коде.

При редактировании существующего кода, в котором уже используется один или другой набор типов, следует придерживаться выбранного типа.

(e) Типы, которые можно использовать в пользовательском пространстве.

В некоторых структурах, видимых в пользовательском пространстве, мы не можем требовать использования типов C99 и не можем применять форму „u32“ выше. Таким образом, мы используем __u32 и подобные типы во всех структурах, которые используются и в пользовательском пространстве.

Возможно, есть и другие случаи, но основное правило состоит в следующем: НИКОГДА НЕ используйте typedef, если вы не соблюдаете одно из этих правил.

В общем, указатель или структура, содержащие элементы, к которым можно получить прямой доступ, **никогда** не должны быть typedef.

Глава 6: Функции

Функции должны быть короткими и приятными, и выполнять только одно действие. Они должны помещаться на одном или двух экранах текста (размер экрана ISO/ANSI 80x24, как мы все знаем), и выполнять одно действие, но делать это хорошо.

Максимальная длина функции обратно пропорциональна сложности функции и уровню отступов. Итак, если у вас есть концептуально простая функция, которая представляет собой лишь один длинный (но простой) оператор вариант case, где вам нужно делать много мелочей для множества разных случаев, длинная функция – это нормально.

Однако, если у вас есть сложная функция, и вы подозреваете, что не слишком одаренный старшеклассник может даже не понять, о чем эта функция, следует придерживаться ограничений. Используйте вспомогательные функции с описательными именами (можно попросить компилятор встроить их, если считаете, что это критически важно для производительности, и он, вероятно, справится лучше).

Другим критерием функции является количество локальных переменных. Их не должно быть больше 5-10, или вы делаете что-то неправильно. Продумайте функцию заново и разбейте ее на более мелкие части. Человеческий мозг обычно легко отслеживает около 7 разных вещей, а больше – и он уже запутается. Вы знаете, что сейчас вы гений, но, возможно, вам через пару недель захочется понять, что именно вы делали.

В исходных файлах разделяйте функции пустой строкой. Если функция экспортируется, макрос EXPORT* должен следовать сразу за строкой с закрывающей фигурной скобкой. Например:

```
int system_is_up(void)
{
    return system_state == SYSTEM_RUNNING;
}
EXPORT_SYMBOL(system_is_up);
```

В прототипах функций включайте имена параметров с типами данных. Хотя для языка C это и не требуется, но рекомендуется для Linux, потому что это простой способ добавить ценную информацию для читателя.

Глава 7: Централизованный выход из функции

Хотя некоторые объявили аналог оператора goto устаревшим, его часто используют компиляторы в виде инструкции безусловной передачи управления.

Оператор goto пригодится, когда функция производит выход из нескольких мест, и необходимо выполнить какие-то общие действия, такие как очистка.

Обоснование:

- безусловные операторы легче понять и выполнять
- уменьшается глубина вложения
- предотвращаются ошибки по причине отсутствия обновления отдельных точек выхода при внесении изменений
- уменьшает объем работы компилятора для оптимизации избыточного кода ;)

```
int fun(int a)
{
    int result = 0;
    char *buffer = kmalloc(SIZE);

    if (buffer == NULL)
        return -ENOMEM;

    if (condition1) {
        while (loop1) {
            ...
        }
        result = 1;
        goto out;
    }
    ...
out:
    kfree(buffer);
    return result;
}
```

Глава 8: Комментирование

Комментарии полезны, но есть и опасность чрезмерного комментирования. НИКОГДА не пытайтесь объяснить в комментарии, КАК работает ваш код: гораздо лучше написать код так, чтобы принцип _работы_ был очевиден, а объяснять плохо написанный код – это пустая трата времени. Как правило, желательно, чтобы комментарии поясняли, ЧТО делает ваш код, а не КАК. Кроме того, постарайтесь не размещать комментарии внутри тела функции: если функция настолько сложна, что нужно отдельно комментировать ее части, скорее всего, вам надо вернуться к главе 6. Можно давать небольшие комментарии, чтобы отметить или предупредить о чем-то особенно умном (или уродливом), но старайтесь избегать лишнего. Вместо этого поставьте комментарии во главе функции, сообщите людям, что она делает, и, возможно, ПОЧЕМУ она это делает.

Комментируя функции API ядра, используйте формат kernel-doc. Более подробную информацию см. в файлах Documentation/kernel-doc-nano-HOWTO.txt и scripts/kernel-doc.

Стиль Linux для комментариев – стиль C89 `/* ... */` . Не используйте стиль C99 `// ...`.

Для длинных (многострочных) комментариев рекомендуется:

```
/*
 * Рекомендуется использовать этот стиль для многострочных
 * комментариев в исходном коде ядра Linux.
 * Просьба использовать его согласованно.
 *
 * Описание:  Столбец звездочек слева,
 * в начале и в конце почти пустые строки.
 */
```

Также важно комментировать данные, являются ли они базовыми или производными типами. Для этого используйте только одно объявление данных в строке (без запятой для объявления массива данных). Это оставляет вам место для небольшого комментария к каждому пункту с объяснением его использования.

Глава 9: Вы устроили беспорядок

Всё в порядке, мы все так делаем. Наверное, опытный пользователь Unix, который вам помогает, сказал, что «GNU emacs» автоматически форматирует исходный код C, и вы заметили, что да, действительно, но используемые по умолчанию значения оставляют желать лучшего (на самом деле, они хуже, чем случайные – несметное количество обезьян, печатающих в GNU emacs, никогда не создаст хорошую программу).

Итак, вы можете либо избавиться от GNU emacs, либо изменить его для использования более адекватных значений. Чтобы сделать последнее, можно вставить следующее в файл .emacs:

```
(defun c-lineup-arglist-tabs-only (ignored)
"Line up argument lists by tabs, not spaces"
(let* ((anchor (c-langelem-pos c-syntactic-element))
    (column (c-langelem-2nd-pos c-syntactic-element))
    (offset (- (1+ column) anchor))
    (steps (floor offset c-basic-offset)))
    (* (max steps 1)
    c-basic-offset)))

(add-hook 'c-mode-common-hook
        (lambda ()
            ;; Add kernel style
            (c-add-style
            "linux-tabs-only"
            '("linux" (c-offsets-alist
                        (arglist-cont-nonempty
                        c-lineup-gcc-asm-reg
                        c-lineup-arglist-tabs-only))))))

(add-hook 'c-mode-hook
        (lambda ()
            (let ((filename (buffer-file-name)))
            ;; Enable kernel mode for the appropriate files
            (when (and filename
                        (string-match (expand-file-name "~/src/linux-trees")
                                      filename))
                (setq indent-tabs-mode t)
                (c-set-style "linux-tabs-only")))))
```

Это заставит emacs лучше работать со стилем программирования ядра для файлов C в ~/src/linux-trees.

Но даже если вам не удастся заставить emacs форматировать нормально, не все потеряно: используйте «indent».

Опять же, у GNU indent такие же безмозглые настройки, как и у GNU emacs, поэтому надо задать для него несколько параметров командной строки. Тем не менее, это не так уж плохо, потому что даже разработчики GNU indent признают авторитет K&R (люди из GNU не злые, они просто серьезно ошибаются в этом вопросе), поэтому вы просто указываете опции «-kr -i8» (означает «K&R, 8 символов отступа») или используйте «scripts/Lindent», которые делают отступы в новейшем стиле.

В «indent» есть много опций, и особенно когда дело доходит до повторного форматирования комментариев, вы можете захотеть взглянуть на страницу руководства. Но помните: «indent» – это не залог хорошего программирования.

Глава 10: Конфигурационные файлы Kconfig

Для всех конфигурационных файлов Kconfig* в дереве источников отступы несколько отличаются. Строки под определением «config» имеют отступы на позицию табуляции, а текст справки с отступом еще на два пробела. Пример:

```
config AUDIT
    bool "Auditing support"
    depends on NET
    help
    Enable auditing infrastructure that can be used with another
    kernel subsystem, such as SELinux (which requires this for
    logging of avc messages output).  Does not do system-call
    auditing without CONFIG_AUDITSYSCALL.
```

Функции, которые все еще могут считаться нестабильными, должны определяться как зависящие от «EXPERIMENTAL»:

```
config SLUB
    depends on EXPERIMENTAL && !ARCH_USES_SLAB_PAGE_STRUCT
    bool "SLUB (Unqueued Allocator)"
    ...
```

тогда как крайне опасные функции (например, поддержка записи для определенных файловых систем) должны подчеркнуть это в строке приглашения:

```
config ADFS_FS_RW
    bool "ADFS write support (DANGEROUS)"
    depends on ADFS_FS
    ...
```

Полную документацию по файлам конфигурации см. в файле Documentation/kbuild/kconfig-language.txt.

Глава 11: Структуры данных

Для структур данных, которые видимы за пределами однопотоковой среды, в которой они создаются и удаляются, всегда должен выполняться подсчет ссылок. В ядре нет сборки мусора (и за пределами ядра сборка мусора производится медленно и неэффективно), а это означает, что абсолютно _необходимо_ подсчитывать ссылки на каждый случай использования.

Подсчет ссылок означает, что можно избежать блокировки и позволить нескольким пользователям получать доступ к структуре данных одновременно – и не нужно беспокоиться о том, что структура внезапно исчезнет только потому, что они спали или делали что-то еще.

Обратите внимание, что блокировка _не_ является заменой для подсчета ссылок. Блокировка используется для обеспечения целостности структур данных, а подсчет ссылок – это метод управления памятью. Обычно необходимо и то, и другое, и их нельзя путать друг с другом.

Для многих структур данных действительно могут быть два уровня подсчета ссылок, когда есть пользователи разных «классов». Подсчет подкласса подсчитывает количество пользователей подкласса и уменьшает глобальный счетчик только один раз, когда подсчет подкласса равен нулю.

Примеры такого многоуровневого подсчета ссылок можно найти в управлении памятью («struct mm_struct»: mm_users и mm_count) и в коде файловой системы («struct super_block»: s_count и s_active).

Следует помнить, что если другой поток может найти вашу структуру данных, и у вас нет счетчика ссылок, почти наверняка возникнет ошибка.

Глава 12: Макросы, перечисления и уровни регистровых передач (RTL)

Имена макросов, определяющих постоянные и метки в перечислениях, пишутся заглавными буквами.

```
#define CONSTANT 0x12345
```

Рекомендуется использовать перечисления при определении нескольких связанных постоянных.

Ценятся имена макросов, написанные ЗАГЛАВНЫМИ буквами, но похожие на функции макросы можно называть, используя буквы в нижнем регистре.

Как правило, рекомендуется использовать встроенные функции для макросов, похожих на функции.

Макросы с несколькими операторами должны быть заключены в блок do - while:

```
#define macrofun(a, b, c)    \
    do {                     \
        if (a == 5)          \
            do_this(b, c);   \
    } while (0)
```

Во время использования макросов постарайтесь избегать следующего:

1. макросы, которые влияют на поток управления:

   ```
   #define FOO(x)                  \
       do {                        \
           if (blah(x) < 0)        \
               return -EBUGGERED;  \
       } while(0)
   ```

   это _очень_ плохая идея. Он выглядит как вызов функции, но выходит из вызывающей функции; не ломайте внутреннего анализатора у тех, кто прочитает код.

2. макросы, которые зависят от наличия локальной переменной с магическим именем:

   ```
   #define FOO(val) bar(index, val)
   ```

   могут показаться хорошей идеей, но они сбивают с толку, когда читаешь код, и такой код склонен ломаться от, казалось бы, невинных изменений.

3. макросы с аргументами, которые используются как l-значения: FOO(x) = y; это вам аукнется, если кто-то, например. сделает FOO встроенной функцией.

4. потеря приоритета: макросы, определяющие постоянные с использованием выражений, должны заключать выражение в круглые скобки. Остерегайтесь аналогичных проблем с макросами с использованием параметров.

```
#define CONSTANT 0x4000
#define CONSTEXP (CONSTANT | 3)
```

   В руководстве cpp подробно рассматриваются макросы. Руководство по внутреннему устройству gcc также рассматривает уровни регистровых передач (RTL), которые часто используются с языком ассемблера в ядре.

Глава 13: Вывод сообщений ядра

Разработчики ядра любят выглядеть грамотными. Обращайте внимание на орфографию в сообщениях ядра, чтобы произвести хорошее впечатление. Не используйте искаженные слова типа «dont»; вместо этого используйте «do not» или «don't». Пусть сообщения будут краткими, ясными и недвусмысленными.

Сообщения ядра не должны заканчиваться точкой.

Вывод номеров в круглых скобках (%d) не повышает их ценность, и его следует избегать.

В <linux/device.h> есть несколько макросов для диагностики модели драйвера, которые следует использовать, чтобы убедиться, что сообщения соотнесены с правильным устройством и драйвером и помечены правильным уровнем: dev_err(), dev_warn(), dev_info() и так далее. Для сообщений, не связанных с определенным устройством, <linux/kernel.h> определяет pr_debug() и pr_info().

Придумать хорошие сообщения отладки может быть довольно сложно; и как только у вас будут такие, они могут стать огромным подспорьем для удаленного устранения неполадок. Такие сообщения должны быть скомпилированы, когда символ DEBUG не определен (то есть, по умолчанию они не включены). Если вы используете dev_dbg() или pr_debug(), это сработает автоматически. Во многих подсистемах есть опции Kconfig для включения -DDEBUG. В соответствующем соглашении VERBOSE_DEBUG используется для добавления сообщений dev_vdbg() в сообщения, которые уже включены с помощью DEBUG.

Глава 14: Выделение памяти

В ядре поддерживаются следующие распределители памяти широкого применения: kmalloc(), kzalloc(), kcalloc(), and vmalloc(). Для получения дополнительной информации обратитесь к документации по API.

Предпочтительна следующая форма передачи размера структуры:

```
p = kmalloc(sizeof(*p), ...);
```

Другая форма, в которой прописывается название структуры, ухудшает читаемость и дает дополнительные возможности для возникновения ошибок при изменении типа переменной указателя, когда соответствующий sizeof, который передается в распределитель ресурсов, не меняется.

Не нужно отбрасывать возвращаемое значение, представляющее собой указатель на объект, тип которого неизвестен. Язык программирования C обеспечивает преобразование из указателя на объект, тип которого неизвестен, на любой другой тип указателя.

Глава 15: Болезнь встраивания (inline)

Похоже, что распространено ошибочное представление о том, что в gcc есть волшебная опция ускорения, называемая встраиванием «inline». Хотя использование встроенных строк может быть оправдано (например, как средство замены макросов, см. Главу 12), довольно часто это не так. Избыток ключевого слова inline приводит к увеличению ядра, что в свою очередь, замедляет работу системы в целом из-за большего объема отпечатка icache для процессора и просто потому, что для pagecache доступно меньше памяти. Просто подумайте: непопадание в pagecache вызывает поиск по диску, который легко занимает 5 миллисекунд. Есть МНОГО циклов процессора, которые могут пройти в эти 5 миллисекунд.

Общее правило состоит в том, чтобы не вводить встраивание в функции, содержащие больше трех строк кода. Исключением из этого правила являются случаи, когда параметр известен как постоянная времени компиляции, и в результате вы *знаете*, что компилятор сможет оптимизировать большую часть ваших функций во время компиляции. Хороший пример последнего случая – встроенная функция kmalloc().

Часто утверждают, что беспроигрышным вариантом будет встраивание статических функций, используемых только один раз, поскольку нет компромиссов пространства. Хотя это технически правильно, gcc способен автоматически встраивать их, а проблема удаления встроенного, если появляется второй пользователь, перевешивает потенциальную ценность подсказки для gcc делать что-то, что он сделал бы в любом случае.

Глава 16: Возвращаемые значения и имена функций

Функции могут возвращать значения множества различных типов, и одним из наиболее распространенных является значение, которое указывает, была функция выполнена или нет. Такое значение может быть представлено как целое число с кодом ошибки (-Exxx = сбой, 0 = выполнено) или логическое значение выполнения (0 = сбой, ненулевое значение = выполнено).

Смешение этих двух видов дает богатую пищу для появления сложных для обнаружения ошибок. Если бы в языке C были явные различия между целыми числами и логическими значениями, тогда компилятор нашел бы для нас эти ошибки... но это не так. Чтобы предотвратить такие ошибки, всегда следуйте этому соглашению:

```
Если имя функции представляет собой действие или команду,
функция должна возвращать целое число с кодом ошибки.  Если имя функции
является утверждением, функция должна возвращать логическое значение выполнения.
```

Например, «add work» (добавить работу) – это команда, а функция add_work() возвращает 0 в случае выполнения или -EBUSY при сбое. Точно так же «PCI device present» (есть PCI-устройство) представляет собой утверждение, а функция pci_dev_present() возвращает 1, если ей удается найти подходящее устройство, или 0, если это не так.

Все экспортируемые функции (EXPORT) должны подчиняться этому соглашению, то же относится и ко всем доступным функциям. Закрытые (статические) функции не должны подчиняться, но это рекомендуется.

Функции, возвращаемое значение которых является фактическим результатом вычисления, а не указанием того, удалось ли выполнить вычисление, не подпадают под это правило. Обычно они указывают на сбой, возвращая некое недопустимое значение. Типичными примерами будут функции, возвращающие указатели; чтобы сообщить об ошибке, они используют NULL или механизм ERR_PTR.

Глава 17: Не изобретайте макросы снова

В файле заголовка include/linux/kernel.h содержатся несколько макросов, которые следует использовать, а не программировать их самостоятельно. Например, если необходимо рассчитать длину массива, воспользуйтесь макросом

```
#define ARRAY_SIZE(x) (sizeof(x) / sizeof((x)[0]))
```

Аналогичным образом, если необходимо рассчитать размер какого-либо элемента структуры, используйте

```
#define FIELD_SIZEOF(t, f) (sizeof(((t*)0)->f))
```

Есть также макросы min() и max(), которые выполняют строгую проверку типов, если понадобится. Не стесняйтесь ознакомиться с этим файлом заголовка, чтобы узнать, что еще не нужно воспроизводить в своем коде.

Глава 18: Редакторские строки режима (modelines) и прочий хлам

Некоторые редакторы могут интерпретировать встроенную в исходные файлы информацию о конфигурации, указанную специальными маркерами. Например, emacs интерпретирует строки, помеченные следующим образом:

```
-*- mode: c -*-
```

Или так:

```
/*
Local Variables:
compile-command: "gcc -DMAGIC_DEBUG_FLAG foo.c"
End:
*/
```

Vim интерпретирует маркеры, которые выглядят так:

```
/* vim:set sw=8 noet */
```

Не включайте их в исходные файлы. У людей есть свои собственные настройки редакторов, и ваши исходные файлы не должны их переопределять. Это относится к маркерам для отступов и конфигурации режима. У других людей могут быть свои собственные режимы или другие волшебные методы для правильной работы отступов.

Приложение I: Источники

- Керниган Брайан В., Ричи Деннис М. Язык программирования Си. Prentice Hall, Inc., 1988. ISBN 0-13-110362-8 (в мягкой обложке), 0-13-110370-9 (в твердом переплете).
- Керниган Брайан В., Пайк Роб. Практика программирования. Addison-Wesley, Inc., 1999. ISBN 0-201-61586-X.
- Рекомендации GNU в соответствии с K&R и данным текстом – для **cpp**, **gcc**, **gcc internals** и **indent**
- Рабочая группа по международной стандартизации языка программирования C WG14
- Стиль программирования ядра, автор greg@kroah.com, презентация на OLS 2002

8.2.4 Руководство по написанию кода на Python

Введение

Данный документ описывает соглашение о том, как писать код для языка Python, включая стандартную библиотеку, входящую в состав Python. Посмотрите также на сопутствующую PEP (Python enhanced proposal – заявку на улучшение языка Python), описывающую, какого стиля следует придерживаться при написании кода на C в реализации языка Python[1].

Данный документ, а также PEP 257 (Документирование кода) созданы на основе оригинала рекомендаций Гуидо ван Россума с добавлениями от Барри[2].

A Foolish Consistency is the Hobgoblin of Little Minds («Безрассудная согласованность сбивает с толку мелкие умы»)

Одна из ключевых идей Гвидо заключается в том, что код читается намного чаще, чем пишется. И рекомендации по стилю программирования предназначены улучшить читаемость кода и сделать его согласованным во множестве проектов на языке Python. Как написано в PEP 20, «Читаемость имеет значение».

В руководстве речь идет о согласованности. Согласованность с руководством очень важна. Согласованность внутри проекта еще важнее. А согласованность в пределах модуля или функции – самое важное.

Но очень важно понимать, когда можно отойти от рекомендаций, потому что руководство неприменимо. Если вы сомневаетесь, используйте свой опыт. Просто посмотрите на другие примеры и решите, какой выглядит лучше. И не бойтесь спросить!

Правила можно нарушить по одной из этих причин:

1. Если применение правила сделает код менее читаемым даже для того, кто привык читать код, написанный по правилам.
2. Чтобы не отступать по стилю от уже написанного не по правилам кода (возможно, в силу исторических причин) – впрочем, это может быть возможность причесать чужой код (в стиле XP).

Размещение кода

Отступы

Используйте 4 пробела на каждый уровень отступа.

Если вы не хотите наводить путаницу в очень старом коде, можете продолжать использовать отступы в 8 пробелов.

Продолжения строк должны выравнивать переносимые элементы либо вертикально, используя подразумевающееся объединение строк в скобках (круглых, квадратных или фигурных), либо с использованием висячего отступа. При использовании висячего отступа необходимо применять следующие соображения: на первой строке не должно быть аргументов, а остальные строки должны четко восприниматься как продолжение строки.

Правильно:

[1] ван Россум Гвидо. PEP 7, Руководство по программированию на языке C
[2] Руководство Барри по GNU Mailman

```
# выравнивание по открывающему разделителю
foo = long_function_name(var_one, var_two,
                         var_three, var_four)

# больше отступов, чтобы данный сегмент отличался от остальных.
def long_function_name(
        var_one, var_two, var_three,
        var_four):
    print(var_one)
```

Неправильно:

```
# запрещены аргументы на первой строке, если не используется вертикальное выравнивание
foo = long_function_name(var_one, var_two,
    var_three, var_four)

# необходимы дополнительные отступы для четких отличий
def long_function_name(
    var_one, var_two, var_three,
    var_four):
    print(var_one)
```

Возможно:

```
# Нет необходимости в дополнительных отступах.
foo = long_function_name(
  var_one, var_two,
  var_three, var_four)
```

Закрывающие круглые/квадратные/фигурные скобки в многострочных конструкциях могут находиться либо под первым символом последней строки списка (не пробелом), например:

```
my_list = [
    1, 2, 3,
    4, 5, 6,
    ]
result = some_function_that_takes_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f',
    )
```

либо под первым символом строки, с которой начинается многострочная конструкция:

```
my_list = [
    1, 2, 3,
    4, 5, 6,
]
result = some_function_that_takes_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f',
)
```

Табуляция или пробелы?

Никогда не смешивайте символы табуляции и пробелы.

Самый распространенный способ отступов в Python – пробелы. На втором месте – отступы только с использованием табуляции. Код, в котором используются и те, и другие типы отступов, следует исправить так, чтобы отступы в нем были расставлены только с помощью пробелов. При вызове интерпретатора в командной строке с параметром `-t` он выдаст предупреждение в случае использовании смешанного стиля в отступах. Запустив интерпретатор с параметром -tt, вы получите в этих местах ошибки. Рекомендуем использовать эти опции!

В новых проектах для отступов настоятельно рекомендуется использовать только пробелы. Во многих редакторах можно легко это делать.

Максимальная длина строки

Ограничьте максимальную длину строки 79 символами.

Пока еще есть немало устройств, где длина строки ограничена 80 символами; к тому же, ограничив ширину окна 80 символами, мы можем расположить несколько окон рядом друг с другом. Автоматический перенос строк на таких устройствах нарушит форматирование, и код будет труднее понять. Поэтому ограничьте длину строки 79 символами. Для длинных блоков текста (строки документации или комментарии) рекомендуется ограничиваться 72 символами.

Предпочтительный способ переноса длинных строк – использование подразумевающегося продолжения строки между обычными, квадратными и фигурными скобками. Длинные строки можно разбить на несколько строк в скобках. Это лучше, чем использовать обратную косую черту для продолжения строки.

Обратную косую черту можно использовать время от времени. Например, длинный оператор with не может работать с неявными продолжениями, так что обратная косая черта здесь подойдет:

```
with open('/path/to/some/file/you/want/to/read') as file_1, \
        open('/path/to/some/file/being/written', 'w') as file_2:
    file_2.write(file_1.read())
```

Еще один такой случай – операторы `assert`.

Делайте правильные отступы для перенесенной строки. Предпочтительнее вставить перенос строки *после* логического оператора, а не перед ним. Например:

```
class Rectangle(Blob):

    def __init__(self, width, height,
                 color='black', emphasis=None, highlight=0):
        if (width == 0 and height == 0 and
            color == 'red' and emphasis == 'strong' or
            highlight > 100):
            raise ValueError("sorry, you lose")
        if width == 0 and height == 0 and (color == 'red' or
                                           emphasis is None):
            raise ValueError("I don't think so -- values are %s, %s" %
                             (width, height))
        Blob.__init__(self, width, height,
                      color, emphasis, highlight)
```

Пустые строки

Отделяйте функции верхнего уровня и определения классов двумя пустыми строками.

Определения методов в пределах класса отделяйте одной пустой строкой.

Также можно добавлять пустые строки (не слишком часто) для выделения групп связанных функций. Пустые строки не стоит добавлять между несколькими связанными программами в одну строку (например, в формальной реализации).

Не слишком часто можно добавлять пустые строки в коде функций, чтобы отделить друг от друга логические части.

Python расценивает символ control+L (или ^L) как пробел. Многие редакторы обрабатывают его как разрыв страницы, поэтому его можно использовать для выделения логических части в файле на разных страницах. Обратите внимание, что не все редакторы распознают control+L и могут на его месте отображать другой символ.

Кодировка (PEP 263)

В коде ядра Python всегда должна использоваться кодировка ASCII или Latin-1 (также известную как ISO-8859-1). Начиная с версии Python 3.0, предпочтительной является кодировка UTF-8, а не Latin-1 (см. PEP 3120).

Для файлов с ASCII не следует объявлять кодировку. Используйте Latin-1 (или UTF-8), только если необходимо указать в комментарии или строке документации имя автора, содержащее в себе символ из Latin-1. В остальных случаях рекомендуется использовать управляющие символы x, u или U, чтобы вставить в строку символы не из ASCII.

Начиная с версии Python 3.0 и выше, в стандартной библиотеке действует следующая политика (см. PEP 3131): все идентификаторы в стандартной библиотеке Python ДОЛЖНЫ содержать только ASCII-символы и означать английские слова везде, где это возможно (во многих случаях используются сокращения или неанглийские технические термины). Кроме того, строки и комментарии также должны содержать лишь ASCII-символы. Исключения составляют: (a) тестовые сценарии для тестирования функций программы в других кодировках, и (b) имена авторов. Авторы, в именах которых есть буквы не из латинского алфавита, должны транслитерировать свои имена в латиницу.

В проектах с открытым кодом для широкой аудитории также рекомендуется использовать это правило.

Импорт

- Импорт разных модулей должен быть на разных строках, например:

```
Yes: import os
     import sys

No:  import sys, os
```

  В то же время, можно писать вот так:

```
from subprocess import Popen, PIPE
```

- Импорт всегда нужно делать в начале файла сразу после комментариев к модулю и строк документации, перед объявлением глобальных переменных и постоянных.

  Группируйте импорты в следующем порядке:

  1. импорты стандартной библиотеки
  2. импорты сторонних библиотек
  3. импорты модулей текущего проекта

Между группами импортов вставляйте пустую строку.

Указывайте все необходимые спецификации __all__ после импортов.

- Относительные импорты крайне не рекомендуются. Всегда указывайте абсолютный путь к модулю для всех видов импорта. Даже сейчас, когда PEP 328 реализован в версии Python 2.5, явно использовать относительные импорты не рекомендуется. Абсолютные импорты более независимы и, как правило, обладают лучшей читаемостью.
- При импорте класса из модуля с классами, обычно можно писать так:

```
from myclass import MyClass
from foo.bar.yourclass import YourClass
```

Если такое написание вызывает конфликт локальных имен, пишите:

```
import myclass
import foo.bar.yourclass
```

И используйте «myclass.MyClass» и «foo.bar.yourclass.YourClass».

Пробелы в выражениях и операторах

Наболевшие вопросы

Избегайте использования пробелов в следующих ситуациях:

- Перед круглыми, фигурными и квадратными скобками и после них:

```
Yes: spam(ham[1], {eggs: 2})
No:  spam( ham[ 1 ], { eggs: 2 } )
```

- Сразу перед запятой, точкой с запятой, двоеточием:

```
Yes: if x == 4: print x, y; x, y = y, x
No:  if x == 4 : print x , y ; x , y = y , x
```

- Сразу перед открывающей скобкой, после которой начинается список аргументов при вызове функции:

```
Yes: spam(1)
No:  spam (1)
```

- Сразу перед открывающей скобкой, после которой идет индекс или срез:

```
Yes: dict['key'] = list[index]
No:  dict ['key'] = list [index]
```

- Больше одного пробела вокруг оператора присваивания (или другого) для того, чтобы выровнять его с другим оператором:

Правильно:

```
x = 1
y = 2
long_variable = 3
```

Неправильно:

```
x             = 1
y             = 2
long_variable = 3
```

Прочие рекомендации

- Всегда окружайте эти знаки двухместных операций пробелами по одному с каждой стороны: присваивание (=), комбинированное присваивание (+=, -= и т.д.), сравнения (==, <, >, !=, <>, <=, >=, in, not in, is, is not), логические операторы (and, or, not).
- Если используются знаки операций с разными приоритетами, рассмотрите возможность добавить пробелы вокруг операций с самым низким приоритетом. Судите сами, однако, никогда не используйте больше одного пробела, и всегда используйте одинаковое количество пробелов по обе стороны от знака.

  Правильно:

```
i = i + 1
submitted += 1
x = x*2 - 1
hypot2 = x*x + y*y
c = (a+b) * (a-b)
```

  Неправильно:

```
i=i+1
submitted +=1
x = x * 2 - 1
hypot2 = x * x + y * y
c = (a + b) * (a - b)
```

- Не используйте пробелы для отделения знака =, когда он употребляется для обозначения аргумента ключевого слова или значения параметра по умолчанию.

  Правильно:

```
def complex(real, imag=0.0):
    return magic(r=real, i=imag)
```

  Неправильно:

```
def complex(real, imag = 0.0):
    return magic(r = real, i = imag)
```

- Не рекомендуется использовать составные операторы (несколько операторов в одной строке).

  Правильно:

```
if foo == 'blah':
    do_blah_thing()
do_one()
do_two()
do_three()
```

  Скорее неправильно:

```
if foo == 'blah': do_blah_thing()
do_one(); do_two(); do_three()
```

- Иногда можно разместить тело цикла if/for/while в той же строке, но если операторов несколько, никогда так не делайте. И избегайте свертывания таких длинных строк!

  Скорее неправильно:

```
if foo == 'blah': do_blah_thing()
for x in lst: total += x
while t < 10: t = delay()
```

  Точно неправильно:

```
if foo == 'blah': do_blah_thing()
else: do_non_blah_thing()

try: something()
finally: cleanup()

do_one(); do_two(); do_three(long, argument,
                             list, like, this)

if foo == 'blah': one(); two(); three()
```

Комментарии

Комментарии, которые противоречат коду, хуже, чем отсутствие комментариев. Всегда считайте первоочередной задачей исправить комментарии, если меняется код!

Комментарии должны представлять собой законченные предложения. Если комментарием будет фраза или предложение, первое слово должно быть написано с заглавной буквы, если только это не идентификатор, который пишется со строчной буквы (никогда не меняйте регистр идентификаторов!).

Если комментарий короткий, точку в конце предложения можно опустить. Блок комментариев обычно состоит из одного или более абзацев, составленных из полных предложений, поэтому каждое предложение должно заканчиваться точкой.

После точки в конце предложения следует ставить два пробела.

Если вы пишете на английском языке, не забывайте о рекомендациях Странка и Уайта по стилю.

Разработчики на языке Python из неанглоязычных стран, пишите комментарии на английском, если только вы не уверены на 120%, что ваш код никогда не будут читать люди, не знающие вашего родного языка.

Блок комментариев

Блок комментариев обычно сопровождает фрагмент кода (или весь код), который за ним следует, и находится на том же уровне отступов, что и сам код. Каждая строка блока комментариев должна начинаться с символа # и одного пробела после него (если только в самом тексте комментария нет отступов).

Абзацы в пределах блока комментариев отделяются строкой, состоящей из одного символа #.

Комментарии в строке с кодом

Старайтесь реже использовать подобные комментарии.

Встроенный комментарий находится в той же строке, что и оператор. Такие комментарии должны отделяться от оператора хотя бы двумя пробелами. Они должны начинаться с символа # и одного пробела.

Комментарии в строке с кодом не нужны и в действительности отвлекают от чтения, если они объясняют очевидное. Не пишите так:

```
x = x + 1                 # Увеличение x
```

Иногда, впрочем, они полезны:

```
x = x + 1                 # Место для рамки окна
```

Строки документации

Соглашения о написании хорошей документации (docstrings) увековечены в PEP 257.

- Пишите документацию для всех доступных модулей, функций, классов, методов. Строки документации необязательны для внутренних методов, но нужно добавить комментарий о том, что делает метод. Комментарий должен идти после строки `def`.
- PEP 257 объясняет, как правильно и хорошо писать документацию. Следует отметить, что очень важно, чтобы закрывающие """ стояли на отдельной строке, а предпочтительно, чтобы перед ними была и пустая строка, например:

  ```
  """Return a foobang

  Optional plotz says to frobnicate the bizbaz first.

  """
  ```

- Для однострочной документации можно оставить закрывающие """ на той же строке.

Контроль версий

Если вам нужно использовать Subversion, CVS или RCS в ваших исходных кодах, делайте это следующим образом:

```
__version__ = "$Revision$"
# $Source$
```

Эти строки следует указывать после документации модуля перед любым другим кодом, отделяя их пустыми строками сверху и снизу.

Соглашения по именованию

Соглашения по именованию переменных в Python довольно запущены, поэтому полной согласованности невозможно будет добиться. Тем не менее, ниже мы приводим список рекомендованных стандартов именования. Новые модули и пакеты (включая сторонние) должны быть написаны в соответствии с этими стандартами, но если уже существующая библиотека написана в другом стиле, предпочтительно поддерживать согласованность.

Описание: Стили имен

Существует много различных стилей именования. Полезно распознавать, какой стиль именования используется независимо от того, для чего он используется.

Обычно различают следующие стили именования:

- `b` (отдельная строчная буква)
- `B` (отдельная заглавная буква)
- `lowercase` (слово в нижнем регистре)
- `lower_case_with_underscores` (слова из строчных букв с символами подчеркивания)
- `UPPERCASE` (заглавные буквы)
- UPPERCASE_WITH_UNDERSCORES (слова из заглавных букв с символами подчеркивания)
- `CapitalizedWords` (слова с заглавными буквами, или CapWords, или CamelCase – называется так, потому что прописные буквы внутри слова напоминают горбы верблюда[3]). Иногда называется StudlyCaps.

  Примечание: когда вы используете аббревиатуры в стиле CapWords, пишите все буквы аббревиатуры заглавными. HTTPServerError выглядит лучше, чем HttpServerError.
- mixedCase (отличается от CapitalizedWords тем, что первое слово начинается со строчной буквы!)
- Capitalized_Words_With_Underscores (слова с заглавными буквами и символами подчеркивания – уродливо!)

Еще есть стиль, в котором к именам из одной логической группы добавляется короткий уникальный префикс. Этот стиль редко используется в Python, но упомянем его для полноты изложения. Например, функция `os.stat()` возвращает кортеж, имена в котором традиционно выглядят так: `st_mode`, `st_size`, `st_mtime` и так далее. (Так сделано, чтобы подчеркнуть соответствие этих полей структуре системных вызовов POSIX, что помогает знакомым с ней разработчикам).

В библиотеке X11 используется префикс X для всех доступных функций. В Python этот стиль считается лишним, потому что перед полями и именами методов стоит имя объекта, а перед именами функций стоит имя модуля.

Кроме того, используются следующие специальные формы записи имен с добавлением символа подчеркивания в начало или конец имени (их можно использовать с любым типом регистра):

- `_single_leading_underscore`: слабый индикатор «для внутреннего пользования». Например, `from M import *` не будет импортировать объекты, имена которых начинаются с символа подчеркивания.
- `single_trailing_underscore_`: используется по соглашению во избежание конфликтов с ключевыми словами Python, например:

```
Tkinter.Toplevel(master, class_='ClassName')
```

- `__double_leading_underscore`: изменяет имя атрибута класса (в классе FooBar, `__boo` становится `_FooBar__boo`; см. ниже).
- `__double_leading_and_trailing_underscore__`: «волшебные» объекты или атрибуты, которые находятся в live in в пространствах имен, управляемых пользователем. Например, `__init__`, `__import__` или `__file__`. Не придумывайте такие имена, используйте их только так, как написано в документации.

[3] Страница Википедии о CamelCase

Предписания: соглашения по именованию

Имена, которых следует избегать

Никогда не используйте символы „l“ (строчная латинская буква эль), „O“ (заглавная латинская буква о) или „I“ (заглавная латинская буква ай) в качестве однобуквенных имен переменных.

В некоторых шрифтах эти символы неотличимы от цифр один и ноль. Если нельзя обойтись без „l“, пишите вместо нее „L“.

Имена модулей и пакетов

Имена модулей должны быть короткими и состоять из строчных букв. Можно использовать и символы подчеркивания, если это улучшает читаемость. Имена пакетов Python также должны быть короткими и состоять из строчных букв, но здесь символы подчеркивания не приветствуются.

Так как имена модулей отображаются в именах файлов, а некоторые файловые системы являются нечувствительными к регистру символов и обрезают длинные имена, очень важно использовать достаточно короткие имена модулей – это не проблема в Unix, но может стать проблемой при переносе кода в старые версии Windows, Mac или DOS.

Если для модуля расширения, написанного на C или C++, есть сопутствующий Python-модуль, содержащий интерфейс более высокого уровня (например, более объектно-ориентированный), модуль C/C++ начинается с символа подчеркивания (например, `_socket`).

Имена классов

Все имена классов должны соответствовать CapWords почти без исключений. Классы для внутреннего использования могут также начинаться с символа подчеркивания.

Имена исключений

Так как исключения должны быть классами, к исключениям применяются правила именования классов. Однако вы можете добавить суффикс «Error» в конце имени (если исключение действительно является ошибкой).

Имена глобальных переменных

(Будем надеяться, что такие имена используются только в пределах одного модуля.) Применяются те же правила, что и для имен функций.

В модули, которые предназначены для использования с помощью `from M import *`, следует добавить механизм `__all__`, чтобы предотвратить экспорт глобальных переменных, или же использовать старое соглашение, добавляя перед именами таких глобальных переменных один символ подчеркивания (которым можно обозначить глобальные переменные, которые используются только внутри модуля).

Имена функций

Имена функций должны состоять из строчных букв, а слова разделяться символами подчеркивания, чтобы улучшить читаемость.

mixedCase допускается только в тех местах, где уже преобладает такой стиль (например, threading.py), для обратной совместимости.

Аргументы функций и методов

Всегда используйте `self` в качестве первого аргумента метода экземпляра.

Всегда используйте `cls` в качестве первого аргумента метода класса.

Если имя аргумента функции конфликтует с зарезервированным ключевым словом, обычно лучше добавить в конец имени символ подчеркивания, а не сокращать слово или искажать его. Таким образом, `class_` лучше, чем `clss`. (Возможно, будет лучше избегать конфликта имен путем подбора синонима).

Имена методов и переменные экземпляров

Используйте тот же стиль, что и для имен функций: они должны состоять из строчных букв, а слова разделяться символами подчеркивания, чтобы улучшить читаемость.

Используйте только один символ подчеркивания в начале слова для внутренних методов и переменных экземпляров.

Чтобы избежать конфликта имен с подклассами, добавьте два символа подчеркивания в начале слова, чтобы включить механизм изменения имен в Python.

Python изменяет эти имена: если в классе Foo есть атрибут с именем `__a`, к нему нельзя обратиться через `Foo.__a`. (Настойчивый пользователь всё равно может получить доступ через `Foo._Foo__a`.) Вообще, двойное подчеркивание в начале имени должно использоваться только во избежание конфликта имен с атрибутами классов, предназначенных для разделения на подклассы.

Примечание: есть некоторые разногласия по поводу использования имен __names (см. ниже).

Постоянные

Постоянные обычно объявляются на уровне модуля и записываются только заглавными буквами, а слова разделяются символами подчеркивания. Например: MAX_OVERFLOW, TOTAL.

Проектирование наследования

Обязательно решите, каким должен быть метод класса или переменная экземпляра класса (в общем, атрибут) – доступными (public) или внутренними (non-public). Если вы сомневаетесь, делайте их внутренними. Потом будет проще открыть к ним доступ, чем наоборот.

Доступные атрибуты – это такие атрибуты, которые будут использовать потребители ваших классов, и вы должны быть уверены в обратной совместимости. Внутренние атрибуты, в свою очередь, не предназначены для использования третьими лицами, поэтому вы можете не гарантировать, что не измените или не удалите эти атрибуты.

Мы не используем термин «закрытый» (private), потому что на самом деле в Python таких атрибутов не бывает (без ненужных дополнительных усилий).

Другой тип атрибутов классов принадлежит так называемому API подклассов (в других языках они часто называются защищенными – «protected»). Некоторые классы предназначены для наследования другими классами, которые расширяют или изменяют поведение базового класса. Когда вы проектируете такой класс, решите и явным образом укажите, какие атрибуты являются доступными (public), какие относятся к API подклассов (subclass API), а какие используются только базовым классом.

С учетом вышесказанного, сформулируем рекомендации:

- В начале имени доступных атрибутов не должно быть символов подчеркивания.
- Если имя доступного атрибута конфликтует с ключевым словом языка, добавьте в конец имени один символ подчеркивания. Это более предпочтительно, чем сокращать слово или искажать его (однако, у этого правила есть исключение: „cls“ – это предпочтительное написание любой переменной или аргумента, который означает класс, а особенно первого аргумента метода класса).

  Примечание 1: См. рекомендации по именам аргументов выше для методов класса.
- Назовите простые открытые атрибуты понятными именами и не пишите сложные методы доступа и изменения (accessor/mutator). Следует помнить, что в Python очень легко расширить поведение функции, если потребуется. В этом случае используйте свойства (properties), чтобы скрыть функциональную реализацию за синтаксисом доступа к атрибутам.

  Примечание 1: Свойства работают только в классах нового стиля (new-style classes).

  Примечание 2: Постарайтесь избавиться от побочных эффектов, связанных с функциональным поведением, хотя такие вещи, как кэширование, вполне допустимы.

  Примечание 3: Избегайте использовать вычислительно затратные операции, потому что из-за записи с помощью атрибутов создается впечатление, что доступ происходит (относительно) быстро.
- Если ваш класс предназначен для разделения на подклассы, но некоторые атрибуты не должны наследоваться подклассами, подумайте о добавлении в имена двух символов подчеркивания в начале и ни одного в конце. Механизм изменения имен в Python сработает так, что имя класса добавится к имени такого атрибута. Это позволит избежать конфликта имен, если в подклассах случайно появятся атрибуты с такими же именами.

  Примечание 1: Обратите внимание, что только имена простых классов используются в измененном имени, поэтому если в подклассе будет то же имя класса и имя атрибута, то снова возникнет конфликт имен.

  Примечание 2: Механизм изменения имен может затруднить отладку или работу с `__getattr__()`. Тем не менее, алгоритм хорошо документирован и легко реализуется вручную.

  Примечание 3: Не всем нравится механизм изменения имен. Постарайтесь достичь компромисса между необходимостью избежать конфликта имен и возможностью доступа к этим атрибутам.

Использованная литература

Защита авторских прав

Автор:

- Гвидо ван Россум <guido@python.org>
- Барри Ворсо <barry@python.org>

8.2.5 Руководство по написанию кода на Lua

Для вдохновения:

- https://github.com/Olivine-Labs/lua-style-guide
- http://dev.minetest.net/Lua_code_style_guidelines

- http://sputnik.freewisdom.org/en/Coding_Standard

Придерживаться стиля в программировании – это искусство. Даже учитывая некоторую произвольность правил, для них есть надежное обоснование. Полезно не только давать значимые советы по стилю, но также понимать основополагающие причины и человеческий аспект того, почему формируются рекомендации по стилю:

- http://mindprod.com/jgloss/unmain.html
- http://www.oreilly.com/catalog/perlbp/
- http://books.google.com/books?id=QnghAQAAIAAJ

Дзен языка программирования Python подходит и здесь; используйте его с умом:

> Красивое лучше, чем уродливое.
> Явное лучше, чем неявное.
> Простое лучше, чем сложное.
> Сложное лучше, чем запутанное.
> Плоское лучше, чем вложенное.
> Разреженное лучше, чем плотное.
> Читаемость имеет значение.
> Особые случаи не настолько особые, чтобы нарушать правила.
> При этом практичность важнее безупречности.
> Ошибки никогда не должны замалчиваться.
> Если не замалчиваются явно.
> Встретив двусмысленность, отбрось искушение угадать.
> Должен существовать один – и, желательно, только один – очевидный способ сделать это.
> Хотя он поначалу может быть и не очевиден.
> Сейчас лучше, чем никогда.
> Хотя никогда зачастую лучше, чем прямо сейчас.
> Если реализацию сложно объяснить – идея плоха.
> Если реализацию легко объяснить – идея, возможно, хороша.
> Пространства имен – отличная штука! Сделаем побольше!
>
> https://www.python.org/dev/peps/pep-0020/

Отступы и форматирование

- 4 пробела, а не табуляция. Библиотека PIL предлагает использовать два пробела, но разработчик читает код от 4 до 8 часов в день, а различать отступы с 4 пробелами легче. Почему именно пробелы? Соблюдение однородности.

  Можно использовать строки режима (modelines) vim:

  ```
  -- vim:ts=4 ss=4 sw=4 expandtab
  ```

- Файл должен заканчиваться на один символ переноса строки, но не должен заканчиваться на пустой строке (два символа переноса строки).
- Отступы всех do/while/for/if/function должны составлять 4 пробела.
- `or`/`and` в `if` должны быть обрамлены круглыми скобками (). Пример:

```
if (a == true and b == false) or (a == false and b == true) then
    <...>
end -- хорошо

if a == true and b == false or a == false and b == true then
    <...>
end -- плохо

if a ^ b == true then
end -- хорошо, но не явно
```

- Преобразование типов

  Не используйте конкатенацию для конвертации в строку или в число (вместо этого воспользуйтесь `tostring`/`tonumber`):

```
local a = 123
a = a .. ''
-- плохо

local a = 123
a = tostring(a)
-- хорошо

local a = '123'
a = a + 5 -- 128
-- плохо

local a = '123'
a = tonumber(a) + 5 -- 128
-- хорошо
```

- Постарайтесь избегать несколько вложенных `if` с общим телом оператора:

```
if (a == true and b == false) or (a == false and b == true) then
    do_something()
end
-- хорошо

if a == true then
    if b == false then
        do_something()
    end
if b == true then
    if a == false then
        do_something()
    end
end
-- плохо
```

- Избегайте множества конкатенаций в одном операторе, лучше использовать `string.format`:

```
function say_greeting(period, name)
     local a = "good  " .. period .. ", " .. name
end
-- плохо
```

(continues on next page)

(продолжение с предыдущей страницы)

```
function say_greeting(period, name)
    local a = string.format("good %s, %s", period, name)
end
-- хорошо

local say_greeting_fmt = "good %s, %s"
function say_greeting(period, name)
    local a = say_greeting_fmt:format(period, name)
end
-- лучше всего
```

- Используйте `and`/`or` для указания значений переменных, используемых по умолчанию,

```
function(input)
    input = input or 'default_value'
end -- хорошо

function(input)
    if input == nil then
        input = 'default_value'
    end
end -- нормально, но избыточно
```

- операторов `if` и возврата:

```
if a == true then
    return do_something()
end
do_other_thing() -- хорошо

if a == true then
    return do_something()
else
    do_other_thing()
end -- плохо
```

- Использование пробелов:
  - не следует вставлять пробелы между именем функции и открывающей круглой скобкой, но аргумент необходимо разделять одним символом пробела

    ```
    function name (arg1,arg2,...)
    end -- плохо

    function name(arg1, arg2, ...)
    end -- хорошо
    ```

  - добавляйте пробел после маркера комментария

    ```
    while true do -- встроенный комментарий
    -- комментарий
    do_something()
    end
    --[[
      многострочный
      комментарий
    ]] --
    ```

– примыкающие конструкции

```
local thing=1
thing = thing-1
thing = thing*1
thing = 'string'..'s'
-- плохо

local thing = 1
thing = thing - 1
thing = thing * 1
thing = 'string' .. 's'
-- хорошо
```

– добавляйте пробел после запятых в таблицах

```
local thing = {1,2,3}
thing = {1 , 2 , 3}
thing = {1 ,2 ,3}
-- плохо

local thing = {1, 2, 3}
-- хорошо
```

– используйте пробелы в определениях ассоциативного массива по сторонам от знаков равенства и запятых

```
return {1,2,3,4} -- плохо
return {
    key1 = val1,key2=val2
} -- плохо

return {
    1, 2, 3, 4
    key1 = val1, key2 = val2,
    key3 = vallll
} -- хорошо
```

также можно применить выравнивание:

```
return {
    long_key  = 'vaaaaalue',
    key       = 'val',
    something = 'even better'
}
```

– также можно добавлять пустые строки (не слишком часто) для выделения групп связанных функций. Пустые строки не стоит добавлять между несколькими связанными программами в одну строку (например, в формальной реализации)

не слишком часто можно добавлять пустые строки в коде функций, чтобы отделить друг от друга логические части

```
if thing then
    -- ...что-то...
end
function derp()
    -- ...что-то...
```

(continues on next page)

(продолжение с предыдущей страницы)

```
end
local wat = 7
-- плохо

if thing then
    -- ...что-то...
end

function derp()
    -- ...что-то...
end

local wat = 7
-- хорошо
```

– Удаляйте символы пробела в конце файла (они категорически запрещаются). Для их удаления в vim используйте `:s/\s\+$//gc`.

Недопущение глобальных переменных

Следует избегать глобальных переменных. В исключительных случаях используйте переменную `_G` для объявления, добавьте префикс или таблицу вместо префикса:

```
function bad_global_example()
end -- глобальная, очень-очень плохо

function good_local_example()
end
_G.modulename_good_local_example = good_local_example -- локальная, хорошо
_G.modulename = {}
_G.modulename.good_local_example = good_local_example -- локальная, лучше
```

Всегда добавляйте префиксы во избежание конфликта имен

Именование

- имена переменных/»объектов» и «методов»/функций: snake_case
- имена «классов»: CamelCase
- частные переменные/методы (в будущем параметры) объекта начинаются с символа подчеркивания `<object>._<name>`. Избегайте `local function private_methods(self) end`
- логическое именование приветствуется `is_<...>`, `isnt_<...>`, `has_`, `hasnt_`.
- для «самых локальных» переменных: - `t` для таблиц - `i`, `j` для индексации - `n` для подсчета - `k`, `v` для получения из `pairs()` (допускаются, `_` если не используются) - `i`, `v` is what you get out of `ipairs()` (допускаются, `_` если не используются) - `k`/`key` для ключей таблицы - `v`/`val`/`value` для передаваемых значений - `x`/`y`/`z` для общих математических величин - `s`/`str`/`string` для строк - `c` для односимвольных строк - `f`/`func`/`cb` для функций - `status, <rv>..` или `ok, <rv>..` для получения из pcall/xpcall - `buf, sz` – это пара (буфер, размер) - `<name>_p` для указателей - `t0`.. для временных отметок - `err` для ошибок
- допускается использование сокращений, если они недвусмысленны, и если вы документируете их.

- глобальные переменные пишутся ЗАГЛАВНЫМИ_БУКВАМИ. Если это системная переменная, для определения используется символ подчеркивания (`_G`/`_VERSION`/..)
- именование модулей – с помощью snake_case (избегайте подчеркивания и дефисов) - „luasql“, а не „Lua-SQL“
- `*_mt` и `*_methods` определяют метатаблицу и таблицу методов

Идиомы и шаблоны

Всегда пользуйтесь круглыми скобками при вызове функций, за исключением множественных случаев (распространенные идиомы в Lua):

- функции `*.cfg{ }` (`box.cfg`/`memcached.cfg`/..)
- функция `ffi.cdef[[ ]]`

Избегайте конструкций такого типа:

- <func>“<name>“ (особенно избегайте require“..“)
- `function object:method() end` (используйте `functon object.method(self) end`)
- не вставляйте точку с запятой в качестве символа-разделителя в таблице (только запятые)
- точки с запятой в конце строки (только для разделения нескольких операторов в одной строке)
- старайтесь избегать создания ненужных функций (closures/..)

Модули

Не начинайте создание модуля с указания лицензии/авторов/описания, это можно сделать в файлах LICENSE/AUTHORS/README соответственно. Для написания модулей используйте один из двух шаблонов (не используйте `modules()`):

```
local M = {}

function M.foo()
...
end

function M.bar()
...
end

return M
```

или

```
local function foo()
...
end

local function bar()
...
end

return {
foo = foo,
```

(continues on next page)

(продолжение с предыдущей страницы)

```
bar = bar,
}
```

Комментирование

Пишите код так, чтобы его не нужно было описывать, но не забывайте о комментировании. Не следует комментировать Lua-синтаксис (примите, что читатель знаком с языком Lua). Постарайтесь рассказать о функциях, именах переменных и так далее.

Многострочные комментарии: используйте соответствующие скобки (`--[[ ]]--`) вместо простых (`--[[ ]]`).

Комментарии к доступным функциям (??):

```
--- Копирование любой таблицы (поверхностное и глубокое)
-- * deepcopy: копирует все уровни
-- * shallowcopy: копирует только первый уровень
-- Поддержка метаметода __copy для копирования специальных таблиц с метатаблицами
-- @function gsplit
-- @table          inp  оригинальная таблица
-- @shallow[opt]   sep  флаг для поверхностной копии
-- @returns             таблица (копия)
```

Тестирование

Используйте модуль `tap`, чтобы написать эффективные тесты. Пример файла с тестом:

```
#!/usr/bin/env tarantool

local test = require('tap').test('table')
test:plan(31)

do -- check basic table.copy (deepcopy)
    local example_table = {
        {1, 2, 3},
        {"help, I'm very nested", {{{ }}} }
    }

    local copy_table = table.copy(example_table)

    test:is_deeply(
        example_table,
        copy_table,
        "checking, that deepcopy behaves ok"
    )
    test:isnt(
        example_table,
        copy_table,
        "checking, that tables are different"
    )
    test:isnt(
        example_table[1],
        copy_table[1],
        "checking, that tables are different"
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    )
    test:isnt(
        example_table[2],
        copy_table[2],
        "checking, that tables are different"
    )
    test:isnt(
        example_table[2][2],
        copy_table[2][2],
        "checking, that tables are different"
    )
    test:isnt(
        example_table[2][2][1],
        copy_table[2][2][1],
        "checking, that tables are different"
    )
end

<...>

os.exit(test:check() and 0 or 1)
```

После тестирования кода вывод будет примерно таким:

```
TAP version 13
1..31
ok - checking, that deepcopy behaves ok
ok - checking, that tables are different
ok - checking, that tables are different
ok - checking, that tables are different
ok - checking, that tables are different
ok - checking, that tables are different
...
```

Обработка ошибок

Принимайте разнообразные значения и выдавайте строго определенные.

В рамках обработки ошибок это означает, что в случае ошибки вы должны предоставить объект ошибки как второе возвращаемое значение. Объектом ошибки может быть строка, Lua-таблица или cdata, в последнем случае должен быть определен метаметод `__tostring`.

В случае ошибки нулевое значение `nil` должно быть первым возвращаемым значением. В таком случае ошибку трудно игнорировать.

При проверке возвращаемых значений функции проверяйте сначала первый аргумент. Если это `nil`, ищите ошибку во втором аргументе:

```
local data, err = foo()
if not data then
    return nil, err
end
return bar(data)
```

Если производительность вашего кода не имеет первоочередное значение, постарайтесь избегать использования более двух возвращаемых значений.

В редких случаях `nil` можно сделать возвращаемым значением. В таком случае можно сначала проверить ошибку, а потом вернуть значение:

```
local data, err = foo()
if not err then
    return data
end
return nil, err
```

Lua Module Index

b

box.cfg, ??
box.ctl, ??
box.error, ??
box.index, ??
box.info, ??
box.schema, ??
box.session, ??
box.slab, ??
box.space, ??
box.tuple, ??
buffer, 566

c

capi_error, ??
clock, 569
console, 572
crypto.cipher, ??
crypto.digest, ??
crypto.hmac, ??
csv, 577

d

debug, ??
decimal, 580
digest, 581

e

errno, 586

f

fiber, 587
fio, 604

h

http.client, ??

i

iconv, 625

j

json, 627

k

key_def, ??

l

log, 635

m

merger, 637
metrics, ??
metrics.http_middleware, ??
metrics.plugins.graphite, ??
metrics.plugins.json, ??
msgpack, 640
my_box.index, ??
my_fiber, ??

n

net_box, ??

o

os, 657

p

pickle, 660

s

schema, 856
socket, 663
strict, 676
string, 676
swim, 681

t

table, 700
tap, 701
tarantool, 707

u

uri, 715
utf8, 709
uuid, 707

x

xlog, 716

y

yaml, 717