Tarantool

Выпуск 1.10.2

мар. 13, 2019

Оглавление

1 Сервер приложений + СУБД **1**

2 Возможности СУБД **3**

3 Руководство пользователя **5**
- 3.1 Предисловие . . . 5
- 3.2 Руководство для начинающих . . . 6
- 3.3 Функциональность СУБД . . . 13
- 3.4 Сервер приложений . . . 55
- 3.5 Администрирование серверной части . . . 93
- 3.6 Репликация . . . 125
- 3.7 Коннекторы . . . 150
- 3.8 Вопросы и ответы . . . 162

4 Справочники **165**
- 4.1 Справочник по встроенным модулям . . . 165
- 4.2 Справочник по сторонним библиотекам . . . 386
- 4.3 Справочник по настройке . . . 438
- 4.4 Утилита *tarantoolctl* . . . 462
- 4.5 Рекомендации по Lua-синтаксису . . . 464

5 Практикум **467**
- 5.1 Практические задания на Lua . . . 467
- 5.2 Практическое задание на C . . . 479
- 5.3 Практические задания по *libslave* . . . 487

6 Примечания к версиям **491**
- 6.1 Версия 1.10 . . . 491
- 6.2 Версия 1.9 . . . 492
- 6.3 Версия 1.7 . . . 494
- 6.4 Версия 1.6 . . . 505

7 Руководство разработчика **511**
- 7.1 Справочник по C API . . . 511
- 7.2 Внутреннее устройство . . . 539
- 7.3 Содействие в разработке . . . 551
- 7.4 Рекомендации . . . 560

ГЛАВА 1

Сервер приложений + СУБД

Tarantool представляет собой сервер приложений на языке Lua, интегрированный с СУБД. В основе Tarantool'а лежат файберы (fibers), что означает, что несколько Tarantool-приложений могут работать в одном потоке (thread), при этом каждый экземпляр Tarantool-сервера может одновременно запускать несколько потоков для обработки ввода-вывода данных и фоновых задач. Tarantool включает в себя LuaJIT (Just In Time) - Lua-компилятор, Lua-библиотеки для наиболее распространенных приложений, а также сервер базы данных Tarantool'а, который представляет собой широко признанную СУБД NoSQL. Таким образом, Tarantool используется для всех тех целей, которые принесли популярность node.js и Twisted, и более того - поддерживает персистентность данных.

Tarantool – это open-source проект. Исходный код открыт для всех и распространяется бесплатно согласно лицензии BSD license. Поддерживаемые платформы: GNU / Linux, Mac OS и FreeBSD.

Создателем Tarantool'а – а также его основным пользователем – является компания Mail.Ru, крупнейшая Интернет-компания России (30 млн пользователей, 25 млн электронных писем в день, вебсайт в списке top 40 международного Alexa-рейтинга). Tarantool используется для обработки самых «горячих» данных Mail.Ru, таких как данные пользовательских онлайн-сессий, настройки онлайн-приложений, кэширование сервисных данных, алгоритмы распределения данных и шардинга, и т.д. Tarantool также используется во всё большем количестве проектов вне стен Mail.Ru. Это, к примеру, онлайн-игры, цифровой маркетинг, социальные сети. Несмотря на то что Mail.Ru спонсирует разработку Tarantool'а, весь процесс разработки, в т.ч. дальнейшие планы и база обнаруженных ошибок, является полностью открытым. В Tarantool включены патчи от большого числа сторонних разработчиков. Усилиями сообщества разработчиков Tarantool'а были написаны (и далее поддерживаются) библиотеки для подключения модулей на внешних языках программирования. А сообщество Lua-разработчиков предоставило сотни полезных пакетов, большинство из которых можно использовать в качестве расширений для Tarantool'а.

Пользователи Tarantool'а могут создавать, изменять и удалять **Lua-функции** прямо во время исполнения кода. Также они могут указывать **Lua-программы**, которые будут загружаться во время запуска Tarantool'а. Такие программы могут служить триггерами, выполнять фоновые задачи и взаимодействовать с другими узлами по сети. В отличие от многих популярных сред разработки приложений, которые используют «реактивный» принцип, сетевое взаимодействие в Lua устроено последовательно, но очень эффективно, т.к. оно использует среду **кооперативной многозадачности** самого Tarantool'а.

Один из встраиваемых Lua-пакетов – это API для функций СУБД. Таким образом, некоторые разра-

ботчики рассматривают Tarantool как СУБД с популярным языком для написания хранимых процедур, другие рассматривают его как Lua-интерпретатор, а третьи – как вариант замены сразу нескольких компонентов в многозвенных веб-приложениях. Производительность Tarantool'а может достигать сотен тысяч транзакций в секунду на ноутбуке, и ее можно наращивать «вверх» или «вширь» за счет новых серверных ферм.

ГЛАВА 2

Возможности СУБД

Компонент «box» – серверная часть с функциями СУБД – это важная часть Tarantool'а, хотя он может работать и без данного компонента.

API для функций СУБД позволяет хранить Lua-объекты, управлять коллекциями объектов, создавать и удалять вторичные ключи, делать атомарные изменения, конфигурировать и мониторить репликацию, производить контролируемое переключение при отказе (failover), а также исполнять код на Lua, который вызывается событиями в базе. А для прозрачного доступа к удаленным (remote) экземплярам баз данных разработан API для вызова удаленных процедур.

В архитектуре серверной части СУБД Tarantool'а реализована концепция «движков» базы данных (storage engines), где в разных ситуациях используются разные наборы алгоритмов и структуры данных. В Tarantool'е есть два встроенных движка: in-memory движок, который держит все данные и индексы в оперативной памяти, и двухуровневый движок для B-деревьев, который обрабатывает данные размером в 10-1000 раз больше того, что может поместиться в оперативной памяти. Все движки в Tarantool'е поддерживают транзакции и репликацию, поскольку они используют единый механизм **упреждающей записи** (WAL = write ahead log). Это механизм обеспечивает согласованность и сохранность данных при сбоях. Таким образом, изменения не считаются завершенными, пока не проходит запись в лог WAL. Подсистема записи в журнал также поддерживает групповые коммиты.

In-memory движок базы данных Tarantool'а (memtx) хранит все данные в оперативной памяти, поэтому у него низкое значение задержки чтения. Кроме того, когда пользователи запрашивают снимки данных (snapshots), этот движок создает персистентные копии данных в энергонезависимой памяти, например на диске. Если экземпляр сервера прекращает работать и данные в оперативной памяти теряются, то при следующем запуске сервер загрузит в память самый свежий снимок и воспроизведет все транзакции из журнала. Таким образом, данные не теряются.

В штатных ситуациях **in-memory движок работает без блокировок**. Вместо многопоточных примитивов, которые предлагает операционная система (таких как mutex'ы), Tarantool использует кооперативную многозадачность для работы с тысячами соединений одновременно. В Tarantool'е есть фиксированное количество независимых потоков управления (thread), и у них нет общего состояния. Для обмена данными между потоками используются очереди сообщений с малой перегрузкой. Хотя такой подход накладывает ограничение на количество процессорных ядер, которые может использовать экземпляр, в то же время он позволяет избежать борьбы за шину памяти, а также дает запас масштабируемости по скорости доступа к памяти и производительности сети. В результате даже при

большой нагрузке экземпляр Tarantool'а в среднем использует процессор менее чем на 10%. Кроме того, Tarantool поддерживает поиск как по первичным, так и по **внешним ключам в индексах**.

Дисковый движок базы данных Tarantool'а совмещает в себе подходы, заимствованные из современных файловых систем, журнально-структурированных деревьев со слиянием (log-structured merge trees) и классических B-деревьев. Все данные разбиты на **диапазоны**. Каждый диапазон представлен файлом на диске. Размер диапазона можно изменять, обычно он равен 64МБ. Каждый диапазон – это набор страниц, которые служат разным целям. После полного слияния диапазона ключи на его страницах не пересекаются. Если диапазоны ключей недавно сильно изменялись, можно провести частичное слияние диапазона. В этом случае на некоторых страницах появились новые ключи и значения. Дисковый движок обновляет данные по принципу дописывания в конец: новые данные никогда не затирают старые. Дисковый движок базы данных называется *vinyl*.

Tarantool поддерживает работу с **составными ключами в индексах**. Возможные типы ключей: HASH, TREE, BITSET и RTREE.

Tarantool также поддерживает **асинхронную репликацию** – как локальную, так и на удаленных серверах. При этом репликацию можно настроить по принципу **мастер-мастер**, когда несколько узлов могут не только обрабатывать входящую нагрузку, но и получать данные от других узлов.

ГЛАВА 3

Руководство пользователя

3.1 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.

3.1.1 Как пользоваться документацией

Для начала можно установить и запустить Tarantool, используя *Docker-контейнер*, *бинарный пакет* 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.

3.1.2 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).

3.1.3 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.

3.2 Руководство для начинающих

В этой главе объясняется, как установить и запустить Tarantool, а также как создать простую базу данных.

Эта глава состоит из следующих разделов:

3.2.1 Использование Docker-образа

Для практики и тестирования мы рекомендуем использовать официальные образы Tarantool'а для Docker. Официальный образ содержит определенную версию Tarantool'а (1.6, 1.10 или 2.0) 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.

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:1
```

Эта команда запускает новый контейнер с именем „mytarantool“. Docker запускает его из официального образа „tarantool/tarantool:1“ с предустановленным Tarantool'ом 1.9 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.

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.

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 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}
---
- [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 выполните команду:

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

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.

3.2.2 Использование бинарного пакета

Для промышленной разработки мы рекомендуем использовать официальные бинарные пакеты. Можно выбрать одну из двух версий Tarantool'а: 1.10 (стабильная) или 2.0 (альфа). Автоматическая система сборки создает, тестирует и публикует пакеты после каждого коммита в соответствующую ветку (`1.10` или `2.0`) репозитория Tarantool'а на GitHub.

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

Запуск экземпляра Tarantool'а

Для запуска экземпляра Tarantool'а выполните эту команду:

```
$ # если вы скачали бинарный пакет с помощью apt-get или yum, введите:
$ /usr/bin/tarantool
$ # если вы скачали бинарный пакет в формате TAR
$ # и разархивировали его в директорию ~/tarantool, введите:
$ ~/tarantool/bin/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.

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:

   (a) 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)
```

   (b) 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}
---
- [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')
```

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`

3.3 Функциональность СУБД

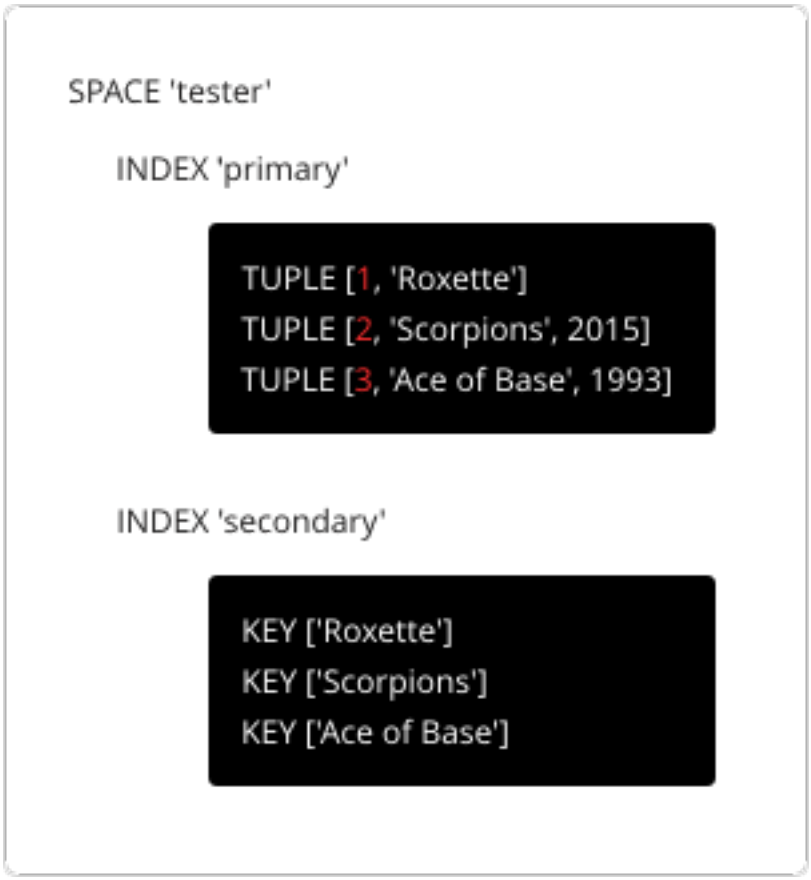
В данной главе мы рассмотрим основные понятия при работе с Tarantool'ом в качестве системы управления базой данных.

Эта глава состоит из следующих разделов:

3.3.1 Модель данных

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

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



Спейс

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.

Кортеж

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.

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]`.

Индекс

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* ниже).

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 = {1, '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 = {2, '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*.

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
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')`.

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’
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	**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) Примечание: в сочетании различных типов порядок будет следующим: логические значения, затем числовые, затем строковые.	TREE или HASH	true -1 1,234 ‘’ ‘ру’

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’а.

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 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 все уровни веса, от L1 до Ln (одинаково),
- сортировка `unicode_ci` принимает во внимание только вес L1, 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 = {{1,'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 = {{1,'str', collation='unicode_ci'}}})
...
tarantool> box.space.S.index.I:select()
---
- - ['ЕЛЕ']
  - ['елейный']
  - ['ёлка']
  - ['еловый']
  - ['елозить']
```

```
  - ['Ёлочка']
  - ['ёлочный']
  - ['ЕЛЬ']
...
tarantool> box.space.S.index.I:select{'ЁлКа'}
---
- - ['ёлка']
...
```

Фактически хорошая сортировка включает в себя гораздо больше, чем простые примеры эквивалентности заглавных букв и строчных в алфавитах. Учитываются также знаки ударения, системы письменности без алфавита и специальные правила, которые применяются в отношении сочетания символов.

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 операция 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}, {{'=', 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:

```
:samp:`box.space.{имя-спейса}:create_index('{имя-индекса}')`
```

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.

Возможны следующие варианты SELECT:

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

   box.space.*space-name*:select(value, {iterator = 'GT'})

   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.

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

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

3. Поиск может производиться как по всему ключу, так и по его частям.

```
-- Suppose an index has two parts
tarantool> box.space.space-name.index.index-name.parts
---
- - type: unsigned
    fieldno: 1
  - type: string
    fieldno: 2
...
-- Suppose the space has three tuples
box.space.space-name:select()
---
- - [1, 'A']
  - [1, 'B']
  - [2, '']
...
```

4. Поиск может производиться по всем полям с использованием таблицы значений:

   box.space.*space-name*:select({1, 'A'})

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

   box.space.*space-name*:select(1)

   Во втором случае Tarantool вернет два кортежа: `{1, 'A'}` и `{1, 'B'}`.

   При необходимости можно задать даже нулевые поля, в результате чего Tarantool вернет все три кортежа (обратите внимание, что поиск по компонентам ключа доступен только для TREE-индексов).

Примеры

- Пример работы с BITSET-индексом:

```
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.

- Пример работы с RTREE-индексом:

```
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.

См. также информацию о других операциях с итераторами с индексом, таких как *alter()* и *drop()* во вложенном модуле *box.index*.

Факторы сложности

Что касается вложенных модулей *box.space* и *box.index*, есть 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.

3.3.2 Контроль транзакций

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
```

Этот запрос будет обработан тремя **потоками** операционной системы:

1. Если мы передадим запрос на удаленный клиент, **сетевой поток** на стороне сервера получит запрос, разберет выражение и преобразует его в выполняемое сообщение сервера, которое уже проверено. Такое сообщение экземпляр сервера может понимать без повторного разбора.

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.

   Поток обработки транзакций экземпляра использует индекс на поле первичного ключа field[1], чтобы найти нужный кортеж. Он проверяет, что данный кортеж можно обновить (мы хотим лишь изменить значение не индексированного поля на более короткое, и вряд ли что-то пойдет не так).

3. Поток обработки транзакций отправляет сообщение в *поток упреждающей записи в журнал (WAL)* для коммита транзакции. По завершении поток WAL отправляет ответ с результатом COMMIT (коммит) или ROLLBACK (откат) на клиент.

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.

Эта модель исключает необходимость любых программных блокировок – кооперативная многозадачность обеспечивает отсутствие многопоточности вокруг ресурса, состояния гонки и проблем с согласованностью данных.

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*.

Чтобы осуществить изоляцию, Tarantool использует простой планировщик с оптимистичным управлением: транзакция подтверждена первой – выигрывает. Если параллельная активная транзакция читает значение, измененное подтвержденной транзакцией, она прерывается.

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.

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.

Операции по изменению базы данных обычно не передают управление, но это зависит от движка:

- В memtx'е чтение и запись не требуют ввода-вывода и не передают управление.
- В vinyl'е не все данные находятся в оперативной памяти, и запрос SELECT часто подразумевает дисковый ввод-вывод и, следовательно, передачу управления, пока запись ожидает освобождения памяти, что также вызывает передачу управления.

В режиме «автокоммита» все операции по изменению данных сопровождаются автоматическим коммитом, который передает управление. Также передает управление явный коммит составной транзакции *box.commit()*.

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

Пример №1

- *Движок = memtx* В `select() insert()` управление передается один раз в конце вставки, что вызвано неявным коммитом; `select()` ничего не записывает в WAL-файл, поэтому не передает управление.
- *Движок = vinyl* В `select() insert()` управление передается от одного до трех раз, поскольку `select()` может передавать управление, если данные не находятся в кэше, `insert()` может передавать управление в ожидании свободной памяти, а при коммите управление передается неявно.
- 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

Предположим, что в спейсе 'tester' существуют кортежи, третье поле которых представляет собой положительную сумму в долларах. Начнем транзакцию, снимем сумму из кортежа №1, внесем ее в кортеж №2 и завершим транзакцию, подтверждая изменения.

```
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.

Если задача интерактивная – отправка запроса на сервер и получение ответа – то она включает в себя сетевой ввод-вывод, поэтому наблюдается неявная передача управления, даже если отправляемый на сервер запрос не представляет собой запрос с неявной передачей управления. Таким образом, последовательность:

```
select
           select
           select
```

приводит к блокировке (в memtx'e), если находится внутри функции или Lua-программы, которая выполняется на экземпляре сервера. Однако она вызывает передачу управления (и в memtx'e, и в vinyl'e), если выполняется как серия передач от клиента, включая клиентов, работающих по telnet, по одному из коннекторов или *модулей MySQL и PostgreSQL* или в интерактивном режиме при *использовании Tarantool'а как клиента*.

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

3.3.3 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.password()* для изменения пароля пользователя и *box.schema.user.passwd()* для получения хеш-пароля.

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" (удаление), например, разрешить выполнение *box.sequence.x:drop* (сейчас можно настроить такие права, но они не действуют)
- „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.

Чтобы **удалить** объект, пользователь должен быть создателем объекта или „admin". Как владелец всей базы данных, „admin" может удалить любой объект, в том числе других пользователей.

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` – это любой тип объекта: „space" (спейс), „index" (индекс), „sequence" (последовательность), „function" (функция), имя роли или „universe";

- `object-name` – это то, на что распространяются права (не указывается, если `object-type` = „universe“);
- `options` – это список параметров, приведенный в скобках, например, `{if_not_exists=true|false}` (как правило, не указывается, поскольку допускаются значения по умолчанию).

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

В данных примерах создатель объекта выдает пользователю „U“ минимально необходимые права на определенные действия.

```
-- Чтобы 'U' мог создавать спейсы:
   box.schema.user.grant('U','create','universe')
   box.schema.user.grant('U','write', 'space', '_schema')
   box.schema.user.grant('U','write', 'space', '_space')
 -- Чтобы 'U' мог создавать индексы (подразумевая, что 'U' создал спейс)
   box.schema.user.grant('U','read', 'space', '_space')
   box.schema.user.grant('U','read,write', 'space', '_index')
 -- Чтобы 'U' мог создавать индексы в спейсы T (подразумевая, что 'U' не создал спейс T)
   box.schema.user.grant('U','create','space','T')
   box.schema.user.grant('U','read', 'space', '_space')
   box.schema.user.grant('U','write', 'space', '_index')
 -- Чтобы 'U' мог изменять индексы в спейсе T (подразумевая, что 'U' не создал индекс)
   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')
 -- Чтобы 'U' мог создавать пользователей или роли:
   box.schema.user.grant('U','create','universe')
   box.schema.user.grant('U','read,write', 'space', '_user')
   box.schema.user.grant('U','write','space', '_priv')
 -- Чтобы 'U' мог создавать последовательности:
   box.schema.user.grant('U','create','universe')
   box.schema.user.grant('U','read,write','space','_sequence')
 -- Чтобы 'U' мог создавать функции:
   box.schema.user.grant('U','create','universe')
   box.schema.user.grant('U','read,write','space','_func')
 -- Чтобы 'U' мог выдавать права на созданные им объекты:
   box.schema.user.grant('U','read','space','_user')
 -- Чтобы 'U' мог производить выборку или получать данные из спейса под названием 'T'
   box.schema.user.grant('U','read','space','T')
 -- Чтобы 'U' мог производить обновление, вставку, удаление или очистку спейса под названием 'T'
   box.schema.user.grant('U','write','space','T')
 -- Чтобы 'U' мог выполнять функцию под названием 'F'
   box.schema.user.grant('U','execute','function','F')
 -- Чтобы 'U' мог использовать функцию "S:next()" для последовательности под названием S
   box.schema.user.grant('U','read,write','sequence','S')
 -- Чтобы 'U' мог использовать функцию "S:set()" или "S:reset()" для последовательности под␣
↪названием S
   box.schema.user.grant('U','write','sequence','S')
```

Пример создания пользователей и объектов и последующей выдачи прав

Здесь создадим Lua-функциб, 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()}
    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*.

3.3.4 Триггеры

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

В Tarantool'е есть четыре типа триггеров:

- *триггеры для обработки соединений*, которые выполняются, когда начинается или заканчивается сессия,
- *триггеры для обработки аутентификации*, которые выполняются при аутентификации, и
- *триггеры для обработки замены*, которые предназначены для событий в базе данных.
- *триггеры для обработки транзакций*, которые выполняются во время коммита или отката.

У всех триггеров есть следующие особенности:

- Триггеры связывают функцию с событием. Запрос «определить триггер» подразумевает передачу функции с триггером в одну из функций обработки событий «on_event()»:
  - *box.session.on_connect()*,
  - *box.session.on_auth()*,
  - *box.session.on_disconnect()*, or
  - *space_object:on_replace()* (после замены), *space_object:before_replace()* (перед заменой), *box.on_commit()* (при коммите) и *box.on_rollback()* (при откате).
- 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 приводят к высокой затрате ресурсов. Если триггер не определен, то затрата ресурсов минимальна: только разыменование указателя и проверка. Если триггер определен, то затрата ресурсов аналогична вызову функции.
- Для одного события можно определить несколько триггеров. В таком случае триггеры выполняются в обратном порядке относительно того, как их определили.
- 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.

Чтобы получить список триггеров, можно использовать следующее:

- on_connect() – без аргументов – чтобы вернуть таблицу со всеми триггерными функциями для обработки соединений;
- on_auth(), чтобы вернуть все триггерные функции для обработки аутентификации;
- on_disconnect(), чтобы вернуть все триггерные функции для обработки отключений;
- on_replace(), чтобы вернуть все триггерные функции для обработки замены, сделанные для on_replace().
- before_replace(), чтобы вернуть все триггерные функции для обработки замены, сделанные для before_replace().

Пример

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

```
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

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)
```

3.3.5 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, 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`).

3.3.6 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 примерное максимальное 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

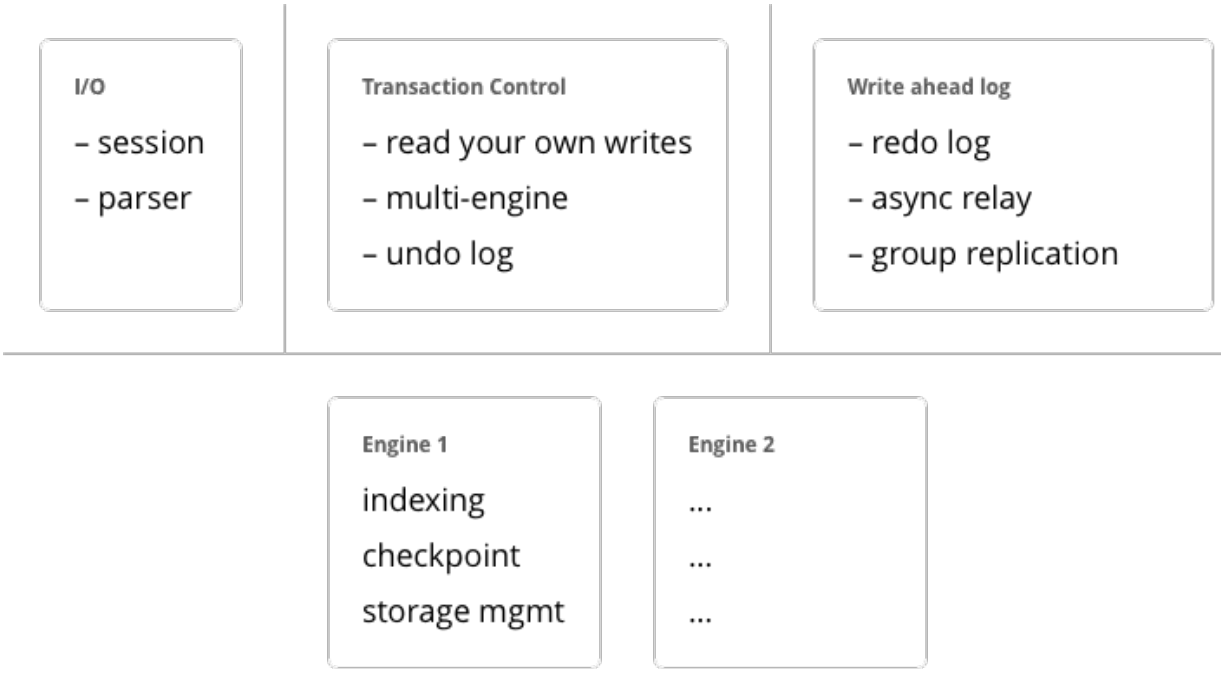
Tarantool – это транзакционная, персистентная СУБД, которая хранит 100% данных в оперативной памяти. Основными преимущества хранения данных оперативной памяти являются скорость и простота использования: нет необходимости в оптимизации, однако производительность остается стабильно высокой.

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



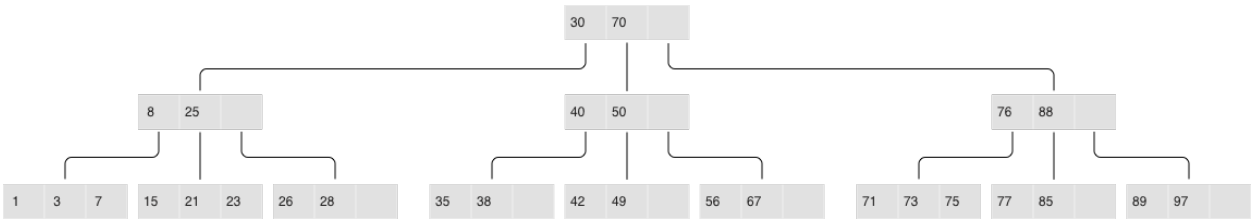
Если изначально проектировать движок с учетом кооперативной многозадачности, можно не только существенно ускорить работу, но и реализовать приемы оптимизации, слишком сложные для многопоточных движков. В общем, использование стороннего решения не привело бы к лучшему результату.

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

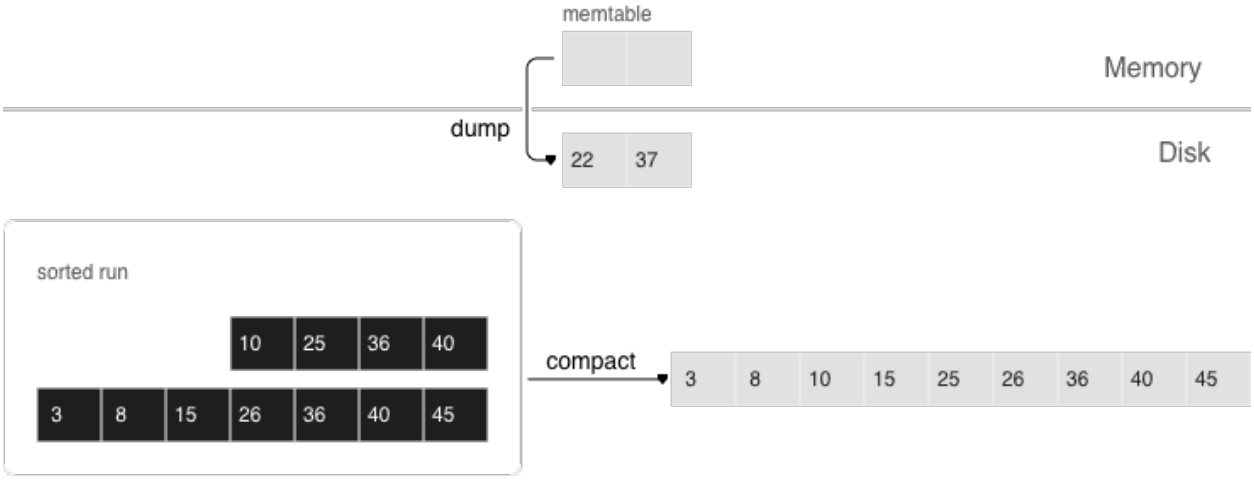
Наполнение LSM-дерева

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

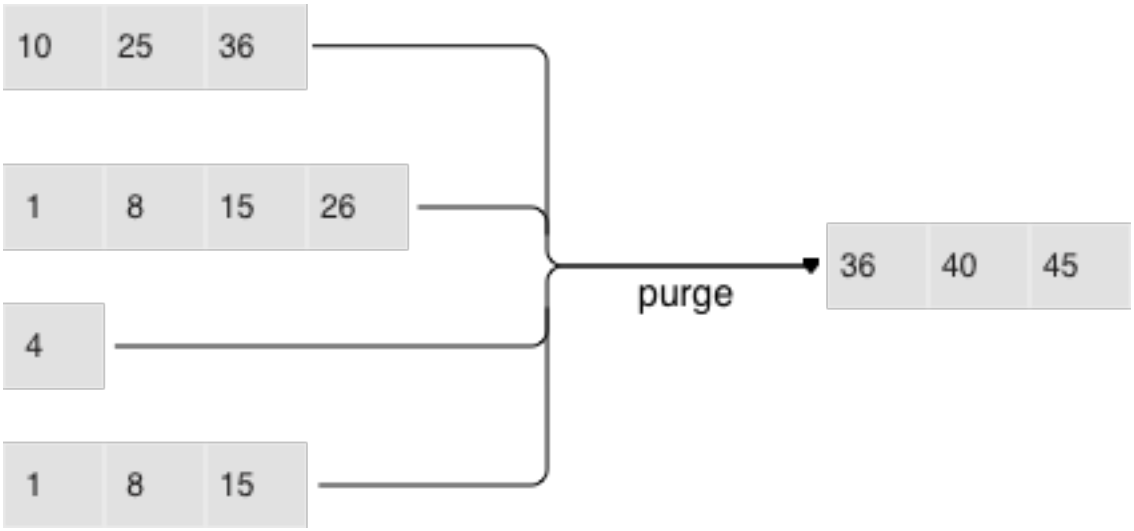
начала. При этом проблема сохранности данных, расположенных в энергозависимой памяти, выносится за рамки алгоритма хранения: ее можно решить разными способами, например, журналированием изменений.

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.

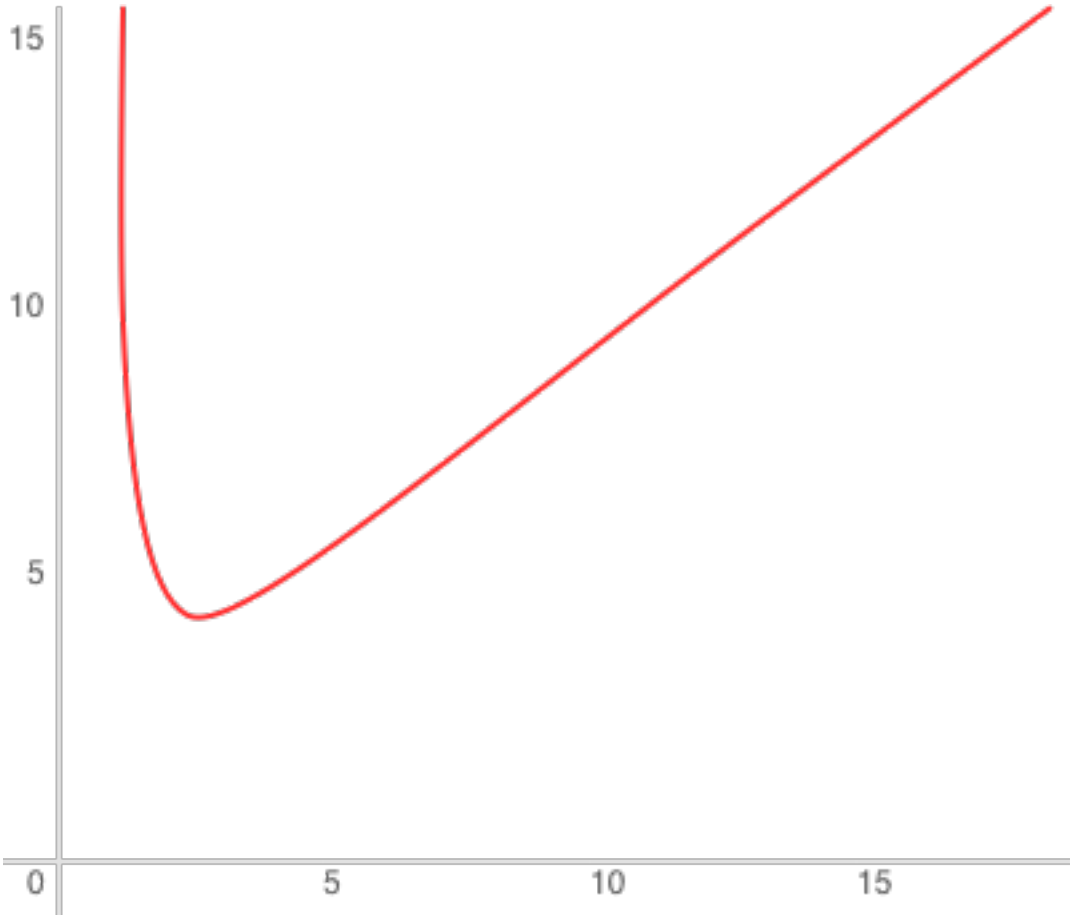
Предположим, что размер L0 составляет 100 МБ, а соотношение размеров файлов на каждом уровне (параметр `vinyl_run_size_ratio`) равно 5, и на каждом уровне может быть не более 2 файлов (пара-

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

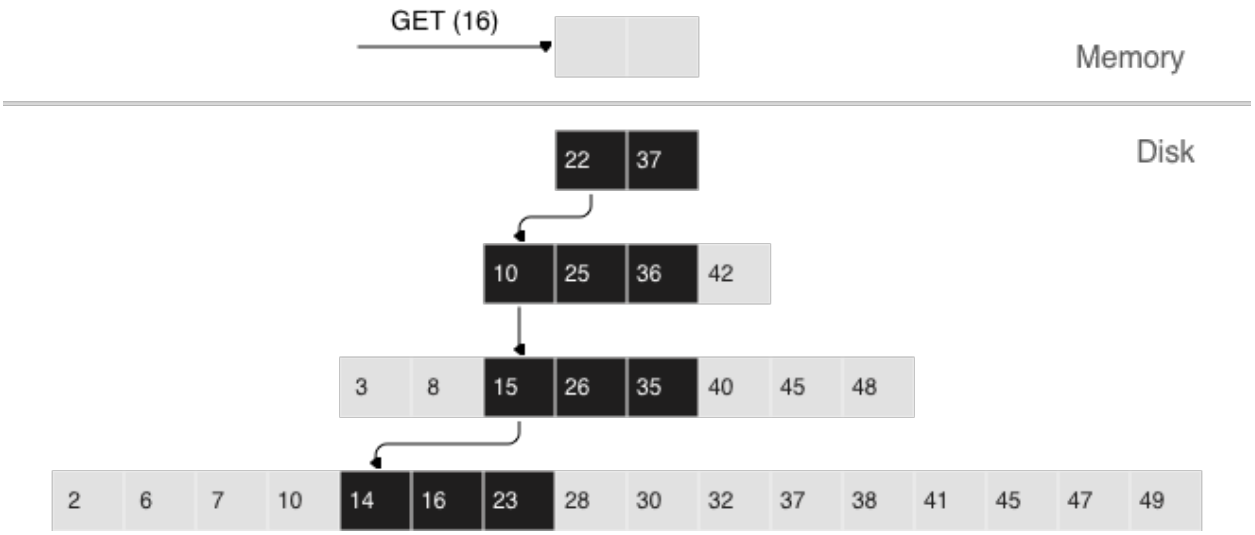
Если число файлов для поиска нужно уменьшить, то соотношение размеров файлов на разных уровнях можно увеличить, и, как следствие, уменьшается число уровней. Если, напротив, необходимо снизить затраты ресурсов, вызванные слиянием, то можно уменьшить соотношение размеров уровней: пирамида будет более высокой, 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}(\frac {N} {L0}) × x` или `x × \frac {ln (\frac {N} {C0})} {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 `\frac {N} {C0}` = 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_level_size_ratio` и `run_count_per_level`, получим коэффициент «паразитной» записи равным примерно 13, коэффициент «паразитной» записи может доходить до 150. Разберемся, почему это происходит.

Поиск

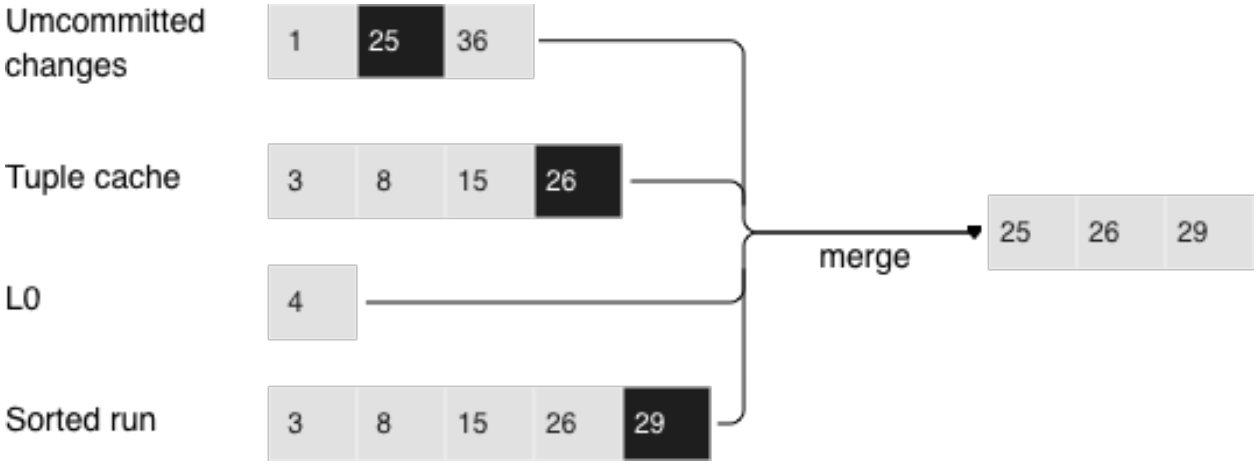
При поиске в LSM-дереве нам необходимо найти не сам элемент, а последнюю операцию с ним. Если это операция удаления, искомый элемент отсутствует в дереве. Если это операция вставки, то искомому элементу соответствует самое верхнее значение в LSM-пирамиде, и поиск можно остановить при первом совпадении 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.

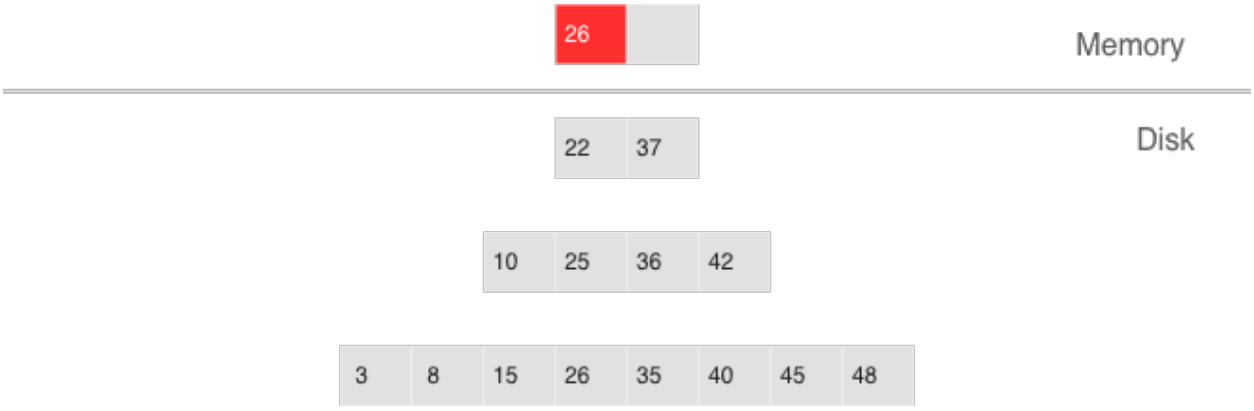
Удаление

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?

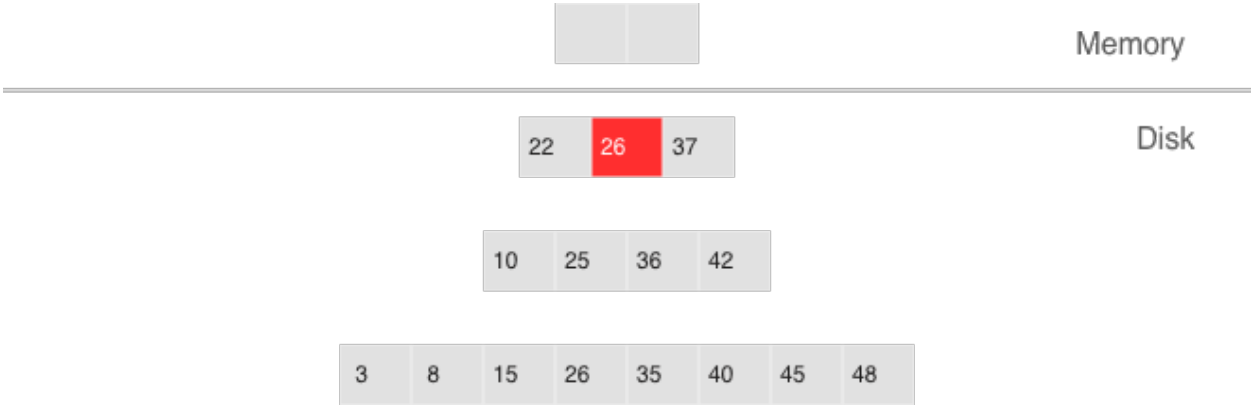
Роль операций удаления при поиске – сообщать об отсутствии искомого значения, а при слиянии – очищать дерево от «мусорных» записей с более старыми LSN.

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.

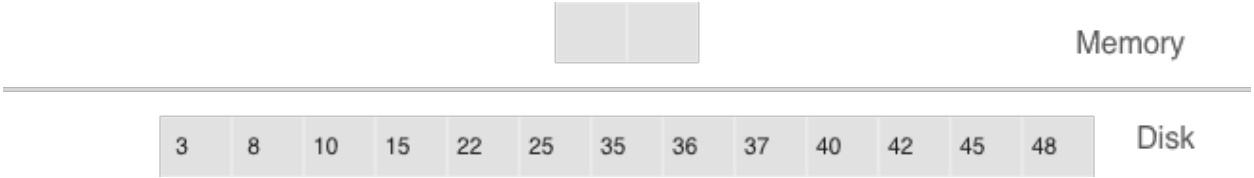
- Нельзя производить удаление из файлов, которые обновляются только путем присоединения новых записей
- Вместо этого на уровень L0 вносятся маркеры удаленных записей (tombstones)



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?

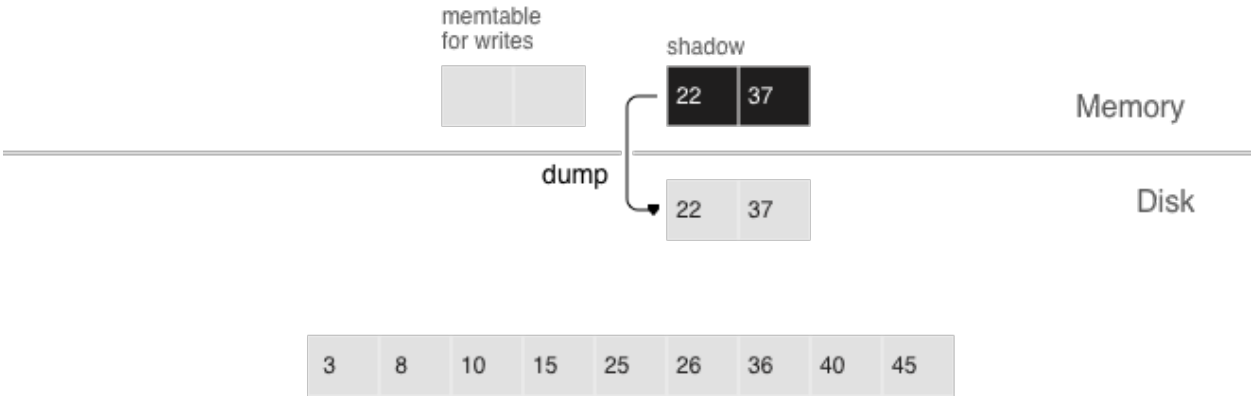
Освобождение L0 подразумевает две долгих операции: запись на диск и освобождение памяти. Чтобы избежать простоя во время записи L0 на диск, Tarantool использует упреждающую запись. Допустим, размер L0 составляет 256 МБ. Скорость записи на диск составляет 10 МБ/с. Тогда для записи L0 на диск понадобится 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.

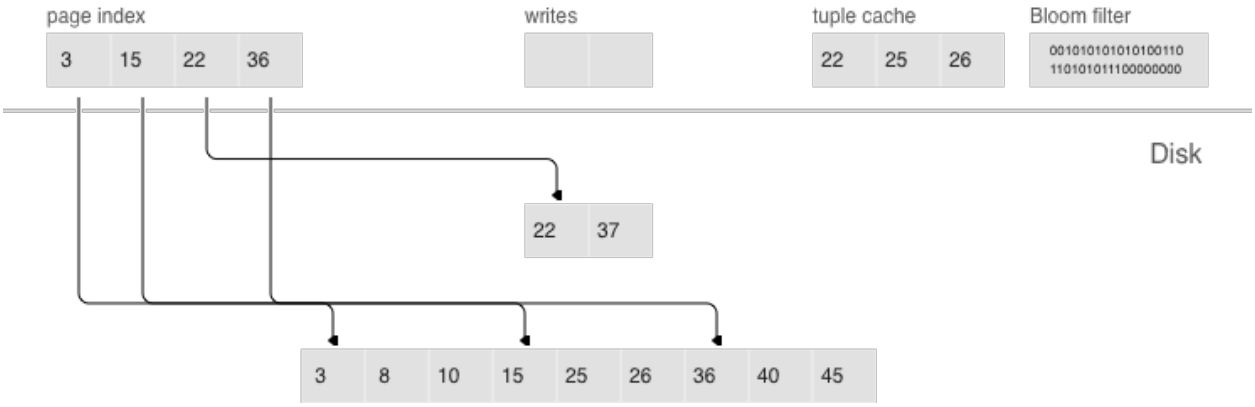


- упреждающий дамп
- загрузка

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 `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.

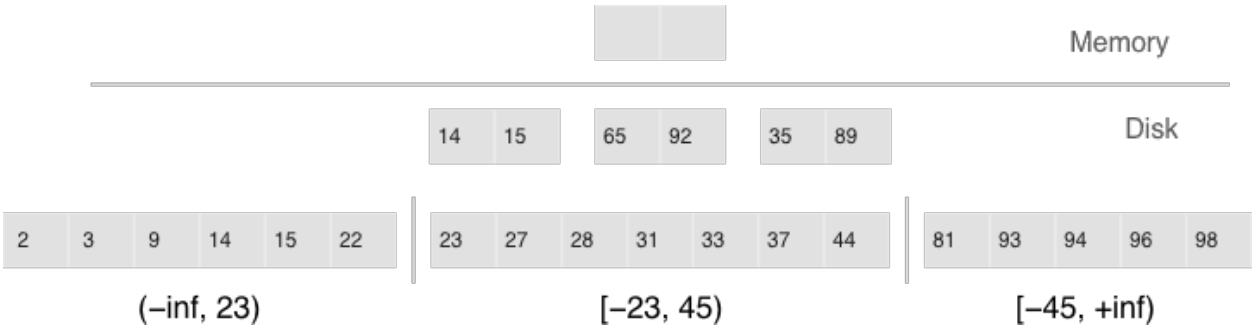
В vinyl'е устройство одного LSM-дерева – это лишь фрагмент мозаики. Vinyl создает и обслуживает несколько LSM-деревьев даже для одной таблицы (так называемого спейса) – по одному дереву на каждый индекс. Но даже один единственный индекс может состоять из десятков LSM-деревьев. Попробуем разобраться, 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 «factorizes» this problem by creating multiple LSM trees for each index. The approximate size of each subtree may be controlled by the *vinyl_range_size* configuration parameter. We call such subtrees «ranges».



Факторизация больших LSM-деревьев с помощью диапазонов

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

Initially, when the index has few elements, it consists of a single range. As more elements are added, its total size may exceed *the maximum range size*. In that case a special operation called «split» divides the tree into two equal parts. The tree is split at the middle element in the range of keys stored in the tree. For example, if the tree initially stores the full range of -inf. . . +inf, then after splitting it at the middle key X, we get two subtrees: one that stores the range of -inf. . . X, and the other storing the range of X. . . +inf. With this approach, we always know which subtree to use for writes and which one for reads. If the tree

contained deletions and each of the neighboring ranges grew smaller as a result, the opposite operation called «coalesce» combines two neighboring trees into one.

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 разрешение *.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.

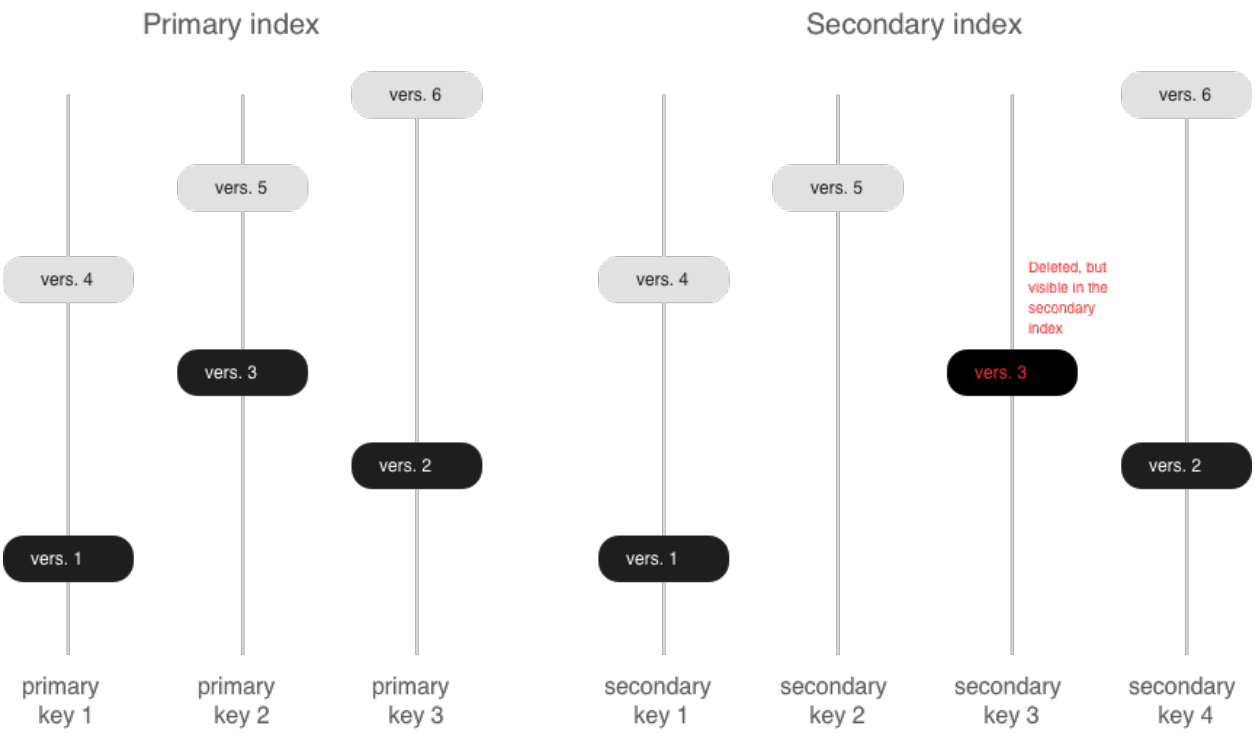
Тем не менее, после пары дней работы процесс Tarantool'а начал потреблять 100 % CPU, а производительность системы упала практически до нуля.

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).

3.4 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:

3.4.1 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 1.9 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 *бинарного пакета* или *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!
 2016-12-19 16:07:14.250 [41436] main/101/myapp.lua C> version 1.7.2-146-g021d36b
 2016-12-19 16:07:14.250 [41436] main/101/myapp.lua C> log level 5
 2016-12-19 16:07:14.251 [41436] main/101/myapp.lua I> mapping 1073741824 bytes for tuple arena...
 2016-12-19 16:07:14.255 [41436] main/101/myapp.lua I> recovery start
 2016-12-19 16:07:14.255 [41436] main/101/myapp.lua I> recovering from `./00000000000000000000.snap
↪'
 2016-12-19 16:07:14.271 [41436] main/101/myapp.lua I> recover from `./00000000000000000000.xlog'
 2016-12-19 16:07:14.271 [41436] main/101/myapp.lua I> done `./00000000000000000000.xlog'
 2016-12-19 16:07:14.272 [41436] main/102/hot_standby I> recover from `./00000000000000000000.xlog'
 2016-12-19 16:07:14.274 [41436] iproto/102/iproto I> binary: started
 2016-12-19 16:07:14.275 [41436] iproto/102/iproto I> binary: bound to [::]:3301
 2016-12-19 16:07:14.275 [41436] main/101/myapp.lua I> done `./00000000000000000000.xlog'
 2016-12-19 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.

3.4.2 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). 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"}
                    }
                }
            }
        }
    },
    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 переменные 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'a *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'a. 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
```

```
    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, поймали 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;
   }

   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
     }
 }

 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 запуска 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*.

3.4.3 Установка модуля

Модули на Lua и C от разработчиков Tarantool'a и сторонних разработчиков доступны здесь:

- Репозиторий модулей Tarantool'a и
- Репозитории deb/rpm Tarantool'a.

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')`.

3.4.4 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.

3.4.5 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()
     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)
  -- do some business
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
            --
            -- hot code reload example
            --

            box.cfg{listen = 3302}})

            -- ATTENTION: unload it all properly!
            local app = package.loaded['app']
            if app ~= nil then
              -- stop the old application version
              app.stop()
              -- unload the application
              package.loaded['app'] = nil
              -- unload all dependencies
              package.loaded['somedep'] = nil
            end

            -- load the application
            log.info('require app')
            app = require('app')

            -- start the application
            app.s{some app options controlled by sysadmins}mins})
```

   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.

3.4.6 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.

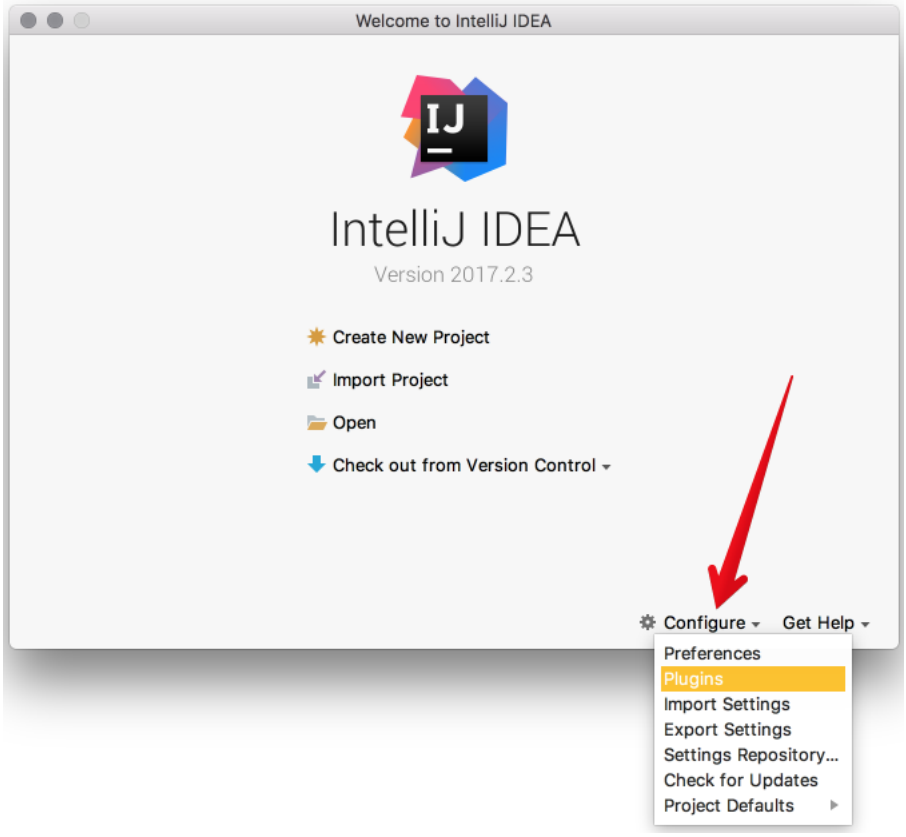
   JetBrains предоставляет специализированные версии для разных языков программирования: IntelliJ IDEA (Java), PHPStorm (PHP), PyCharm (Python), RubyMine (Ruby), CLion (C/C++), WebStorm (Web) и другие. Поэтому загрузите версию, которая подходит предпочитаемому языку.

   Для всех версий поддерживается интеграция с Tarantool'ом.

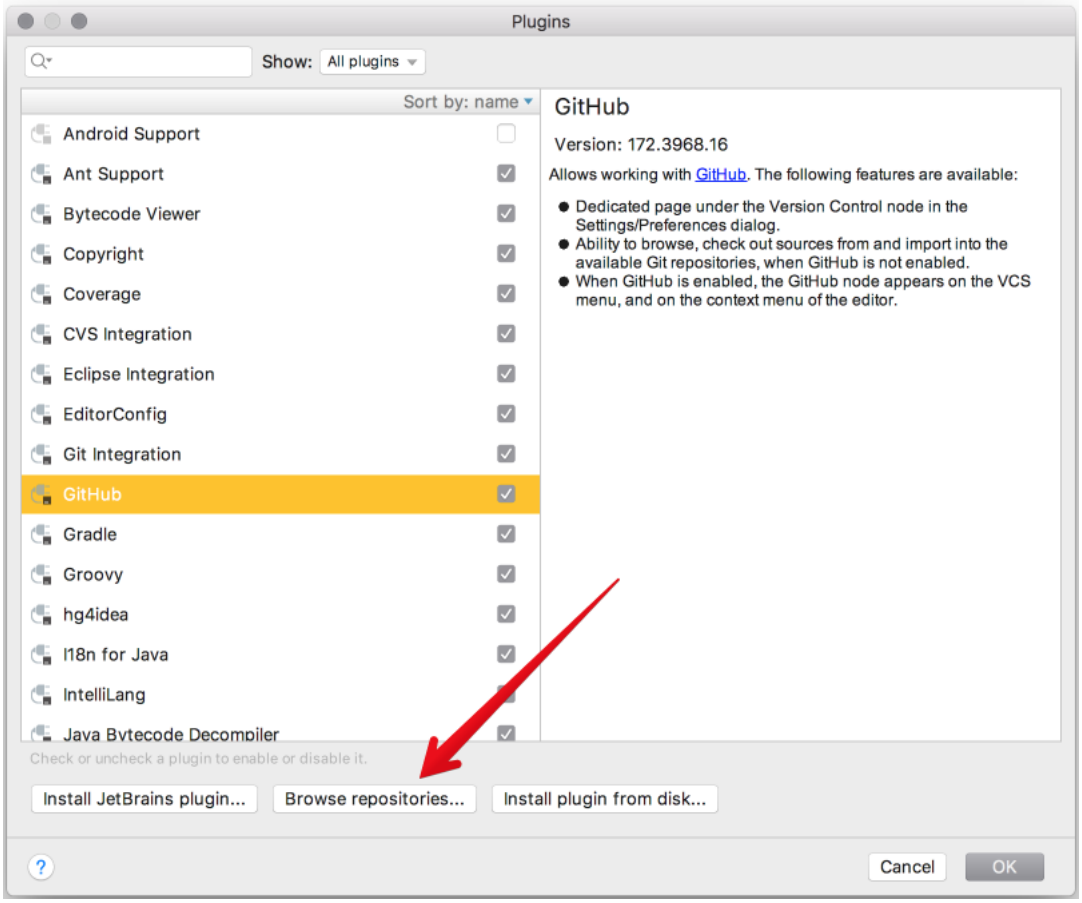
2. Настройте IDE:

   (a) Запустите IntelliJ IDEA.

   (b) Нажмите кнопку `Configure` и выберите `Plugins`.

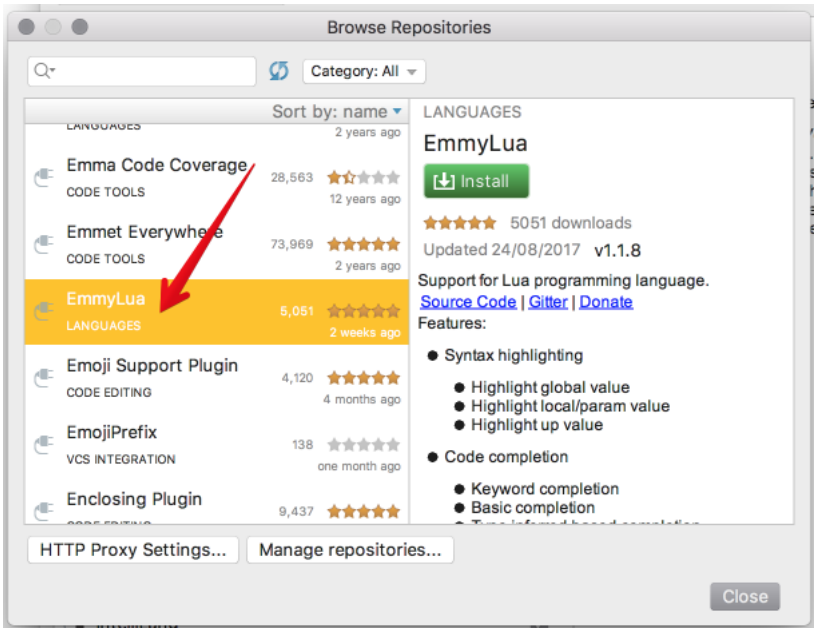


   (c) Нажмите `Browse repositories`.



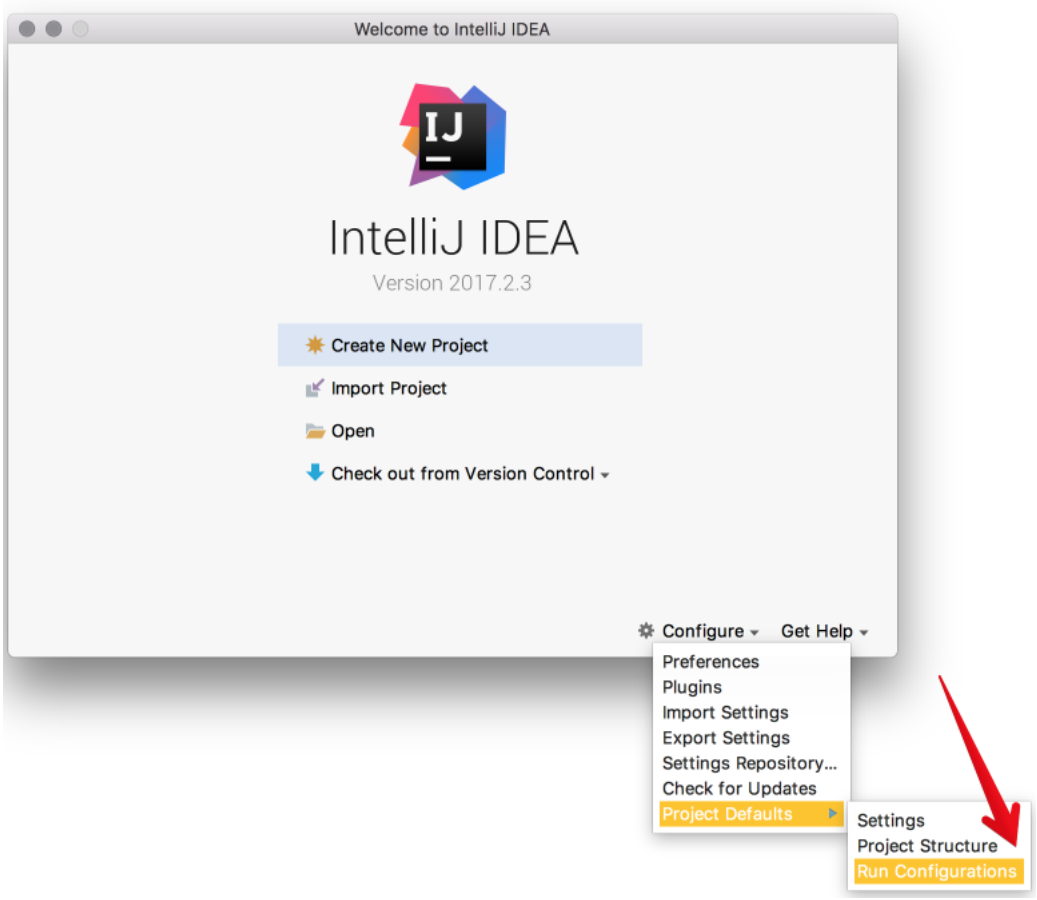
(d) Установите плагин `EmmyLua`.

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`.

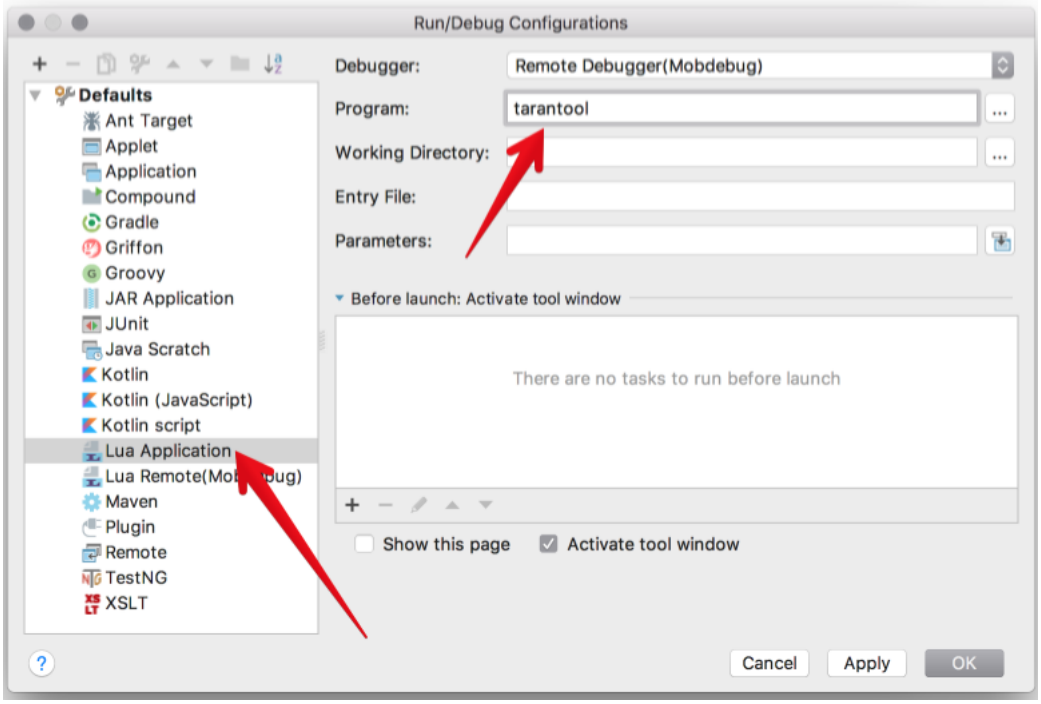


(g) Найдите `Lua Application` в боковой панели слева.

(h) В `Program` введите путь к установленному бинарному файлу `tarantool`.

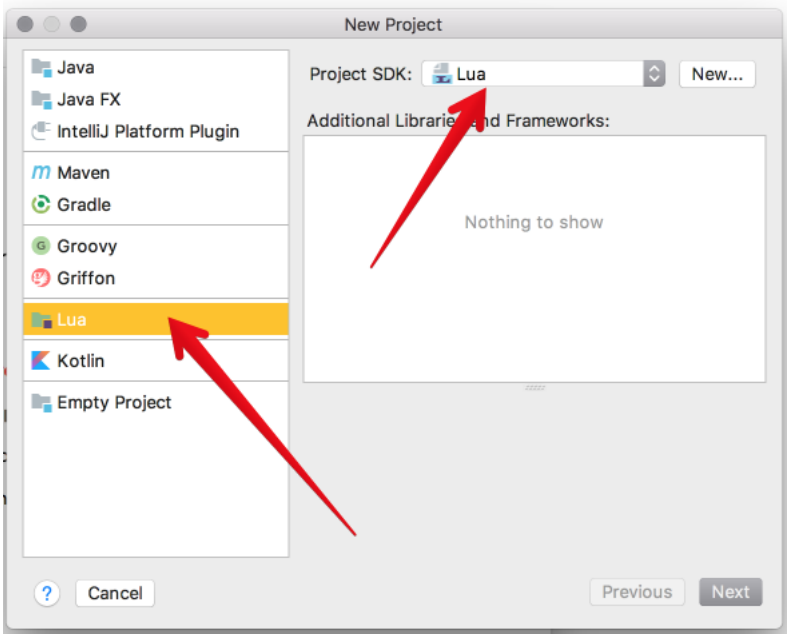
По умолчанию, это `tarantool` или `/usr/bin/tarantool` на большинстве платформ.

Если вы установили `tarantool` из источников в другую директорию, укажите здесь правильный путь.

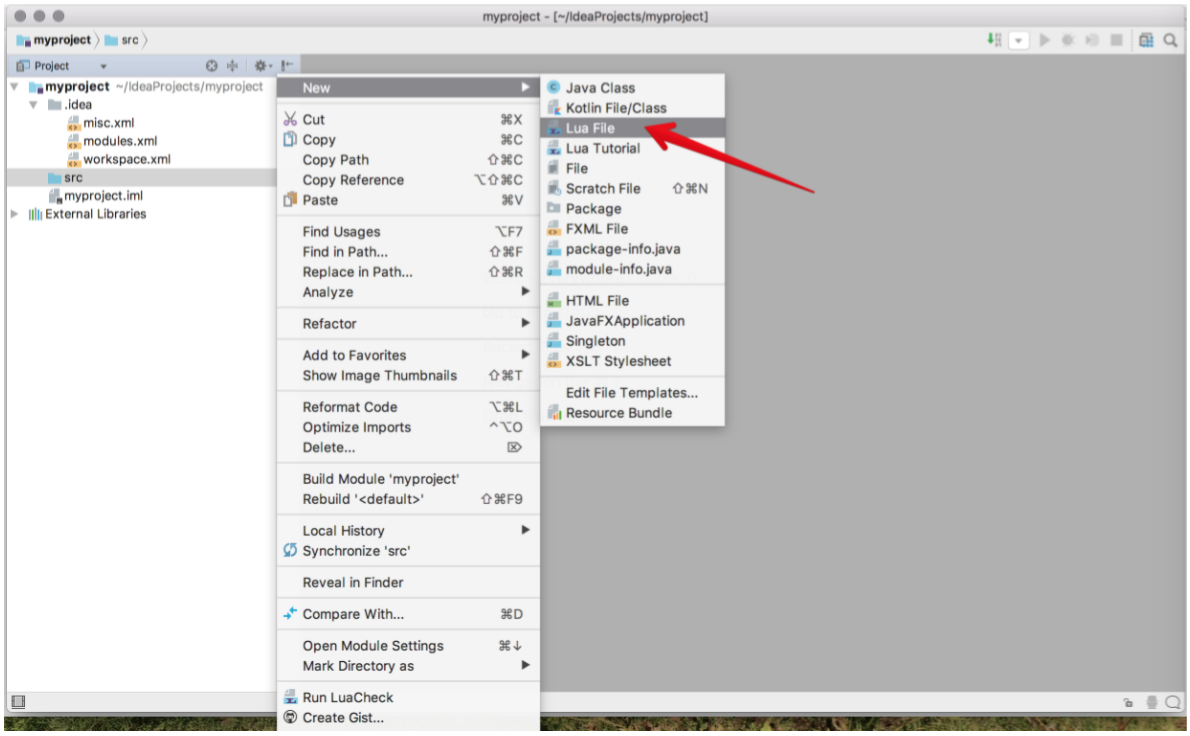


Теперь IntelliJ IDEA можно использовать с Tarantool'ом.

3. Создайте новый проект на Lua.

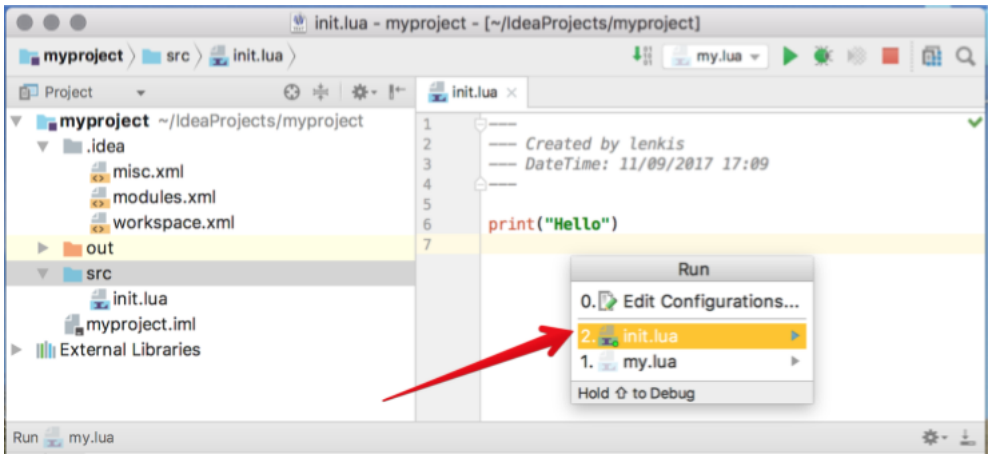


4. Добавьте новый Lua-файл, например, `init.lua`.



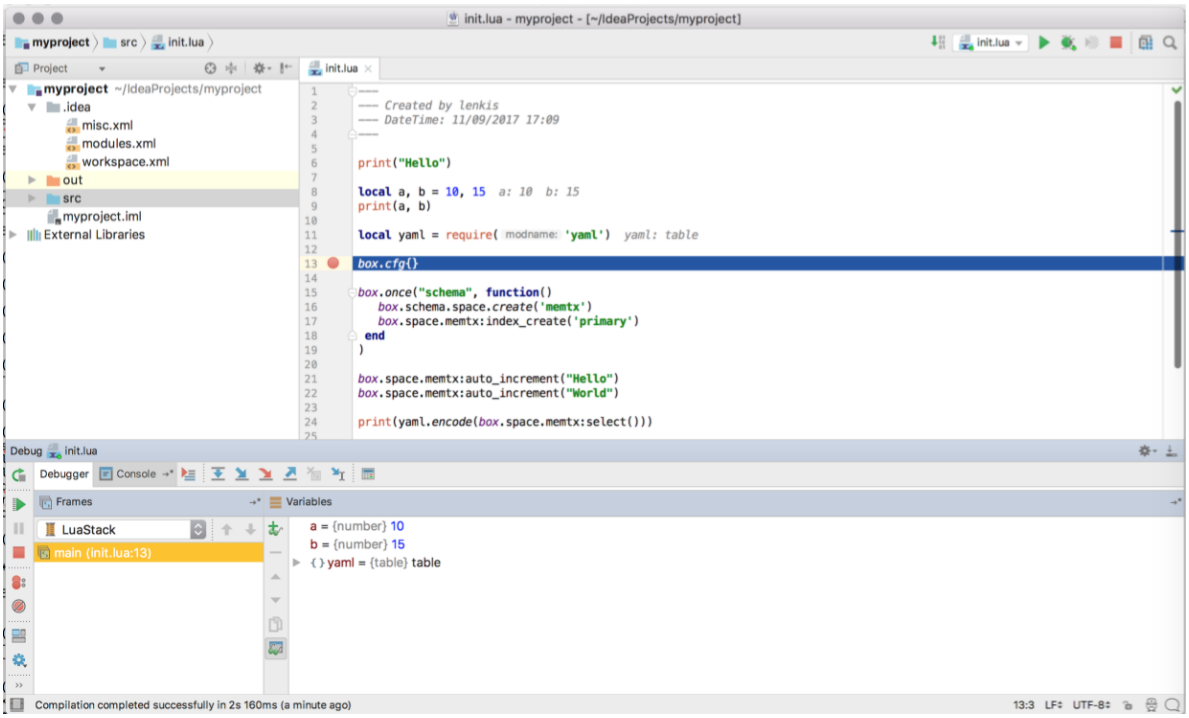
5. Разработайте код, сохраните файл.

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.



3.4.7 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*
- *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*

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'}})
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)
    assert(res == 0)
    return ffi.string(buf, buflen[0])
end

-- Простой код тестов
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
```

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 значений nils.

```
#!/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 eof
        end
        if not s:write("pong: "..line) then
            break -- ошибка или eof
        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)
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

    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('http://api.openweathermap.org/data/2.5/weather?q=Oakland,us')
if r.status ~= 200 then
    print('Failed to get weather forecast ', r.reason)
    return
end
local data = json.decode(r.body)
print('Oakland wind speed: ', data.wind.speed)
```

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) -- анализировать связь с *:8080
        server:route({ path = '/' }, handler)
        server:start()
        -- подключиться к localhost:8080 и читать 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 означает '*'
 server: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>
```

3.5 Администрирование серверной части

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:

3.5.1 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",
  }
  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.
- `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`.

В качестве полноценного примера можно использовать скрипт example.lua, который поставляется вместе с Tarantool и задает все конфигурационные параметры.

3.5.2 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 exampl[e]
  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

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
```

3.5.3 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
          /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.

3.5.4 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 1.10.0 (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.

3.5.5 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
  ...
  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
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,

Можно также попробовать воспользоваться Lua-модулем tarantool/prometheus, который облегчает сбор метрик (например, использование памяти или количество запросов) с Tarantool-приложений и баз данных и их публикацию через протокол Prometheus.

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'а. 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'а. *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'а:

```
$ 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
  #22 box_lua_call (request=request@entry=0x7f72130401d8, out=out@entry=0x7f7213020600) at /usr/
↪src/tarantool/src/box/lua/call.c:405
  #23 0x00000000004c0f27 in box_process_call (request=request@entry=0x7f72130401d8,␣
↪out=out@entry=0x7f7213020600) at /usr/src/tarantool/src/box/box.cc:1074
  #24 0x000000000041326c in tx_process_misc (m=0x7f7213040170) at /usr/src/tarantool/src/box/
↪iproto.cc:942
  #25 0x0000000000504554 in cmsg_deliver (msg=0x7f7213040170) 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'а. 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'а 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'а:

```
$ 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
                        | | |
                        | | --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

The jit.p profiler comes with the Tarantool application server, to load it one only needs to say `require('jit.p')` or `require('jit.profile')`. There are many options for sampling and display, they are described in the documentation for The LuaJIT Profiler.

Пример

Make a function that calls a function named f1 that does 500,000 inserts and deletes in a Tarantool space. Start the profiler, execute the function, stop the profiler, and show what the profiler sampled.

```
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
```

Typically the result will show that the sampling happened within f1() many times, but also within internal Tarantool functions, whose names may change with each new version.

3.5.6 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
  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'a,
- «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'a: 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)
                #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`.

3.5.7 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-lsn 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. Переключите все узлы в *режим только для чтения* и отключите командой *box.backup.start()* создание контрольных точек. Последнее действие необходимо, чтобы сборщик мусора автоматически не удалил более старые контрольные точки.
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).

3.5.8 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.

Функции backup.start() и backup.stop()

Данные функции используются для резервного копирования в определенных ситуациях.

`box.backup.start()` сообщает серверу, что следует отложить некоторые действия, которые могут помешать резервному копированию – отложить создание контрольной точки, отложить сборку мусора и фактически перейти в режим только для чтения.

Затем `box.backup.stop()` сообщает серверу, что можно возобновить нормальную работу. Начиная с версии Tarantool'а 1.10.1, добавлен необязательный аргумент `box.backup.start(n)`, где `n` обозначает используемую контрольную точку относительно последней контрольной точки – например, `n = 0` означает, что резервное копирование будет выполняться по последней контрольной точке, `n = 1` означает, что резервное копирование будет выполняться по первой контрольной точке от последней контрольной точки (счет в обратном порядке)» и так далее, а по умолчанию для `n` используется ноль.

`box.backup.start()` возвращает таблицу с именами снимка и файлов vinyl'а, которые следует копировать. Например:

```
tarantool> box.backup.start()
---
- - ./00000000000000000015.snap
  - ./00000000000000000000.vylog
  - ./513/0/00000000000000000002.index
  - ./513/0/00000000000000000002.run
...
```

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 приостановит сборку мусора до вызова `box.backup.stop()` и покажет список файлов для резервирования.
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.

3.5.9 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...
 grant read access to 'public' role for _vindex view
 set schema version to 1.7.0
 ---
 ...
```

Обновление экземпляра Tarantool'а

Tarantool поддерживает обратную совместимость между двумя последовательными версиями. Например, обновление Tarantool 1.6 до 1.7 или Tarantool 1.7 до 1.8 не должно вызвать затруднений, тогда как миграции с Tarantool 1.6 прямиком на 1.8 могут препятствовать несовместимые изменения.

Как обновить Tarantool 1.6 до 1.7 / 1.10

Этот процесс предназначен для обновления индивидуальных экземпляров Tarantool'а с 1.6.x до 1.7.x (или до 1.10.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'а 1.6 поддерживаются при обновлении, но после непродолжительного использования Tarantool'а 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'а 1.7 / 1.9 / 1.10.

Чтобы обновить Tarantool 1.6 до 1.7 (или до 1.10.x):

1. Уточните у разработчиков, необходимо ли обновлять файлы приложения из-за наличия несовместимых изменений (см. Примечания к версиям Tarantool'a 1.7 / 1.9 / 1.10). 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 (а также Tarantool 1.9 и 1.10) 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.

3.5.10 Замечания по поводу некоторых операционных систем

Mac OS

Администрирование экземпляров Tarantool'а 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'а*. 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
```

3.5.11 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.

3.5.12 Руководство по разрешению проблем

В данном руководстве используется сторонний модуль stat. Для его установки выполните команду:

```
$ sudo yum install tarantool-stat
$ # -- ИЛИ --
$ sudo apt-get install tarantool-stat
```

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> require('stat').stat()['slab.arena_used_ratio']

  -- запрашиваем значение quota_used_ratio
  tarantool> require('stat').stat()['slab.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> require('stat').stat()['slab.quota_used_ratio']

 -- запрашиваем значение items_used_ratio
 tarantool> require('stat').stat()['slab.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> require('stat').stat()['stat.op.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> require('stat').stat()['stat.op.<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 1.9+, 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
 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.

3.6 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:

3.6.1 Архитектура механизма репликации

Механизм репликации

Набор экземпляров, которые работают на копиях одной базы данных, составляют **набор реплик**. 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.

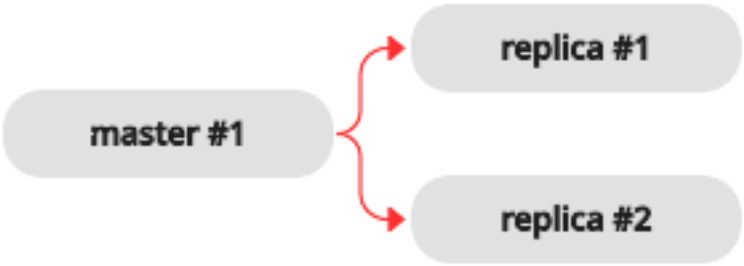
Both these parameters are «dynamic». This allows a replica to become a master and vice versa on the fly with the help of a `box.cfg{}` request.

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).

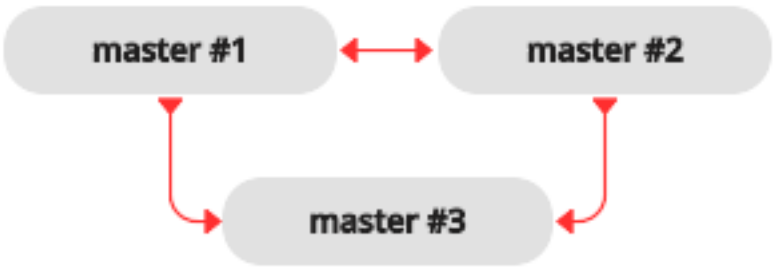
В конфигурации мастер-реплика каждое изменение, сделанное на мастере, будет отображаться на репликах, но не наоборот.



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.

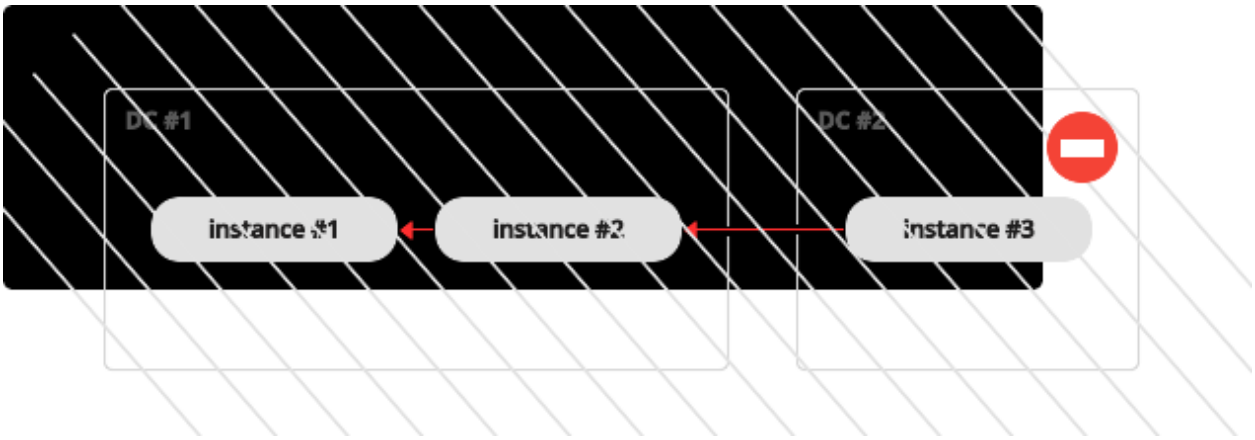
Однако операции обновления UPDATE могут с легкостью привести к рассинхронизации. Например, операции присваивания и увеличения не обладают коммутативностью и могут привести к различным результатам, если применять их в различном порядке на разных экземплярах.

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

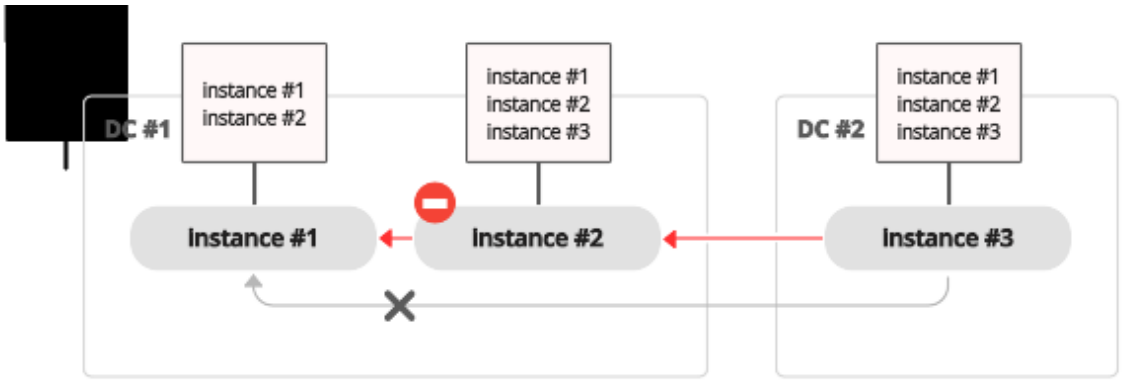
Топология репликации определяется в конфигурационном параметре *replication*. Рекомендована **полная ячеистая** конфигурация, поскольку она облегчает возможное восстановление после сбоя.

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`.

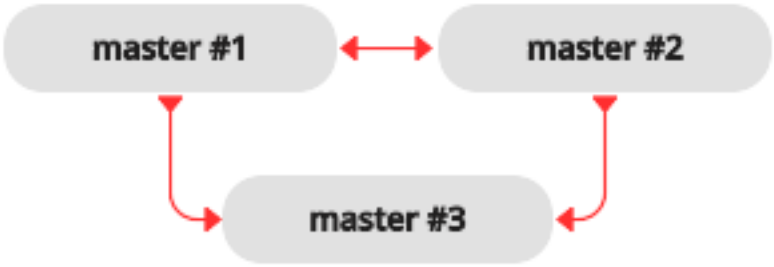


Теперь экземпляр №2 неисправен. Экземпляр №3 пытается подключиться к экземпляру №1, как к новому мастеру, но мастер отклоняет подключение, поскольку не содержит запись для экземпляра №3.

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.

3.6.2 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")
  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.

  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'})
  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
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
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
```

3.6.3 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:

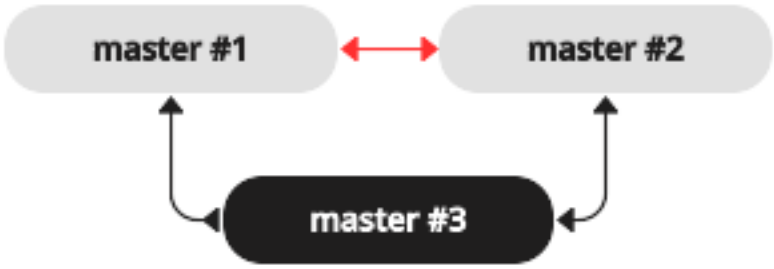
```
$ # запуск реплики №2
$ tarantool replica2.lua
2017-06-14 14:54:33.927 [46945] main/101/replica2.lua C> version 1.7.4-52-g980d30092
2017-06-14 14:54:33.927 [46945] main/101/replica2.lua C> log level 5
2017-06-14 14:54:33.928 [46945] main/101/replica2.lua I> mapping 268435456 bytes for tuple arena...
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, -- temporarily read-only
}
box.once("schema", function()
   box.schema.user.create('replicator', {password = 'password'})
   box.schema.user.grant('replicator', 'replication') -- предоставить роль "replication"
   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
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».

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 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. Подключение к количеству узлов, указанному в *replication_connect_quorum*.
3. Синхронизация 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'a 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.

3.6.4 Удаление экземпляров

Чтобы правильно удалить экземпляр из набора реплик, выполните следующие действия:

1. Выполните `box.cfg{}` с пустым источником репликации на экземпляре:

   ```
   tarantool> box.cfg{replication=''}
   ---
   ...
   ```

   Остальные экземпляры продолжают работать. Если выбывший экземпляр снова возвращается в кластер, то он получит информацию о всех изменениях, которые произошли на остальных экземплярах за время его отсутствия.

2. Если экземпляр больше не будет использоваться, удалите записи об экземпляре из следующих мест:

   (a) параметр *replication* на всех работающих экземплярах в наборе реплик:

   ```
   tarantool> box.cfg{replication=...}
   ```

   (b) кортеж *box.space._cluster* на любом мастере в наборе реплик. Например, для записи с ID экземпляра = 3:

   ```
   tarantool> box.space._cluster:select{}
   ---
   - - [1, '913f99c8-aee3-47f2-b414-53ed0ec5bf27']
     - [2, 'eac1aee7-cfeb-46cc-8503-3f8eb4c7de1e']
     - [3, '97f2d65f-2e03-4dc8-8df3-2469bd9ce61e']
   ...
   tarantool> box.space._cluster:delete(3)
   ---
   - [3, '97f2d65f-2e03-4dc8-8df3-2469bd9ce61e']
   ...
   ```

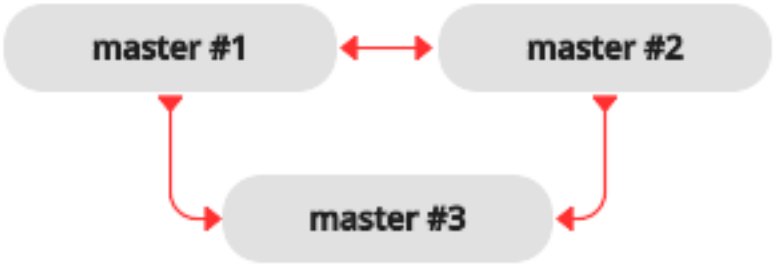
3.6.5 Мониторинг набора реплик

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.

  Реплика отправляет сообщения контрольного сигнала на мастер каждую секунду, и мастер запрограммирован на автоматическое переподключение, если он не получает сообщения контрольного сигнала в течение количества секунд, указанного в *replication_timeout*.

  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

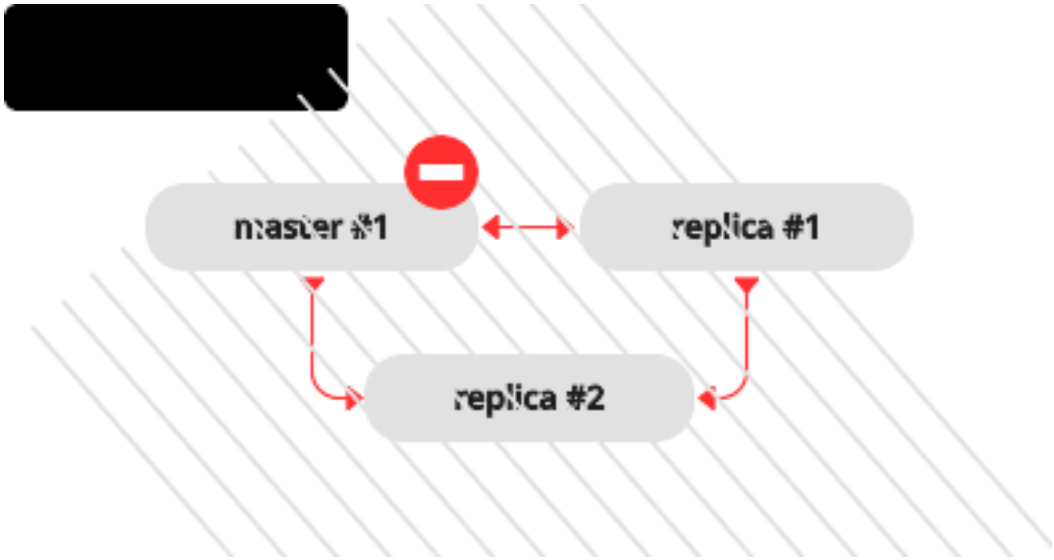
записи события в *журнал упреждающей записи* на этом мастере.

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.

3.6.6 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.

3.6.7 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.

3.6.8 Предотвращение дублирующихся действий

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.

Остановка репликации

Предположим, что в наборе реплик с двумя мастерами мастер №1 пытается сделать что-то, что уже было сделано мастером №2. Например, попробуйте 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:
    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.

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}})
# at master #2
tarantool> box.space.tester:upsert({1}, {{'=', 2, box.info.uuid}})
```

... можно получить следующие результаты в зависимости 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.

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.

3.7 Коннекторы

В этой главе описаны API для различных языков программирования.

3.7.1 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.

3.7.2 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.

3.7.3 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')
```

3.7.4 Java

См. http://github.com/tarantool/tarantool-java/.

3.7.5 Go

См. https://github.com/mialinx/go-tarantool.

3.7.6 R

См. https://github.com/thekvs/tarantoolr.

3.7.7 Erlang

См. Erlang-драйвер для Tarantool'a.

3.7.8 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
$ 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.

3.7.9 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.

3.7.10 Python

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
```

Для подготовки сохраните код в файл с именем `example.py` и установите коннектор `tarantool-python`. Для установки коннектора воспользуйтесь либо командой samp:*pip install tarantool>0.4* для установки в директорию `/usr` (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 проверьте, что у экземпляра задан порт для *прослушивания* на `localhost:3301`, и в базе создан спейс `examples`, как *описано выше*. Чтобы запустить программу, выполните команду `python example.py`. Программа установит соединение с Tarantool-сервером, пошлет *INSERT-запрос* и не выбросит никакого исключения, если всё прошло хорошо. Если такой кортеж уже существует, то программа выбросит 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.

3.7.11 Node.js

Самый используемый драйвер для node.js – Node Tarantool driver. Он не входит в репозиторий Tarantool'а, 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.

3.7.12 C#

Самый используемый драйвер для C# – progaudi.tarantool, который раньше назывался `tarantool-csharp`. Он не входит в репозиторий Tarantool'а, 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.

3.7.13 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
```

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, выполните команду `./example`. Программа установит соединение с экземпляром Tarantool'а и отправит запрос. Если Tarantool не запущен на localhost на прослушивание по порту 3301, то программа выдаст сообщение об ошибке “Connection refused”. Если вставка не сработает, программа выдаст сообщение об ошибке «Insert failed» и код ошибки (все коды ошибок см. в исходном файле /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

Далее приводится еще один пример полноценной программы на языке C, которая осуществляет выборку по индекс-ключу `[99999]` из спейса `examples` с помощью высокоуровневого Tarantool API для языка C. Для вывода результатов в этой программе используются функции из библиотеки MsgPuck. Эти функции нужны для декодирования массивов значений в формате MessagePack.

```
#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);
            }
        }
    }
    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.

3.7.14 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*.

3.8 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 различные 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 4

Справочники

4.1 Справочник по встроенным модулям

В данном справочнике рассматриваются встроенные Lua-модули Tarantool'а.

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'а.

4.1.1 Модуль *box*

Помимо выполнения фрагментов кода на Lua или определения собственных функций, с помощью модуля `box` и вложенных модулей можно использовать функции хранилища Tarantool'а.

Содержимое модуля `box` можно просмотреть во время исполнения кода с помощью команды `box` без аргументов. Модуль `box` включает в себя следующее:

Вложенный модуль *box.cfg*

Вложенный модуль `box.cfg` предназначен для системных администраторов, чтобы указать все *параметры конфигурации сервера*.

Введите команду `box.cfg` без фигурных скобок для просмотра текущей конфигурации, например:

```
tarantool> box.cfg
 ---
 - checkpoint_count: 2
   too_long_threshold: 0.5
   slab_alloc_factor: 1.1
   memtx_max_tuple_size: 1048576
   background: false
```

```
  <...>
...
```

Чтобы установить параметры, введите команду `box.cfg{...}`, например:

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

Если ввести `box.cfg{}` без параметров, Tarantool применит настройки по умолчанию:

```
tarantool> box.cfg{}
tarantool> box.cfg -- sorted in the alphabetic order
---
- background                   = false
  checkpoint_count             = 2
  checkpoint_interval          = 3600
  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_timeout  = 4
  replication_skip_conflict    = false
  replication_sync_lag         = 10
  replication_sync_timeout     = 300
  replication_timeout          = 1
  rows_per_wal                 = 500000
  slab_alloc_factor            = 1.05
  snap_io_rate_limit           = nil
  too_long_threshold           = 0.5
  username                     = nil
  vinyl_bloom_fpr              = 0.05
  vinyl_cache                  = 128
  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          = 2
wal_dir                      = '.'
wal_dir_rescan_delay         = 2
wal_max_size                 = 256 * 1024 * 1024
wal_mode                     = 'write'
worker_pool_threads          = 4
work_dir                     = nil
```

Первый вызов `box.cfg{...}` (с параметрами или без них) запускает модуль базы данных Tarantool'а под названием *box*. Чтобы выполнить любые операции с базой данных, необходимо сначала вызвать `box.cfg{...}`.

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 (ошибки могут возникать 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 (ошибки могут возникать из-за превышения времени ожидания или прерывания работы файбера)

Пример:

```
tarantool> box.ctl.wait_rw(0.1)
           ---
           ...
```

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-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.

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

Type	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'a 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
  }
  if (search-key-part[i] == index-key-part[i])
  {
    if (iterator is LT or GT) return FALSE
    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

Type	Аргументы	Описание
box.index.ALL	index	Все ключи индекса являются совпадениями. Найденные кортежи упорядочены по возрастанию по хешу ключа индекса, который будет выглядеть случайным.
box.index.EQ или „EQ“	index, 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“	index, 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

Type	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

Type	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.
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 значения	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')
---
...
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
- `options.iterator` – тип итератора
- `options.limit` (*number*) – максимальное количество кортежей
- `options.offset` (*number*) – 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',
         >   limit = 1000
         > })
```

Результатом будет следующая таблица кортежа:

```
---
    - - [2, 'Y', 'Row with field[2]=Y']
      - [3, 'Z', 'Row with field[2]=Z']
    ...
```

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: Параметр *`field-value [, значение поля ...]`* необязателен. Если он пропущен, то каждый ключ в индексе будет считаться совпадением независимо от типа итератора. Таким образом, для примера выше, `box.space.tester:select{}` выберет каждый кортеж в спейсе tester по первому индексу (первичный ключ).

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` вернет ошибку.

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}
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. В будущей версии Tarantool'a index: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. В будущей версии Tarantool'a 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 ключей индекса.

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.

тип возвращаемого значения кортеж

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}*)
Alter an index. It is legal in some circumstances to change one or more of the index characteristics, for example its type, its sequence options, its parts, and whether it is unique, Usually this causes rebuilding of the space, except for the simple case where a part's `is_nullable` flag is changed from `false` to `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,
- the primary-key index cannot be changed to `{unique = false}`.

Note re storage engine: vinyl does not support `alter()` of a primary-key index unless the space is empty.

Пример:

```
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` – общее количество байтов;
- `index_object:stat().disk.rows` – 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.compact` 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().cache.evict` содержит количество освобождений кэша.
- `index_object:stat().range_size` – maximum number of bytes in a range

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 *конфигурационного параметра времени ожидания timeout*, 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;
- вернуть ошибку, 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
...
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'})
---
- - [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. :ref: *ниже <box_info_gc>*.
- **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** соответствует статусу **replication.upstream.status** (см. ниже).
- **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 для времени исполнения Lua-кода.
- **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»*):

- **replication.id** – это короткий числовой идентификатор экземпляра в рамках набора реплик.
- **replication.uuid** – это глобально-уникальный идентификатор экземпляра. Данное значение также хранится в системном спейсе *box.space._cluster*.
- **replication.lsn** – это *номер в журнале* (LSN) для последней записи в *журнале упреждающей записи* (WAL) экземпляра.
- **replication.upstream** содержит статистику по реплицируемым данным, которые переданы экземпляром.
- **replication.upstream.status** – это репликационный статус экземпляра:
  - `auth` означает, что экземпляр проходит *аутентификацию* для установки соединения с источником репликации.
  - `connecting` означает, что экземпляр пытается установить соединение с источниками репликации, перечисленными в параметре *replication*.
  - `disconnected` означает, что экземпляр не подключен к набору реплик (по причине проблем в сети, а не ошибок репликации).
  - `follow` означает, что идет репликация.
  - `running` означает, что роль экземпляра – «мастер» (не только для чтения), и идет репликация.
  - `stopped` означает, что репликация остановилась по причине ошибки репликации (например, *повторяющийся ключ*).
  - Статус `orphan` означает, что экземпляр (еще) не подключился к необходимому количеству мастеров (см. *статус orphan*).
  - `synch` means that the master and replica are synchronizing to have the same data.
- **replication.upstream.idle** – это время (в секундах) с момента получения экземпляром последнего события от мастера. Это основной индикатор работоспособности репликации. Более подробную информацию см. в разделе *Мониторинг набора реплик*.
- **replication.upstream.peer** содержит имя пользователя, запустившего репликацию, IP-адрес хоста и номер порта, используемый для экземпляра. Более подробную информацию см. в разделе *Мониторинг набора реплик*.
- **replication.upstream.lag** – это 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. Более подробную информацию см. в разделе *Мониторинг набора реплик*.
- **replication.upstream.message** содержит сообщение об ошибке в случае *системного сбоя*, в противном случае не заполнен.
- **replication.downstream** содержит статистику по реплицируемым данным, которые запрошены и загружены с экземпляра.

- **replication.downstream.vclock** содержит *векторные часы*, что представляет собой таблицу из пар „**id**, **lsn**“, например `vclock: {1: 3054773, 4: 8938827, 3: 285902018}`. Даже если экземпляр *удален*, тем не менее, значения с него появятся здесь.
- **replication.downstream.status** = `disconnected` отображается, если последующий экземпляр отключается от предыдущего. В остальных случаях такой статус не отображается.

`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()
 ---
 - version: 1.7.6-68-g51fcffb77
   id: 2
   ro: true
   vclock: {1: 5}
   uptime: 917
   lsn: 0
   vinyl: []
   cluster:
     uuid: 783e2285-55b1-42d4-b93c-68dcbb7a8c18
   pid: 35341
   status: running
   signature: 5
   replication:
     1:
       id: 1
       uuid: 471cd36e-cb2e-4447-ac66-2d28e9dd3b67
       lsn: 5
       upstream:
         status: follow
         idle: 124.98795700073
         peer: replicator@192.168.0.101:3301
         lag: 0
       downstream:
         vclock: {1: 5}
     2:
       id: 2
       uuid: ac45d5d2-8a16-4520-ad5e-1abba6baba0a
       lsn: 0
   uuid: ac45d5d2-8a16-4520-ad5e-1abba6baba0a
 ...
```

Функция *box.once*

`box.once(`*key*, *function*`[`, ...`]`)`

Выполнение функции при условии, что она раньше не выполнялась. Передаваемое значение про-

веряется 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)
    ---
    ...
```

Параметры

- `key` (`string`) – значение для проверки
- `function` (*function*) – функция
- `...` – аргументы, которые следует передать в функцию

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()	Создание спейса
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}*])

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

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
         > })
---
```

```
...
```

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 `object-type, object-name` 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` (имя роли) (см. раздел *Роли*).

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*, *privilege*, *object-type*, *object-name*)
`box.schema.user.revoke`(*user-name*, *role-name*)

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.

Должен существовать пользователь, должен существовать объект, но ошибка не выпадет, если у пользователя нет прав.

Вариант: вместо `object-type, object-name` введите „universe“, что означает „все типы объектов и все объекты“.

Вариант: вместо `privilege, object-type, object-name` введите `role-name` (имя роли) (см. раздел *Роли*).

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*])

Возврат описания *прав* 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. Необязательный параметр; если не указать, информация будет для авторизованного пользователя.

Пример:

```
box.schema.user.info()
    box.schema.user.info('Lena')
```

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 `object-type, object-name` 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 `object-type, object-name` 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}*]`)`
Создание *кортежа* с функцией. Сама функция не создается – это делается с помощью Lua – но если необходимо выдать права функции, следует сначала выполнить box.schema.func.create. Чтобы получить информацию о том, как происходит управление данными функций в Tarantool'е, см. справочник по спейсу *_func*.

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` (по умолчанию, false) – значение `true` (правда) заставит Tarantool рассматривать пользователя, вызвавшего функцию, в качестве владельца функции с полными правами. Следует помнить, что SETUID работает только по *бинарным портам*. SETUID не сработает, если вызвать функцию через *административную консоль* или в Lua-скрипте.
- `language` = „LUA“|“C“ (выбор языка из Lua и C; по умолчанию, 'LUA').

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.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.
- `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 вернуть ошибку (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=200,
                    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]* ...)

Можно использовать опцию `sequence=`*`имя-последовательности`* (или `sequence=`*`id-последовательности`*, или `sequence=true`) при *создании* или *изменении* первичного индекса. Происходит ассоциация последовательности с индексом, так что следующий вызов `insert()` поместит следующее сгенерированное число в поле первичного ключа, если в противном случае поле было бы nil.

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
      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{nil,0}
    ---
    - [1, 0]
    ...
```

Примечание: Если вы используете отрицательные числа в параметрах последовательности, убедитесь, что тип ключа индекса будет целое число „integer“. В противном случае, тип ключа может быть либо „integer“, либо „unsigned“ (без знака).

A sequence cannot be dropped if it is associated with an index. However, *index_object:alter()* can be used to say that a sequence is not associated with an index, for example `box.space.T.index.I:alter({sequence=false})`.

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 1, если сессия есть; 0, если сессии нет.

тип возвращаемого значения число

`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

`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
```

```
---
...
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.

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.

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.

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)
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
-- 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: риск высокий, если высоки значения и `arena_used_ratio`, и `quota_used_ratio` (90-95%).

Если значение `quota_used_ratio` низкое, то высокое значение `arena_used_ratio` и/или `items_used_ratio` указывает на низкую фрагментацию памяти (т.е. память используется эффективно).

Если значение `quota_used_ratio` высокое (достигает 100%), то низкое значение `arena_used_ratio` (50-60%) указывает на значительную фрагментацию памяти. Весьма вероятно, что в данном случае непосредственного риска нехватки памяти нет, но такую проблему следует тщательно рассмотреть. Например, есть риск того, что вся квота памяти используется на кортежи, а для части индекса slab'ов нет. Или все slab'ы выделены на хранение кортежей, а в действительности все они наполовину пусты.

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` / `slab_count` * `slab_size` (это slab'ы, которые используются только для кортежей, не для индексов);
- `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'a* (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'a.

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'a `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
    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

Вложенный модуль `box.space` включает в себя функции по управлению данными `select` (выборка), `insert` (вставка), `replace` (замена), `update` (обновление), `upsert` (обновление и вставка), `delete` (удаление), `get` (получение), `put` (выдача). Также в модуле есть такие элементы, как id, и указание на активность спейса. Код вложенного модуля находится в файле src/box/lua/schema.lua.

Индекс

Ниже приведен перечень всех функций и элементов модуля `box.space`.

Имя	Использование
space_object:auto_increment()	Генерация ключа + вставка кортежа
space_object:bsize()	Подсчет байтов
space_object:count()	Подсчет кортежей
space_object:create_index()	Создание индекса
space_object:delete()	Удаление кортежа
space_object:drop()	Удаление спейса
space_object:format()	Объявление имен и типов полей
space_object:frommap()	Конвертация ассоциативного массива в кортеж или таблицу
space_object:get()	Выбор кортежа
space_object:insert()	Вставка кортежа

Continued on next page

Таблица 4.1 – continued from previous page

Имя	Использование
space_object:len()	Подсчет кортежей
space_object:on_replace()	Создание триггера замены с функцией, которая не может изменять кортеж
space_object:before_replace()	Создание триггера замены с функцией, которая может изменять кортеж
space_object:pairs()	Подготовка к итерации
space_object:put()	Вставка или замена кортежа
space_object:rename()	Переименование спейса
space_object:replace()	Вставка или замена кортежа
space_object:run_triggers()	Включение/отключение триггера замены
space_object:select()	Выбор одного или более кортежей
space_object:truncate()	Удаление всех кортежей
space_object:update()	Обновление кортежа
space_object:upsert()	Обновление кортежа
space_object:user_defined()	Любая функция / метод, которые хочет добавить любой пользователь
space_object.enabled	Флаг, если спейс активен – true
space_object.field_count	Необходимое количество полей
space_object.id	Числовой идентификатор спейса
space_object.index	Контейнер для индексов спейса
box.space._cluster	(Метаданные) Список наборов реплик
box.space._func	(Метаданные) Список кортежей с функциями
box.space._index	(Метаданные) Список индексов
box.space._vindex	(Метаданные) Список индексов, доступных текущему пользователю
box.space._priv	(Метаданные) Список прав
box.space._vpriv	(Метаданные) Список прав, доступных текущему пользователю
box.space._schema	(Метаданные) Список схем
box.space._sequence	(Метаданные) Список последовательностей
box.space._sequence_data	(Метаданные) Список последовательностей
box.space._space	(Метаданные) Список спейсов
box.space._vspace	(Метаданные) Список спейсов, доступных текущему пользователю
box.space._user	(Метаданные) Список пользователей
box.space._vuser	(Метаданные) Список пользователей, доступных текущему пользователю

`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“ или „string“, или „integer“, или „number“, или „boolean“, или „array“, или „scalar“, возможна сортировка, возможно значение is_nullable}	`{1, '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

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*.

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 = {1, 'unsigned', 2, 'string'}
↪})
---
...
```

Подробнее о типах полей индекса:

Семь типов полей индекса (unsigned | string | integer | number | boolean | 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.
- **array**: массив чисел. Используется в *RTREE-индексах* в memtx'e.
- **scalar**: логические значения (true или false), целые числа от integers between -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 или строки. При использовании нескольких типов, порядок ключей должен быть следующим: логические значения, затем числа, затем строки. Используется в индексах типа TREE или HASH в memtx'e и в TREE-индексах в vinyl'e.

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“
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	логические значения (true или false), 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, строки	индексы типа TREE или HASH в memtx'e, TREE-индексы в vinyl'e	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={{2,'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'}}})
```

Примечание про движок базы данных: vinyl поддерживает только TREE-индексы, и следует создать в vinyl'е вторичные индексы до вставки кортежей.

`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.

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 допустимый тип для *индексируемых полей*: unsigned | string | integer | number | boolean | array | scalar (такое же требование, как для *«Параметров для space_ object:create_ index»*);
- 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={2,'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
---
- [['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'}].

Примечание про движок базы данных: vinyl поддерживает форматирование не пустых спейсов. Определение первичного индекса форматировать нельзя.

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}
```

Использование имен полей вместо номеров полей: в *get()* можно использовать имена полей, описанные в необязательном операторе *space_ object:format()*. Это аналогично стандартной Lua-функции, где на компонент можно ссылаться по имени, а не по номеру. Например, может форматировать спейс *tester* с полем под названием *x* и использовать имя *x* в определении индекса:

```
box.space.tester:format({{name='x',type='scalar'}})
box.space.tester:create_index('I',{parts={'x'}})
```

Тогда если get или select вернут отдельный кортеж, можно сослаться на поле „x“ в кортеже по имени:

```
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: Если уже существует кортеж с тем же уникальным значением ключа, возвращается `ER_TUPLE_FOUND`.

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
...
```

Note re storage engine: vinyl supports `len()` but the result may be approximate. If an exact result is necessary then use *count()* or `#select(...)`.

`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
- `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.

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)
```

В функции `trigger-function` могут быть два параметра: старый кортеж, новый кортеж. Например, следующий код вызывает вывод nil при обработке запроса на вставку и вывод [1, „Hi“] 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) print(old) end
box.space.space_1:on_replace(on_replace_function)
box.space.space_1:insert{1,'Hi'}
box.space.space_1:delete{1}
```

Примеры:

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 = {1, '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
```

`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
- `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.

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, гл 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*.

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 = ''
---
...
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: Если уже существует другой кортеж с тем же уникальным значением ключа, возвращается ER_TUPLE_FOUND. (Это случится 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(\[*key*\[, *options*\]\])

Поиск кортежа или набора кортежей в заданном спейсе.

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 = {1,'unsigned', 2, '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> -- необходимо совпадение с двумя полями первичного ключа
tarantool> s:select{1,'B'}
---
- - [1, 'B']
...
tarantool> -- необходимо совпадение только одного поля первичного ключа
tarantool> s:select{1}
---
- - [1, 'A']
  - [1, 'B']
  - [1, 'C']
...
tarantool> -- необходимо совпадение с 0 полей, поэтому возвращает все кортежи
tarantool> s:select{}
```

```
---
- - [1, 'A']
  - [1, 'B']
  - [1, 'C']
  - [2, 'D']
...
tarantool> -- первое поле должно быть больше 0,
tarantool> -- пропуск первого кортежа и возврат до
tarantool> -- 2 кортежей. Все параметры в данном примере
tarantool> -- зависят от характеристик индекса, поэтому см.
tarantool> -- более подробное описание в 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*.

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.

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.

тип возвращаемого значения кортеж

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_value*, *{{operator*, *field_no*, *value}*, *...}*`)`
Обновление или вставка кортежа.

Если существует кортеж, который совпадает с полями ключа `tuple_value`, 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_value`, запрос приведет к тому же результату, что и *space_object:insert()*, и используется параметр `{tuple_value}`. 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.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:

```
# проверка количества индексов для спейса 'tester'
tarantool> #box.space.tester.index
```

```
---
- 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 – это системный спейс, который содержит кортежи с функциями, созданными с помощью *box.schema.func.create()*.

  Кортежи в данном спейсе включают в себя следующие поля:

  - числовой идентификатор функции, число,
  - имя функции,
  - флаг,
  - название языка (необязательно): „LUA“ (по умолчанию) or „C“.

  Спейс _func не содержит саму функцию. Lua-функции создаются по-прежнему с помощью function имя_функции () ... end без каких-либо добавлений в спейс _func. Спейс _func предназначен лишь для хранения кортежей с функциями так, чтобы их имена могли использоваться в функциях *выдачи/отмены прав*.

  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»),

- тип объекта: „space“ (спейс), „function“ (функция), „sequence“ (последовательность) или „universe“ (вселенная),
- 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{}
---
- - ['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).

`box.space._space`
`_space` – это системный спейс.

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  '
...
```

Примеры:

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}
---
- - [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')
---
...
```

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()
---
- - _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
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'а* 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'а 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'а.

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`.

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*.

```
-- Настройка базы данных --
box.cfg{}
format = {}
format[1] = {'field1', 'unsigned'}
format[2] = {'field2', 'unsigned'}
format[3] = {'field3', 'unsigned'}
s = box.schema.create_space('test', {format = format})
-- Создание первичного индекса --
pk = s:create_index('pk', {parts = {{'field1'}}})
-- Создание уникального вторичного индекса --
sk_uniq = s:create_index('sk_uniq', {parts = {{'field2'}}})
-- Создание неуникального вторичного индекса --
sk_non_uniq = s:create_index('sk_non_uniq', {parts = {{'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]
...
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()
---
...
```

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 = {{1, 'unsigned'}, {2, 'unsigned'}}})
---
...
tarantool> s2:insert{1, 1}
---
- [1, 1]
...
tarantool> -- Удалить по компоненту ключа: ошибка --
tarantool> s2:delete{1}
---
- error: Invalid key part count in an exact match (expected 2, got 1)
...
tarantool> -- Удалить по ключу целиком: разрешено --
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}
---
- [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{}
---
- - [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}
---
- [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()`, чтобы просмотреть статистику сетевой активности: количество отправленных и полученных пакетов, а также общее количество запросов в секунду.

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*.

```
tarantool> box.stat()
---
- 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
  UPDATE:
    total: 0
    rps: 0
...
tarantool> box.stat().DELETE -- выбранный пункт таблицы
---
- total: 0
  rps: 0
...
tarantool> box.stat.net()
---
- SENT:
    total: 0
    rps: 0
  RECEIVED:
    total: 0
    rps: 0
...
tarantool> box.stat.vinyl().tx.commit -- выбранный пункт таблицы
---
- 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 па-

кеты 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*.

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.dump` содержит объем данных из последних изменений, для которых был создан дамп, а также счетчик дампов.

  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*.

- `box.stat.vinyl().disk.compact` содержит объем данных из последних изменений, для которых было произведено *слияние*. Он подразделяется на `disk.compact.in` (объем данных текущего слияния), `disk.compact.queue` (объем данных в ожидании слияния) и `disk.compact.out` (объем данных после слияния, который, предположительно, меньше `disk.compact.in`).
- `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 дампа `disk.dump`.

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 *page_size* плюс *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.

Функция *box.snapshot*

`box.snapshot()`

Создает снимок всех данных и сохраняет его в *memtx_dir*`/<latest-lsn>.snap`. Чтобы сделать снимок, сначала Tarantool входит в режим 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, замененные одновременно с ходом процесса.

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 создание ответвления, 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 на экземпляр. Хотя это может быть удобно, не рекомендуется использовать такой метод в автоматическом процессе: сигнал не дает возможность проверить, был ли корректно сделан снимок.

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:transform()	Удаление (и замена) полей кортежа
tuple_object:unpack()	Получение полей кортежа
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:pairs()	Подготовка к итерации
tuple_object:update()	Обновление кортежа

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*)

Если `t` – это экземпляр кортежа, то `t['field-name']` вернет поле под названием *field-name* в

кортеже. У полей есть имена, если кортеж был получен из спейса с определенным *форматом*.

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'}
---
...
tarantool> t:find('a')
---
- 1
...
tarantool> t:findall('a')
---
- 1
- 4
...
tarantool> t:findall(2, 'a')
---
- 4
...
```

`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')
---
- ['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

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: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:pairs()`

В языке Lua метод lua-table-value:pairs() возвращает: `функция`, `значение-Lua-таблицы`, `nil`. В Tarantool'е метод расширен так, что `tuple-value:pairs()` возвращает: `функция`, `значение-кортежа`, `nil`, – что используется для Lua-итераторов, поскольку они обходят компоненты значения до тех пор, пока не достигнут маркера.

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: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'}
---
...
tarantool> t:update({{'=', 2, 'B'}})
---
- ['Fld#1', 'B', 'Fld#3', 'Fld#4', 'Fld#5']
...
```

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
...
```

Вложенный модуль *box.error*

Общие сведения

Функция `box.error` предназначена для вызова ошибки. Разница между этой функцией и встроенной Lua-функцией error в том, что когда клиент получает ошибку, код ошибки сохраняется. В отличие от этого, ошибки в Lua всегда передаются на клиент в виде `ER_PROC_LUA`.

Индекс

Ниже приведен перечень всех функций модуля `box.error`.

Имя	Использование
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

- code (*integer*) –
- reason (string) –

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'e, 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

Например:

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
...
```

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 отправляются 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

Нельзя использовать запросы, которые могут приводить к изменению определения данных – создание, изменение, удаление, очистка.

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`
is_in_txn()	Обозначение наличия активной транзакции

`box.begin()`
Начало транзакции. Отключение *неявной передачи управления* до окончания транзакции. Сигнал о записи в :ref :*журнал упреждающей записи* <*internals-wal*> будет задержан до окончания транзакции. Фактически файбер, который выполняет функцию `box.begin()`, начинает «активную транзакцию со множеством запросов» с блокировкой всех остальных файберов.

возвращается ошибка, если такая операция не допускается, потому что уже есть активная транзакция.

возвращается ошибка, если по некоторой причине нельзя выделить память.

`box.commit()`
Окончание транзакции и применение результатов всех операций по изменению данных.

возвращается ошибка и прерывание транзакции в случае конфликта.

возвращается ошибка, если операция не может выполнить запись на диск.

возвращается ошибка, если по некоторой 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.

возвращается ошибка, если по некоторой 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.

возвращается ошибка, если точка сохранения отсутствует.

Пример:

```
function f()
  box.begin()           -- start transaction
  box.space.t:insert{1} -- this will not be rolled back
  local s = box.savepoint()
  box.space.t:insert{2} -- this will be rolled back
  box.rollback_to_savepoint(s)
  box.commit()          -- end transaction
end
```

`box.atomic(`*function-name*[, *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.

возвращается любая ошибка, которую могут вернуть *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`.

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).

4.1.2 Модуль *buffer*

Модуль `buffer` возвращает буфер, допускающий динамическое изменение размера, который используется только в качестве опции для методов *модуля net.box*.

Как правило, модуль `net.box` возвращает Lua-таблицу. Если используется опция `buffer`, то методы модуля `net.box` возвращают неформатированную строку MsgPack. Это экономит время работы на сервере, если в клиентском приложении есть собственная процедура декодирования MsgPack-строк.

buffer.ibuf()

возвращается дескриптор буфера.

тип возвращаемого значения cdata.

Пример:

Предположим, что Tarantool-сервер настроен на прослушивание на farhost:3301. Предположим, что на нем есть спейс `T` с одним кортежем: `'ABCDE', 12345`. В данном примере запустим сервер на localhost:3302, а затем используем процедуры `net.box` для подключения к farhost. Затем создадим буфер и используем его как опцию для вызова `conn.space...select()`. Результат получим в формате MsgPack. Чтобы показать это, используем *msgpack.decode_unchecked()* на `ibuf.rpos` («позиция для чтения» в буфере). Таким образом, мы проведем декодирование не на удаленном сервере, а на локальном.

```
box.cfg{listen=3302}
  buffer = require('buffer')
  ibuf = buffer.ibuf()
  net_box = require('net.box')
  conn = net_box.connect('farhost:3301')
  buffer = require('buffer')
  conn.space.T:select({},{buffer=ibuf})
  msgpack = require('msgpack')
  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'а в данном случае следует использовать функцию `msgpack.ibuf_decode(ibuf.rpos)`. Начиная с версии 1.7.7 Tarantool'а , `ibuf_decode` объявлена устаревшей.

4.1.3 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.

Индекс

Ниже приведен перечень всех функций модуля `clock`.

Имя	Использование
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.

возвращается 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.

возвращается 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.

возвращается 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.

возвращается 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.

возвращается **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)
    return 88
  end
  clock.bench(f, 10)
```

4.1.4 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*.

Индекс

Ниже приведен перечень всех функций модуля `console`.

Имя	Использование
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.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

возвращается 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`([*true*/*false*])
Установка флага автодополнения ввода. Если значение автодополнения = *true* (правда), и пользователь 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 многостроковых 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('')!
  ---
  ...
```

4.1.5 Модуль *crypto*

Общие сведения

«Crypto» is short for «Cryptography», which generally refers to the production of a digest value from a function (usually a Cryptographic hash function), applied against a string. Tarantool's `crypto` module supports ten types of cryptographic hash functions (AES, DES, DSS, MD4, MD5, MDC2, RIPEMD, SHA-1, SHA-2). Some of the crypto functionality is also present in the *Модуль digest* module.

Индекс

Ниже приведен перечень всех функций модуля `crypto`.

Имя	Использование
crypto.cipher.{algorithm}.{cipher_mode}.encrypt()	Шифрование строки
crypto.cipher.{algorithm}.{cipher_mode}.decrypt()	Расшифрование строки
crypto.digest.{algorithm}()	Получение дайджеста

`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*`)`

Pass or return a digest derived from the string. The eleven algorithm choices:

- 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')
```

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
  print(crypto.cipher.aes192.cbc.encrypt('AB', 'key'))
  c = crypto.cipher.aes192.cbc.encrypt.new()
  c:init()
  c:update('A', 'key')
  c:update('B', 'key')
  print(c:result())
  c:free()

  -- 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')
```

4.1.6 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.

Возможные параметры, передаваемые 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)

Индекс

Ниже приведен перечень всех функций модуля `csv`.

Имя	Использование
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*

возвращается 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()`

возвращается 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*

возвращается 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'а (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
...
```

4.1.7 Модуль *digest*

Общие сведения

A «digest» is a value which is returned by a function (usually a Cryptographic hash function), applied against a string. Tarantool's `digest` module supports several types of cryptographic hash functions (AES, MD4, MD5, SHA-1, SHA-2, PBKDF2) as well as a checksum function (CRC32), two functions for base64, and two non-cryptographic hash functions (guava, murmur). Some of the digest functionality is also present in the *crypto* module.

Индекс

Ниже приведен перечень всех функций модуля `digest`.

Имя	Использование
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.

`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 параметры:

- `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`.

Например:

```
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(`$\left[$*seed*$\right]$`)`
Запуск инкрементного вычисления с помощью MurmurHash. См. примечания по *инкрементным методам*.

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'
           >   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'
  ...
```

4.1.8 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*.

Индекс

Ниже приведен перечень всех функций модуля `errno`.

Имя	Использование
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 integer (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 string (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
    <...>
  ...
```

4.1.9 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()`.

Индекс

Ниже приведен перечень всех функций и элементов модуля `fiber`.

Имя	Использование
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

Continued on next page

Таблица 4.2 – continued from previous page

Имя	Использование
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.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()
          >   fiber.sleep(1000)
          > end
 ---
 ...
 tarantool> fiber_object = fiber.create(function_name)
 ---
 ...
```

`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()
         >   fiber.sleep(1000)
         > end
---
```

```
...
tarantool> fiber_object = fiber.new(function_name)
---
...
```

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.

Пример:

```
tarantool> fiber.sleep(1.5)
 ---
 ...
```

fiber.yield()

Передача управления планировщику. Работает аналогично *fiber.sleep(0)*, только *fiber.sleep(0)* зависит от таймера, *fiber.yield()* – нет.

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 string (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.

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 string (строка)

fiber.self():name() может также быть выражен как fiber.name().

Пример:

```
tarantool> fiber.self():name()
---
- interactive
...
```

fiber_object:name(*name*)

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*.

Параметры

- fiber_object – как правило, это объект, полученный в результате вызова *fiber.create*, *fiber.self* или *fiber.find*
- name (string) – новое имя файбера.

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 string (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:id()* вызовет ошибку с указанием недоступности файбера `error: the fiber is dead`.

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
---
- error: fiber is cancelled
...
```

`fiber_object.storage`
Локальное хранилище в пределах файбера. 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. Сборщик мусора в Lua отметит или освободит локальное хранилище при вызове *fiber_object*`:cancel()`.

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 true, если выполнено, false, если не выполнено

Тип возвращаемого значения boolean (логический)

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 num

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 num

Пример:

```
tarantool> fiber.time(), fiber.time64()
 ---
 - 1448466351.2708
 - 1448466351270762
 ...
```

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
 ---
 ...
```

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.

возвращается 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

возвращается 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

возвращается 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).

возвращается `true` (правда), если канал пуст. В противном случае, `false` (ложь).

тип возвращаемого значения boolean (логический)

`channel_object:count()`
Определение количества сообщений в канале.

возвращается количество сообщений.

тип возвращаемого значения число

`channel_object:is_full()`
Проверка заполненности канала.

возвращается `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()`.

возвращается `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()`.

возвращается `true` (правда), если на канале есть писатели в состоянии ожидании. В противном случае, `false` (ложь).

тип возвращаемого значения boolean (логический)

`channel_object:is_closed()`

возвращается `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
         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.

возвращается 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.

возвращается 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.

тип возвращаемого значения nil

`cond_object:broadcast()`
Пробуждение всех файберов, которые выполнили `wait()` для той же переменной.

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.

4.1.10 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).

Индекс

Ниже приведен перечень всех функций и элементов модуля `fio`.

Имя	Использование
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.copyfile()	Копирование файла
fio.chown() fio.chmod()	Управление правами на использование и правами владения объекта
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.

возвращается путь к файлу

тип возвращаемого значения string (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

возвращается имя файла

тип возвращаемого значения string (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

возвращается имя директории, то есть путь к файлу без имени файла.

тип возвращаемого значения string (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

возвращается имя директории, то есть путь к файлу с именем файла.

тип возвращаемого значения string (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.

`path.exists`(*path-name*)

Параметры

- `path-name` (`string`) – путь к директории или файлу.

возвращается 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)

`path.is_dir`(*path-name*)

Параметры

- `path-name` (`string`) – путь к директории или файлу.

возвращается true (правда), если путь к файлу ссылается на директорию; в противном случае, false (ложь)

тип возвращаемого значения boolean (логический)

`path.is_file`(*path-name*)

Параметры

- `path-name` (`string`) – путь к директории или файлу.

возвращается true (правда), если путь к файлу ссылается на файл; в противном случае, false (ложь)

тип возвращаемого значения boolean (логический)

`path.is_link`(*path-name*)

Параметры

- `path-name` (`string`) – путь к директории или файлу.

возвращается true (правда), если путь к файлу ссылается на символьную ссылку; в противном случае, false (ложь)

тип возвращаемого значения boolean (логический)

`path.lexists`(*path-name*)

Параметры

- `path-name` (`string`) – путь к директории или файлу.

возвращается 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.

возвращается 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.

возвращается (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.

возвращается (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.

возвращается (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 похож на результат команды `ls`.

Параметры

- `path-name` (`string`) – путь к директории.

возвращается (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.

возвращается 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 введения команды cp -r. Директория to-path должна быть пустой.

Параметры

- from-path (string) – путь.
- to-path (string) – путь.

возвращается (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 введения команды `mkdir`.

Параметры

- `path-name` (`string`) – путь.

возвращается (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 введения команды `rmdir`. Директория должна быть пустой.

Параметры

- `path-name` (`string`) – путь.

возвращается (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.

возвращается (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.

возвращается (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.copyfile`(*path-name*, *new-path-name*)

Копирование файла. Результат аналогичен результату введения команды `cp`.

Параметры

- `path-name` (`string`) – путь к первоначальному файлу.
- `new-path-name` (`string`) – путь к новому файлу.

возвращается (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

возвращается 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`*) –

возвращается (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`.

возвращается 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.

возвращается (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:

Использование `fio.open()` с `tonumber('N', 8)` для определения прав в виде восьмеричного числа:

```
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()`.

возвращается 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` – откуда читать (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 нулевое значение nil, 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 or buffer` (string) – записываемое значение
- `count` (*number*) – количество записываемых байтов (если формат – `pwrite(buffer, count, offset)`)
- `offset` (*number*) – смещение в файле – где начинается запись

возвращается 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*.)

```
ibuf = require('buffer').ibuf()
 ---
 ...

 tarantool> fh:pwrite(ibuf, 1, 0)
 ---
 - true
 ...
```

`file-handle:read(`[*count*]`)`
`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` и `fh:write` влияют на положение поиска по файлу, и это следует учитывать при работе нескольких файберов над одним файлом. Существует возможность ограничения или запрета на доступ к файлу с помощью `fiber.ipc`.

Параметры

- `fh` (*`userdata`*) – дескриптор файла, который возвращает `fio.open()`.
- `buffer` – откуда читать (если формат – `read(buffer, count)`)
- `count` (*`number`*) – количество байтов для чтения

Если формат – `read()` – без `count` – считываются все байты в файле.

Если формат – `read()` или `read([count])`, возвращается строка с данными, прочитанными из файла, либо нулевое значение nil, если не выполнено.

Если формат – `read(buffer, count)`, возвращаются данные в буфер. (Буферы можно ввести с помощью *buffer.ibuf*.)

```
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` и `fh:write` влияют на положение поиска по файлу, и это следует учитывать при работе нескольких файберов над одним файлом. Существует возможность ограничения или запрета на доступ к файлу с помощью `fiber.ipc`.

Параметры

- `fh` (*userdata*) – дескриптор файла, который возвращает `fio.open()`.
- `new-string or buffer` (string) – записываемое значение
- `count` (*number*) – количество записываемых байтов (если формат – `write(buffer, count)`)

возвращается 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
         ...
```

`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()`.

возвращается 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.

возвращается 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()`.

возвращается 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()`.

возвращается 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
     <...>
 ...
```

4.1.11 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'а. 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
---
...
```

4.1.12 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.

Индекс

Ниже приведен перечень всех функций модуля `http`.

Имя	Использование
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`) – максимальное количество записей в кэше соединения.

возвращается 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` - delay, in seconds, that the operating system will wait while the connection is idle before sending keepalive probes. See also CURLOPT_TCP_KEEPIDLE and the note below about keepalive_interval.
  - `keepalive_interval` - the interval, in seconds, that the operating system will wait between sending keepalive probes. See also CURLOPT_TCP_KEEPINTVL. If both keepalive_idle and keepalive_interval are set, then Tarantool will also set HTTP keepalive headers: Connection:Keep-Alive and Keep-Alive:timeout=<keepalive_idle>. Otherwise Tarantool will send 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“.

возвращается 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

Для запросов существуют следующие ускоренные методы:

- `http_client:get(url, options)` – ускоренный 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)` - shortcut for `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.

4.1.13 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.

Индекс

Ниже приведен перечень всех функций модуля `iconv`.

Имя	Использование
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.

возвращается 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»)

возвращается 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.

```
tarantool> string.hex('Д')
 ---
 - d094
 ...

 tarantool> converter = require('iconv').new('UTF16BE', 'UTF8')
 ---
 ...

 tarantool> utf16_string = converter('Д')
 ---
 ...
```

```
tarantool> string.hex(utf16_string)
---
- '0414'
...
```

4.1.14 Модуль *json*

Общие сведения

Модуль `json` обеспечивает процедуры работы с форматом JSON. Он основан на модуле Lua-CJSON от Mark Pulford. Полное руководство по Lua-CJSON включено в официальную документацию (the official documentation).

Индекс

Ниже приведен перечень всех функций и элементов модуля `json`.

Имя	Использование
json.encode()	Конвертация Lua-объекта в JSON-строку
json.decode()	Конвертация JSON-строки в Lua-объект
json.NULL	Аналог «nil» в языке Lua

`json.encode`(*lua-value*)

Конвертация Lua-объекта в JSON-строку.

Параметры

- `lua_value` – скалярное значение или значение из Lua-таблицы.

возвращается оригинальное значение, преобразованное в JSON-строку.

тип возвращаемого значения string (строка)

Пример:

```
tarantool> json=require('json')
       ---
       ...
       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*)

Конвертация JSON-строки в Lua-объект.

Параметры

- `string` (`string`) – строка в формате JSON.

возвращается 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*.

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
  - 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"}'
...
```

Структуру JSON-вывода можно указать с помощью `__serialize`:

- `__serialize="seq"` для массива
- `__serialize="map"` для ассоциативного массива

Сериализация „A“ и“ B“ различными значениями `__serialize` приводит к различным результатам:

```
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"})})
      ---
      - '[[]]'
      ...
```

Параметры конфигурации

Следующие параметры конфигурации определяют, как Tarantool кодирует недопустимые числа или типы. Значения параметров – логические `true`/`false` (правда/ложь).

- `cfg.encode_invalid_numbers` (по умолчанию, true) – разрешить nan и inf
- `cfg.encode_use_tostring` (по умолчанию, false) – использовать tostring для нераспознаваемых типов
- `cfg.encode_invalid_as_nil` (по умолчанию, false) – использовать null для всех нераспознаваемых типов
- `cfg.encode_load_metatables` (по умолчанию, false) – загрузить метатаблицы

Например, следующий код интерпретирует 0/0 (что не является числом) и 1/0 (что представляет собой бесконечность) в качестве специальных значений, а не как null или ошибку:

```
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]
 ...
```

Такие параметры конфигурации применяются для формата JSON, для *MsgPack* и для *YAML*.

4.1.15 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}
```

Индекс

Ниже приведен перечень всех функций и элементов модуля `log`.

Имя	Использование
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` (`string`) – Выходное значение будет представлять собой строку, которая содержит следующее: * текущая временная отметка, * название модуля, * „E“, „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*. 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.

возвращается nil

`log.logger_pid()`

возвращается PID регистратора записи в журнале

`log.rotate()`
Ротация журнала.

возвращается 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.

4.1.16 Модуль *msgpack*

Общие сведения

Модуль `msgpack` берет строки в формате MsgPack и декодирует их или берет ряд значений в ином формате и кодирует их в формат MsgPack. MsgPack интенсивно используется в Tarantool'е, поскольку кортежи *хранятся* в виде массивов в формате MsgPack.

Индекс

Ниже приведен перечень всех функций и элементов модуля `msgpack`.

Имя	Использование
msgpack.encode()	Конвертация Lua-объекта в MsgPack-строку
msgpack.decode()	Конвертация MsgPack-строки в Lua-объект
msgpack.decode_unchecked()	Конвертация MsgPack-строки в Lua-объект
msgpack.NULL	Аналог «nil» в языке Lua

`msgpack.encode(`*lua_value*`)`
Конвертация Lua-объекта в MsgPack-строку.

Параметры

- `lua_value` – скалярное значение или значение из Lua-таблицы.

возвращается оригинальное значение, преобразованное в MsgPack-строку.

тип возвращаемого значения string (строка)

`msgpack.decode(`*msgpack_string*`[`, *start_position*`])`
Конвертация MsgPack-строки в Lua-объект.

Параметры

- `msgpack_string` (`string`) – строка в формате MsgPack.
- `start_position` (*`integer`*) – откуда начинать, минимальное значение = 1, максимальное = длина строки, по умолчанию = 1.

возвращается

- (если `msgpack_string` в правильном MsgPack-формате) содержимое `msgpack_string` в формате Lua-таблицы, (в противном случае) скалярное значение, строка или число;
- «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 таблица и число

`msgpack.decode_unchecked(`*string*`)`
Конвертация MsgPack-строки в Lua-объект. Поскольку проверка не проводится, `decode_unchecked()` может работать с указателями строки в буфере в отличие от `decode()`. Пример см. в модуле *buffer*.

Параметры

- `string` – строка в формате MsgPack.

возвращается

- оригинальное содержание в формате Lua-таблицы;
- количество декодированных байтов.

тип возвращаемого значения Lua-объект.

`msgpack.NULL`
Значение, сопоставимое с нулевым значением «nil» в языке Lua, которое можно использовать в качестве объекта-заполнителя в кортеже.

Пример

```
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]
 ...
```

Структуру MsgPack-вывода можно указать с помощью `__serialize`:

- `__serialize = "seq"или "sequence"` для массива
- `__serialize = "map"или "mapping"` для ассоциативного массива

Сериализация „A“ и“ B“ различными значениями `__serialize` приводит к различным результатам. Чтобы показать различия, ниже приведена процедура кодирования *{„A“,“B“}* в массив и в ассоциативный массив с выводом каждого результата в виде шестнадцатеричного числа.

```
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
```

На странице спецификации MsgPack по ссылке Specification объясняется, что значение первого результата кодирования:

```
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 (ложь)	„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

Кроме того, некоторые параметры конфигурации MsgPack для кодировки можно изменить так же, как и для *JSON*.

4.1.17 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.

Индекс

Ниже приведен перечень всех функций модуля `net.box`.

Имя	Использование
net_box.connect() net_box.new()	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 более 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:call()	Вызов хранимой процедуры
conn:eval()	Оценка и выполнение выражения в строке
conn:timeout()	Установка времени ожидания

`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.

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.

Возможные опции

- *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).
- *connect_timeout*: количество секунд ожидания до возврата ошибки «error: Connection timed out».

Параметры

- `URI` (`string`) – *URI* объекта подключения
- `options` – возможные опции: *wait_connected*, *reconnect_after*, *call_16* и *console*

возвращается 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 conn`

`conn:ping()`

Выполнение команды проверки состояния PING.

возвращается true (правда), если выполнено, false (ложь) в случае ошибки

тип возвращаемого значения boolean (логический)

Пример:

```
net_box.self:ping()
```

conn:wait_connected([*timeout*])
Ожидание активности или закрытия подключения.

Параметры

- timeout (*number*) – в секундах

возвращается 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.

возвращается 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

возвращается 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{...}`.

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(...)`.

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(...)`.

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(...)`.

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(...)`.

`conn.space.<space-name>:delete({field-value, ...} [, {options}])`
`conn.space.`*имя-спейса*`:delete(...)` – это удаленный вызов, аналогичный локальному вызову `box.space.`*имя-спейса*`:delete(...)`.

`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.

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:

```
conn:call('function5')
 conn:call('fx',{1,'B'},{timeout=99})
```

`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')`.

Пример:

```
conn:eval('return 5+5')
 conn:eval('return ...', {1,2,3})
 conn:eval('return 5+5, {}, {timeout=0.1})
```

`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,
- `future:wait_result(timeout)` будет ждать, когда результат запроса будет доступен, а затем получит его.

- `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()*.

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.

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
   - '  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
...
```

4.1.18 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.

Индекс

Ниже приведен перечень всех функций модуля `os`.

Имя	Использование
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
 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> require('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
 ...
```

4.1.19 Модуль *pickle*

Индекс

Ниже приведен перечень всех функций модуля `pickle`.

Имя	Использование
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

возвращается бинарная строка, которая содержит все аргументы, упакованные в соответствии со спецификаторами формата.

тип возвращаемого значения string (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"]
 ...
 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`) –

возвращается 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'))
 ---
 ...
```

4.1.20 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. Имена и сигнатуры функций по большей части совместимы с luasocket.

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`.

Индекс

Ниже приведен перечень всех функций модуля `socket`.

Имя	Использование
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

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.

возвращается 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

возвращается подключенный сокет, если нет ошибки.

тип возвращаемого значения пользовательские данные

Пример:

```
socket.tcp_connect('127.0.0.1', 3301)
```

socket.getaddrinfo(*host*, *type*[, *{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*.

возвращается 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
 ...
```

socket.tcp_server(*host*, *port*, *handler-function*[, *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/table*) – что выполнить после подключения
- timeout (*number*) – количество секунд ожидания

Параметр handler-function может представлять собой имя функции (например, function_55), объявление функции (например, function () print('!') end) или таблицу с handler = функция (например, {handler=function_55, name='A'}).

Пример:

```
    socket.tcp_server('localhost', 3302, function () end)
```

```
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.

возвращается 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

возвращается 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 по ссылке *this description*.

`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*.

возвращается строка запрошенной длины, если выполнено.

тип возвращаемого значения string (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*]`)`

Read from a connected socket until some condition is true, and return the bytes that were read. Reading goes on until `limit` bytes have been read, or a delimiter has been read, or a timeout has expired. Unlike `socket_object:recv` (which uses an internal read-ahead buffer), `socket_object:read` depends on the socket's buffer.

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'}`.

возвращается 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 string (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»

возвращается пустая строка, если нет данных для чтения, либо нулевое значение nil в случае ошибки, либо строка, ограниченная количеством байтов в `size`.

тип возвращаемого значения string (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

возвращается 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`.

возвращается 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.

возвращается 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

возвращается 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*.

возвращается 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.

возвращается 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.

возвращается 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.

возвращается 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*) –

возвращается 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.

возвращается 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.

возвращается 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.

возвращается „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()`.

возвращается 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()`.

возвращается 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)`

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 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
...
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, function(s)
     while true do
       local request
       request = s:read("\n");
       if request == "" or request == nil then
         break
       end
       print(request)
     end
   end)
```

Вышеуказанный код означает: использовать *tcp_server()* для ожидания подключения с любого хоста по порту 3302. 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. Вызвать 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».

4.1.21 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
 ---
 ...
```

4.1.22 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.

Ниже приведен перечень всех функций библиотеки `string`.

Имя	Использование
string.ljust()	Выравнивание строки по левому полю
string.rjust()	Выравнивание строки по правому полю
string.hex()	Получение шестнадцатеричного значения строки
string.startswith()	Проверка, начинается ли строка с заданной подстроки
string.endswith()	Проверка, заканчивается ли строка на заданную подстроку
string.lstrip()	Удаление пробелов слева от строки
string.rstrip()	Удаление пробелов справа от строки
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` – (строка) строка для выравнивания по левому краю
- `width` – (целое число) ширина строки после выравнивания по левому краю
- `pad-character` – (строка) отдельный символ, по умолчанию = 1 пробел

Возвращается выровненная по левому краю строка (не изменяется, если ширина <= длине строки)

Тип возвращаемого значения string (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` – (строка) строка для выравнивания по правому краю
- `width` – (целое число) ширина строки после выравнивания по правому краю
- `pad-character` – (строка) отдельный символ, по умолчанию = 1 пробел

Возвращается выровненная по правому краю строка (не изменяется, если ширина <= длине строки)

Тип возвращаемого значения string (строка)

Пример:

```
tarantool> string = require('string')
 ---
 ...
 tarantool> string.rjust('', 5, 'X')
 ---
 - 'XXXXX'
 ...
```

`string.hex(`*input-string*`)`
Возврат шестнадцатеричного значения введенной строки.

Параметры

- `input-string` – (строка) обрабатываемая строка

Возвращается шестнадцатеричное число, два символа шестнадцатеричных цифр для каждого введенного символа

Тип возвращаемого значения string (строка)

Пример:

```
tarantool> string = require('string')
 ---
 ...
 tarantool> string.hex('ABC ')
 ---
 - '41424320'
 ...
```

`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` – (строка) строка, где производится поиск данных из `start-string`
- `start-string` – (строка) искомая строка
- `start-pos` – (целое число) положение: где начинать искать в пределах `input-string`
- `end-pos` – (целое число) положение: где заканчивать искать в пределах `input-string`

Возвращается true или false

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)
 ---
 - 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` – (строка) строка, где производится поиск данных из `end-string`

- `end-string` – (строка) искомая строка
- `start-pos` – (целое число) положение: где начинать искать в пределах `input-string`
- `end-pos` – (целое число) положение: где заканчивать искать в пределах `input-string`

Возвращается true или false

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.split(`*input-string*[, *split-string*]`)`

Разделение `input-string` на одну или более выводимых строк в таблице. Места разделения указаны в `split-string`.

Параметры

- `input-string` – (строка) строка для разделения
- `split-string` – (строка) искомая строка в пределах `input-string`. По умолчанию = пробел.

Возвращается таблица строк, которые были разделены из `input-string`

Тип возвращаемого значения таблица

Пример:

```
tarantool> fiber = require('string')
 ---
 ...
 tarantool> string.split("A*BXX C", "XX")
 ---
 - - A*B
   - ' C'
 ...
```

`string.lstrip(`*input-string*`)`

Возврат значения введенной строки без пробелов слева.

Параметры

- `input-string` – (строка) обрабатываемая строка

Возвращается результат после удаления пробелов из введенной строки

Тип возвращаемого значения string (строка)

Пример:

```
tarantool> string = require('string')
 ---
 ...
 tarantool> string.lstrip(' ABC ')
 ---
 - 'ABC '
 ...
```

`string.rstrip`(*input-string*)

Возврат значения введенной строки без пробелов справа.

Параметры

- `input-string` – (строка) обрабатываемая строка

Возвращается результат после удаления пробелов из введенной строки

Тип возвращаемого значения string (строка)

Пример:

```
tarantool> string = require('string')
 ---
 ...
 tarantool> string.rstrip(' ABC ')
 ---
 - ' ABC'
 ...
```

`string.strip`(*input-string*)

Возврат значения введенной строки без пробелов слева или справа.

Параметры

- `input-string` – (строка) обрабатываемая строка

Возвращается результат после удаления пробелов из введенной строки

Тип возвращаемого значения string (строка)

Пример:

```
tarantool> string = require('string')
 ---
 ...
 tarantool> string.strip(' ABC ')
 ---
 - ABC
 ...
```

4.1.23 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.

Выполнив команду «table», можно увидеть доступные функции:

```
tarantool> table
 ---
 - maxn: 'function: builtin#90'
```

```
  copy: 'function: 0x41e9d300'
  new: 'function: builtin#94'
  clear: 'function: builtin#95'
  move: 'function: 0x41e918e0'
  foreach: 'function: 0x41e91588'
  sort: 'function: builtin#93'
  remove: 'function: 0x41e917c8'
  foreachi: 'function: 0x41e914b8'
  deepcopy: 'function: 0x41e9d2e0'
  getn: 'function: 0x41e91620'
  concat: 'function: builtin#92'
  insert: 'function: builtin#91'
...
```

В данном разделе мы рассматриваем только дополнительную функцию, добавленную разработчиками Tarantool'a: `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) 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
...
```

4.1.24 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.

Индекс

Ниже приведен перечень всех функций модуля `tap`.

Имя	Использование
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

`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.

возвращается taptest

тип возвращаемого значения пользовательские данные

```
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.

возвращается taptest

тип возвращаемого значения userdata или строка

`taptest:plan(`*count*`)`
Указание количества проводимых тестов.

Параметры

- `count` (*`number`*) –

возвращается 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.

возвращается true (правда) или false (ложь).

тип возвращаемого значения boolean (логический)

`taptest:diag(`*message*`)`
Отображение сообщения диагностики.

Параметры

- `message` (`string`) – отображаемое сообщение.

возвращается 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

возвращается 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

возвращается 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

возвращается 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

возвращается 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

возвращается 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.

возвращается 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.

возвращается 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

возвращается 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

возвращается true (правда) или false (ложь).

тип возвращаемого значения boolean (логический)

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
```

4.1.25 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
           ---
           - build:
               target: Linux-x86_64-RelWithDebInfo
               options: cmake . -DCMAKE_INSTALL_PREFIX=/usr -DENABLE_BACKTRACE=ON
               mod_format: so
               flags: ' -fno-common -fno-omit-frame-pointer -fno-stack-protector -fexceptions
                 -funwind-tables -fopenmp -msse2 -std=c11 -Wall -Wextra -Wno-sign- compare -Wno-
↪strict-aliasing
                 -fno-gnu89-inline'
               compiler: /usr/bin/x86_64-linux-gnu-gcc /usr/bin/x86_64-linux-gnu-g++
             uptime: 'function: 0x408668e0'
             version: 1.7.0-66-g9093daa
             pid: 'function: 0x40866900'
           ...
           tarantool> tarantool.pid()
           ---
           - 30155
           ...
           tarantool> tarantool.uptime()
           ---
           - 108.64641499519
           ...
```

4.1.26 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.

Индекс

Ниже приведен перечень всех функций и элементов модуля `uuid`.

Имя	Использование
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()

возвращается UUID

тип возвращаемого значения cdata.

uuid.bin()

возвращается UUID

тип возвращаемого значения 16-байтная строка

uuid.str()

возвращается UUID

тип возвращаемого значения 36-байтная двоичная строка

uuid.fromstr(*uuid_str*)

Параметры

- uuid_str – UUID в 36-байтной шестнадцатеричной строке

возвращается конвертированный UUID

тип возвращаемого значения cdata.

uuid.frombin(*uuid_bin*)

Параметры

- uuid_str – UUID в 16-байтной двоичной строке

возвращается 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'.

возвращается UUID, сконвертированный из введенного значения формата cdata.

тип возвращаемого значения 16-байтная двоичная строка

uuid_object:str()

возвращается 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`.

возвращается true (правда), если значение состоит из одних нулей, в противном случае false (ложь).

тип возвращаемого значения логическое значение bool

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
 ...
```

4.1.27 Модуль *utf8*

Общие сведения

`utf8` is Tarantool's module for handling UTF-8 strings. It includes some functions which are compatible with ones in Lua 5.3 but Tarantool has much more. For example, because internally Tarantool contains a complete copy of the «International Components For Unicode» library, there are comparison functions which understand the default ordering for Cyrillic (Capital Letter Zhe Ж = Small Letter Zhe ж) and Japanese (Hiragana A = Katakana A).

Модуль является встроенным, поэтому нет необходимости выполнять команду `require('utf8')`.

Имя	Использование
casecmp and *cmp*	Сравнения
lower and *upper*	Замена регистра
isalpha, *isdigit*, *islower* and *isupper*	Определение типа символа
sub	Подстроки
length	Длина в символах
next	Посимвольная итерация

`utf8.casecmp(`*UTF8-string*, *utf8-string*`)`

Параметры

- `string` (*UTF8-string*) – строка в формате UTF-8

возвращается -1 означает «меньше», 0 означает «равно», +1 означает «больше»

тип возвращаемого значения число

Compare two strings with the Default Unicode Collation Element Table (DUCET) for the Unicode Collation Algorithm. Thus „å“ is less than „B“, even though the code-point value of å (229) is greater than the code-point value of B (66), because the algorithm depends on the values in the Collation Element Table, not the code-point values.

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

возвращается строка в UTF-8

тип возвращаемого значения string (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

возвращается -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.

The comparison is done with at least three weights. Therefore the elements which affect secondary or later weights (such as «case» in Latin or Cyrillic alphabets, or «kana differentiation» in Japanese) are not ignored. and upper case comes after lower case.

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

возвращается true или false

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

возвращается true или false

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

возвращается true или false

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

возвращается true или false

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.length(`*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

возвращается 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.

If the string contains a byte sequence which is not valid in UTF-8, each byte in the invalid byte sequence will be counted as one character.

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

возвращается та же строка в нижнем регистре

тип возвращаемого значения string (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

возвращается 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

возвращается строка в формате UTF-8, «подстрока» введенного значения

тип возвращаемого значения string (строка)

Позиции символа в начале и в конце могут быть отрицательными, что указывает на отсчет с конца строки, а не с начала.

The default value for end-character is the length of the input string. Therefore, saying `utf8.sub(1, 'abc')` will return „abc“, the same as the input string.

Пример:

```
tarantool> utf8.sub('åγжabcdefg', 5, 8)
---
- abcd
...
```

utf8.upper(*UTF8-string*)

Параметры

- string (*UTF8-string*) – строка в формате UTF-8

возвращается та же строка в верхнем регистре

тип возвращаемого значения string (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
...
```

4.1.28 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'а URI включает в себя процедуры для разложения URI-строк на компоненты или объединения компонентов в URI-строку.

Индекс

Ниже приведен перечень всех функций модуля `uri`.

Имя	Использование
uri.parse()	Получение таблицы URI-компонентов
uri.format()	Создание URI из компонентов

`uri.parse(`*URI-string*`)`

Параметры

- `URI-string` – Унифицированный идентификатор ресурса

возвращается 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')
 ---
 - 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.

возвращается URI-строка. Таким образом, uri.format() – это операция, обратная uri.parse().

тип возвращаемого значения string (строка)

Пример:

```
tarantool> uri.format({host = 'x.html', scheme = 'http', fragment = 'y'})
 ---
 - http://x.html#y
 ...
```

4.1.29 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}}
```

4.1.30 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.

Индекс

Ниже приведен перечень всех функций и элементов модуля `yaml`.

Имя	Использование
yaml.encode()	Конвертация Lua-объекта в YAML-строку
yaml.decode()	Конвертация YAML-строки в Lua-объект
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.

возвращается оригинальное значение, преобразованное в YAML-строку.

тип возвращаемого значения string (строка)

`yaml.decode(`*string*`)`
Конвертация YAML-строки в Lua-объект.

Параметры

- `string` – строка в формате YAML.

возвращается оригинальное содержание в формате Lua-таблицы.

тип возвращаемого значения таблица

`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
  ...
...
tarantool> yaml.encode(setmetatable({'A', 'B'}, { __serialize="seq"}))
---
- |
  ---
  ['A', 'B']
  ...
...
tarantool> yaml.encode({setmetatable({f1 = 'A', f2 = 'B'}, { __serialize="map"})})
---
- |
  ---
  - {'f2': 'B', 'f1': 'A'}
  ...
...
tarantool> yaml.encode({setmetatable({f1 = 'A', f2 = 'B'}, { __serialize="mapping"})})
---
- |
  ---
  - f2: B
    f1: A
  ...
...
```

Кроме того, некоторые параметры конфигурации YAML для кодировки можно изменить так же, как и для *JSON*.

4.1.31 Разное

Индекс

Ниже приведен перечень разных доступных функций.

Имя	Использование
tonumber64()	Конвертация строки или Lua-числа в 64-битное целое число
dostring()	Анализ и выполнение произвольного Lua-кода

`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')
 ---
 ...
 tarantool> type(i), i / 2, i - 2, i * 2, i + 2, i % 2, i ^ 2
 ---
 - number
 - 500000000
 - 999999998
 - 2000000000
 - 1000000002
 - 0
 - 1000000000000000000
 ...
```

`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.

возвращается 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
         >       return nil
         >     end
         >   end
         >   return f(...)]], 1)
---
- null
...
```

4.1.32 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.

4.1.33 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.

4.1.34 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.

Индекс

Ниже приведен перечень всех функций библиотеки `debug`.

Имя	Использование
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.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

возвращается среда объекта `object`

`debug.gethook(`[*thread*]`)`

возвращается 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

возвращается 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

возвращается 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

возвращается метатаблица объекта `object` или `nil`, если метатаблица отсутствует

debug.getregistry()

возвращается таблица реестра

debug.getupvalue(*func*, *up*)

Параметры

- func (*function*) – функция, для которой будет получено сопоставляющее значение
- up (*number*) – индекс сопоставляющего значения функции

возвращается 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

возвращается 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

возвращается 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

возвращается имя сопоставляющего значения или `nil`, если сопоставляющее значение с данным индексом отсутствует

`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)

возвращается строка с обратной трассировкой стека вызовов

4.2 Справочник по сторонним библиотекам

В данном справочнике описаны сторонние Lua-модули для Tarantool'а.

4.2.1 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'a 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'a rocks.tarantool.org. Затем выполните:

```
luarocks install mysql [MYSQL_LIBDIR = path]
                       [MYSQL_INCDIR = path]
                       [--local]
```

Например:

```
$ 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,
     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
 $ # Проверьте, что создана поддиректория include, путем поиска
 $ # .../include/mysql.h. (Если нет, то можно проверить
 $ # .../include/mysql/mysql.h.)
 $ [ -f $TMDIR/include/mysql.h ] && echo "OK" || echo "Error"
 OK

 $ # Проверьте, что создана поддиректория library, а в ней
 $ # необходимый файл .so.
 $ [ -f $TMDIR/lib/libmysqlclient.so ] && echo "OK" || echo "Error"
 OK

 $ # Проверьте, что mysql-клиент может подключиться, с помощью настроек
 $ # по умолчанию: порт = 3306, пользователь = 'root', пароль пользователя = '',
 $ # база данных = 'test'. Эти настройки можно изменить, используя
 $ # измененные значения.
 $ $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 ...

 $ # Вставьте строку в базу данных test и завершите работу.
 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

 $ # Установите luarocks
 $ sudo apt-get -y install luarocks | grep -E "Setting up|already"
 Setting up luarocks (2.0.8-2) ...

 $ # Настройте список сторонних модулей Tarantool'а в ~/.luarocks,
 $ # следуя инструкциям по ссылке rocks.tarantool.org
 $ mkdir ~/.luarocks
 $ echo "rocks_servers = {[[http://rocks.tarantool.org/]]}" >> \
     ~/.luarocks/config.lua

 $ # Убедитесь, что при следующей установке будут использованы файлы из главного
 $ # хранилища Tarantool'а. Получаем результат, нормальный для Ubuntu
 $ # 12.04 Precise Pangolin
 $ cat /etc/apt/sources.list.d/tarantool.list
 deb http://tarantool.org/dist/1.7/ubuntu/ precise main
 deb-src http://tarantool.org/dist/1.7/ubuntu/ precise main

 $ # Установите tarantool-dev. Строка на экране должна показать версию 1.6
 $ sudo apt-get -y install tarantool-dev | grep -E "Setting up|already"
 Setting up tarantool-dev (1.6.6.222.g48b98bb~precise-1) ...
```

```
$

$ # Используйте luarocks для локальной установки, то есть в $HOME
$ luarocks install mysql MYSQL_LIBDIR=/usr/local/mysql/lib --local
Installing http://rocks.tarantool.org/mysql-scm-1.rockspec...
... (здесь будет еще информация о сборке драйвера Tarantool/MySQL)
mysql scm-1 is now built and installed in ~/.luarocks/

$ # Убедитесь, что driver.so создан в месте,
$ # где Tarantool будет искать его
$ find ~/.luarocks -name "driver.so"
~/.luarocks/lib/lua/5.1/mysql/driver.so

$ # Измените директорию на директорию, которую можно использовать для
$ # временного тестирования. В данном примере предполагаем, что имя
$ # этой директории будет /home/pgulutzan/tarantool_sandbox.
$ # (Измените "/home/pgulutzan" на фактическую корневую директорию
$ # пользователя машины, используемой для тестирования.)
$ cd /home/pgulutzan/tarantool_sandbox

$ # Запустите экземпляр Tarantool-сервера. Не используйте файл инициализации Lua.

$ tarantool
tarantool: version 1.7.0-222-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'a 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'a rocks.tarantool.org. Затем выполните:

```
luarocks install pg [POSTGRESQL_LIBDIR = path]
                    [POSTGRESQL_INCDIR = path]
                    [--local]
```

Например:

```
$ 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 знаки вопроса («?») 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.

```
$ # Проверьте, что создана поддиректория include, путем поиска
 $ # /usr/include/postgresql/libpq-fe-h.
 $ [ -f /usr/include/postgresql/libpq-fe.h ] && echo "OK" || echo "Error"
 OK

 $ # Проверьте, что создана поддиректория library, а в ней необходимый файл .so.
 $ [ -f /usr/lib/x86_64-linux-gnu/libpq.so ] && echo "OK" || echo "Error"
 OK

 $ # Проверьте, что psql-клиент может подключиться, с помощью настроек по умолчанию:
 $ # порт = 5432, пользователь = 'postgres', пароль пользователя = 'postgres',
 $ # база данных = 'postgres'. Эти настройки можно изменить, используя
 $ # измененные значения. Вставьте строку в базу данных postgres и завершите работу.
 $ 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
 postgres=# INSERT INTO test VALUES (1,'PostgreSQL row');
 INSERT 0 1
 postgres=# \q
 $

 $ # Установите luarocks
 $ sudo apt-get -y install luarocks | grep -E "Setting up|already"
 Setting up luarocks (2.0.8-2) ...

 $ # Настройте список сторонних модулей Tarantool'а в ~/.luarocks,
 $ # следуя инструкциям по ссылке rocks.tarantool.org
 $ mkdir ~/.luarocks
 $ echo "rocks_servers = {[[http://rocks.tarantool.org/]]}" >> \
        ~/.luarocks/config.lua

 $ # Убедитесь, что при следующей установке будут использованы файлы из главного
 $ # хранилища Tarantool'а. Получаем результат, нормальный для Ubuntu 12.04 Precise Pangolin
 $ cat /etc/apt/sources.list.d/tarantool.list
 deb http://tarantool.org/dist/1.7/ubuntu/ precise main
 deb-src http://tarantool.org/dist/1.7/ubuntu/ precise main

 $ # Установите tarantool-dev. Строка на экране должна показать версию 1.7
 $ sudo apt-get -y install tarantool-dev | grep -E "Setting up|already"
 Setting up tarantool-dev (1.7.0.222.g48b98bb~precise-1) ...
 $

 $ # Используйте luarocks для локальной установки, то есть в $HOME
```

```
$ luarocks install pg POSTGRESQL_LIBDIR=/usr/lib/x86_64-linux-gnu --local
Installing http://rocks.tarantool.org/pg-scm-1.rockspec...
... (здесь будет еще информация о сборке драйвера Tarantool/PostgreSQL)
pg scm-1 is now built and installed in ~/.luarocks/

$ # Убедитесь, что driver.so создан в месте,
$ # где Tarantool будет искать его
$ find ~/.luarocks -name "driver.so"
~/.luarocks/lib/lua/5.1/pg/driver.so

$ # Измените директорию на директорию, которую можно использовать для
$ # временного тестирования. В данном примере предполагаем, что имя
$ # папе этой директории будет $HOME/tarantool_sandbox.
$ # (Измените "$HOME" на фактическую корневую директорию
$ # машины, используемой для тестирования.)
cd $HOME/tarantool_sandbox

$ # Запустите экземпляр Tarantool-сервера. Не используйте файл инициализации Lua.

$ tarantool
tarantool: version 1.7.0-412-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{}
 ...
 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.

4.2.2 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
...
    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)
   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, `expiration` 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 `expiration.run_task`:

- `force` (логическое значение) – выполнение задачи даже на реплике. По умолчанию: `force=false`, поэтому, как правило, `expiration` 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.)

4.2.3 Модуль *shard*

Во время шардинга кортежи из набора кортежей распределяются по нескольким узлам, на каждом из которых есть экземпляр сервера базы данных Tarantool'а. При таком распределении каждый экземпляр обрабатывает только подмножество общих данных, поэтому появляется возможность обрабатывать данные при больших нагрузках путем простого добавления большего количества компьютеров в сеть.

Модуль Tarantool'а *shard* позволяет создавать шарды, а также аналоги функций по управлению данными из библиотеки box (select, insert, replace, update, delete).

Для начала введем терминологию:

Консистентное хеширование Модуль *shard* распределяет данные в соответствии с алгоритмом хеширования, то есть применяет хеш-функцию к значению первичного ключа кортежа, что определить к какому шарду относится кортеж. Хеш-функция является консистентной, поэтому изменение количества серверов не повлияет на результат для множества ключей. Модуль *shard* использует специальную хеш-функцию *digest.guava* из модуля digest.

Экземпляр Запущенная in-memory копия Tarantool-сервера иногда называется экземпляром сервера. Как правило, каждый шард ассоциирован с одним экземпляром, или же, если выполняется и шардинг, и репликация, каждый шард ассоциирован с одним набором реплик.

Очередь Временный список последних запросов обновления. Иногда называется «пакетная обработка». Поскольку обновления в базу данных с шардингом могут быть замедлены, ускорить выполнение можно путем отправки запросов в очередь вместо ожидания окончания обновления на каждом узле. В модуле *shard* присутствуют функции для добавления запросов в очередь, которые будут затем обработаны без дополнительных действий. Использование очереди необязательно.

Резервирование по принципу избыточности Количество копий реплицируемых данных в каждом шарде.

Реплика Экземпляр, который входит в набор реплик.

Набор реплик Часто отдельный шард ассоциирован с отдельным экземпляром. Однако, часто шард реплицируется. Когда шард реплицируется, множество экземпляров («реплики»), которые обрабатывают реплицируемые данные шарда, составляют «набор реплик».

Реплицируемые данные Полная копия данных. Модуль *shard* обрабатывает как шардинг, так и репликацию. Один шард может содержать одну или несколько копий реплицируемых данных. Попытки записи производятся по очереди на каждую копию реплицируемых данных. Модуль *shard* не использует встроенную функцию репликации.

Шард Подмножество кортежей в базе данных, разделенное по значению, которое возвращает консистентная хеш-функция. Как правило, каждый шард находится на отдельном узле или отдельном наборе узлов (например, если резервирование = 3, то шард будет на трех узлах).

Зона Физическое местоположение, где узлы тесно связаны, с одинаковыми точками безопасности, резервного копирования и доступа. Простейшим примером зоны является один компьютер с одним экземпляром Tarantool-сервера. Копии реплицируемых данных на шарде должны находиться в разных зонах.

Пакет *shard* распространяется отдельно от основного пакета *Tarantool*. Для работы с ним выполните установку отдельно:

- либо на версии Tarantool'a 1.7.4+ выполните команду:

```
$ tarantoolctl rocks install shard
```

- либо установите с помощью *yum* или *apt*, например, на Ubuntu выполните команду:

```
$ sudo apt-get install tarantool-shard
```

- либо скачайте из GitHub *tarantool/shard* и используйте Lua-файлы, как описано в файле README.

Затем перед использованием модуля выполните команду `shard = require('shard')`.

Самой необходимой функцией модуля является

```
shard.init(*настройка-шарда*)
```

Ее следует вызывать для каждого шарда.

Настройка шарда представляет собой таблицу со следующими полями:

- *servers* – серверы, т.е. список URI узлов и зон, в которых находятся узлы
- *login* – имя пользователя, которое используется для доступа по модулю *shard*
- *password* – пароль для имени пользователя
- *redundancy* – резервирование, число, минимум 1
- *binary* – номер порта, на котором настроено прослушивание для текущего хоста (отличный от порта „listen“, который определяет *box.cfg*)

Возможные ошибки:

- значение параметра redundancy (резервирование) не должно быть больше количества серверов;
- серверы должны быть рабочими;
- две копии реплицируемых данных одного шарда не должны находиться в одной зоне.

Пример: синтаксис *shard.init* для одного шарда

- Количество копий реплицируемых данных на один шард (redundancy – резервирование) равно 3.
- Количество экземпляров равно 3.
- Модуль *shard* делает вывод, что существует только один шард.

```
tarantool> cfg = {
         >   servers = {
         >     { uri = 'localhost:33131', zone = '1' },
         >     { uri = 'localhost:33132', zone = '2' },
         >     { uri = 'localhost:33133', zone = '3' }
         >   },
         >   login = 'test_user',
         >   password = 'pass',
         >   redundancy = '3',
         >   binary = 33131,
         > }
---
...
tarantool> shard.init(cfg)
---
...
```

Пример: синтаксис *shard.init* для трех шардов

Здесь описаны три шарда. Каждый шард содержит две копии реплицируемых данных. Поскольку количество серверов равно 7, количество копий реплицируемых данных на один шард равно 2, а деление 7 на 2 дает в остатке 1, – один из серверов не будет использоваться. Это необязательно должно быть ошибкой, поскольку один из серверов может быть нерабочим.

```
tarantool> cfg = {
         >   servers = {
         >     { uri = 'host1:33131', zone = '1' },
         >     { uri = 'host2:33131', zone = '2' },
         >     { uri = 'host3:33131', zone = '3' },
         >     { uri = 'host4:33131', zone = '4' },
         >     { uri = 'host5:33131', zone = '5' },
         >     { uri = 'host6:33131', zone = '6' },
         >     { uri = 'host7:33131', zone = '7' }
         >   },
         >   login = 'test_user',
         >   password = 'pass',
         >   redundancy = '2',
         >   binary = 33131,
         > }
---
...
tarantool> shard.init(cfg)
---
...
```

Каждой функции взаимодействия с данными модуля *box* соответствует функция в модуле *shard*:

```
shard[*имя-спейса*].insert{...}
shard[*имя-спейса*].replace{...}
```

```
shard[*имя-спейса*].delete{...}
shard[*имя-спейса*].select{...}
shard[*имя-спейса*].update{...}
shard[*имя-спейса*].auto_increment{...}
```

Например, чтобы выполнить вставку в таблицу T в базе данных с шардингом, просто выполните команду `shard.T:insert{...}` вместо `box.space.T:insert{...}`.

Запрос `shard.T:select{}` без первичного ключа вызовет ошибку.

Каждой функции модуля *box* для взаимодействия с данными, поставленной в очередь, соответствует функция в модуле *shard*:

```
shard[*имя-спейса*].q_insert{...}
shard[*имя-спейса*].q_replace{...}
shard[*имя-спейса*].q_delete{...}
shard[*имя-спейса*].q_select{...}
shard[*имя-спейса*].q_update{...}
shard[*имя-спейса*].q_auto_increment{...}
```

Пользователь должен добавить *operation_id*. Чтобы получить дополнительную информацию о функциях для взаимодействия с данными, поставленными в очередь, и о функциях, предназначенных для обслуживания, см. файл README.

Пример: шард, минимальная настройка

Создан только один шард, который содержит только одну копию реплицируемых данных. Таким образом, данный пример не иллюстрирует возможности репликации или шардинга, он показывает синтаксис и отображаемые сообщения, что каждый может повторить за пару минут лишь с помощью вырезания и вставки.

```
$ mkdir ~/tarantool_sandbox_1
$ cd ~/tarantool_sandbox_1
$ rm -r *.snap
$ rm -r *.xlog
$ ~/tarantool-1.7/src/tarantool

tarantool> box.cfg{listen = 3301}
tarantool> box.schema.space.create('tester')
tarantool> box.space.tester:create_index('primary', {})
tarantool> box.schema.user.create('test_user', {password = 'pass'})
tarantool> box.schema.user.grant('test_user', 'read,write,execute', 'universe')
tarantool> cfg = {
         >   servers = {
         >       { uri = 'localhost:3301', zone = '1' },
         >   },
         >   login = 'test_user';
         >   password = 'pass';
         >   redundancy = 1;
         >   binary = 3301;
         > }
tarantool> shard = require('shard')
tarantool> shard.init(cfg)
tarantool> -- Now put something in ...
tarantool> shard.tester:insert{1,'Tuple #1'}
```

Если вырезать и вставить вышеуказанное, то результат с запросами и ответами только для *shard.init* и *shard.tester* должен выглядеть примерно так:

```
<...>
tarantool> shard.init(cfg)
2017-09-06 ... I> Sharding initialization started...
2017-09-06 ... I> establishing connection to cluster servers...
2017-09-06 ... I> connected to all servers
2017-09-06 ... I> started
2017-09-06 ... I> redundancy = 1
2017-09-06 ... I> Adding localhost:3301 to shard 1
2017-09-06 ... I> shards = 1
2017-09-06 ... I> Done
---
- true
...
tarantool> -- Введите что-то...
---
...
tarantool> shard.tester:insert{1,'Tuple #1'}
---
- - [1, 'Tuple #1']
...
```

Пример: шард, горизонтальное масштабирование

Созданы два шарда, каждый из которых содержит одну копию реплицируемых данных. В реальной жизни два узла будут представлены двумя компьютерами, для примера же требуется использовать две оболочки, которые мы назовем «терминал №1» и «терминал №2».

В первом терминале введите:

```
$ mkdir ~/tarantool_sandbox_1
$ cd ~/tarantool_sandbox_1
$ rm -r *.snap
$ rm -r *.xlog
$ ~/tarantool-1.7/src/tarantool

tarantool> box.cfg{listen = 3301}
tarantool> box.schema.space.create('tester')
tarantool> box.space.tester:create_index('primary', {})
tarantool> box.schema.user.create('test_user', {password = 'pass'})
tarantool> box.schema.user.grant('test_user', 'read,write,execute', 'universe')
tarantool> console = require('console')
tarantool> cfg = {
         >   servers = {
         >     { uri = 'localhost:3301', zone = '1' },
         >     { uri = 'localhost:3302', zone = '2' },
         >   },
         >   login = 'test_user',
         >   password = 'pass',
         >   redundancy = 1,
         >   binary = 3301,
         > }
tarantool> shard = require('shard')
tarantool> shard.init(cfg)
tarantool> -- Введите что-нибудь ...
```

```
tarantool> shard.tester:insert{1,'Tuple #1'}
```

Во втором терминале введите:

```
$ mkdir ~/tarantool_sandbox_2
$ cd ~/tarantool_sandbox_2
$ rm -r *.snap
$ rm -r *.xlog
$ ~/tarantool-1.7/src/tarantool

tarantool> box.cfg{listen = 3302}
tarantool> box.schema.space.create('tester')
tarantool> box.space.tester:create_index('primary', {})
tarantool> box.schema.user.create('test_user', {password = 'pass'})
tarantool> box.schema.user.grant('test_user', 'read,write,execute', 'universe')
tarantool> console = require('console')
tarantool> cfg = {
         >   servers = {
         >     { uri = 'localhost:3301', zone = '1' };
         >     { uri = 'localhost:3302', zone = '2' };
         >   };
         >   login = 'test_user';
         >   password = 'pass';
         >   redundancy = 1;
         >   binary = 3302;
         > }
tarantool> shard = require('shard')
tarantool> shard.init(cfg)
tarantool> -- Выведите что-нибудь ...
tarantool> shard.tester:select{1}
```

На терминале №1 появится цикл сообщений с ошибками типа «Connection refused» (в подключении отказано) и «server check failure» (отказ проверки сервера). Это нормально. Сообщения будут появляться, пока не начнется процесс на терминале №2.

В конце, на терминале №2 появится примерно следующее:

```
tarantool> shard.tester:select{1}
---
- - - [1, 'Tuple #1']
...
```

Данный пример показывает, что введенная на терминале №1 информация может быть извлечена на терминале №2 с помощью модуля *shard*.

Для получения подробной информации см. файл README.

4.2.4 Модуль *vshard*

Обзор

В модуле `vshard` реализована новая функция шардинга (сегментирования), которая позволяет осуществлять горизонтальное масштабирование в Tarantool'е.

С ростом проекта масштабируемость баз данных становится проблемой. Если отдельный сервер не может справиться с нагрузкой, необходимо применять средства масштабирования.

Есть два различных подхода к масштабированию данных: вертикальное и горизонтальное масштабирование:

- *Вертикальное масштабирование* подразумевает увеличение производительности системы отдельного сервера.
- *Горизонтальное масштабирование* подразумевает секционирование набора данных и распределение данных по множеству серверов. При добавлении новых серверов набор данных повторно равномерно распределяется по всем серверам, новым и старым.

Шардинг, или сегментирование, представляет собой архитектуру базы данных, которая предоставляет возможность горизонтального масштабирования.

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. Секционирование набора данных осуществляется с помощью сегментных ключей.

Модуль `vshard` основан на концепции виртуальных сегментов: набор кортежей распределяется на большое количество абстрактных виртуальных узлов (виртуальных сегментов, или сегментов), а не на малое количество физических узлов.

Хеширование сегментного ключа в большое количество виртуальных сегментов позволяет незаметно для пользователя изменять количество серверов в кластере. Механизм балансирования распределяет сегменты между шардами, если некоторые серверы добавляются или убираются.

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 функции, аналогичные функциям по управлению данными библиотеки Tarantool'а `box` (select, insert, replace, update, delete).

Установка

Пакет `vshard` распространяется отдельно от основного пакета Tarantool'а. Для работы с ним выполните установку отдельно:

```
$ tarantoolctl rocks install vshard
```

Примечание: Для работы с модулем `vshard` необходима версия Tarantool'а 1.9+.

Краткое руководство

В директории `vshard/example/` находится предварительно настроенный кластер из 1 `роутера` и 2 наборов реплик из 2 узлов (2 `хранилища`) в каждом, что составляет всего 5 экземпляров Tarantool'а в целом:

- `router_1` – экземпляр `роутера`
- `storage_1_a` – экземпляр `хранилища`, мастер первого набора реплик
- `storage_1_b` – экземпляр `хранилища`, реплика из первого набора реплик
- `storage_2_a` – экземпляр `хранилища`, мастер второго набора реплик

- `storage_2_b` – экземпляр `хранилища`, реплика из второго набора реплик

Управление всеми экземплярами осуществляется с помощью утилиты `tarantoolctl` из корневой директории проекта.

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
<...>
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)
- `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 API `router`:

```
unix/:./data/router_1.control> 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: []
...
```

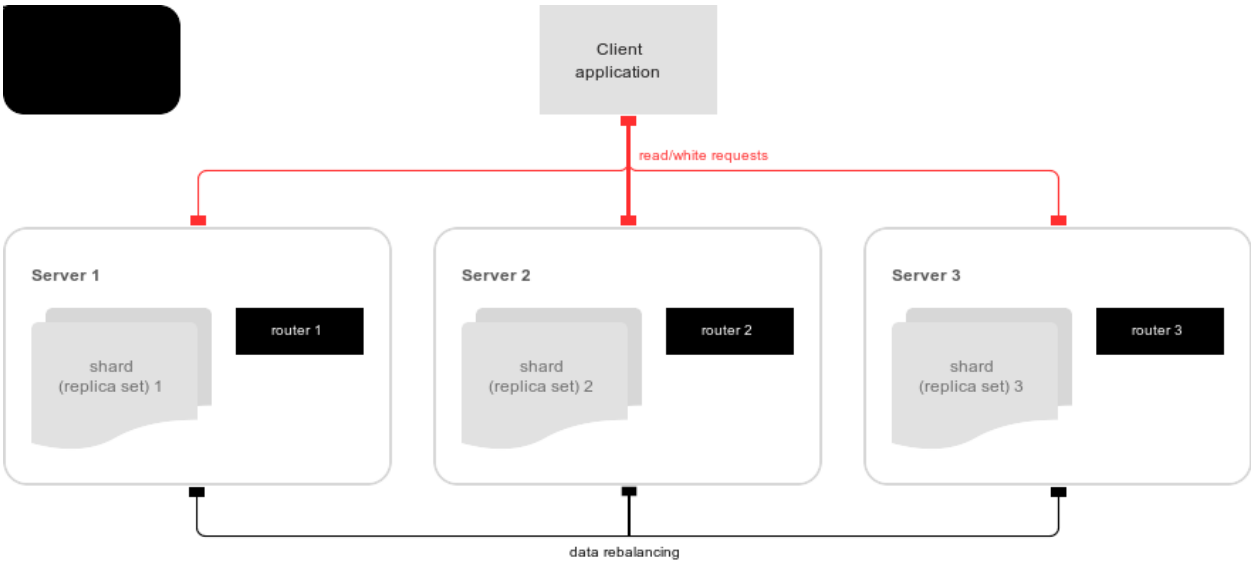
Архитектура

Сегментированный кластер в Tarantool'е состоит из хранилищ, роутеров и балансировщика.

Хранилище (storage) – это узел, который хранит подмножество набора данных. Развертывание нескольких реплицируемых хранилищ осуществляется в виде наборов реплик, чтобы обеспечить резерв (набор реплик также можно называть шардом или сегментом).

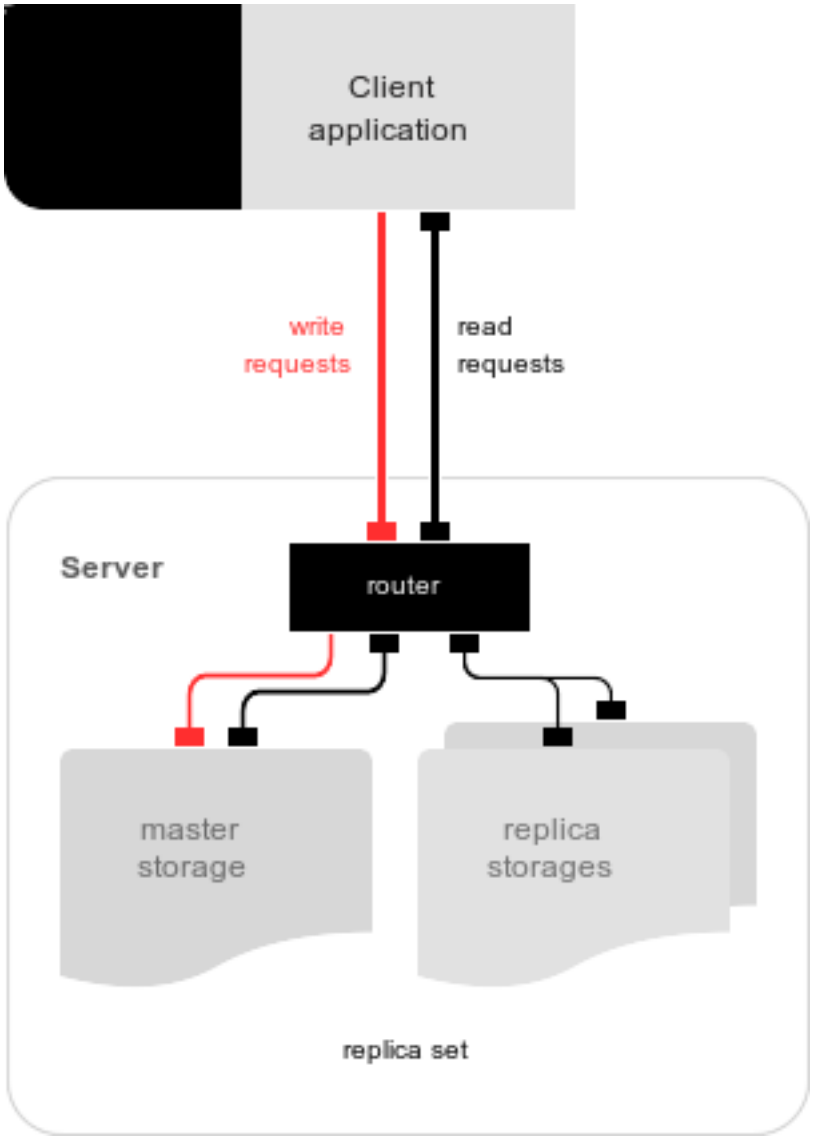
Роутер (router) – это автономный компонент ПО, который обеспечивает маршрутизацию запросов чтения и записи от клиентского приложения к шардам.

Балансировщик (rebalancer) – это внутренний компонент, который равномерно распределяет набор данных между всеми шардами в случае добавления или удаления серверов. Он также занимается выравниванием нагрузки с учетом производительности существующих наборов реплик.



Хранилище

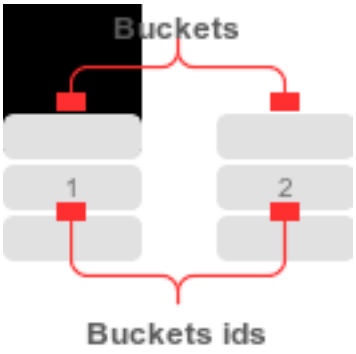
Хранилище (storage) – это узел, который хранит подмножество набора данных. Несколько реплицируемых хранилищ составляют набор реплик. У каждого хранилища в наборе реплик есть роль: **мастер** или **реплика**. Мастер обрабатывает запросы на чтение и запись. Реплики обрабатывают запросы на чтение, но не могут обрабатывать запросы на запись.



Виртуальные сегменты

Набор данных при шардинге распределяется на большое количество абстрактных узлов, которые называются **виртуальные сегменты** (далее по тексту **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. Одна сегмент не может относиться к нескольким наборам реплик одновременно.

Общее количество сегментов определяет администратор, который настраивает первоначальную конфигурацию кластера.

Каждый спейс Tarantool'а, который планируется сегментировать, должен включать в себя проиндексированное поле с идентификатором сегмента. Спейсы без индексов идентификаторов сегментов не участвуют в шардинге, но могут использоваться в качестве обычных спейсов. По умолчанию, имя индекса совпадает с идентификатором сегмента.

Миграция сегментов

Балансировщик представляет собой фоновый процесс балансировки, который обеспечивает равномерное распределение сегментов по шардам. Во время балансировки происходит миграция сегментов по наборам реплик.

Набор реплик, из которого переносится сегмент, называется «исходный» (source); а набор реплик, куда переносится сегмент, называется «целевой» (destination).

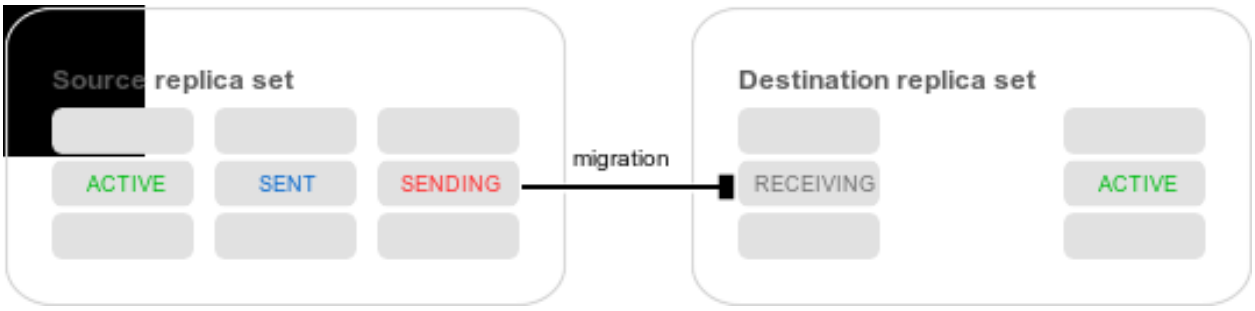
Блокировка набора реплик позволяет набору реплик оставаться невидимым для балансировщика. Набор реплик с блокировкой не может ни принимать новые сегменты, ни мигрировать свои собственные.

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) – произошла миграция сегмента в целевой набор реплик. *Роутер* использует такой статус, чтобы рассчитать новое местоположение сегмента. Отправленный сегмент в статусе SENT автоматически переходит в статус мусора GARBAGE через 0,5 секунды после миграции (это время задается в параметре BUCKET_SENT_GARBAGE_DELAY).

- 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.



В целом, 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 помечается отправленным в статусе SENT и перестает принимать запросы.
4. Сегмент в целевом наборе реплик переходит в активный статус ACTIVE и начинает принимать все запросы.

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`.

Роутер

Все запросы из приложения приходят в сегментированный кластер через `роутер`. `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.

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.

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 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. Операция выполняется с помощью функции vshard.router.call():

```
result = vshard.router.call(<идентификатор_сегмента>, <режим(read:write)>, <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).

Администрирование

Конфигурация сегментированного кластера

Минимальный рабочий сегментированный кластер должен состоять 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` и `хранилища` 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 на производстве.

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. Подразумевается, что само приложение отвечает за взаимодействие с такой системой и передачу параметров шардинга.

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',
                    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` и два `хранилища`. Каждое `хранилище` 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 `vshard/example`.

Вес реплики

`Роутер` отправляет все запросы только на мастер-экземпляр. Задав вес реплики, можно разрешить

отправку запросов 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 буквально ближайшую реплику.

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 в 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, -- ...
    },
    [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
        [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. Если во всех наборах реплик это число остается неизменным, то сегменты распределяются равномерно.

Эталонное число рассчитывается автоматически с учетом количества сегментов в кластере и веса наборов реплик.

Например: Пользователь указал количество сегментов = 3000, а вес 3 наборов реплик равен 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.

`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
```

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.

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 равноценно блокированию 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.

Появление блокировки набора реплик и закрепления всех сегментов обусловлено необходимостью возможной изоляции 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, взгляните на 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 учета блокировок и закрепления совершенно разные, хотя с точки зрения функциональности 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 на набор 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 этом шаге 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. Описанная 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 =
                                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).

   Если в сегменте есть ссылки типа RW, его нельзя перемещать. Однако, если балансировщику требуется отправка этого сегмента, он блокирует его для новых запросов на запись, ожидает завершения всех текущих запросов, а затем отправляет сегмент.

   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 будет закреплен в хранилище до перезапуска экземпляра.

Чтобы узнать количество ссылок в сегменте, используйте *vshard.storage.buckets_info([идентификатор_сегмента])* (параметр `идентификатор_сегмента` необязателен).

Например:

```
vshard.storage.buckets_info(1)
---
- 1:
    status: active
    ref_rw: 1
    ref_ro: 1
    ro_lock: true
    rw_lock: true
    id: 1
```

Определение спейса

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)
```

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 вмешательства человека.

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. раздел *Процесс балансировки*.

Сборщик мусора

Файбер **сборки мусора** работает в фоновом режим 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 для 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.

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*

`sharding`
Поле, которое определяет логическую топологию сегментированного кластера Tarantool'а.

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`
Индекс по идентификатору сегмента.

Тип: непустая строка или неотрицательное целое число
По умолчанию: совпадает с числом идентификатора 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. В свою очередь, недостаточное число сегментов может привести к снижению степени детализации при балансировке.

Тип: число
По умолчанию: 3000
Динамический: нет

`collect_bucket_garbage_interval`
Интервал между действия 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`
Максимальная 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

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:

Справочник по API

Общедоступные API роутера

- *vshard.router.bootstrap()*
- *vshard.router.cfg(cfg)*
- *vshard.router.new(name, cfg)*
- *vshard.router.call(bucket_id, mode(read:write), 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.route(bucket_id)*
- *vshard.router.routeall()*
- *vshard.router.bucket_id(key)*
- *vshard.router.bucket_count()*
- *vshard.router.sync(timeout)*
- *vshard.router.discovery_wakeup()*
- *vshard.router.info()*
- *vshard.router.buckets_info()*
- *replicaset.call()*
- *replicaset.callro()*
- *replicaset.callrw()*

`vshard.router.bootstrap()`
Выполнение первоначальной настройки кластера и распределение всех сегментов по наборам реплик.

`vshard.router.cfg(`*cfg*`)`
Настройка базы данных и начало шардинга указанного `роутера`. См. *образец конфигурации* выше.

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 при вызове перед его методами необходимо указывать двоеточие, а (`vshard.router:метод(...)`), а перед методами статичного роутера – точку (`vshard.router.метод(...)`).

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. Используется в качестве префикса в журналах роутера, должно быть уникальным в пределах экземпляра
- `cfg` – конфигурационная таблица. *Образец конфигурации* описан выше.

Возвращается экземпляр роутера, если он создан; в противном случае, nil и ошибка

`vshard.router.call(`*bucket_id*, *mode(read:write)*, *function_name*, *{argument_list}*, *{options}*`)`
Вызов пользовательской функции на шарде, где хранится сегмент с указанным идентификатором. Для получения подробной информации о работе функции см. раздел *Обработка запросов*.

Параметры

- `bucket_id` – идентификатор сегмента
- `mode` – тип функции: чтение или запись
- `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.

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 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.

Пример:

Для вызова функции `customer_add` из `vshard/example` выполните команду:

```
result = vshard.router.call(100, 'write', 'customer_add', {{customer_id = 2, bucket_id = 100,␣
↪name = 'name2', accounts = {}}}, {timeout = 100})
```

`vshard.router.callro(`*bucket_id*, *function_name*, *{argument_list}*, *{options}*`)`

Вызов пользовательской функции на шарде, где хранится сегмент с указанным идентификатором, в режиме только чтения. 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.

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 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.

`vshard.router.callrw(`*bucket_id*, *function_name*, *{argument_list}*, *{options}*`)`

Вызов пользовательской функции на шарде, где хранится сегмент с указанным идентификатором, в режиме записи. 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.

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 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.

`vshard.router.route(`*bucket_id*`)`
Возврат объекта набора реплик для сегмента с указанным идентификатором.

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 объект набора реплик

Пример:

```
replicaset = vshard.router.routeall()
```

`vshard.router.bucket_id(`*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:

```
bucket_id = vshard.router.bucket_id(18374927634039)
```

`vshard.router.bucket_count()`
Возврат общего количества сегментов, указанных в *vshard.router.cfg()*'.

Возвращается общее количество сегментов

Тип возвращаемого значения число

`vshard.router.sync(`*timeout*`)`
Ожидание синхронизации набора данных на репликах.

Параметры

- `timeout` – время ожидания в секундах

`vshard.router.discovery_wakeup()`
Принудительный запуск файбера обнаружения сегментов.

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` – the number of buckets which replica sets are not known to the `router`

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}

replicaset.call(*replicaset*, *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.call аналогичен replicaset.callrw.

Параметры

- replicaset – UUID набора реплик
- 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.

replicaset.callrw(*replicaset*, *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.callrw аналогичен replicaset.call.

Параметры

- replicaset – UUID набора реплик
- 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.

`replicaset.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.callro()` рекомендуется вызывать исключительно функции, доступные только для чтения. поскольку такие функции можно выполнять не только на мастере, но и на репликах.

Параметры

- `replicaset` – UUID набора реплик
- `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.

Внутренние API роутера

- *vshard.router.bucket_discovery(bucket_id)*

`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

Общедоступные API хранилища

- *vshard.storage.cfg(cfg, name)*
- *vshard.storage.info()*
- *vshard.storage.call(bucket_id, mode(read:write), 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()*
- *vshard.storage.buckets_info()*
- *vshard.storage.buckets_count()*
- *vshard.storage.sharded_spaces()*

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
        uuid: 810d85ef-4ce4-4066-9896-3c352fec9e64
...
```

vshard.storage.call(*bucket_id*, *mode(read:write)*, *function_name*, *{argument_list}*)
Вызов пользовательской функции на текущем экземпляре «хранилища».

Параметры

- `bucket_id` – идентификатор сегмента
- `mode` – тип функции: 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

`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

возвращается `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

возвращается `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/RW.

Параметры

- `bucket_id` – идентификатор сегмента
- `mode` – чтение или запись

возвращается `true` (правда), если выполнение 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` – чтение или запись

возвращается `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 сегмента в статусе мусора.

Параметры

- `bucket_index` – индекс спейса с частью идентификатора спейса
- `control` – контроллер сборщика мусора. Если увеличивается масштаб создания сегментов, поиск следует прервать.

возвращается 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:

```
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()
Флаг, указывающий на ход процесса балансировки. Его значение будет true (правда), если в настоящий момент узел применять маршруты, полученные от узла балансировки.

vshard.storage.is_locked()
Флаг, указывающий на блокировку балансировщика.

vshard.storage.rebalancer_disable()
Отключение балансировки. Отключенный балансировщик находится в режиме ожидания до повторного запуска.

vshard.storage.rebalancer_enable()
Запуск балансировки.

vshard.storage.sharded_spaces()
Отображение спейсов, которые доступны балансировщику и сборщику мусора.

Внутренние API хранилища

- *vshard.storage.bucket_ stat(bucket_ id)*
- *vshard.storage.bucket_ recv(bucket_ id, from, data)*
- *vshard.storage.bucket_ delete_ garbage(bucket_ id)*
- *vshard.storage.bucket_ collect(bucket_ id)*
- *vshard.storage.bucket_ force_ create(first_ bucket_ id, count)*

- *vshard.storage.bucket_force_drop(bucket_id, to)*
- *vshard.storage.bucket_send(bucket_id, to)*
- *vshard.storage.buckets_discovery()*
- *vshard.storage.rebalancer_request_state()*

vshard.storage.bucket_recv(*bucket_id*, *from*, *data*)
Получение идентификатора сегмента из удаленного набора реплик.

Параметры

- `bucket_id` – идентификатор сегмента
- `from` – UUID исходного набора реплик
- `data` – данные, хранящиеся логически в идентификаторе сегмента 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:

```
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 идентификатора сегмента, 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 идентификаторе сегмента:

```
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 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*`)`
Перемещение идентификатора 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.

возвращается количество сегментов в активном статусе, если таковые обнаружены; в противном случае, nil

`vshard.storage.buckets_discovery()`
Сбор массива идентификаторов активных сегментов для обнаружения.

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.

Горизонтальное масштабирование 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 Сегментный ключ, который принадлежность сегмента к определенному набору реплик.

Роутер Прокси-сервер, который отвечает за запросы маршрутизации от приложения к узлам в кластере.

4.2.5 Модуль *tdb*

Отладчик Tarantool'а (сокращенно `tdb`) можно использовать с любой Lua-программой. Рабочие функции: определение точек прерывания, исследование переменных, перебор строк по одной, обратная трассировка и отображение информации о файберах. Функции вывода: использование различных цветов в разных ситуациях, включая номера строк, и добавление подсказок.

Модуль не поставляется в репозитории Tarantool'а, его следует устанавливать отдельно. Это обычно делается следующим образом:

```
$ git clone --recursive https://github.com/Sulverus/tdb
$ cd tdb
$ make
$ sudo make install prefix=/usr/share/tarantool/
```

Чтобы запустить `tdb` в рамках Lua-программы и определить точку прерывания, включите в программу следующие строки:

```
tdb = require('tdb')
tdb.start()
```

Чтобы начать сессию отладки, выполните Lua-программу: выполнение остановится на точке прерывания, и можно будет вводить команды отладчика.

Команды отладчика

bt Обратная трассировка – отображение стека (красным) с именами программы/функции и номерами строк, выполнение которых привело к текущей строке.

c Продолжение выполнения до следующей точки прерывания или до окончания программы.

e Вход в режим вычисления. Когда программа находится в режиме вычисления, можно выполнять определенные запросы, которые будут действовать с точки зрения контекста. Это особенно полезно для отображения значений переменных программы. Другие команды отладчика не будут работать, пока не выйти из режима оценки, набрав: -e.

-e Выход из режима оценка.

f Отображение идентификатора файбера, имени программы и процентного соотношения использованной памяти в виде таблицы.

n Переход на следующую строку с пропуском любых вызовов функций.

globals Отображение имен переменных или функций, которые являются глобальными.

h Отображение списка команд отладчика.

locals Отображение имен и значений переменных, например, переменных для управления Lua-оператором «for».

q Немедленный выход.

Пример сессии

Сохраните следующую программу в директории по умолчанию и назовите ее «example.lua»:

```
tdb = require('tdb')
tdb.start()
i = 1
j = 'a' .. i
print('end of program')
```

Запустите Tarantool, используя example.lua в качестве файла инициализации.

```
$ tarantool example.lua
```

Теперь вывод на экране выглядит следующим образом:

```
$ tarantool example.lua
(TDB)  Tarantool debugger v.0.0.3. Type h for help
example.lua
(TDB)  [example.lua]
(TDB)  3: i = 1
(TDB)>
```

Запросы отладчика выделены синим, подсказки и информация – зеленым, а текущая строка – 3 строка программы example.lua – цветом, который используется по умолчанию. Введите 6 команд отладчика:

```
n  -- переход на следующую строку
n  -- переход на следующую строку
e  -- вход в режим оценки
j  -- отображение j
-e -- выход из режима оценки
q  -- выход
```

Теперь вывод на экране выглядит следующим образом:

```
$ tarantool example.lua
(TDB)  Tarantool debugger v.0.0.3. Type h for help
example.lua
(TDB)  [example.lua]
(TDB)  3: i = 1
(TDB)> n
(TDB)  4: j = 'a' .. i
(TDB)> n
(TDB)  5: print('end of program')
(TDB)> e
(TDB)  Eval mode ON
(TDB)> j
j       a1
(TDB)> -e
(TDB)  Eval mode OFF
(TDB)> q
```

Другой пример работы отладчика можно найти здесь.

4.3 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 ]
```

4.3.1 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 асинхронной сетевой интегрированной среды программирования многократного использования.

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.

4.3.2 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*.

4.3.3 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"),
   memtx_memory        = 100000,
   pid_file            = "tarantool.pid",
   rows_per_wal        = 50
}
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()*.

4.3.4 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*

`background`
: Запуск сервера в виде фоновой задачи. Параметры *log* и *pid_file* должны быть не равны нулю, что это сработало.

  Тип: логический
  По умолчанию: false (ложь)
  Динамический: нет

`custom_proc_title`
: 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`

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`

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`

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`
: 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 (временный)	нет	да	нет	нет
is_local	нет	да	нет	да

`vinyl_dir`
: 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`
: 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`
: Имя пользователя в UNIX, на которое переключается система после запуска.

Тип: строка

По умолчанию: null

Динамический: нет

`wal_dir`

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`

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`

Максимальное количество потоков, используемых во время исполнения определенных внутренних процессов (сейчас *socket.getaddrinfo()* и *coio_call()*).

Тип: целое число

По умолчанию: 4

Динамический: да

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`
: Количество памяти, которое Tarantool выделяет для фактического хранения кортежей, в байтах. 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

  Динамический: **да**, но нельзя уменьшить

`memtx_max_tuple_size`
: Размер наибольшего блока выделения памяти в байтах 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

  Динамический: нет

`memtx_min_tuple_size`
: Размер наименьшего блока выделения памяти в байтах. Его можно уменьшить, если кортежи очень малого размера. Значение должно быть от 8 до 1 048 280 включительно.

  Тип: целое число

  По умолчанию: 16

  Динамический: нет

`vinyl_bloom_fpr`
: 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`
Размер кэша для движка базы данных vinyl в байтах. Размер кэша можно изменить динамически.

Тип: целое число
По умолчанию = 128 * 1024 * 1024 = 134217728
Динамический: **да**

`vinyl_max_tuple_size`
Размер наибольшего блока выделения памяти в байтах 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
Динамический: нет

`vinyl_memory`
Максимальное количество байтов оперативной памяти, которые использует vinyl.

Тип: целое число
По умолчанию = 128 * 1024 * 1024 = 134217728
Динамический: **да**, но нельзя уменьшить

`vinyl_page_size`
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 :ref:‘Параметры space_object:create_index() ‘.

Тип: целое число
По умолчанию = 8 * 1024 = 8192
Динамический: нет

`vinyl_range_size`
The default maximum range size for a vinyl index, in bytes. The maximum range size affects the decision whether to split a range.

If `vinyl_range_size` is not nil and not 0, then it is used as the default value for the `range_size` option in the *Options for space_object:create_index()* chart.

If `vinyl_range_size` is nil or 0, and `range_size` is not specified when the index is created, then Tarantool sets a value later depending on performance considerations. To see the actual value, use *index_object:stat().range_size*.

In Tarantool versions prior to 1.10.2, `vinyl_range_size` default value was 1073741824.

Тип: целое число
Default = nil
Динамический: нет

`vinyl_run_count_per_level`
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 :ref:‘Параметры space_object:create_index() ‘.

Тип: целое число
По умолчанию = 2
Динамический: нет

`vinyl_run_size_ratio`
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 :ref:‘Параметры space_object:create_index() ‘.

Тип: число с плавающей запятой
По умолчанию = 3.5
Динамический: нет

`vinyl_read_threads`
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`
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*

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 на 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`
: 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.

  Например:

```
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`
: 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. После этого самый старый снимок (а также любые связанные с ним WAL-файлы) перед созданием 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

Записи в бинарный журнал и создание снимков

- *force_recovery*,
- *rows_per_wal*,
- *wal_max_size*,
- *snap_io_rate_limit*,
- *wal_mode*,
- *wal_dir_rescan_delay*

`force_recovery`
: 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

`rows_per_wal`
: Количество записей журнала, которое хранится в отдельном WAL-файле. При достижении предельного значения Tarantool создает другой WAL-файл под названием *<первый-lsn-в-журнале>*.`xlog`. Эту функцию можно использовать для простого резервного копирования на основе rsync.

  Тип: целое число

  По умолчанию: 500000

  Динамический: нет

`wal_max_size`
: 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)

  Динамический: нет

`snap_io_rate_limit`
: 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`
: 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: **да**

`wal_dir_rescan_delay`
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`
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_timeout*
- *replication_connect_timeout*
- *replication_connect_quorum*
- *replication_skip_conflict*
- *replication_sync_lag*
- *replication_sync_timeout*
- *replicaset_uuid*
- *instance_uuid*

`replication`

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_timeout`

Реплика отправляет сообщения контрольного сигнала на мастер каждую секунду, и мастер запрограммирован на автоматическое переподключение, если он не получает сообщения контрольного сигнала дольше количества секунд, указанного в `replication_timeout`.

См. дополнительную информацию в разделе *Мониторинг набора реплик*.

Тип: целое число

По умолчанию: 1

Динамический: **да**

`replication_connect_timeout`

Количество секунд, в течение которых реплика ожидает попытки подключения к мастеру в кластере. Для получения подробной информации, см. *статус orphan*.

Данный параметр отличается от *replication_timeout*, который используется только для автоматического переподключения репликации, когда отсутствуют сообщения контрольного сигнала.

Тип: число с плавающей запятой

По умолчанию: 4

Динамический: **да**

`replication_connect_quorum`

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`

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.

Пример:

```
box.cfg{replication_skip_conflict=true}
```

Тип: логический

По умолчанию: false (ложь)

Динамический: **да**

`replication_sync_lag`

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`

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**

`replicaset_uuid`

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 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`
: 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 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`
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`
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`

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'}
-- or
box.cfg{log = 'syslog:facility=user'}
-- or
box.cfg{log = 'syslog:identity=tarantool,facility=user'}
-- or
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`

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.

Тип: логический

По умолчанию: true

Динамический: нет

`too_long_threshold`

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`

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.

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'а.

```
ps -A | grep tarantool
```

На терминале №2 используйте команду `kill -HUP` для отправки сигнала SIGHUP на экземпляр Tarantool'а. 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`

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`

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`
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*

`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. Нижнее значение 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.

4.4 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*.

4.4.1 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`.

4.4.2 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`.

`tarantoolctl stop NAME` Остановка экземпляра Tarantool'а.

`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` Вход в интерактивную Lua-консоль экземпляра.

`tarantoolctl eval NAME FILE` Оценка 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.

4.4.3 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] [--re`
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.

4.4.4 Команды для управления модулями Tarantool'a

`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.

4.5 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.

4.5.1 Способы ссылки на объект

Во-первых, есть пять *способов 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
            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 пяти 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`*`:`).

4.5.2 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, ка 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.

4.5.3 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 5

Практикум

5.1 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*.

5.1.1 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_funciton()
         >   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
         >   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
         >   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.

5.1.2 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
4    sum = 0
5    for v, t in box.space.tester:pairs() do
6      is_valid_json, lua_table = pcall(json.decode, t[2])
7      if is_valid_json then
8        field_value = lua_table[field_name]
9        if type(field_value) == "number" then sum = sum + field_value end
10     end
11   end
12   return sum
13 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.

5.1.3 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")
    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
    -- 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. Та-

ким 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.")
```

Мы получим следующий результат:

```
tarantool> indexed_pattern_search("t", 2, "ABC.E.")
---
- - [7, 'ABCDEF', 'a']
...
```

5.2 Практическое задание на C

Ниже приводится практическое занятие на языке C: *Хранимые процедуры на языке C*.

5.2.1 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 1.10
- 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
```

Файл `msgpuck.h` есть в системе, если Tarantool был установлен из исходных файлов. В противном случае, следует установить пакет «msgpuck» по ссылке https://github.com/rtsisyk/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`.

На данный момент функция `harder()` начнет использовать функции, определенные в msgpuck.h. Процедуры, которые начинаются с «mp» – это функции msgpuck, которые обрабатывают данные в формате MsgPack. Передача и возврат всегда осуществляются в этом формате, поэтому следует ознакомиться с msgpuck для того, чтобы овладеть навыками работы с API для языка C.

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;
  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 тапл. 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 тапла + 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.

5.3 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'а. 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'а (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::Slave::binlog_pos_t sBinlogPos = sl->getLastBinlog();
    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;
    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.setMasterLogNamePos("mysql-bin", 0);
        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 6

Примечания к версиям

Примечания к версиям содержат краткое описание значимых изменений в следующих версиях Tarantool'a: *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 вышущенных стабильных релизах (milestone = closed) на GitHub.

6.1 Версия 1.10

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) допустимость *резервного копирования любой контрольной точки*, 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.
- (Репликации) 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.

6.2 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.

6.3 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 Tarantoool 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 количестве 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-filep $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.

6.4 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 7

Руководство разработчика

7.1 Справочник по C API

7.1.1 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

7.1.2 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)

7.1.3 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 ошибочного ввода-вывода (eio) с указанной функцией и аргументами. Передает управление и ожидает окончания задачи или истечения времени ожидания. 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);
         return open(filename, flags);
 }

 if (coio_call(openfile_cb, 0.10, "/tmp/file", 0) == -1)
     // обработка ошибок.
 ...
```

int `coio_getaddrinfo`(const char **host*, const char **port*, const struct addrinfo **hints*, struct addrinfo ***res*, double *timeout*)

Версия *getaddrinfo(3)* для файбера.

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)*

7.1.4 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.

7.1.5 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()*.

7.1.6 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()*

7.1.7 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

7.1.8 Модуль *lua/utils*

void *`luaL_pushcdata`(struct lua_State *L*, uint32_t *ctypeid*)
Принудительная передача 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.

7.1.9 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);
```

7.1.10 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_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.

7.1.11 Модуль *trivia/config*

API_EXPORT
: Внешний модификатор для всех доступных функций.

`PACKAGE_VERSION_MAJOR`
: Мажорная версия пакета – 1 в 1.9.2.

`PACKAGE_VERSION_MINOR`
: Минорная версия пакета – 9 в 1.9.2.

`PACKAGE_VERSION_PATCH`
: Патч-версия пакета – 2 в 1.9.2.

`PACKAGE_VERSION`
: Строка с идентификатором версии: мажорная-минорная-патч-коммит-идентификатор, например, 1.9.2-0-g113ade24e.

`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'a.

`TARANTOOL_CXX_FLAGS`
: Флаги компиляции CXX, используемые для сборки Tarantool'a.

`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`.

7.1.12 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*. Возвращаемый 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*)

7.1.13 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

7.2 Внутреннее устройство

7.2.1 Бинарный протокол Tarantool'a

Бинарный протокол Tarantool'a представляет собой бинарный запросно-ответный протокол.

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
 +--------------------------------------+
 |                                      |
 | 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.

Протокол использует ассоциативные массивы, которые содержат несколько целочисленных постоянных, в качестве ключей. Эти постоянные указаны по ссылке src/box/iproto_constants.h. Ниже приведены часто используемые постоянные:

```
-- пользовательские ключи
<iproto_sync>          ::= 0x01
<iproto_schema_id>     ::= 0x05  /* также 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 /* также expression */
<iproto_ops>           ::= 0x28
<iproto_data>          ::= 0x30
<iproto_error>         ::= 0x31
```

```
-- -- Значение ключа <code> в запросе может быть следующим:
-- Ключи для команд пользователя
<iproto_select>       ::= 0x01
<iproto_insert>       ::= 0x02
<iproto_replace>      ::= 0x03
<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
-- Коды для команд администратора
-- (включая коды для инициализации набора реплик и выбора мастера)
<iproto_ping>         ::= 0x40
<iproto_join>         ::= 0x41 /* i.e. replication join */
<iproto_subscribe>    ::= 0x42
<iproto_request_vote> ::= 0x43

-- -- Значение для ключа <code> в ответе может быть следующим:
<iproto_ok>           ::= 0x00
<iproto_type_error>   ::= 0x8XXX /* где XXX -- это значение в 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
```

Они различаются лишь набором допустимых ключей и значений. Ключ определяет тип следующего за ним значения. Если в теле сообщения нет ключей, может отсутствовать весь ассоциативный массив в формате msgpack для тела сообщения. Так и случится при запросе проверки связи <ping>. `schema_id` может отсутствовать в заголовке запроса, что означает отсутствие проверки версии, но этот ключ обязательно должен присутствовать в ответе. Если `schema_id` отправляется в заголовке, будет выполнена соответствующая проверка.

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
```

```
    * Вставка      OP = '!'
      вставить <argument> до поля <field_no>
    * Присвоение      OP = '='
      присвоить <argument> полю <field_no>.
      увеличит кортеж, если <field_no> == <max_field_no> + 1

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:

+======================+==================+
|                      |                  |
|   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:

+======================+==================+
|                      |                  |
|   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.

Удобные макросы для определения шестнадцатеричных постоянных для возвращаемых кодов можно найти по ссылке src/box/errcode.h

Структура пакета при репликации

```
-- ключи для репликации
 <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

 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 предыдущем разделе.

7.2.2 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 ди-

ректорию *wal_dir*. Новый WAL-файл создается для количества записей, определенного в параметре *rows_per_wal*, или для количества байтов, указанного в *rows_per_wal* . 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`.

Помимо номера записи в журнале (LSN) и запроса на изменение данных (в формате *бинарного протокола Tarantool'a*), каждая запись в WAL-файле содержит заголовок, некоторые метаданные, а также данные, форматированные по правилам msgpack. Например, так выглядит WAL-файл после первого запроса вставки INSERT («s:insert({1})») для базы данных из песочницы, созданной в упражнениях в *«Руководстве для начинающих»*. Слева представлены шестнадцатеричные байты, которые можно просмотреть с помощью:

```
$ 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}
```

Для чтения файлов в формате .xlog в Tarantool'е предусмотрен *модуль xlog*.

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. В процесс записи, запущенный в потоке журнала упреждающей записи, отправляется сообщение о необходимости внесения записи в WAL-файл. Экземпляр переключается на работу со следующим запросом, пока запись не будет подтверждена.
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.

7.2.3 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'a.

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.

7.2.4 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.

7.3 Содействие в разработке

7.3.1 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+.
   - библиотека ReadLine любой версии

- библиотека ncurses любой версии
- библиотека OpenSSL версии 1.0.1+
- библиотека cURL версии 0.725+
- библиотека LibYAML версии 0.1.4+
- библиотека ICU последней 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 coreutils sed \
      libreadline-dev libncurses5-dev libyaml-dev libssl-dev \
      libcurl4-openssl-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/Fedora, выполните команду:

```
$ yum install -y gcc gcc-c++ cmake coreutils sed \
      readline-devel ncurses-devel libyaml-devel openssl-devel \
      libcurl-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 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 \
       readline ncurses libyaml openssl curl libunwind-headers icu4c \
       && pip install python-daemon \
       msgpack-python pyyaml configargparse six gevent
```

  В противном случае, загрузите стандартный пакет Xcode для разработки:

```
$ xcode-select --install
$ xcode-select -switch /Applications/Xcode.app/Contents/Developer
```

- Если вы используете FreeBSD (команды для FreeBSD 10.1), выполните команду:

```
$ pkg install -y sudo git cmake gmake gcc coreutils \
      readline ncurses libyaml openssl curl libunwind icu \
      python27 py27-pip py27-setuptools py27-daemon \
      py27-msgpack-python 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. Используйте `git`, чтобы загрузить последний исходный код Tarantool'а из репозитория на GitHub `tarantool/tarantool` (ветка 1.10) в локальную директорию `~/tarantool`, например:

```
$ git clone --recursive https://github.com/tarantool/tarantool.git -b 1.10 ~/tarantool
```

В редких случаях вложенные модули необходимо снова обновить с помощью команды:

```
$ 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/`.

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 ]
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

7.3.2 Сборка документации

Документация Tarantool'а создается с помощью системы упрощенной разметки под названием `Sphinx` (see `http://sphinx-doc.org`). Вы можете создать локальную версию документации, а также содействовать в разработке версии Tarantool'а.

Необходимо установить следующие пакеты:

- `git` (программа для скачивания репозиториев исходного кода)
- `CMake` версии 2.8 или более новой (программа для управления процессом сборки)
- `Python` версии выше 2.6 – рекомендуется 2.7 – и ниже 3.0 (Sphinx – это средство на основе Python)
- `LaTeX` (система для подготовки документации; название устанавливаемого пакета обычно начинается со слов „texlive“ или „tetex“, на Ubuntu называется „texlive-latex-base“)
- `ImageMagick` (система для конвертации изображений; на MacOS установите, используя `brew`)

Необходимо установить следующие модули Python:

- pip любой версии
- Sphinx версии 1.4.4 или новее

Примечание: Если на Mac появится сообщение ошибки «Missing SPHINX_EXECUTABLE» Mac, экспортируйте переменную PATH вручную:

```
export PATH=$PATH:/User/user_name/Library/Python/2.7/bin
```

- sphinx-intl версии 0.9.9

Примечание: Если на Mac появится сообщение ошибки «Missing SPHINX_INTL_DIR» Mac, экспортируйте переменную SPHINX_INTL_DIR вручную:

```
export SPHINX_INTL_DIR=/User/user_name/Library/Python/2.7/bin
```

- lupa – любой версии

Примечание: Для правильной установки модулей Python на Mac следует указать флаг `--user`.

Более подробную информацию об установке см. в разделе *Сборка из исходных файлов* данного руководства.

1. Используйте `git` для загрузки последней версии исходного кода документации из репозитория GitHub `tarantool/doc` (ветка 1.10). Например, для загрузки локальной директории под названием `~/tarantool-doc`:

```
$ git clone https://github.com/tarantool/doc.git ~/tarantool-doc
```

2. Используйте `CMake`, чтобы начать сборку.

```
$ cd ~/tarantool-doc
$ make clean          # необязательно, добавлено на удачу
$ rm CMakeCache.txt   # необязательно, добавлено на удачу
$ cmake .             # начать
```

3. Создайте локальную версию документации.

   Выполните команду `make` с соответствующей опцией, чтобы указать версию собираемой документации.

```
$ cd ~/tarantool-doc
$ make sphinx-html            # многостраничная английская версия
$ make sphinx-singlehtml      # одностраничная английская версия
$ make sphinx-html-ru         # многостраничная русская версия
$ make sphinx-singlehtml-ru   # одностраничная русская версия
$ make all                    # все версии плюс веб-сайт полностью
```

   Документация будет создана в поддиректориях `/output`:

   - `/output/en` (файлы английской версии)
   - `/output/ru` (файлы русской версии)

   Точкой входа в каждую версию будет файл `index.html` в соответствующей директории.

4. Настройте веб-сервер.

   - Один способ сделать это – выполнить команду `make sphinx-webserver`. Веб-сервер будет настроен и запущен по порту `8000`:

```
$ cd ~/tarantool-doc
$ make sphinx-html       # в качестве примера соберем многостраничную версию␣
↪документации на английском языке
$ make sphinx-webserver  # настройка и запуск веб-сервера
```

     Если порт `8000` уже используется, можно указать любой другой номер порта свыше `1000` в файле `tarantool-doc/CMakeLists.txt` (найдите его по `sphinx-webserver`) и повторно собрать файлы cmake:

```
$ cd ~/tarantool-doc
$ git clean -qfxd        # очистка старых файлов cmake
$ cmake .                # повторная сборка файлов cmake
$ make sphinx-html       # в качестве примера соберем многостраничную версию␣
↪документации на английском языке
$ make sphinx-webserver  # настройка и запуск веб-сервера по указанному порту
```

     Или можно освободить порт:

```
$ sudo lsof -i :8000  # получение идентификатора процесса (PID)
COMMAND PID USER FD TYPE DEVICE SIZE/OFF NODE NAME
Python 19516 user 3u IPv4 0xe7f8gc6be1b43c7 0t0 TCP *:irdmi (LISTEN)
$ sudo kill -9 19516  # удаление процесса
```

- Другой способ – это запустить встроенный веб-сервер на Python. Убедитесь, что запускаете его из папки документации `output`:

```
$ cd ~/tarantool-doc/output
$ python -m SimpleHTTPServer 8000
```

  Если порт `8000` уже используется, можно указать любой другой номер порта свыше `1000` в файле.

5. Откройте браузер и введите `127.0.0.1:8000/en/doc/1.10/` в адресной строке (или `127.0.0.1:8000/ru/doc/1.10/` для документации на русском). Обратите внимание на завершающую косую черту «/» в адресной строке.

   Если сборка локальной документации выполнена правильно, руководство появится в окне браузера.

6. Чтобы содействовать в разработке документации, используйте формат REST для создания чернового варианта и отправьте изменения на рассмотрение в виде запроса на включение в проект через GitHub.

   Чтобы текст соответствовал стилю и формату, воспользуйтесь *рекомендациями*, предоставленными в документации, здравым смыслом и уже существующими документами.

Примечание:

- Если вы предлагаете создать новый раздел документации (отдельную страницу), его следует сохранить в соответствующий раздел на GitHub.
- Если вы хотите содействовать в локализации данной документации (например, на русский), добавьте перевод в файлы формата `.po`, которые хранятся в директории соответствующей локали (например, `/locale/ru/LC_MESSAGES/` для русского языка). Более подробную информацию о локализации с помощью Sphinx см. по ссылке `http://www.sphinx-doc.org/en/stable/intl.html`

7.3.3 Управление версиями

Политика управления версия

Версия Tarantool'а определяется тремя цифрами, например, 1.7.7. Мы пользуемся цифрами по определению, данному на сайте `http://semver.org`:

- Первая цифра означает МАЖОРНУЮ версию. **Мажорная** версия может содержать *несовместимые изменения.*
- Вторая цифра означает МИНОРНУЮ версию; такая версия не содержит несовместимых изменений и используется для релиза нового *функционала* с обратной совместимостью.
- Третья цифра используется для ПАТЧ-версий, которые содержат только **исправления дефектов** с обратной совместимостью.

Цифра МИНОРНОЙ версии также отражает ее стабильность:

- 0 означает альфа-версию,
- 1 означает бета-версию,
- от 1 до 10 означает стабильную версию, а
- 10 означает окончательную версию с долгосрочной технической поддержкой.

Таким образом, каждая МАЖОРНАЯ версия проходит через жизненный цикл разработки МИНОРНЫХ версий следующим образом:

1. **Альфа**. Один раз в несколько месяцев выходят несколько альфа-версий, например, 2.0.1, 2.0.2.

   Альфа-версии могут содержать несовместимые изменения, сбои и другие дефекты.

2. **Бета**. Когда готовы значительные изменения, необходимые для включения новых основных функций, мы выпускаем несколько бета-версий, например, 2.1.3, 2.1.4.

   Бета-версии могут приводить к сбоям, но не содержат несовместимых изменений, поэтому их можно использовать для разработки новых приложений.

4. **Стабильная**. Наконец, после того, как бета-версии успешно отработают примерно несколько месяцев, во время которых мы исправляем поступающие дефекты и добавляем некоторые небольшие функции, мы объявляем эту МАЖОРНУЮ версию стабильной.

Как и в Ubuntu, мы различаем два вида стабильных версий:

- **LTS (Long Term Support - Долгосрочная техническая поддержка)** такие версии поддерживаются в течение 3 лет (сообщество) и до 5 лет (платежеспособные клиенты). **LTS**-версию можно идентифицировать по МИНОРНОЙ версии = 10.
- **Стандартные стабильные версии** поддерживаются в течении нескольких месяцев после выхода.

«Support» (поддержка) означает, что мы продолжаем исправлять ошибки в этой версии.

Мы добавляем коммиты одновременно в три МАЖОРНЫЕ версии:

- **LTS** – это стабильная версия, которая не получает новые функции, а только исправления обратной совместимости. Следовательно, по правилам семантической версификации в LTS-версии никогда не увеличивается МАЖОРНАЯ или МИНОРНАЯ, а только ПАТЧ-версия.
- **СТАБИЛЬНАЯ** – это наша текущая стабильная версия, в которую могут быть добавлены новые функции. Когда выходит следующая СТАБИЛЬНАЯ версия, увеличивается МИНОРНАЯ версия. Между МИНОРНЫМИ версиями у нас могут увеличиваться промежуточные уровни ПАТЧ-версии, в которых будут только исправлены дефекты. Мы поддерживаем ПАТЧ-уровни для двух СТАБИЛЬНЫХ версий – текущей и предыдущей – для сообщества разработчиков.
- **СЛЕДУЮЩАЯ** – это следующая МАЖОРНАЯ версия, которая проходит процесс зрелости, описанный в начале раздела. Когда СЛЕДУЮЩАЯ версия находится в альфа-стадии, МИНОРНАЯ остается на уровне 0 и увеличивается, когда версия переходит в БЕТА-стадию. После того, как СЛЕДУЮЩАЯ версия становится СТАБИЛЬНОЙ, мы переключаемся на выдачу небольших функций, обозначая предыдущую стабильную версию как LTS, и выпускаем ее с МИНОРНОЙ версией = 10.

Итак, раз в квартал выходят:

- следующая LTS-версия, например, 2.10.6, 2.10.7 или 2.10.8
- следующая СТАБИЛЬНАЯ версия,например, 3.6, 3.7 или 3.8
- (возможно) альфа-стадия или бета-стадия СЛЕДУЮЩЕЙ версии, например, 4.0.1, 4.0.2 или 4.0.3

Для всех поддерживаемых версий мы также выпускаем ПАТЧИ, как только обнаружим и устраним уязвимость.

Мы также публикуем ночные сборки и используем четвертый слот в идентификаторе версии для обозначения номера ночной сборки.

Пример идентификатора версии:

- 2.0.3 - третья альфа-стадия версии 2.0

- 2.1.3 - бета-стадия версии 2.0
- 2.2 - стабильная версия серии 2.0, но еще не LTS
- 2.10 - LTS-версия

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.

Как выпустить Docker-контейнер

Чтобы выдать новую версию Docker-контейнера:

1. В главной ветке `master` в репозитории tarantool/docker найдите Dockerfile, который соответствует **мажорной** версии коммита (например, https://github.com/tarantool/docker/blob/master/2.4/Dockerfile for Tarantool version 2.x), и укажите необходимый коммит в `TARANTOOL_VERSION`, например, `TARANTOOL_VERSION=2.4.0-11-gcd17b77f9`.

   Снова загрузите Dockerfile в главную ветку.

3. В том же репозитории создайте ветку с именем на основании версий коммита `<major>.<minor>`, например, ветка `2.4` для коммита 2.4.0-11-gcd17b77f9.

4. В настройках сборки контейнера Tarantool'а в `hub.docker.com` (https://hub.docker.com/r/tarantool/tarantool/~/settings/automated-builds/) добавьте новую строку:

   ```
   Branch: x.y, /x, x.y
   ```

   где `x` и `y` соответствуют мажорной и минорной версиям коммита.

   Нажмите **Save changes** (сохранить изменения).

Вскоре будет создан новый Docker-контейнер.

7.4 Рекомендации

7.4.1 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'а 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'а 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.

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@freelists.org. Просьба подписаться на рассылку https://www.freelists.org/list/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
    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@freelists.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.

7.4.2 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'а, 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'а (например, `/etc/tarantool/instances.enabled` – активные экземпляры).

Неправильно: «С помощью репликации несколько *серверов* Tarantool'а 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

возвращается 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)

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.

7.4.3 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.

1. 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».
2. Остальные главы документа «Стиль программирования ядра Linux» изменяются следующим образом:

Общие рекомендации

Для управления версиями мы пользуемся Git. Последние разработки ведутся в ветке, используемой по умолчанию (сейчас `2.0`). Наш git-репозиторий находится на github, его можно посмотреть выгрузить с помощью `git clone git://github.com/tarantool/tarantool.git` (для анонимного пользователя доступ только для чтения).

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
 *
 * @reval  1  Ошибка (не EINTR)
 * /
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.

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, для комментариев уменьшается до 66 символов, и этому следует уделить особое внимание.

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:

1. 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.

2. 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.

3. когда вы используете разрыв, чтобы буквально создать _новый_ тип для проверки типов.

4. 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.

5. 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

7.4.4 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].

[1] ван Россум Гвидо. PEP 7, Руководство по программированию на языке C
[2] Руководство Барри по GNU Mailman

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:

```
# 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. Автома-

тический 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.

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“.

[3] Страница Википедии о CamelCase

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>

7.4.5 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

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...
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,
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 -- проверка базовой table.copy (глубокая копия)
    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"
    )
    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() == true 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
    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
    return data
end
return nil, err
```

Lua Module Index

b

box.ctl, ??
box.error, ??
box.index, ??
box.schema, ??
box.session, ??
box.slab, ??
box.space, ??
box.tuple, ??
buffer, 273

c

capi_error, ??
clock, 274
console, 276
crypto.cipher, ??
crypto.digest, ??
csv, 280

d

debug, ??
digest, 284

e

errno, 288

f

fiber, 290
fio, 304

h

http.client, ??

i

iconv, 323

j

json, 325

l

log, 328

m

msgpack, 329
my_box.index, ??
my_fiber, ??

n

net_box, ??

o

os, 341

p

pickle, 344

s

schema, 532
shard, 399
socket, 346
strict, 357
string, 357

t

table, 361
tap, 362
tarantool, 367

u

uri, 375
utf8, 370
uuid, 368

x

xlog, 377

y

yaml, 377