
Tarantool

Выпуск 1.7.1

мар. 14, 2019

1	Общие сведения	1
1.1	Сервер приложений + СУБД	1
1.2	Возможности СУБД	2
2	Вопросы и ответы	4
3	Руководство пользователя	6
3.1	Предисловие	6
3.2	Начало работы	7
3.3	Функционал СУБД	10
3.4	Репликация	82
3.5	Справочник по конфигурированию	94
3.6	Администрирование серверной части	105
3.7	Коннекторы	120
3.8	Модули	129
3.9	Приложение А. Коды ошибок	132
3.10	Приложение В. Детали реализации	132
3.11	Приложение С. Учебные примеры на языке Lua	136
3.12	Приложение D. Vinyl	147
3.13	Приложение Е. Готовые рецепты	154
3.14	Приложение F. Учебные примеры на языке C	164
3.15	Appendix G. Some significant changes in version 1.7	168
4	Справочник по Lua API	169
4.1	Module <i>box</i>	169
4.2	Модуль <i>clock</i>	169
4.3	Модуль <i>console</i>	171
4.4	Модуль <i>crypto</i>	172
4.5	Module <i>csv</i>	174
4.6	Модуль <i>digest</i>	177
4.7	Вложенный модуль <i>box.error</i>	180
4.8	Модуль <i>fiber</i>	182
4.9	Модуль <i>fiber-ipc</i>	188
4.10	Module <i>fio</i>	191
4.11	Module <i>fun</i>	199
4.12	Module <i>jit</i>	199
4.13	Module <i>json</i>	201

4.14	Module <i>log</i>	203
4.15	Модуль <i>msgpack</i>	204
4.16	Модуль <i>net.box</i>	206
4.17	Функция <i>box.once</i>	210
4.18	Module <i>os</i>	210
4.19	Модуль <i>pickle</i>	212
4.20	Модуль <i>socket</i>	214
4.21	Модуль <i>strict</i>	223
4.22	Module <i>tap</i>	224
4.23	Модуль <i>tarantool</i>	227
4.24	Module <i>uuid</i>	228
4.25	Module <i>yaml</i>	230
4.26	Разное	232
5	Справочник по Lua rocks	234
5.1	Модули SQL СУБД	234
5.2	Модуль <i>expirationd</i>	244
5.3	Модуль <i>shard</i>	246
5.4	Module <i>tdb</i>	251
6	Справочник по C API	253
6.1	Модуль <i>box</i>	253
6.2	Модуль <i>clock</i>	256
6.3	Модуль <i>coio</i>	256
6.4	Модуль <i>error</i>	257
6.5	Модуль <i>fiber</i>	261
6.6	Модуль <i>index</i>	263
6.7	Модуль <i>latch</i>	267
6.8	Модуль <i>lua/utills</i>	267
6.9	Модуль <i>say</i> (логирование)	269
6.10	Модуль <i>schema</i>	270
6.11	Модуль <i>trivia/config</i>	271
6.12	Модуль <i>tuple</i>	271
6.13	Модуль <i>txn</i>	275
7	Руководство разработчика	276
7.1	Состав документации	276
7.2	Сборка из исходных файлов	276
7.3	Сборка документации	280
7.4	Соглашения по разработке	281
7.5	Соглашения по документации	281
7.6	Протокол IProto	285
7.7	Соглашения по разработке на языке C	294
7.8	Соглашения по разработке на языке Python	307
7.9	Работа с релизами	318
	Lua Module Index	320

1.1 Сервер приложений + СУБД

Tarantool представляет собой сервер приложений на языке Lua, интегрированный с СУБД. В основе Tarantool лежат фиберы (fibers), что позволяет нескольким Lua-приложениям работать в одном потоке (thread), при этом Tarantool-сервер может одновременно запускать другие потоки для обработки ввода-вывода данных и фоновых сервисных задач. Tarantool включает в себя LuaJIT (Just In Time) Lua-компилятор, Lua-библиотеки для решения наиболее частых задач, а также сервер баз данных Tarantool, представляющий собой NoSQL СУБД. Таким образом, Tarantool может всё то же, что node.js и Twisted, а кроме того обеспечивает сохранность 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'a – а также его основным пользователем – является компания [Mail.Ru](#), крупнейшая Интернет-компания России (30 млн пользователей, 25 млн электронных писем в день, веб-сайт 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, кеширование 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'a, 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'a 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'a.

Пользователи Tarantool'a 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'a. Такие программы могут служить триггерами, выполнять фоновые задачи и взаимодействовать с другими программами по сети. В отличие от многих популярных сред разработки приложений, которые используют "реактивный" принцип, сетевое взаимодействие в Lua устроено по-

следовательно, но очень эффективно, т.к. оно использует среду **взаимной многозадачности** самого Tarantool'a.

Один из встраиваемых Lua-пакетов – это API для функционала СУБД. Таким образом, некоторые разработчики 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'a может достигать сотен тысяч транзакций в секунду на ноутбуке, и ее можно наращивать "вверх" или "вширь" за счет новых серверных ферм.

1.2 Возможности СУБД

Компонент "боксерверная часть с функционалом СУБД – это важная часть Tarantool'a, хотя он может работать и без данного компонента.

API для функционала 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'a реализована концепция "движков"базы данных (storage engines), где в разных ситуациях используются разные наборы алгоритмов и структуры данных. В Tarantool'e 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'e 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 логирования также поддерживает групповые коммиты.

In-memory движок (memtx) хранит все данные в оперативной памяти, поэтому время ожидания при чтении у него очень мало. Также, когда пользователи запрашивают статические снимки (snapshots), этот движок создает персистентные копии данных в постоянной памяти, например на жестком диске. Если Tarantool-сервер прекращает работать и данные в оперативной памяти теряются, то при следующем запуске Tarantool-сервер загружает в память самую свежую копию данных с диска и применяет к ней все транзакции из лога, которые были сделаны с момента создания копии. Таким образом, данные при сбое не теряются.

В штатных ситуациях **in-memory движок работает без блокировок**. Вместо низкоуровневых механизмов параллельной обработки данных, которые предлагает операционная система (например, mutex'ов), Tarantool использует среду взаимной многозадачности, и таким образом может работать с тысячами соединений одновременно. В Tarantool'e есть фиксированное количество независимых нитей (thread), и у них нет общего состояния. Для обмена данными между нитями используются очереди сообщений, что позволяет уменьшить накладные расходы. Такой подход накладывает ограничение на количество процессорных ядер, которые Tarantool-сервер может использовать, но в то же время он позволяет избежать конкуренции за шину памяти, а также дает запас масштабируемости по скорости доступа к памяти и производительности сети. В результате даже сильно нагруженный Tarantool-сервер в среднем использует процессор не более чем на 10%. Кроме того, Tarantool поддерживает поиск как по первичным, так и по **вторичным ключам в индексах**.

Дисковый движок (*vinyl*) совмещает в себе подходы, заимствованные из современных файловых систем, журналируемых merge-деревьев (log-structured merge trees) и классических B-деревьев. Все данные в этом движке разбиты по забегам (runs), где каждый забег представляет собой файл на диске. Максимальный размер забега обычно равен 64МБ, и его можно настраивать. Каждый забег –

это набор страниц, которые нужны для каких-то целей. Если забег полностью смержен, то диапазоны ключей на его страницах не пересекаются. Если же диапазоны ключей в забеге в какой-то момент сильно изменялись, то мы имеем дело с частично смерженным забегом. В этом случае на некоторых страницах появились новые ключи и значения, которых ранее не было в данном забеге. Дискковый движок обновляет данные по принципу дописывания в конец; новые данные никогда не затирают старые.

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 2

Вопросы и ответы

Вопрос:	Почему Tarantool? Tarantool – это in-memory сервер баз данных последнего поколения, предназначенный для работы с веб-приложениями. Разработка ведется с учетом многолетнего опыта использования Tarantool'a в стенах Mail.Ru. Первый релиз состоялся в 2008 году.
Ответ:	
Вопрос:	Почему Lua? Lua – это легкий, быстрый и расширяемый язык, позволяющий использовать различные парадигмы программирования. Lua также легко встраивается в различные приложения. Ко-рутины (coroutines) в Lua близко соотносятся с файберами (fibers) в Tarantool'e, а вся Lua-архитектура гладко ложится на его внутреннюю реализацию. Lua – это первый язык, на котором можно писать хранимые процедуры для Tarantool'a. В будущем список поддерживаемых языков планируется расширить.
Ответ:	
Вопрос:	В чем состоит главное преимущество Tarantool'a? Tarantool предоставляет богатый набор функций по работе с базами данных (HASH, TREE, RTREE, BitSet-индексы, вторичные индексы, составные индексы, транзакции, триггеры, асинхронная репликация), и всё это – в гибкой среде Lua-интерпретатора. Эти два обстоятельства делают Tarantool быстрым и надежным in-memory сервером баз данных с атомарными операциями и сложной логикой на стороне сервера. Преимуществом Tarantool'a по сравнению с традиционными SQL СУБД является его высокая производительность: низкие накладные расходы и безблокировочная архитектура позволяют Tarantool'у обрабатывать на порядок больше запросов в секунду на аналогичном оборудовании. Преимущество же перед другими NoSQL СУБД заключается в большей гибкости Tarantool'a: язык Lua позволяет гибко обрабатывать данные, хранящиеся в компактном, ненормализованном формате.
Ответ:	
Вопрос:	Каковы ваши планы по развитию? Мы постоянно улучшаем производительность серверной части. Наши главные цели на 2016-2018 год – это автоматический шардинг и синхронная репликация, а также частичная поддержка SQL. План работ находится в открытом доступе, и все желающие могут оставлять запросы на добавление функционала.
Ответ:	
Вопрос:	Кто разрабатывает Tarantool? Во-первых, этим занимается команда разработки в Mail.Ru – см. историю коммитов на github.com/tarantool . Вся разработка ведется открытым образом.
Ответ:	Кроме того, активную роль играют члены сообщества разработчиков Tarantool'a. Их силами было создано большинство коннекторов и ведутся доработки под разные дистрибутивы.
Вопрос:	Насколько серьезны планы Mail.Ru в отношении Tarantool'a? Tarantool – это проект с открытым кодом, распространяемый под лицензией BSD, поэтому он не зависит от внешних спонсоров. В то же время, Tarantool – это часть технологического "костяка" Mail.Ru, и поэтому он пользуется сильной поддержкой со стороны Mail.Ru.
Ответ:	
Вопрос:	Возникают ли проблемы из-за того, что Tarantool является in-memory решением? Основной движок баз данных в Tarantool'e 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'e используются технологии сжатия и распределения данных, которые позволяют использовать все виды памяти наиболее эффективно. Если Tarantool стабильно
Ответ:	

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

Знакомство с Tarantool'ом вы можете начать с того, что скачаете готовый установочный пакет (как описано в начале главы 2 "Начало работы") или воспользуетесь нашим онлайн-стендом на <http://try.tarantool.org>. В любом случае для первого знакомства вы можете выполнить тренировочные примеры из второй части главы 2 "Первичный запуск Tarantool'a и создание базы данных".

В главе 3 "Функционал СУБД" рассказано о возможностях Tarantool'a как NoSQL СУБД. В конце этой главы также приводится справочник по библиотекам Tarantool'a. Но если вы собираетесь использовать Tarantool только как сервер приложений Lua, то вы можете пропустить и эту, и следующую за ней главу 4 "Репликация".

Глава 5 "Справочник по конфигурированию" и глава 6 "Администрирование серверной части" предназначены в первую очередь для системных администраторов. Тем не менее любому пользователю Tarantool'a полезно знать, как производится настройка Tarantool-сервера, поэтому мы настоятельно рекомендуем прочесть раздел про параметры `box.cfg`. Глава 7 "Коннекторы" нужна только тем пользователям, кто хочет устанавливать соединение с Tarantool'ом с помощью программ на других языках программирования (например C, Perl или Python) – для прочих пользователей эта глава неактуальна. В главе 8. "Модули" показано, как устанавливать существующие и создавать новые модули (библиотеки) на языках Lua и C.

В первых двух Приложениях, А и В, приводится справочная информация по внутренней реализации Tarantool'a: коды ошибок, логика внутренних процессов (восстановления и репликации) и собственные форматы файлов.

Три длинных учебных примера в Приложении С – "Вставка 1 млн таплов с помощью хранимой процедуры на языке Lua "Подсчет суммы по JSON-полям во всех таплах" и "Индексированный поиск по шаблону" предназначены для пользователей, которые еще плохо знакомы с языком Lua или NoSQL СУБД.

В Приложении D "Vinyl" подробно рассказывается про дисковый движок Tarantool'a под названием *vinyl*.

В Приложении E "Готовые рецепты" собраны рабочие фрагменты кода, которые пригодятся для решения типичных или, наоборот, особо трудных задач.

В Приложении F содержится один длинный учебный пример по работе с хранимыми процедурами на языке C.

Опытным пользователям могут быть интересны Справочники и Руководство разработчика, а также подробные комментарии в исходном коде.

3.1.2 Как связаться с сообществом разработчиков Tarantool'a

Оставить сообщение о найденных дефектах или сделать запрос на новый функционал можно тут: <http://github.com/tarantool/tarantool/issues>

Пообщаться напрямую с командой разработки Tarantool'a можно в [telegram](#) или на форумах ([английском](#) или [русскоязычном](#)).

3.2 Начало работы

В этой главе рассказывается, как скачать, установить и начать работать с Tarantool'ом с нуля.

Для промышленной эксплуатации рекомендуется скачать бинарный (исполняемый) пакет. Тогда вы гарантированно получите сборку той же версии, что и у разработчиков. Это существенно упростит поиск ошибок, если вам в будущем понадобится помощь, а также позволит избежать проблем из-за того, что вы использовали инструменты или параметры отличные от тех, что использовали при сборке сами разработчики. См. раздел "[Скачивание и установка бинарного пакета](#)".

Для разработческих целей вы можете скачать исходные файлы и собрать бинарный пакет самостоятельно с помощью компилятора C/C++ и обычных инструментов для сборки. Хотя это и более трудный способ получить бинарный пакет, но он дает вам больший контроль над результатом. Также в состав исходных файлов входят дополнительные пакеты, например набор тестов для Tarantool'a. См. раздел [Сборка из исходных файлов](#)@ "[Сборка из исходных файлов](#)".

После установки вы можете сразу опробовать Tarantool в действии. Ниже вы найдете инструкции по созданию безопасной тестовой среды. Всего за несколько минут вы сможете запустить Tarantool-сервер и задать несколько инструкций по манипулированию данными. См. раздел [Первичный запуск Tarantool'a и создание базы данных](#)@ "[Первичный запуск Tarantool'a и создание базы данных](#)".

3.2.1 Скачивание и установка бинарного пакета

Бинарные пакеты для двух версий Tarantool'a – стабильной 1.6 и самой свежей 1.7 – выложены на странице <http://tarantool.org/download.html>. При каждом изменении исходного кода на GitHub (репозиторий <http://github.com/tarantool/tarantool>, ветка "1.7") происходит сборка, автоматическое тестирование и выкладка бинарных пакетов на вышеуказанную страницу.

Чтобы скачать и установить бинарный пакет для вашей операционной системы, откройте терминал с командной строкой и введите инструкции, которые даны для вашей операционной системы на странице <http://tarantool.org/download.html>.

3.2.2 Первичный запуск Tarantool'а и создание базы данных

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

Создайте новую директорию. Она понадобится только для тестовых целей, и ее можно будет удалить по окончании экспериментов.

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

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

Запустите Tarantool-сервер. Имя программы – tarantool.

```
$ # 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
$ # Если вы собрали Tarantool из исходных файлов, введите:
$ ~/tarantool/src/tarantool
```

Tarantool-сервер запускается в интерактивном режиме и выводит приглашение командной строки. Чтобы создать базу данных, задайте ее настройки с помощью вызова *box.cfg*. Вот пример минимальной конфигурации:

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

Если все в порядке, то Tarantool-сервер начнет в процессе отображать процесс инициализации, например:

```
tarantool> box.cfg{listen = 3301}
2015-08-07 09:41:41.077 ... version 1.7.0-1216-g73f7154
2015-08-07 09:41:41.077 ... log level 5
2015-08-07 09:41:41.078 ... mapping 1073741824 bytes for a shared arena...
2015-08-07 09:41:41.079 ... initialized
2015-08-07 09:41:41.081 ... initializing an empty data directory
2015-08-07 09:41:41.095 ... creating './00000000000000000000000000000000.snap.inprogress'
2015-08-07 09:41:41.095 ... saving snapshot './00000000000000000000000000000000.snap.inprogress'
2015-08-07 09:41:41.127 ... done
2015-08-07 09:41:41.128 ... primary: bound to 0.0.0.0:3301
2015-08-07 09:41:41.128 ... ready to accept requests
```

Поскольку Tarantool-сервер уже запущен, вы можете запустить новый терминал и присоединиться к основному порту Tarantool-сервера, введя следующую команду:

```
$ telnet 0 3301
```

но пока что будет лучше оставить Tarantool-сервер работать в интерактивном режиме. На промышленных серверах *интерактивный режим* нужен лишь для администрирования, однако для наглядности большинство примеров в данном руководстве даны именно в интерактивном режиме. Итак, Tarantool ждет от вас ввода инструкций.

Создайте первый спейс и первый *индекс*:

```
tarantool> s = box.schema.space.create('tester')
tarantool> s:create_index('primary', {
  >   type = 'hash',
  >   parts = {1, 'unsigned'}
  > })
```

Выполните вставку трех "таплов"(так в Tarantool'е называются "записи") в первый "спейс"из вашей базы данных:

```
tarantool> t = s:insert({1})
tarantool> t = s:insert({2, 'Music'})
tarantool> t = s:insert({3, 'Length', 93})
```

Произведите выборку тапла из первого спейса в базе по первому указанному ключу:

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

Вот что должно отображаться на вашем терминале к этому моменту:

```
tarantool> s = box.schema.space.create('tester')
2015-06-10 12:04:18.158 ... creating './00000000000000000000000000000000.xlog.inprogress'
---
...
tarantool>s:create_index('primary', {type = 'hash', parts = {1, 'unsigned'}})
---
...
tarantool> t = s:insert{1}
---
...
tarantool> t = s:insert{2, 'Music'}
---
...
tarantool> t = s:insert{3, 'Length', 93}
---
...
tarantool> s:select{3}
---
- - [3, 'Length', 93]
...
tarantool>
```

Далее, чтобы подготовиться к тестовому примеру в следующем разделе, введите:

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

3.2.3 Установка удаленного соединения

В предыдущем разделе ваш первый запрос был `box.cfg{listen = 3301}`. Значением `listen` может быть любой URI (универсальный код ресурса), в данном случае – просто номер локального порта (3301). Вы можете отправлять запросы на URI для прослушивания с помощью:

1. telnet,
2. коннектора (см. главу *Коннекторы*@*"Коннекторы"*),
3. другого экземпляра Tarantool'a с помощью *модуля console*,


```

| | НАБОР ТАПЛОВ 'tester' | | | | |
| | +-----+ | |
| | | Тапл: [ 1 ] | |
| | | Тапл: [ 2, 'Music' ] | |
| | | Тапл: [ 3, 'length', 93 ] | |
| | +-----+ | |
| | | | | | |
| | ИНДЕКС 'primary' | |
| | +-----+ | |
| | | Ключ: 1 | |
| | | Ключ: 2 | |
| | | Ключ: 3 | |
| | +-----+ | |
| | | | | | |
| | +-----+ | |
+-----+

```

Спейс

Спейс с именем 'tester' в данном примере – это контейнер.

Когда Tarantool используется для хранения данных, то он создает по меньшей мере один спейс. В общем же случае спейсов может быть много. Каждый спейс имеет уникальное имя, заданное пользователем, а также уникальный числовой идентификатор, который может быть задан пользователем, но, как правило, назначается автоматически самим Tarantool'ом. Пространства всегда содержат один набор таплов и один или более индексов.

Набор таплов

Набор таплов – в данном примере он назван 'tester' – это группа таплов.

Каждый спейс всегда содержит один набор таплов. Идентификатор набора таплов совпадает с именем самого спейса, в данном примере – *tester*.

Тапл выполняет ту же роль, что "строка" или "запись" а компоненты тапла (которые мы называем "поля") выполняют ту же роль, что "поле столбца, соответствующее данной строке" или "поле в записи" за тем исключением, что поля тапла могут быть составными (например, они могут быть массивами или отображениями) и им не нужны имена. Поэтому нет необходимости предварительно определять набор таплов при создании спейса, а каждый тапл может иметь различное количество элементов. Таплы хранятся в виде [MsgPack](#)-массивов.

Тапл может иметь любое количество полей, и это могут быть поля разных типов. Идентификатором поля является его номер. Поля нумеруются, начиная с 1. Так, например, "1" может использоваться в некоторых контекстах для обозначения первого поля тапла.

Когда Tarantool возвращает значение тапла, он берет строки в одинарные кавычки, отделяет поля с запятыми и заключает тапл в квадратные скобки. Например, [3, 'length', 93].

Индекс

Индекс – в данном примере, первичный – это совокупность значений ключей и указателей.

Чтобы набором таплов было можно пользоваться, в спейсе необходим по крайней мере один индекс. Вообще же индексов в спейсе может быть много. Как и в случае со спейсами, пользователь может

– и должен – указать имя индекса, а Tarantool подставляет уникальный числовой идентификатор («идентификатор индекса»). В нашем примере всего один индекс с именем “primary”.

Индекс может быть *составным*. Значение ключа в таком индексе составляется из значений двух или более полей табла, причем они могут браться в любом порядке. Индекс может быть *уникальным*. В этом случае один и тот же ключ не может встречаться в индексе более одного раза. Также индекс может быть одного из следующих *четырёх типов*: HASH (он самый быстрый и самый экономный в плане использования памяти, но он должен быть уникальным), TREE (он позволяет делать поиск по части ключа и получать отсортированные результаты), BITSET (он хорош для поиска с '=' и больших количеством AND-условий) или RTREE (для пространственных координат). Первый индекс называется “*первичным*” (primary) и должен быть уникальным. Все остальные индексы называются “*вторичными*” (secondary).

Индекс может содержать идентификаторы полей табла и типы данных для этих полей. Индексированные поля могут содержать данные следующих типов:

- **unsigned** (беззнаковое целое число в диапазоне от 0 до 18,446,744,073,709,551,615)
- **integer** (знаковое целое число в диапазоне от -9,223,372,036,854,775,808 до 9,223,372,036,854,775,807)
- **number** (беззнаковое целое число, либо знаковое целое число, либо число с плавающей точкой)
- **string** (строковое значение, т.е. любая последовательность октетов)
- **scalar** (логическое значение, либо число, либо строковое значение)
- **array** (последовательность чисел для *RTREE-индексов*)

В рамках нашего примера рассмотрим следующий запрос:

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

В результате у всех таблов в спейсе *tester* должно быть поле с номером 1, содержащее беззнаковое целое число.

Определения спейсов и индексов хранятся в системных спейсах. Можно (с некоторыми ограничениями) на ходу добавлять, удалять и менять эти определения. Правила синтаксиса в определениях спейсов и индексов даны в разделе *Библиотека "box" @*.

Типы данных

Tarantool работает с числами (numbers), строками (strings), логическими значениями (booleans), таблицами (tables) и пользовательскими типами данных (userdata).

Общий тип	Особый тип	Результат Lua type()	Пример
scalar	number	"number"	12345
scalar	string	"string"	'A B C'
scalar	boolean	"boolean"	true
scalar	nil	"nil"	nil
compound	Lua table	"table"	table: 0x410f8b10
compound	tuple	"Userdata"	12345: {'A B C'}

В языке Lua тип *number* (число) – это число с плавающей точкой двойной точности, но в Tarantool'e можно использовать как целые числа, так и числа с плавающей запятой. Tarantool по возможности сохраняет числовые значения в виде чисел с плавающей точкой, если числовое значение содержит десятичную запятую или если оно очень велико (более 100 триллионов = 1e14). Если в формате с плавающей точкой сохранить не удастся, то 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 различных представлениях (обычном, экспоненциальном, с суффиксом ULL и с использованием функции `tonumber64`): `-55`, `-2.7e+20`, `10000000000000ULL`, `tonumber64('18446744073709551615')`.

Для хранения данных в базе Tarantool использует формат MsgPack. Данные при хранении имеют переменную длину, поэтому для самого маленького числа потребуется только один байт, а самое большое число потребует девять байтов. Если поле имеет индекс `'unsigned'`, то оно может содержать только целые беззнаковые числа со значениями в диапазоне от 0 до 18,446,744,073,709,551,615.

Тип `string` (строка) – это последовательность байтов, имеющая переменную длину. Как правило, строки представлены в виде алфавитно-числовых символы, заключенных в одинарные кавычки.

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

Тип `nil` (нулевой) может иметь только одно значение, также называемое `nil`, но часто отображаемое как `null`. 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 `yaml.NULL`, либо `json.NULL`, либо `msgpack.NULL`.

Тип `tuple` возвращается в формате YAML, например – `[120, 'a', 'b', 'c']`. Некоторые функции могут возвращать таблицы с несколькими таплами. Скалярная величина может быть конвертирована в тапл с 1 полем. Lua-таблица может содержать все типы полей, разрешенные для таплов, кроме нулевого типа (`nil`).

Некоторые из этих типов данных подходят для *индексированных полей*.

См. также примеры таплов в разделе про пакет `box.tuple`.

Операции

Основные операции – это пять операций для изменения данных (INSERT, UPDATE, UPSERT, DELETE, REPLACE) и одна операция для возвращения данных (SELECT). Также в Tarantool'e поддерживаются второстепенные операции типа PING, которые можно использовать только в рамках бинарного протокола. Кроме того, в Tarantool'e есть операции для *индекс-итераторов*, которые можно использовать только в коде на языке Lua. (Индекс-итераторы нужны для обхода индексов от одного ключа к другому и дают возможность пользоваться преимуществами разных типов индексов, например вычислять значение выражений логического типа при обходе BITSET-индексов или двигаться в порядке убывания значений при обходе TREE-индексов.)

Шесть примеров основных операций:

```
-- Добавляем новый тапл в набор таплов с именем tester.
-- Первое поле, field[1], будет равно 999 (тип = unsigned).
-- Второе поле, field[2], будет равно 'Taranto' (тип = string).
tarantool> box.space.tester:insert{999, 'Taranto'}

-- Обновляем тапл, меняем значение поля field[2].
-- Условие "{999}", содержащее значение ключа, которое нужно
-- искать в первичном индексе, построенном по первому полю
-- тапла, является обязательным, поскольку запросам update()
-- всегда требуется условие, определяющее значение первичного
-- ключа, в данном случае field[1].
-- Условие "{{'=', 2, 'Tarantino'}}" определяет, что полю field[2] нужно
-- присвоить новое значение.
tarantool> box.space.tester:update({999}, {'=', 2, 'Tarantino'})
```



```
-- Выполняем операцию upsert() для тапла и снова меняем
-- значение поля field[2].
-- Синтаксис запроса upsert() аналогичен синтаксису update(),
-- но возвращаемые значения у этих запросов разные.
tarantool> box.space.tester:upsert({999}, {'=' , 2, 'Tarantism'})

-- Производим замену тапла с помощью replace(), добавляем новое поле.
-- Это можно сделать и с помощью запроса update(),
-- но такой вариант часто оказывается более сложным.
tarantool> box.space.tester:replace{999, 'Tarantella', 'Tarantula'}

-- Возвращаем значение тапла.
-- Условие "{999}" все еще обязательно, хотя оно и не должно
-- содержать значение первичного ключа.
tarantool> box.space.tester:select{999}

-- Удаляем тапл.
-- Условие, определяющее значение первичного ключа,
-- снова является обязательным.
tarantool> box.space.tester:delete{999}
```

Как Tarantool выполняет основные операции? Давайте рассмотрим это на следующем примере:

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

Это аналогично следующему выражению на языке SQL:

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

1. Если запрос делается с удаленного клиента, то клиент разбирает введенное выражение, проверяет на корректность и переводит его в формат инструкций бинарного протокола, которые Tarantool-сервер сможет понять без повторного разбора. Затем клиент отправляет всё в виде пакета на сторону сервера.
2. Поток, выполняющий обработку транзакций на стороне сервера, использует первичный индекс по полю field[1], чтобы найти нужный тапл в памяти. Он проверяет, что данный тапл можно обновить (мы хотим лишь изменить значение не индексированного поля, и вряд ли что-то пойдет не так в таком простом случае).
3. Поток обработки транзакций посылает сообщение другому потоку, который занимается записью в WAL.

В этот момент происходит *передача управления* (yield). Чтобы понять важность этого события – а оно действительно важно, – нужно пояснить несколько фактов и ввести ряд новых терминов.

ФАКТ #1:

В Tarantool'е есть только один поток обработки транзакций. Многие люди уже привыкли к мысли, что потоков для обработки 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.

ФАКТ #2:

Поток обработки транзакций может управлять множеством *файберов*. Файбер – это набор инструкций, среди которых могут быть и сигналы "передать управление". Поток обработки транзакций выполняет инструкции, пока не увидит такой сигнал, и тогда он переключается на выполнение инструкций 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 файбер #2.

ФАКТ #3:

Без передачи управления не обойтись, т.к. иначе поток обработки транзакций сможет выполнять инструкции только для одного файбера. Передача управления может осуществляться один из двух способов. Первый – это *неявная передача*. Она делается при каждой операции, связанной с изменением данных, при каждом обращении к сетевому соединению, а также при каждом запросе, который передается через Tarantool-клиент. Второй способ передачи управления – это явная передача. Ее можно – и нужно – вызывать внутри Lua-функций с помощью инструкции “yield”, чтобы не допустить захвата процессора одним файбером. Это называется *взаимная многозадачность*.

Поскольку все операции, связанные с изменением данных, заканчиваются неявной передачей управления и неявным коммитом, и поскольку каждая такая операция может затрагивать не более одного тапла, то не возникает нужды в блокировках. Для примера рассмотрим следующую Lua-функцию, которая осуществляет три операции в Tarantool'е:

```
s:select{999}           -- не происходит ни передачи управления, ни коммита
s:update({...},{...})  -- происходит и передача управления, и коммит
s:select{999}           -- не происходит ни передачи управления, ни коммита
```

Последовательность операций “SELECT + UPDATE” является атомарной транзакцией: функция сохраняет базу данных в согласованном виде, пока не отработает UPDATE. А в случае “UPDATE + SELECT” согласованности нет, поскольку после операции UPDATE поток обработки транзакций может переключиться на другой файбер и удалить тот тапл, что был обновлен в рамках предыдущей операции UPDATE.

Примечание про движок: в движке vinyl передача управления происходит по-другому, см. раздел про *различия между движками memtx и vinyl*.

Примечание: Примечание про составные транзакции

есть способ отложить передачу управления, см. раздел про *атомарные операции*.

Поскольку блокировки не используются, а запись на диск производится только при работе с WAL-файлом, то транзакции в Tarantool'е обычно совершаются быстро. Кроме того, если мы имеем дело с мощным многоядерным процессором, то Tarantool-сервер может задействовать для работы не все потоки такого процессора, и продвинутые пользователи могут безболезненно запускать второй Tarantool-сервер на том же процессоре.

См. также примеры с запросами в регрессионных тестах для Tarantool'a (<https://github.com/tarantool/tarantool/tree/1.7/test/box>). Полное описание грамматики поддерживаемых в Tarantool'е функций для манипулирования данными см. далее в этой главе.

Не все операции в Tarantool'е можно выразить с помощью функций по манипулированию данными или с помощью языка Lua. Чтобы получить доступ ко всем возможностям манипулирования данными, вам понадобится *коннектор для Perl, PHP, Python или другого языка программирования*. Клиент-серверный протокол для коннекторов является открытым. Документация доступна в виде аннотированных *BNF-диаграмм*.

Сохранение данных на диск

Tarantool сохраняет данные и информацию об изменениях в нескольких WAL-файлах (write-ahead log). Записью в WAL занимается отдельный поток. Он ловит все запросы, которые могут привести к изменению данных в базе, например `box.schema.create` или `box.space.insert`. Как правило, запись о запросе, включая служебные поля и флаги, делается в WAL-файл немедленно. Это обеспечивает сохранность данных, поскольку, даже если данные из памяти утеряны вследствие перебоя в электроснабжении, Tarantool восстановит их автоматически при следующем старте: он загрузит данные из WAL-файлов, а затем применит все записанные в WAL-файлах запросы (это называется "процесс восстановления"). Пользователи могут менять частоту записи или вовсе отключать запись в WAL с помощью параметра `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. Снимки могут делаться автоматически, или же пользователи могут создавать их сами в любой момент с помощью запроса `box.snapshot()`.

См. подробности о работе потока записи в WAL в разделе [Детали реализации](#).

Манипулирование данными

Основные запросы для *манипулирования данными* – это `insert`, `replace`, `update`, `upsert`, `delete`, `select`. Все они реализованы в библиотеке `box`. Многие из этих запросов могут возвращать данные. Как правило, и вводимые, и возвращаемые значения являются Lua-таблицами.

Lua-синтаксис в данных функциях может различаться. Далее приводятся варианты таких различий на примере SELECT-запросов. Аналогичные правила существуют и для остальных функций. В каждом из приведенных примеров выполняются следующие действия: производится выборка по набору туплов из спейса с именем 'tester', где значение поля, которое соответствует ключу в первичном индексе, равно 1. Также во всех примерах мы подразумеваем, что числовой идентификатор спейса 'tester' равен 512, но это верно только для нашей тестовой базы.

Во-первых, есть пять *способов ссылки на объект*:

```
-- #1 имя_модуля . имя_вложенного_модуля . имя_объекта
tarantool> box.space.tester:select{1}
-- #2 вместо имени объекта указываем литерал в квадратных скобках
tarantool> box.space['tester']:select{1}
-- #3 вместо имени объекта указываем числовой идентификатор в квадратных скобках
tarantool> box.space[512]:select{1}
-- #4 вместо литерала, обозначающего имя объекта, указываем переменную
tarantool> variable = 'tester'
tarantool> box.space[variable]:select{1}
-- #5 указываем переменную вместо ссылки на весь объект
tarantool> s = box.space.tester
tarantool> s:select{1}
```

Для примеров в остальной части документации мы будем, как правило, использовать вариант синтаксиса #1, например `"box.space.tester:"`. Но вы можете с тем же успехом пользоваться любым из пяти описанных выше вариантов.

Также мы в дальнейшем будем использовать синтаксис типа `"space_object:"` для ссылки на спейсы (как в приведенных выше примерах) и `"index_object:"` для ссылки на индексы (например, `box.space.tester.index.primary:`).

Во-вторых, есть семь *способов задания параметров*:

```
-- #1
tarantool> box.space.testers:select{1}
-- #2
tarantool> box.space.testers:select({1})
-- #3
tarantool> box.space.testers:select(1)
-- #4
tarantool> box.space.testers.select(box.space.testers,1)
-- #5
tarantool> box.space.testers:select({1},{iterator='EQ'})
-- #6
tarantool> variable = 1
tarantool> box.space.testers:select{variable}
-- #7
tarantool> variable = {1}
tarantool> box.space.testers:select(variable)
```

Значение первичного ключа заключается в фигурные скобки. Если же этот первичный ключ является составным, то и значение будет составным, например `...select{1,2,3}`. Фигурные скобки в свою очередь могут заключаться в круглые скобки – например, `...select({...})`. Это опциональный вариант синтаксиса, и он необходим только в том случае, если нужно передать что-то помимо первичного ключа, как в примере #5. Вместо значений-литералов – например, 1 (скалярное значение) или {1} (Lua-таблица) – можно использовать имена переменных, как в примерах #6 и #7. Хотя в некоторых случаях фигурные скобки можно опускать, мы рекомендуем всегда их использовать. Так вы явно обозначите, что значение имеет тип "Lua-таблица". В примерах и описаниях в документации мы везде используем фигурные скобки, например "{1}". Но как и в случае со ссылками на объект, вы можете пользоваться любым допустимым вариантом синтаксиса.

Все функции для манипулирования данными оперируют наборами таплов. Однако, поскольку первичные ключи всегда уникальны, количество таплов в таком наборе всегда равно 0 или 1. Единственным исключением является функция `box.space...select`, которая может брать на вход как первичный, так и вторичный ключ.

Операции с индексами

Операции с индексами производятся автоматически. Если запрос по манипулированию данными меняет данные в тапле, то меняются и ключи в индексе для данного тапла. Поэтому пользователю нужно знать только как и зачем задавать индексы.

Простая операция для создания индекса, которую мы рассматривали ранее, имела следующий вид:

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

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

Также возможны следующие варианты:

1. Индексируемое поле может быть строкой, а не числом.

```
box.space.space-name:create_index('index-name', {parts = {1, 'string'}})
```

Обычный индекс, как правило, строится по полям одного из двух типов: 'NUM' = числовой (numeric) = любое неотрицательное целое число, либо 'STR' = строка (string) = любая последовательность байтов. Числа в индексе упорядочены по числовой прямой (например, число 2345

больше, чем число 500), а строки – по коду первого байта, затем по коду второго, третьего и т.д. (и теперь строка '2345' будет меньше, чем строка '500').

Подробнее о других типах индексов см. в описании функции `create_index`.

2. Индекс может строиться по нескольким полям.

```
box.space.space-name:create_index('index-name', {
  parts = {3, 'unsigned', 2, 'string'}
})
```

В обычном индексе может быть максимум 255 частей. Каждая часть характеризуется номером поля и его типом.

3. Индекс может быть неуникальным.

```
box.space.*space-name*:create_index('index-name', { unique = false })
```

Первичный индекс для тапла должен строиться по уникальным значениям полей, но остальные (вторичные) индексы могут строиться по неуникальным значениям.

4. Индекс может представлять собой не только дерево.

```
box.space.*space-name*:create_index('index-name', { type = 'hash' })
```

Чаще всего индекс – это дерево (по умолчанию) или хеш (в этом случае индекс должен быть уникальным; в определенных случаях такой индекс занимает меньше места и поиск по нему работает быстрее). Третий тип индекса – это набор битов (bitset); это неуникальный индекс, предназначенный для работы с различными бинарными значениями. Четвертый тип индекса – это R-дерево; это тоже неуникальный индекс, предназначенный для работы с массивами, а не со строками или беззнаковыми числами.

Наличие индексов никак не влияет на синтаксис запросов на изменение данных. А вот SELECT-запросы, благодаря индексам, становятся более разнообразными.

Вот простой SELECT-запрос, который мы рассматривали ранее:

```
:extsamp: `box.space.{*{имя-спейса}*}:select({*{значение}*})`
```

По умолчанию, такой запрос ищет нужный тапл по значению в первом (первичном) индексе. Поскольку первичный индекс всегда уникален, то данный запрос вернет не более одного тапла.

Также возможны следующие варианты:

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), REQ (неравно), GE (больше или равно), GT (больше). Сравнения имеют смысл только для индексов типа 'tree'.

Этот вариант поиска может вернуть более одного тапла. В таком случае таплы 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, '']
...

```

Поиск может производиться по всем полям (в этом случае используется таблица 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 тапла: {1, 'A'} и {1, 'B'}. При необходимости можно задать даже нулевые поля, в результате чего Tarantool вернет все три тапла.

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.

При поиске по BITSET-индексам можно использовать операторы BITS_ANY_SET, BITS_ALL_SET, BITS_ALL_NOT_SET, EQ и ALL.

- 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.

При поиске по RTREE-индексам можно использовать операторы GT, GE, LT, LE, OVERLAPS и NEIGHBOR.

Библиотека "box"

Наряду с исполнением кусков кода на Lua и заданием собственных функций, вы можете использовать функционал СУБД, предоставляемый Lua-библиотекой `box` и ее вложенными модулями.

3.3.2 Вложенные модули из библиотеки "box"

Содержимое библиотеки "box" можно просматривать во время исполнения с помощью команды `box` (без аргументов). Библиотека `box` содержит следующие вложенные модули: `box.schema`, `box.tuple`, `box.space`, `box.index`, `box.cfg`, `box.info`, `box.slab`, `box.stat`. Каждый вложенный модуль содержит по крайней мере одну Lua-функцию, а некоторые вложенные модули также содержат поля-члены. Функции во вложенных модулях предназначены для определения данных (операции CREATE и DROP), манипулирования данными (INSERT DELETE UPDATE UPSERT SELECT REPLACE) и просмотра данных (просмотр содержимого спейсов и конфигурации сервера).

Факторы, которые могут влиять на быстродействие функций для манипулирования данными из библиотеки box

Размер индекса	Количество ключей в индексе равно количеству таплов 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 используемой 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 наборе есть более одного тапла.
Количество обращений к индексам	Обычно для выборки значений одного тапла используется только один индекс. Но при обновлении значений в тапле требуется N обращений, если у набора таплов есть N индексов.
Количество обращений к таплам	Некоторые запросы, например SELECT, могут возвращать несколько таплов. Как правило, это наименее важный фактор из всех.
Настройка WAL	Важным параметром для записи в WAL является <code>wal_mode</code> . 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.

Далее в описании каждой функции для манипулирования данными будет дано примечание, какие из вышеприведенных факторов могут влиять на ее быстродействие.

3.3.3 Два движка базы данных: memtx и vinyl

Движок для работы с данными – это набор инструкций очень низкого уровня, которые задают логику хранения и доступа к значениям полей таплов. В Tarantool'е есть 2 движка: memtx (для работы с данными в памяти) и vinyl (для работы с данными на диске). Чтобы в конфигурации Tarantool'a указать движок vinyl, добавьте условие `engine = 'vinyl'`. В текущей документации главный акцент сделан на использовании движка memtx, поскольку он используется по умолчанию и появился раньше vinyl'a. В то же время vinyl является полностью рабочим движком для работы с данными вида "ключ-значение и он подойдет тем, кто хочет хранить данные непосредственно на диске и таким образом уменьшить время восстановления и получить возможность работы с базой большего объема. Пояснения по архитектуре и результаты сравнительных тестов на быстродействие приведены в Приложении D: [vinyl](#). С другой стороны, в vinyl'е нет некоторых функций и опций, которые есть в memtx'е. Далее в документации для всех таких случаев даны примечания, начинающиеся со слов "Примечание про движок: vinyl". А в конце данной главы приведен обзор всех *отличий между движками memtx и vinyl*.

3.3.4 Справочник по библиотекам

Вложенный модуль `box.schema`

The `box.schema` submodule has data-definition functions for spaces, users, roles, and function tuples.

`box.schema.space.create(space-name[, {options}])`
Create a space.

Параметры

- `space-name` (*string*) – name of space, which should not be a number and should not contain special characters
- `options` (*table*) – see "Options for `box.schema.space.create`" chart, below

Return space object

Rtype userdata

Options for `box.schema.space.create`

Name	Effect	Type	Default
<code>temporary</code>	space is temporary	boolean	false
<code>id</code>	unique identifier	number	last space's id, +1
<code>field_count</code>	fixed field count	number	0 i.e. not fixed
<code>if_not_exists</code>	no error if duplicate name	boolean	false
<code>engine</code>	storage engine = <i>'memtx' or 'vinyl'</i>	string	'memtx'
<code>user</code>	user name	string	current user's name
<code>format</code>	field names+types	table	(blank)

Параметры

- `space-id` (*num*) – the numeric identifier established by `box.schema.space.create`

There are five *syntax variations* for object references targeting space objects, for example `box.schema.space.drop(space-id)` will drop a space. However, the common approach is to use functions attached to the space objects, for example `space_object:drop()`.

Note re storage engine: vinyl does not support temporary spaces.

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
  > })
```

For an illustration with the `format` clause, see `box.space._space` example.

After a space is created, usually the next step is to *create an index* for it, and then it is available for insert, select, and all the other `box.space` functions.

```
box.schema.user.create(user-name[, {options}])
```

Create a user. For explanation of how Tarantool maintains user data, see section [Users and the _user space](#).

Параметры

- `user-name` (*string*) – name of user, which should not be a number and should not contain special characters
- `options` (*table*) – `if_not_exists`, `password`

Return `nil`

Example:

```
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}])
```

Drop a user. For explanation of how Tarantool maintains user data, see section [Users and the _user space](#).

Параметры

- `user-name` (*string*) – the name of the user
- `options` (*table*) – `if_exists`

Example:

```
box.schema.user.drop('Lena')
box.schema.user.drop('Lena', {if_exists=false})
```

```
box.schema.user.exists(user-name)
```

Return true if a user exists; return false if a user does not exist.

Параметры

- `user-name` (*string*) – the name of the user

Rtype `bool`

Example:

```
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}])
```

Grant [privileges](#) to a user.

Параметры

- `user-name` (*string*) – the name of the user
- `privileges` (*string*) – 'read' or 'write' or 'execute' or a combination,
- `object-type` (*string*) – 'space' or 'function'.
- `object-name` (*string*) – name of object to grant permissions to
- `role-name` (*string*) – name of role to grant to user.

- options (*table*) – grantor, if_not_exists

If 'function', 'object-name' is specified, then a `_func` tuple with that object-name must exist.

Variation: instead of object-type, object-name say 'universe' which means 'all object-types and all objects'.

Variation: instead of privilege, object-type, object-name say role-name (see section [Roles](#)).

Example:

```
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)`

Revoke *privileges* from a user.

Параметры

- user-name (*string*) – the name of the user
- privilege (*string*) – 'read' or 'write' or 'execute' or a combination
- object-type (*string*) – 'space' or 'function'
- object-name (*string*) – the name of a function or space

The user must exist, and the object must exist, but it is not an error if the user does not have the privilege.

Variation: instead of object-type, object-name say 'universe' which means 'all object-types and all objects'.

Variation: instead of privilege, object-type, object-name say role-name (see section [Roles](#)).

Example:

```
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)`

Return a hash of a password.

Параметры

- password (*string*) – password

Rtype string

Example:

```
box.schema.user.password('ЛЕНА')
```

`box.schema.user.passwd([user-name], password)`

Associate a password with the user who is currently logged in. or with another user. Users who wish to change their own passwords should use `box.schema.user.passwd(password)`. Administrators who wish to change passwords of other users should use `box.schema.user.passwd(user-name, password)`.

Параметры

- `user-name` (*string*) – user-name
- `password` (*string*) – password

Example:

```
box.schema.user.passwd('ЛЕНА')
box.schema.user.passwd('Lena', 'ЛЕНА')
```

```
box.schema.user.info([user-name])
```

Return a description of a user's privileges.

Параметры

- `user-name` (*string*) – the name of the user. This is optional; if it is not supplied, then the information will be for the user who is currently logged in.

Example:

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

```
box.schema.role.create(role-name[, {options}])
```

Create a role. For explanation of how Tarantool maintains role data, see section [Roles](#).

Параметры

- `role-name` (*string*) – name of role, which should not be a number and should not contain special characters
- `options` (*table*) – `if_not_exists`

Return nil

Example:

```
box.schema.role.create('Accountant')
box.schema.role.create('Accountant', {if_not_exists = false})
```

```
box.schema.role.drop(role-name)
```

Drop a role. For explanation of how Tarantool maintains role data, see section [Roles](#).

Параметры

- `role-name` (*string*) – the name of the role

Example:

```
box.schema.role.drop('Accountant')
```

```
box.schema.role.exists(role-name)
```

Return true if a role exists; return false if a role does not exist.

Параметры

- `role-name` (*string*) – the name of the role

Rtype bool

Example:

```
box.schema.role.exists('Accountant')
```

```
box.schema.role.grant(user-name, privilege, object-type, object-name[, option])
```

Grant *privileges* to a role.

Параметры

- *user-name* (*string*) – the name of the role
- *privilege* (*string*) – 'read' or 'write' or 'execute' or a combination
- *object-type* (*string*) – 'space' or 'function'
- *object-name* (*string*) – the name of a function or space
- *option* (*bool*) – {if_not_exists=true} or {if_not_exists=false}

The role must exist, and the object must exist.

Variation: instead of *object-type*, *object-name* say 'universe' which means 'all object-types and all objects'.

Variation: instead of *privilege*, *object-type*, *object-name* say *role-name* – to grant a role to a role.

Example:

```
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(user-name, privilege, object-type, object-name)
```

Revoke *privileges* from a role.

Параметры

- *user-name* (*string*) – the name of the role
- *privilege* (*string*) – 'read' or 'write' or 'execute' or a combination
- *object-type* (*string*) – 'space' or 'function'
- *object-name* (*string*) – the name of a function or space

The role must exist, and the object must exist, but it is not an error if the role does not have the privilege.

Variation: instead of *object-type*, *object-name* say 'universe' which means 'all object-types and all objects'.

Variation: instead of *privilege*, *object-type*, *object-name* say *role-name*.

Example:

```
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])
```

Return a description of a role's privileges.

Параметры

- `role-name` (*string*) – the name of the role.

Example:

```
box.schema.role.info('Accountant')
```

```
box.schema.func.create(func-name[, {options}])
```

Create a function tuple. This does not create the function itself – that is done with Lua – but if it is necessary to grant privileges for a function, `box.schema.func.create` must be done first. For explanation of how Tarantool maintains function data, see section [Functions and the `_func` space](#).

Параметры

- `func-name` (*string*) – name of function, which should not be a number and should not contain special characters
- `options` (*table*) – `if_not_exists`, `setuid`, `language`.

Return nil

Example:

```
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)
```

Drop a function tuple. For explanation of how Tarantool maintains function data, see section [Functions and the `_func` space](#).

Параметры

- `func-name` (*string*) – the name of the function

Example:

```
box.schema.func.drop('calculate')
```

```
box.schema.func.exists(func-name)
```

Return true if a function tuple exists; return false if a function tuple does not exist.

Параметры

- `func-name` (*string*) – the name of the function

Rtype bool

Example:

```
box.schema.func.exists('calculate')
```

Вложенный модуль `box.space`

The `box.space` submodule has the data-manipulation functions `select`, `insert`, `replace`, `update`, `upsert`, `delete`, `get`, `put`. It also has members, such as `id`, and whether or not a space is enabled. Submodule source code is available in file [src/box/lu/schema.lua](#).

A list of all `box.space` functions follows, then comes a list of all `box.space` members.

The functions and members of `box.space`

Name	Use
<code>space_object:create_index()</code>	Create an index
<code>space_object:insert()</code>	Insert a tuple
<code>space_object:select()</code>	Select one or more tuples
<code>space_object:get()</code>	Select a tuple
<code>space_object:drop()</code>	Destroy a space
<code>space_object:rename()</code>	Rename a space
<code>space_object:replace()</code>	Insert or replace a tuple
<code>space_object:put()</code>	Insert or replace a tuple
<code>space_object:update()</code>	Update a tuple
<code>space_object:upsert()</code>	Update a tuple
<code>space_object:delete()</code>	Delete a tuple
<code>space_object:count()</code>	Get count of tuples
<code>space_object:len()</code>	Get count of tuples
<code>space_object:truncate()</code>	Delete all tuples
<code>space_object:auto_increment()</code>	Generate key + Insert a tuple
<code>space_object:pairs()</code>	Prepare for iterating
<code>space_object.id</code>	.Numeric identifier of space
<code>space_object.enabled</code>	.Flag, true if space is enabled
<code>space_object.field_count</code>	.Required number of fields
<code>space_object.index</code>	.Container of space's indexes
<code>box.space.__schema</code>	.(Metadata) List of schemas
<code>box.space.__space</code>	.(Metadata) List of spaces
<code>box.space.__index</code>	.(Metadata) List of indexes
<code>box.space.__user</code>	.(Metadata) List of users
<code>box.space.__priv</code>	.(Metadata) List of privileges
<code>box.space.__cluster</code>	.(Metadata) List of clusters

object `space_object`

`space_object:create_index(index-name[, {options}])`

Create an index. It is mandatory to create an index for a tuple set before trying to insert tuples into it, or select tuples from it. The first created index, which will be used as the primary-key index, must be unique.

Parameters: `space_object` = an *object reference*; `index_name` (type = string) = name of index, which should not be a number and should not contain special characters; `options`.

Return index object

Rtype index_object

Options for `space_object:create_index`:

Name	Effect	Type	Default
type	type of index	string ('HASH' or 'TREE' or 'BITSET' or 'RTREE')	'TREE'
id	unique identifier	number	last index's id, +1
unique	index is unique	boolean	true
if_not_exists	no error if duplicate name	boolean	false
parts	field-numbers + types	{field_no, 'unsigned' or 'string' or 'integer' or 'number' or 'array' or 'scalar'}	{1, 'unsigned'}

Possible errors: too many parts. Index '...' already exists. Primary key must be unique.

Note re storage engine: vinyl supports only the TREE index type, and vinyl secondary indexes must be created before tuples are inserted.

```
tarantool> s = box.space.space55
---
...
tarantool> s:create_index('primary', {unique = true, parts = {1, 'unsigned', 2, 'string'}
↪})
---
...
```

Details about index field types: The six index field types (unsigned | string | integer | number | array | scalar) differ depending on what values are allowed, and what index types are allowed. **unsigned**: unsigned integers between 0 and 18446744073709551615, about 18 quintillion. May also be called 'uint' or 'num', but 'num' is deprecated. Legal in memtx TREE or HASH indexes, and in vinyl TREE indexes. **string**: any set of octets, up to the *maximum length*. May also be called 'str'. Legal in memtx TREE or HASH or BITSET indexes, and in vinyl TREE indexes. **integer**: integers between -9223372036854775808 and 18446744073709551615. May also be called 'int'. Legal in memtx TREE or HASH indexes, and in vinyl TREE indexes. **number**: integers between -9223372036854775808 and 18446744073709551615, single-precision floating point numbers, or double-precision floating point numbers. Legal in memtx TREE or HASH indexes, and in vinyl TREE indexes. **array**: array of integers between -9223372036854775808 and 9223372036854775807. Legal in memtx RTREE indexes. **scalar**: booleans (true or false), or integers between -9223372036854775808 and 18446744073709551615, or single-precision floating point numbers, or double-precision floating-point numbers, or strings. When there is a mix of types, the key order is: booleans, then numbers, then strings. Legal in memtx TREE or HASH indexes, and in vinyl TREE indexes.

Index field types to use in create_index

Тип поля для индексирования	What can be in it	Where is it legal	Примеры:
unsigned	integers between 0 and 18446744073709551615	memtx TREE or HASH indexes, vinyl TREE indexes	123456
string	strings – any set of octets	memtx TREE or HASH indexes vinyl TREE indexes	'A B C' '\65 \66 \67'
integer	integers between -9223372036854775808 and 18446744073709551615	memtx TREE or HASH indexes, vinyl TREE indexes	-2 ⁶³
number	integers between -9223372036854775808 and 18446744073709551615, single-precision floating point numbers, double-precision floating point numbers	memtx TREE or HASH indexes, vinyl TREE indexes	1.234 -44 1.447e+44
array	array of integers between -9223372036854775808 and 9223372036854775807	memtx RTREE indexes	{10, 11} {3, 5, 9, 10}
scalar	booleans (true or false), integers between -9223372036854775808 and 18446744073709551615, single-precision floating point numbers, double-precision floating point numbers, strings	memtx TREE or HASH indexes, vinyl TREE indexes	true -1 1.234 " "

`space_object:insert(tuple)`

Insert a tuple into a space.

Parameters: *space_object* = an *object reference*; tuple (type = Lua table or tuple) = tuple to be inserted.

Return the inserted tuple

Rtype tuple

Possible errors: If a tuple with the same unique-key value already exists, returns `ER_TUPLE_FOUND`.

Example:

```
tarantool> box.space.test:insert{5000, 'tuple number five thousand'}
---
- [5000, 'tuple number five thousand']
...
```

`space_object:select(key)`

Search for a tuple or a set of tuples in the given space.

Parameters: *space_object* = an *object reference*; key (type = Lua table or scalar) = key to be matched against the index key, which may be multi-part.

Return the tuples whose primary-key fields are equal to the passed field-values. If the number of passed field-values is less than the number of fields in the primary key, then only the passed field-values are compared, so `select{1,2}` will match a tuple whose primary key is `{1,2,3}`.

Rtype tuple

Possible errors: No such space; wrong type.

Complexity Factors: Index size, Index type.

Example:

```
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> -- must equal both primary-key fields
tarantool> s:select{1,'B'}
---
- - [1, 'B']
...
tarantool> -- must equal only one primary-key field
tarantool> s:select{1}
---
- - [1, 'A']
- - [1, 'B']
- - [1, 'C']
...
tarantool> -- must equal 0 fields, so returns all tuples
tarantool> s:select{}
---
- - [1, 'A']
- - [1, 'B']
- - [1, 'C']
- - [2, 'D']
...
```

For examples of complex `select` requests, where one can specify which index to search and what condition to use (for example "greater than" instead of "equal to") and how many tuples to return, see the later section [index_object:select](#).

`space_object:get(key)`

Search for a tuple in the given space.

Parameters: *space_object* = an *object reference*; *key* (type = Lua table or scalar) = key to be matched against the index key, which may be multi-part.

Return the tuple whose index key matches *key*, or null.

Rtype tuple

Possible errors: If *space_object* does not exist.

Complexity Factors: Index size, Index type, Number of indexes accessed, WAL settings.

The `box.space...select` function returns a set of tuples as a Lua table; the `box.space...get` function returns at most a single tuple. And it is possible to get the first tuple in a tuple set by appending `[1]`. Therefore `box.space.testers:get{1}` has the same effect as `box.space.testers:select{1}[1]`, if exactly one tuple is found.

Example:

```
box.space.testers:get{1}
```

`space_object:drop()`

Drop a space.

Parameters: *space_object* = an *object reference*.

Return nil

Possible errors: If *space_object* does not exist.

Complexity Factors: Index size, Index type, Number of indexes accessed, WAL settings.

Example:

```
box.space.space_that_does_not_exist:drop()
```

`space_object:rename(space-name)`

Rename a space.

Parameters: *space_object* = an *object reference*; *space-name* (type = string) = new name for space.

Return nil

Possible errors: *space_object* does not exist.

Example:

```
tarantool> box.space.space55:rename('space56')
---
...
tarantool> box.space.space56:rename('space55')
---
...
```

`space_object:replace(tuple)`

`space_object:put(tuple)`

Insert a tuple into a space. If a tuple with the same primary key already exists, `box.space...:replace()` replaces the existing tuple with a new one. The syntax variants `box.space...:replace()` and `box.space...:put()` have the same effect; the latter is sometimes used to show that the effect is the converse of `box.space...:get()`.

Parameters: *space_object* = an *object reference*; tuple (type = Lua table or tuple) = tuple to be inserted.

Return the inserted tuple.

Rtype tuple

Possible errors: If a different tuple with the same unique-key value already exists, returns `ER_TUPLE_FOUND`. (This will only happen if there is a secondary index. By default secondary indexes are unique.)

Complexity Factors: Index size, Index type, Number of indexes accessed, WAL settings.

Example:

```
box.space.testers:replace{5000, 'tuple number five thousand'}
```

`space_object:update(key, {{operator, field_no, value}, ...})`

Update a tuple.

The `update` function supports operations on fields — assignment, arithmetic (if the field is numeric), cutting and pasting fragments of a field, deleting or inserting a field. Multiple operations can be combined in a single update request, and in this case they are performed atomically and sequentially. Each operation requires specification of a field number. When multiple operations are present, the field number for each operation is assumed to be relative to the most recent state of the tuple, that is, as if all previous operations in a multi-operation update have already been applied. In other words, it is always safe to merge multiple `update` invocations into a single invocation, with no change in semantics.

Possible operators are:

- + for addition (values must be numeric)
- - for subtraction (values must be numeric)
- & for bitwise AND (values must be unsigned numeric)
- | for bitwise OR (values must be unsigned numeric)
- ^ for bitwise XOR (exclusive OR) (values must be unsigned numeric)
- : for string splice
- ! for insertion
- # for deletion
- = for assignment

For ! and = operations the field number can be -1, meaning the last field in the tuple.

Parameters: *space_object* = an *object reference*; key (type = Lua table or scalar) = primary-key field values, must be passed as a Lua table if key is multi-part; {operator, field_no, value} (type = table): a group of arguments for each operation, indicating what the operation is, what field the operation will apply to, and what value will be applied. The field number can be negative, meaning the position from the end of tuple (`#tuple + negative field number + 1`).

Return the updated tuple.

Rtype tuple

Possible errors: it is illegal to modify a primary-key field.

Complexity Factors: Index size, Index type, number of indexes accessed, WAL settings.

Thus, in the instruction:

```
s:update(44, {'+', 1, 55 }, {'=', 3, 'x'})
```

the primary-key value is 44, the operators are '+' and '=' meaning *add a value to a field and then assign a value to a field*, the first affected field is field 1 and the value which will be added to it is 55, the second affected field is field 3 and the value which will be assigned to it is 'x'.

Example:

Assume that the initial state of the database is `tester` that has one tuple set and one primary key whose type is `unsigned`. There is one tuple, with `field[1] = 999` and `field[2] = 'A'`.

In the update:

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

The first argument is `tester`, that is, the affected space is `tester`. The second argument is 999, that is, the affected tuple is identified by primary key value = 999. The third argument is =, that is, there is one operation — *assignment to a field*. The fourth argument is 2, that is, the affected field is `field[2]`. The fifth argument is 'B', that is, `field[2]` contents change to 'B'. Therefore, after this update, `field[1] = 999` and `field[2] = 'B'`.

In the update:

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

the arguments are the same, except that the key is passed as a Lua table (inside braces). This is unnecessary when the primary key has only one field, but would be necessary if the primary key had more than one field. Therefore, after this update, `field[1] = 999` and `field[2] = 'B'` (no change).

In the update:

```
box.space.tester:update({999}, {'=', 3, 1})
```

the arguments are the same, except that the fourth argument is 3, that is, the affected field is `field[3]`. It is okay that, until now, `field[3]` has not existed. It gets added. Therefore, after this update, `field[1] = 999`, `field[2] = 'B'`, `field[3] = 1`.

In the update:

```
box.space.tester:update({999}, {'+', 3, 1})
```

the arguments are the same, except that the third argument is '+', that is, the operation is addition rather than assignment. Since `field[3]` previously contained 1, this means we're adding 1 to 1. Therefore, after this update, `field[1] = 999`, `field[2] = 'B'`, `field[3] = 2`.

In the update:

```
box.space.tester:update({999}, {'|', 3, 1}, {'=', 2, 'C'})
```

the idea is to modify two fields at once. The formats are '|' and =, that is, there are two operations, OR and assignment. The fourth and fifth arguments mean that `field[3]` gets OR'ed with 1. The seventh and eighth arguments mean that `field[2]` gets assigned 'C'. Therefore, after this update, `field[1] = 999`, `field[2] = 'C'`, `field[3] = 3`.

In the update:

```
box.space.tester:update({999}, {'#', 2, 1}, {'-', 2, 3})
```

The idea is to delete `field[2]`, then subtract 3 from `field[3]`. But after the delete, there is a renumbering, so `field[3]` becomes `field[2]` before we subtract 3 from it, and that's why the seventh argument is 2, not 3. Therefore, after this update, `field[1] = 999`, `field[2] = 0`.

In the update:

```
box.space.testers:update({999}, {'=', 2, 'XYZ'})
```

we're making a long string so that splice will work in the next example. Therefore, after this update, `field[1] = 999`, `field[2] = 'XYZ'`.

In the update

```
box.space.testers:update({999}, {':', 2, 2, 1, '!!!'})
```

The third argument is ':', that is, this is the example of splice. The fourth argument is 2 because the change will occur in `field[2]`. The fifth argument is 2 because deletion will begin with the second byte. The sixth argument is 1 because the number of bytes to delete is 1. The seventh argument is '!!!', because '!!!' is to be added at this position. Therefore, after this update, `field[1] = 999`, `field[2] = 'X!!!'`.

`space_object:upsert(tuple_value, {operator, field_no, value}, ...)`

Update or insert a tuple.

If there is an existing tuple which matches the key fields of `tuple_value`, then the request has the same effect as `space_object:update()` and the `{operator, field_no, value}, ...` parameter is used. If there is no existing tuple which matches the key fields of `tuple_value`, then the request has the same effect as `space_object:insert()` and the `{tuple_value}` parameter is used. However, unlike `insert` or `update`, `upsert` will not read a tuple and perform error checks before returning – this is a design feature which enhances throughput but requires more caution on the part of the user.

Parameters: `space_object` = an *object reference*; `tuple_value` (type = Lua table or scalar) = field values, must be passed as a Lua table; `{operator, field_no, value}` (type = Lua table) = a group of arguments for each operation, indicating what the operation is, what field the operation will apply to, and what value will be applied. The field number can be negative, meaning the position from the end of the tuple (`#tuple + negative field number + 1`).

Return null.

Possible errors: it is illegal to modify a primary-key field. It is illegal to use `upsert` with a space that has a unique secondary index.

Complexity factors: Index size, Index type, number of indexes accessed, WAL settings.

Example:

```
box.space.testers:upsert({12, 'c'}, {'=', 3, 'a'}, {'=', 4, 'b'})
```

`space_object:delete(key)`

Delete a tuple identified by a primary key.

Parameters: `space_object` = an *object reference* `key` (type = Lua table or scalar) = key to be matched against the index key, which may be multi-part.

Return the deleted tuple

Rtype tuple

Complexity Factors: Index size, Index type

Note re storage engine: vinyl will return `nil`, rather than the deleted tuple.

Example:

```

tarantool> box.space.testster:delete(1)
---
- [1, 'My first tuple']
...
tarantool> box.space.testster:delete(1)
---
...
tarantool> box.space.testster:delete('a')
---
- error: 'Supplied key type of part 0 does not match index part type:
  expected unsigned'
...

```

space_object.id

Ordinal space number. Spaces can be referenced by either name or number. Thus, if space `tester` has `id = 800`, then `box.space.testster:insert{0}` and `box.space[800]:insert{0}` are equivalent requests.

Parameters: *space_object* = an *object reference*.

Example:

```

tarantool> box.space.testster.id
---
- 512
...

```

space_object.enabled

Whether or not this space is enabled. The value is `false` if the space has no index.

Parameters: *space_object* = an *object reference*.

space_object.field_count

The required field count for all tuples in this space. The `field_count` can be set initially with:

```

box.schema.space.create(..., {
  ... ,
  field_count = field_count_value ,
  ...
})

```

The default value is 0, which means there is no required field count.

Parameters: *space_object* = an *object reference*.

Example:

```

tarantool> box.space.testster.field_count
---
- 0
...

```

space_object.index

A container for all defined indexes. There is a Lua object of type *box.index* with methods to search tuples and iterate over them in predefined order.

Parameters: *space_object* = an *object reference*.

Rtype table

Example:

```
tarantool> #box.space.testster.index
---
- 1
...
tarantool> box.space.testster.index.primary.type
---
- TREE
...
```

`space_object:count([key][, iterator])`

Parameters: *space_object* = an *object reference*; key (type = Lua table or scalar) = key to be matched against the primary index key, which may be multi-part; iterator = comparison method.

Return Number of tuples.

Example:

```
tarantool> box.space.testster:count(2, {iterator='GE'})
---
- 1
...
```

`space_object:len()`

Parameters: *space_object* = an *object reference*.

Return Number of tuples in the space.

Example:

```
tarantool> box.space.testster:len()
---
- 2
...
```

Note re storage engine: vinyl does not support `len()`. One possible workaround is to say `#select(., .)`.

`space_object:truncate()`

Deletes all tuples. .

Parameters: *space_object* = an *object reference*.

Complexity Factors: Index size, Index type, Number of tuples accessed.

Return nil

Примечание: Note that `truncate` must be called only by the user who created the space OR under a *setuid* function created by that user. Read more about *setuid* functions [here](#)

Note re storage engine: vinyl does not support `truncate()`.

Example:

```
tarantool> box.space.testster:truncate()
---
...
tarantool> box.space.testster:len()
---
```



```
- 0
...
```

`space_object:auto_increment(field-value [, field-value ...])`

Insert a new tuple using an auto-increment primary key. The space specified by `space_object` must have an `unsigned` or `integer` or `numeric` primary key index of type `TREE`. The primary-key field will be incremented before the insert.

Note re storage engine: vinyl does not support `auto_increment()`.

Parameters: *space_object* = an *object reference*; *field-value(s)* (type = Lua table or scalar) = tuple's fields, other than the primary-key field.

Return the inserted tuple.

Rtype tuple

Complexity Factors: Index size, Index type, Number of indexes accessed, WAL settings.

Possible errors: index has wrong type or primary-key indexed field is not a number.

Example:

```
tarantool> box.space.testster:auto_increment{'Fld#1', 'Fld#2'}
---
- [1, 'Fld#1', 'Fld#2']
...
tarantool> box.space.testster:auto_increment{'Fld#3'}
---
- [2, 'Fld#3']
...
```

`space_object:pairs()`

A helper function to prepare for iterating over all tuples in a space.

Parameters: *space_object* = an *object reference*.

Return function which can be used in a for/end loop. Within the loop, a value is returned for each iteration.

Rtype function, tuple

Example:

```
tarantool> s = box.schema.space.create('space33')
---
...
tarantool> -- index 'X' has default parts {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
...

```

box.space._schema

_schema is a system tuple set. Its single tuple contains these fields: 'version', major-version-number, minor-version-number.

Example:

The following function will display all fields in all tuples of _schema:

```

function example()
  local ta = {}
  local i, line
  for k, v in box.space._schema:pairs() do
    i = 1
    line = ''
    while i <= #v do
      line = line .. v[i] .. ' '
      i = i + 1
    end
    table.insert(ta, line)
  end
  return ta
end

```

Here is what example() returns in a typical installation:

```

tarantool> example()
---
- - 'cluster 1ec4e1f8-8f1b-4304-bb22-6c47ce0cf9c6 '
- 'max_id 520 '
- 'version 1 7 0 '
...

```

box.space._space

_space is a system tuple set. Its tuples contain these fields: id, uid, space-name, engine, field_count, temporary, format. These fields are established by [space.create\(\)](#).

Example:

The following function will display all simple fields in all tuples of _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
    end
    i = i + 1
  end
  return ta
end

```

```

    end
    table.insert(ta, line)
  end
  return ta
end

```

Here is what `example()` returns in a typical installation:

```

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 '
...

```

Example:

The following requests will create a space using `box.schema.space.create` with a `format` clause. Then it retrieves the `_space` tuple for the new space. This illustrates the typical use of the `format` clause, it shows the recommended names and data types for the fields.

```

tarantool> box.schema.space.create('TM', {
  >   format = {
  >     [1] = [{"name"} = "field_1"],
  >     [2] = [{"type"} = "unsigned"]
  >   }
  > })
---
- index: []
  on_replace: 'function: 0x41c67338'
  temporary: false
  id: 522
  engine: memtx
  enabled: false
  name: TM
  field_count: 0
- created
...
tarantool> box.space._space:select(522)
---
- - [522, 1, 'TM', 'memtx', 0, {}, [{"name": 'field_1'}, {'type': 'unsigned'}]]
...

```

`box.space._index`

`_index` is a system tuple set. Its tuples contain these fields: `space-id index-id index-name index-type is-index-unique [tuple-field-no, tuple-field-type ...]`.

The following function will display all fields in all tuples of `_index`: (notice that the fifth field gets

special treatment as a map value and the sixth or later fields get special treatment as arrays):

```
function example()
  local ta = {}
  local i, line, value
  for k, v in box.space._index:pairs() do
    i = 1
    line = ''
    while v[i] ~= nil do
      if i < 5 then
        value = v[i]
      end
      if i == 5 then
        if v[i].unique == true then
          value = 'true'
        end
      end
      if i > 5 then
        value = v[i][1][1] .. ' ' .. v[i][1][2]
      end
      line = line .. value .. ' '
      i = i + 1
    end
    table.insert(ta, line)
  end
  return ta
end
```

Here is what `example()` returns in a typical installation:

```
tarantool> example()
---
- - '272 0 primary tree true 0 str '
- - '280 0 primary tree true 0 num '
- - '280 1 owner tree tree 1 num '
- - '280 2 name tree true 2 str '
- - '281 0 primary tree true 0 num '
- - '281 1 owner tree tree 1 num '
- - '281 2 name tree true 2 str '
- - '288 0 primary tree true 0 num '
- - '288 2 name tree true 0 num '
- - '289 0 primary tree true 0 num '
- - '289 2 name tree true 0 num '
- - '296 0 primary tree true 0 num '
- - '296 1 owner tree tree 1 num '
- - '296 2 name tree true 2 str '
- - '297 0 primary tree true 0 num '
- - '297 1 owner tree tree 1 num '
- - '297 2 name tree true 2 str '
- - '304 0 primary tree true 0 num '
- - '304 1 owner tree tree 1 num '
- - '304 2 name tree true 2 str '
- - '305 0 primary tree true 0 num '
- - '305 1 owner tree tree 1 num '
- - '305 2 name tree true 2 str '
- - '312 0 primary tree true 1 num '
- - '312 1 owner tree tree 0 num '
- - '312 2 object tree tree 2 str '
- - '313 0 primary tree true 1 num '
```

```

- '313 1 owner tree tree 0 num '
- '313 2 object tree tree 2 str '
- '320 0 primary tree true 0 num '
- '320 1 uuid tree true 1 str '
- '512 0 primary tree true 0 num '
- '513 0 primary tree true 0 num '
- '516 0 primary tree true 0 STR '
...

```

`box.space._user`

`_user` is a system tuple set for support of the *authorization feature*.

`box.space._priv`

`_priv` is a system tuple set for support of the *authorization feature*.

`box.space._cluster`

`_cluster` is a system tuple set for support of the *replication feature*.

Example: use `box.space` functions to read `_space` tuples

This function will illustrate how to look at all the spaces, and for each display: approximately how many tuples it contains, and the first field of its first tuple. The function uses Tarantool `box.space` functions `len()` and `pairs()`. The iteration through the spaces is coded as a scan of the `_space` system tuple set, which contains metadata. The third field in `_space` contains the space name, so the key instruction `space_name = v[3]` means `space_name` is the `space_name` field in the tuple of `_space` that we've just fetched with `pairs()`. The function returns a table:

```

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

```

And here is what happens when one invokes the function:

```

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
...

```

Example: use `box.space` functions to organize a `_space` tuple

The objective is to display field names and field types of a system space – using metadata to find metadata.

To begin: how can one select the `_space` tuple that describes `_space`?

A simple way is to look at the constants in `box.schema`, which tell us that there is an item named `SPACE_ID == 288`, so these statements will retrieve the correct tuple: `box.space._space:select{288}` or `box.space._space:select{box.schema.SPACE_ID}`

Another way is to look at the tuples in `box.space._index`, which tell us that there is a secondary index named `'name'` for space number 288, so this statement also will retrieve the correct tuple: `box.space._space.index.name:select{'_space'}`

```

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': '*}]] ...

```

It looks disorganized because field number 7 has been formatted with recommended names and data types. How can one get those specific sub-fields? Since it's visible that field number 7 is an array of maps, this *for* loop will do the organizing: `local tuple_of_space, field_name, field_type tuple_of_space = box.space._space.index.name:select{'_space'}[1] for i = 1, #tuple_of_space[7], 1 do field_name = tuple_of_space[7][i]['name'] field_type = tuple_of_space[7][i]['type'] print(field_name .. ', ' .. field_type) end`

And here is what happens when one executes the *for* loop: `id,num owner,num name,str engine,str field_count,num flags,str format,*`

Вложенный модуль `box.index`

The `box.index` submodule provides read-only access for index definitions and index keys. Indexes are contained in `box.space.space-name.index` array within each space object. They provide an API for ordered iteration over tuples. This API is a direct binding to corresponding methods of index objects of type `box.index` in the storage engine.

object `index_object`

`index_object.unique`

True if the index is unique, false if the index is not unique.

Parameters:

- `index_object` = an *object reference*.

Rtype boolean

`index_object.type`

Index type, 'TREE' or 'HASH' or 'BITSET' or 'RTREE'.

Parameters:

- `index_object` = an *object reference*.

`index_object.parts`

An array describing index key fields.

Parameters:

- `index_object` = an *object reference*.

Rtype table

Example:

```
tarantool> box.space.test.index.primary
---
- unique: true
  parts:
  - type: unsigned
    fieldno: 1
  id: 0
  space_id: 513
  name: primary
  type: TREE
...

```

`index_object:pairs(bitset-value | search-value, iterator-type)`

This method provides iteration support within an index. The *bitset-value* or *search-value* parameter specifies what must match within the index. The *iterator-type* parameter specifies the rule for matching and ordering. Different index types support different iterators. For example, a TREE index maintains a strict order of keys and can return all tuples in ascending or descending order, starting from the specified key. Other index types, however, do not support ordering.

To understand consistency of tuples returned by an iterator, it's essential to know the principles of the Tarantool transaction processing subsystem. An iterator in Tarantool does not own a consistent read view. Instead, each procedure is granted exclusive access to all tuples and spaces until there is a "context switch": which may happen due to [the-implicit-yield-rules](#), or by an explicit call to [fiber.yield](#). When the execution flow returns to the yielded procedure, the data set could have changed significantly. Iteration, resumed after a yield point, does not preserve the read view, but continues with the new content of the database. The tutorial [Indexed pattern search](#) shows one way that iterators and yields can be used together.

Parameters:

- `index_object` = an *object reference*;
- `bitset-value` | `search-value...` = what to search for
- `iterator-type` = as defined in tables below.

Return this method returns an iterator closure, i.e. a function which can be used to get the next value on each invocation

Rtype function, tuple

Possible errors: Selected iteration type is not supported for the index type, or search value is not supported for the iteration type.

Complexity Factors: Index size, Index type, Number of tuples accessed.

A search-value can be a number (for example 1234), a string (for example 'abcd'), or a table of numbers and strings (for example {1234, 'abcd'}). Each part of a search-value will be compared to each part of an index key.

Iterator types for TREE indexes

Примечание: Formally the logic for TREE index searches is: comparison-operator is = or >= or > or <= or < depending on iterator-type

```

for i = 1 to number-of-parts-of-search-value
  if (search-value-part[i] is nil and <comparison-operator> is "=") or
    (search-value-part[i] <comparison-operator> index-key-part[i] is true)
  then
    comparison-result[i] is true
  endif

```

if all comparison-results are true, then search-value "matches" index key.

Notice how, according to this logic, regardless what the index-key-part contains, the comparison-result for equality is always true when a search-value-part is nil or is missing. This behavior of searches with nil is subject to change.

Type	Argument	Description
box.index.EQ or 'EQ'	Search value	The comparison operator is '=' (equal to). If an index key is equal to a search value, it matches. Tuples are returned in ascending order by index key. This is the default.
box.index.REQ or 'REQ'	Search value	Matching is the same as for <code>box.index.EQ</code> . Tuples are returned in descending order by index key.
box.index.GT or 'GT'	Search value	The comparison operator is '>' (greater than). If an index key is greater than a search value, it matches. Tuples are returned in ascending order by index key.
box.index.GE or 'GE'	Search value	The comparison operator is '>=' (greater than or equal to). If an index key is greater than or equal to a search value, it matches. Tuples are returned in ascending order by index key.
box.index.ALL or 'ALL'	Search value	Same as <code>box.index.GE</code> .
box.index.LT or 'LT'	Search value	The comparison operator is '<' (less than). If an index key is less than a search value, it matches. Tuples are returned in descending order by index key.
box.index.LE or 'LE'	Search value	The comparison operator is '<=' (less than or equal to). If an index key is less than or equal to a search value, it matches. Tuples are returned in descending order by index key.

Iterator types for HASH indexes

Type	Argument	Description
box.index.none	none	All index keys match. Tuples are returned in ascending order by hash of index key, which will appear to be random.
box.index.EQ or 'EQ'	value	The comparison operator is '=' (equal to). If an index key is equal to a search value, it matches. The number of returned tuples will be 0 or 1. This is the default.
box.index.GT or 'GT'	value	The comparison operator is '>' (greater than). If a hash of an index key is greater than a hash of a search value, it matches. Tuples are returned in ascending order by hash of index key, which will appear to be random. Provided that the space is not being updated, one can retrieve all the tuples in a space, N tuples at a time, by using {iterator='GT', limit=N} in each search, and using the last returned value from the previous result as the start search value for the next search.

Iterator types for BITSET indexes

Type	Argument	Description
box.index.ALL or 'ALL'	none	All index keys match. Tuples are returned in their order within the space.
box.index.EQ or 'EQ'	bitset value	If an index key is equal to a bitset value, it matches. Tuples are returned in their order within the space. This is the default.
box.index.BITS_ALL	bitset value	If all of the bits which are 1 in the bitset value are 1 in the index key, it matches. Tuples are returned in their order within the space.
box.index.BITS_ANY	bitset value	If any of the bits which are 1 in the bitset value are 1 in the index key, it matches. Tuples are returned in their order within the space.
box.index.BITS_NO	bitset value	If none of the bits which are 1 in the bitset value are 0 in the index key, it matches. Tuples are returned in their order within the space.

Iterator types for RTREE indexes

Type	Argument	Description
box.index.ALL or 'ALL'	None	All keys match. Tuples are returned in their order within the space.
box.index.EQ or 'EQ'	Search value	If all points of the rectangle-or-box defined by the search value are the same as the rectangle-or-box defined by the index key, it matches. Tuples are returned in their order within the space. "Rectangle-or-box" means "rectangle-or-box as explained in section about <i>RTREE</i> ". This is the default.
box.index.GT or 'GT'	Search value	If all points of the rectangle-or-box defined by the search value are within the rectangle-or-box defined by the index key, it matches. Tuples are returned in their order within the space.
box.index.GE or 'GE'	Search value	If all points of the rectangle-or-box defined by the search value are within, or at the side of, the rectangle-or-box defined by the index key, it matches. Tuples are returned in their order within the space.
box.index.LT or 'LT'	Search value	If all points of the rectangle-or-box defined by the index key are within the rectangle-or-box defined by the search key, it matches. Tuples are returned in their order within the space.
box.index.LE or 'LE'	Search value	If all points of the rectangle-or-box defined by the index key are within, or at the side of, the rectangle-or-box defined by the search key, it matches. Tuples are returned in their order within the space.
box.index.OVERLAPS or 'OVERLAPS'	Search value	OVERLAPS Some points of the rectangle-or-box defined by the search value are within the rectangle-or-box defined by the index key, it matches. Tuples are returned in their order within the space.
box.index.NEIGHBOR or 'NEIGHBOR'	Search value	NEIGHBOR Some points of the rectangle-or-box defined by the defined by the key are within, or at the side of, defined by the index key, it matches. Tuples are returned in order: nearest neighbor first.

First Example of index pairs():

Default 'TREE' Index and pairs() function:

```
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()
---
...

```

Second Example of index pairs():

This Lua code finds all the tuples whose primary key values begin with 'XY'. The assumptions include that there is a one-part primary-key TREE index on the first field, which must be a string. The iterator loop ensures that the search will return tuples where the first value is greater than or equal to 'XY'. The conditional statement within the loop ensures that the looping will stop when the first two letters are not '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

```

Third Example of index pairs():

This Lua code finds all the tuples whose primary key values are greater than or equal to 1000, and less than or equal to 1999 (this type of request is sometimes called a "range search" or a "between search"). The assumptions include that there is a one-part primary-key TREE index on the first field, which must be a number. The iterator loop ensures that the search will return tuples where the first value is greater than or equal to 1000. The conditional statement within the loop ensures that the looping will stop when the first value is greater than 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)`

This is an alternative to `box.space...select()` which goes via a particular index and can make use of additional parameters that specify the iterator type, and the limit (that is, the maximum number of tuples to return) and the offset (that is, which tuple to start with in the list).

Parameters:

- *index_object* = an *object reference*;
- *search-key* = values to be matched against the index key;

- option(s) any or all of
 - `iterator = iterator-type`,
 - `limit = maximum-number-of-tuples`,
 - `offset = start-tuple-number`.

Return the tuple or tuples that match the field values.

Rtype tuple set as a Lua table

Example:

```
-- Create a space named tester.
tarantool> sp = box.schema.space.create('tester')
-- Create a unique index 'primary'
-- which won't be needed for this example.
tarantool> sp:create_index('primary', {parts = {1, 'unsigned' }})
-- Create a non-unique index 'secondary'
-- with an index on the second field.
tarantool> sp:create_index('secondary', {
  >   type = 'tree',
  >   unique = false,
  >   parts = {2, 'string'}
  > })
-- Insert three tuples, values in field[2]
-- equal to 'X', 'Y', and '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'}
-- Select all tuples where the secondary index
-- keys are greater than 'X'.`
tarantool> sp.index.secondary:select({'X'}, {
  >   iterator = 'GT',
  >   limit = 1000
  > })
```

The result will be a table of tuple and will look like this:

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

```

Примечание: `index.index-name` is optional. If it is omitted, then the assumed index is the first (primary-key) index. Therefore, for the example above, `box.space.tester:select({1}, {iterator = 'GT'})` would have returned the same two rows, via the 'primary' index.

Примечание: `iterator = iterator-type` is optional. If it is omitted, then `iterator = 'EQ'` is assumed.

Примечание: `field-value [, field-value ...]` is optional. If it is omitted, then every key in the index is considered to be a match, regardless of iterator type. Therefore, for the

example above, `box.space.tester:select{}` will select every tuple in the tester space via the first (primary-key) index.

Примечание: `box.space.space-name.index.index-name:select(...)[1]``. can be replaced by `box.space.space-name.index.index-name:get(...)`. That is, `get` can be used as a convenient shorthand to get the first tuple in the tuple set that would be returned by `select`. However, if there is more than one tuple in the tuple set, then `get` returns an error.

Example with BITSET index:

The following script shows creation and search with a BITSET index. Notice: BITSET cannot be unique, so first a primary-key index is created. Notice: bit values are entered as hexadecimal literals for easier reading.

```
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)`

Search for a tuple via the given index, as described [earlier](#).

Parameters: *space_object* = an *object reference*; key (type = Lua table or scalar) = key to be matched against the index key, which may be multi-part.

Return the tuple whose index-key fields are equal to the passed key values.

Rtype tuple

Possible errors: No such index; wrong type; more than one tuple matches.

Complexity Factors: Index size, Index type. See also [space_object:get\(\)](#).

Example:

```
tarantool> box.space.testers.index.primary:get(2)
---
- [2, 'Music']
...
```

`index_object:min(key-value)`

Find the minimum value in the specified index.

Parameters:

- *index_object* = an *object reference*;
- key-value.

Return the tuple for the first key in the index. If optional *key-value* is supplied, returns the first key which is greater than or equal to *key-value*.

Rtype tuple

Possible errors: index is not of type 'TREE'.

Complexity Factors: Index size, Index type.

Example:

```
tarantool> box.space.testers.index.primary:min()
---
- ['Alpha!', 55, 'This is the first tuple!']
...
```

`index_object:max(key-value)`

Find the maximum value in the specified index.

Parameters:

- *index_object* = an *object reference*;
- key-value.

Return the tuple for the last key in the index. If optional *key-value* is supplied, returns the last key which is less than or equal to *key-value*.

Rtype tuple

Possible errors: index is not of type 'TREE'.

Complexity Factors: Index size, Index type.

Example:

```
tarantool> box.space.testster.index.primary:max()
---
- ['Gamma!', 55, 'This is the third tuple!']
...
```

`index_object:random(random-value)`

Find a random value in the specified index. This method is useful when it's important to get insight into data distribution in an index without having to iterate over the entire data set.

Parameters:

- *index_object* = an *object reference*;
- *random-value* (type = number) = an arbitrary non-negative integer.

Return the tuple for the random key in the index.

Rtype tuple

Complexity Factors: Index size, Index type.

Note re storage engine: vinyl does not support `random()`.

Example:

```
tarantool> box.space.testster.index.secondary:random(1)
---
- ['Beta!', 66, 'This is the second tuple!']
...
```

`index_object:count(key[[, iterator]])`

Iterate over an index, counting the number of tuples which match the key-value.

Parameters:

- *index_object* = an *object reference*;
- *key-value* (type = Lua table or scalar) = the value which must match the key(s) in the specified index. The type may be a list of field-values, or a tuple containing only the field-values; *iterator* = comparison method.

Return the number of matching index keys.

Rtype number

Example:

```
tarantool> box.space.testster.index.primary:count(999)
---
- 0
...
tarantool> box.space.testster.index.primary:count('Alpha!', { iterator = 'LE' })
---
- 1
...
```

`index_object:update(key, {{operator, field_no, value}, ...})`

Update a tuple.

Same as `box.space...update()`, but key is searched in this index instead of primary key. This index ought to be unique.

Parameters:

- *index_object* = an *object reference*;
- *key* (type = Lua table or scalar) = key to be matched against the index key;
- *operator, field_no, value* (type = Lua table) = update operations (see: [box.space...update\(\)](#)).

Return the updated tuple.

Rtype tuple

`index_object:delete(key)`

Delete a tuple identified by a key.

Same as [box.space...delete\(\)](#), but key is searched in this index instead of in the primary-key index. This index ought to be unique.

Parameters:

- *index_object* = an *object reference*;
- *key* (type = Lua table or scalar) = key to be matched against the index key.

Return the deleted tuple.

Rtype tuple

Note re storage engine: vinyl will return nil, rather than the deleted tuple.

`index_object:alter({options})`

Alter an index.

Parameters:

- *index_object* = an *object reference*;
- *options* = options list, same as the options list for [create_index](#).

Return nil

Possible errors: Index does not exist, or the first index cannot be changed to {unique = false}, or the alter function is only applicable for the memtx storage engine.

Note re storage engine: vinyl does not support `alter()`.

Example:

```
tarantool> box.space.space55.index.primary:alter({type = 'HASH'})
---
...
```

`index_object:drop()`

Drop an index. Dropping a primary-key index has a side effect: all tuples are deleted.

Parameters:

- *index_object* = an *object reference*.

Return nil.

Possible errors: Index does not exist, or a primary-key index cannot be dropped while a secondary-key index exists.

Example:

```
tarantool> box.space.space55.index.primary:drop()
---
...
```

`index_object:rename(index-name)`

Rename an index.

Parameters:

- *index_object* = an *object reference*;
- *index-name* (type = string) = new name for index.

Return nil

Possible errors: `index_object` does not exist.

Example:

```
tarantool> box.space.space55.index.primary:rename('secondary')
---
...
```

Complexity Factors: Index size, Index type, Number of tuples accessed.

`index_object:bsize()`

Return the total number of bytes taken by the index.

Parameters:

- *index_object* = an *object reference*.

Return number of bytes

Rtype number

Example showing use of the box functions

This example will work with the sandbox configuration described in the preface. That is, there is a space named `tester` with a numeric primary key. The example function will:

- select a tuple whose key value is 1000;
- return an error if the tuple already exists and already has 3 fields;
- **Insert or replace the tuple with:**
 - `field[1]` = 1000
 - `field[2]` = a uuid
 - `field[3]` = number of seconds since 1970-01-01;
- Get `field[3]` from what was replaced;
- Format the value from `field[3]` as `yyyy-mm-dd hh:mm:ss.ffff`;
- Return the formatted value.

The function uses Tarantool box functions *box.space...select*, *box.space...replace*, *fiber.time*, *uuid.str*. The function uses Lua functions *os.date()* and *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
```

... And here is what happens when one invokes the function:

```
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'
...
```

Example showing a user-defined iterator

Here is an example that shows how to build one's own iterator. The *paged_iter* function is an "iterator function which will only be understood by programmers who have read the Lua manual section [Iterators and Closures](#). It does paginated retrievals, that is, it returns 10 tuples at a time from a table named "t" whose primary key was defined with `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"
  end
end
```

```

return page
end
end

```

Programmers who use `paged_iter` do not need to know why it works, they only need to know that, if they call it within a loop, they will get 10 tuples at a time until there are no more tuples. In this example the tuples are merely printed, a page at a time. But it should be simple to change the functionality, for example by yielding after each retrieval, or by breaking when the tuples fail to match some additional criteria.

```

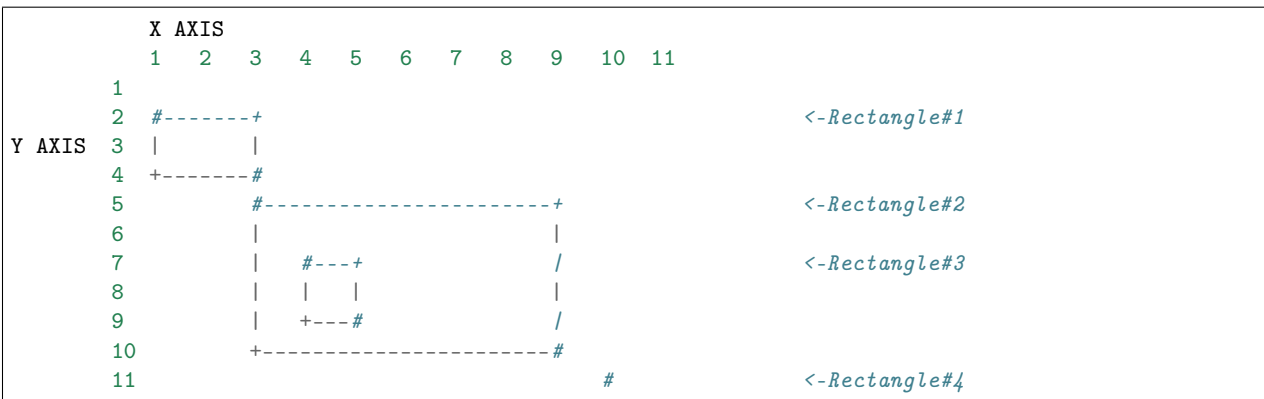
for page in paged_iter("X", 10) do
  print("New Page. Number Of Tuples = " .. #page)
  for i=1,#page,1 do print(page[i]) end
end

```

Submodule `box.index` with index type = RTREE for spatial searches

The `box.index` submodule may be used for spatial searches if the index type is RTREE. There are operations for searching *rectangles* (geometric objects with 4 corners and 4 sides) and *boxes* (geometric objects with more than 4 corners and more than 4 sides, sometimes called hyperrectangles). This manual uses the term *rectangle-or-box* for the whole class of objects that includes both rectangles and boxes. Only rectangles will be illustrated.

Rectangles are described according to their X-axis (horizontal axis) and Y-axis (vertical axis) coordinates in a grid of arbitrary size. Here is a picture of four rectangles on a grid with 11 horizontal points and 11 vertical points:



The rectangles are defined according to this scheme: {X-axis coordinate of top left, Y-axis coordinate of top left, X-axis coordinate of bottom right, Y-axis coordinate of bottom right} – or more succinctly: {x1,y1,x2,y2}. So in the picture ... Rectangle#1 starts at position 1 on the X axis and position 2 on the Y axis, and ends at position 3 on the X axis and position 4 on the Y axis, so its coordinates are {1,2,3,4}. Rectangle#2's coordinates are {3,5,9,10}. Rectangle#3's coordinates are {4,7,5,9}. And finally Rectangle#4's coordinates are {10,11,10,11}. Rectangle#4 is actually a "point" since it has zero width and zero height, so it could have been described with only two digits: {10,11}.

Some relationships between the rectangles are: "Rectangle#1's nearest neighbor is Rectangle#2 and "Rectangle#3 is entirely inside Rectangle#2".

Now let us create a space and add an RTREE index.

```

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'}
> })

```

Field#1 doesn't matter, we just make it because we need a primary-key index. (RTREE indexes cannot be unique and therefore cannot be primary-key indexes.) The second field must be an "array which means its values must represent {x,y} points or {x1,y1,x2,y2} rectangles. Now let us populate the table by inserting two tuples, containing the coordinates of Rectangle#2 and Rectangle#4.

```

tarantool> s:insert{1, {3, 5, 9, 10}}
tarantool> s:insert{2, {10, 11}}

```

And now, following the description of *RTREE iterator types*, we can search the rectangles with these requests:

```

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]]
...

```

Request#1 returns 1 tuple because the point {10,11} is the same as the rectangle {10,11,10,11} ("Rectangle#4" in the picture). Request#2 returns 1 tuple because the rectangle {4,7,5,9}, which was "Rectangle#3" in the picture, is entirely within {3,5,9,10} which was Rectangle#2. Request#3 returns 2 tuples, because the NEIGHBOR iterator always returns all tuples, and the first returned tuple will be {3,5,9,10} ("Rectangle#2" in the picture) because it is the closest neighbor of {1,2,3,4} ("Rectangle#1" in the picture).

Now let us create a space and index for cuboids, which are rectangle-or-boxes that have 6 corners and 6 sides.

```

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'}
> })

```

The additional field here is `dimension=3`. The default dimension is 2, which is why it didn't need to be specified for the examples of rectangle. The maximum dimension is 20. Now for insertions and selections there will usually be 6 coordinates. For example:

```

tarantool> s:insert{1, {0, 3, 0, 3, 0, 3}}
tarantool> r:select({1, 2, 1, 2, 1, 2}, {iterator = box.index.GT})

```

Now let us create a space and index for Manhattan-style spatial objects, which are rectangle-or-boxes that have a different way to calculate neighbors.

```
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'}
  > })
```

The additional field here is `distance='manhattan'`. The default distance calculator is 'euclid', which is the straightforward as-the-crow-flies method. The optional distance calculator is 'manhattan', which can be a more appropriate method if one is following the lines of a grid rather than traveling in a straight line.

```
tarantool> s:insert{1, {0, 3, 0, 3}}
tarantool> r:select({1, 2, 1, 2}, {iterator = box.index.NEIGHBOR})
```

More examples of spatial searching are online in the file [R tree index quick start and usage](#).

Вложенный модуль `box.session`

The `box.session` submodule allows querying the session state, writing to a session-specific temporary Lua table, or setting up triggers which will fire when a session starts or ends. A *session* is an object associated with each client connection.

`box.session.id()`

Return the unique identifier (ID) for the current session. The result can be 0 meaning there is no session.

Rtype number

`box.session.exists(id)`

Return 1 if the session exists, 0 if the session does not exist.

Rtype number

`box.session.peer(id)`

This function works only if there is a peer, that is, if a connection has been made to a separate server.

Return The host address and port of the session peer, for example "127.0.0.1:55457". If the session exists but there is no connection to a separate server, the return is null. The command is executed on the server, so the "local name" is the server's host and port, and the "peer name" is the client's host and port.

Rtype string

Possible errors: 'session.peer(): session does not exist'

`box.session.sync()`

Return the value of the `sync` integer constant used in the [binary protocol](#).

Rtype number

`box.session.storage`

A Lua table that can hold arbitrary unordered session-specific names and values, which will last until the session ends.

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.'
...

```

See the section [Triggers](#) for instructions about defining triggers for connect and disconnect events with `box.session.on_connect()` and `box.session.on_disconnect()`. See the section [Access control](#) for instructions about `box.session` functions that affect user identification and security.

Вложенный модуль `box.tuple`

The `box.tuple` submodule provides read-only access for the `tuple` userdata type. It allows, for a single tuple: selective retrieval of the field contents, retrieval of information about size, iteration over all the fields, and conversion to a Lua table.

`box.tuple.new(value)`

Construct a new tuple from either a scalar or a Lua table. Alternatively, one can get new tuples from tarantool's [select](#) or [insert](#) or [replace](#) or [update](#) requests, which can be regarded as statements that do `new()` implicitly.

Параметры

- `value (lua-value)` – the value that will become the tuple contents.

Return a new tuple

Rtype tuple

In the following example, `x` will be a new table object containing one tuple and `t` will be a new tuple object. Saying `t` returns the entire tuple `t`.

Example:

```

tarantool> x = box.space.testers: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>

The # operator in Lua means "return count of components". So, if `t` is a tuple instance, `#t` will return the number of fields.

Rtype number

In the following example, a tuple named `t` is created and then the number of fields in `t` is returned.

```

tarantool> t = box.tuple.new{'Fld#1', 'Fld#2', 'Fld#3', 'Fld#4'}
---
...
tarantool> #t
---
- 4
...

```

tuple_object:bsize()

If `t` is a tuple instance, `t:bsize()` will return the number of bytes in the tuple. It is useful to check this number when making changes to data, because there is a fixed maximum: one megabyte. Every field has one or more "length" bytes preceding the actual contents, so `bsize()` returns a value which is slightly greater than the sum of the lengths of the contents.

Return number of bytes

Rtype number

In the following example, a tuple named `t` is created which has three fields, and for each field it takes one byte to store the length and three bytes to store the contents, and a bit for overhead, so `bsize()` returns $3*(1+3)+1$.

```

tarantool> t = box.tuple.new{'aaa', 'bbb', 'ccc'}
---
...
tarantool> t:bsize()
---
- 13
...

```

<tuple_object>(field-number)

If `t` is a tuple instance, `t[field-number]` will return the field numbered `field-number` in the tuple. The first field is `t[1]`.

Return field value.

Rtype lua-value

In the following example, a tuple named `t` is created and then the second field in `t` is returned.

```

tarantool> t = box.tuple.new{'Fld#1', 'Fld#2', 'Fld#3', 'Fld#4'}
---
...
tarantool> t[2]
---
- Fld#2
...

```

`tuple_object:find([field-number], search-value)`

`tuple_object:findall([field-number], search-value)`

If `t` is a tuple instance, `t:find(search-value)` will return the number of the first field in `t` that matches the search value, and `t:findall(search-value [, search-value ...])` will return numbers of all fields in `t` that match the search value. Optionally one can put a numeric argument `field-number` before the search-value to indicate “start searching at field number `field-number`.”

Return the number of the field in the tuple.

Rtype number

In the following example, a tuple named `t` is created and then: the number of the first field in `t` which matches 'a' is returned, then the numbers of all the fields in `t` which match 'a' are returned, then the numbers of all the fields in `t` which match 'a' and are at or after the second field are returned.

```

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, ...])`

If `t` is a tuple instance, `t:transform(start-field-number, fields-to-remove)` will return a tuple where, starting from field `start-field-number`, a number of fields (`fields-to-remove`) are removed. Optionally one can add more arguments after `fields-to-remove` to indicate new values that will replace what was removed.

Параметры

- `start-field-number` (*integer*) – base 1, may be negative
- `fields-to-remove` (*integer*) –
- `field-value(s)` (*lua-value*) –

Return tuple

Rtype tuple

In the following example, a tuple named `t` is created and then, starting from the second field, two fields are removed but one new one is added, then the result is returned.

```
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]])`

If `t` is a tuple instance, `t:unpack()` will return all fields, `t:unpack(1)` will return all fields starting with field number 1, `t:unpack(1,5)` will return all fields between field number 1 and field number 5.

Return field(s) from the tuple.

Rtype lua-value(s)

In the following example, a tuple named `t` is created and then all its fields are selected, then the result is returned.

```
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:pairs()`

In Lua, `lua-table-value:pairs()` is a method which returns: `function, lua-table-value, nil`. Tarantool has extended this so that `tuple-value:pairs()` returns: `function, tuple-value, nil`. It is useful for Lua iterators, because Lua iterators traverse a value's components until an end marker is reached.

Return function, tuple-value, nil

Rtype function, lua-value, nil

In the following example, a tuple named `t` is created and then all its fields are selected using a Lua for-end loop.

```
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({format, field_number, value}, ...)`
Update a tuple.

This function updates a tuple which is not in a space. Compare the function `box.space.space-name:update{key, format, {field_number, value}...}`, which updates a tuple in a space.

Parameters: briefly: `format` indicates the type of update operation such as '=' for 'assign new value', `field_number` indicates the field number to change such as 2 for field number 2, `value` indicates the string which operates on the field such as 'B' for a new assignable value = 'B'.

For details: see the description for `format`, `field_number`, and `value` in the section [box.space.space-name:update{key, format, {field_number, value}...}](#).

Return new tuple

Rtype tuple

In the following example, a tuple named `t` is created and then its second field is updated to equal '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

This function will illustrate how to convert tuples to/from Lua tables and lists of scalars:

```

tuple = box.tuple.new({scalar1, scalar2, ... scalar_n}) -- scalars to tuple
lua_table = {tuple:unpack()} -- tuple to Lua table
scalar1, scalar2, ... scalar_n = tuple:unpack() -- tuple to scalars
tuple = box.tuple.new(lua_table) -- Lua table to tuple

```

Then it will find the field that contains 'b', remove that field from the tuple, and display how many bytes remain in the tuple. The function uses Tarantool `box.tuple` functions `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:unpack()}
  scalar1, scalar2, scalar3 = tuple1:unpack()
  tuple2 = box.tuple.new(luatable1)
  field_number = tuple2:find('b')
  tuple2 = tuple2:transform(field_number, 1)
  return 'tuple2 = ' , tuple2 , ' # of bytes = ' , tuple2:bsize()
end

```

... And here is what happens when one invokes the function:

```
tarantool> example()
---
- tuple2 =
- ['a', 'c']
- '# of bytes = '
- 5
...
```

Просмотр состояния сервера

Submodule *box.cfg*

The `box.cfg` submodule is for administrators to specify all the server configuration parameters; the full description of the parameters is in section [book-cfg](#). Use `box.cfg` without braces to get read-only access to those parameters.

Example:

```
tarantool> box.cfg
---
- snapshot_count: 6
  too_long_threshold: 0.5
  slab_alloc_factor: 1.1
  slab_alloc_maximal: 1048576
  background: false
  <...>
...
```

Submodule *box.info*

The `box.info` submodule provides access to information about server variables. Some important ones:

- **server.uid** holds the unique identifier of the server. This value is also in the `box.space._cluster` system space.
- **pid** is the process ID of the server. This value is also shown by the `tarantool` module.
- **version** is the Tarantool version. This value is also shown by `tarantool -version`.
- **uptime** is the number of seconds since the server started.

`box.info()`

Since `box.info` contents are dynamic, it's not possible to iterate over keys with the Lua `pairs()` function. For this purpose, `box.info()` builds and returns a Lua table with all keys and values provided in the submodule.

Return keys and values in the submodule.

Rtype table

Example:

```
tarantool> box.info()
---
- server:
```

```

lsn: 158
ro: false
uuid: a2684219-b2b1-4334-88ab-50b0722283fd
id: 1
version: 1.7.0-1216-g73f7154
pid: 12932
status: running
vclock:
- 158
replication:
  status: off
uptime: 908
...
tarantool> box.info.pid
---
- 12932
...
tarantool> box.info.status
---
- running
...
tarantool> box.info.uptime
---
- 1065
...
tarantool> box.info.version
---
- 1.7.0-66-g9093daa
...

```

Submodule *box.slab*

The `box.slab` submodule provides access to slab allocator statistics. The slab allocator is the main allocator used to store tuples. This can be used to monitor the total memory use and memory fragmentation.

The display of slabs is broken down by the slab size – 64-byte, 136-byte, and so on. The example omits the slabs which are empty. The example display is saying that: there are 16 items stored in the 64-byte slab (and $16 \cdot 64 = 1024$ so `bytes_used = 1024`); there is 1 item stored in the 136-byte slab (and $136 \cdot 1 = 136$ so `bytes_used = 136`); the `arena_used` value is the total of all the `bytes_used` values ($1024 + 136 = 1160$); the `arena_size` value is the `arena_used` value plus the total of all the `bytes_free` values ($1160 + 4193200 + 4194088 = 8388448$). The `arena_size` and `arena_used` values are the amount of the % of `slab_alloc_arena` that is already distributed to the slab allocator.

Example:

```

tarantool> box.slab.info().arena_used
---
- 4194304
...
tarantool> box.slab.info().arena_size
---
- 104857600
...
tarantool> box.slab.stats()
---
- - mem_free: 16248

```

```

mem_used: 48
item_count: 2
item_size: 24
slab_count: 1
slab_size: 16384
- mem_free: 15736
  mem_used: 560
  item_count: 14
  item_size: 40
  slab_count: 1
  slab_size: 16384
  <...>
...
tarantool> box.slab.stats()[1]
---
- mem_free: 15736
  mem_used: 560
  item_count: 14
  item_size: 40
  slab_count: 1
  slab_size: 16384
...

```

Submodule *box.stat*

The `box.stat` submodule provides access to request and network statistics. Show the average number of requests per second, and the total number of requests since startup, broken down by request type and network events statistics.

```

tarantool> type(box.stat), type(box.stat.net) -- virtual tables
---
- table
- table
...
tarantool> box.stat, box.stat.net
---
- net: []
- []
...
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 -- a selected item of the table
---
- total: 0
  rps: 0
...
tarantool> box.stat.net()
---
- SENT:
  total: 0
  rps: 0
EVENTS:
  total: 2
  rps: 0
LOCKS:
  total: 6
  rps: 0
RECEIVED:
  total: 0
  rps: 0
...

```

Служебные запросы

To learn which functions are considered to be administrative, type `help()`. A reference description also follows below:

`box.snapshot()`

Take a snapshot of all data and store it in `snap_dir/<latest-lsn>.snap`. To take a snapshot, Tarantool first enters the delayed garbage collection mode for all data. In this mode, tuples which were allocated before the snapshot has started are not freed until the snapshot has finished. To preserve consistency of the primary key, used to iterate over tuples, a copy-on-write technique is employed. If the master process changes part of a primary key, the corresponding process page is split, and the snapshot process obtains an old copy of the page. In effect, the snapshot process uses multi-version concurrency control in order to avoid copying changes which are superseded while it is running.

Since a snapshot is written sequentially, one can expect a very high write performance (averaging to 80MB/second on modern disks), which means an average database instance gets saved in a matter of minutes. Note: as long as there are any changes to the parent index memory through concurrent updates, there are going to be page splits, and therefore one needs to have some extra free memory to run this command. 10% of `slab_alloc_arena` is, on average, sufficient. This statement waits until a

snapshot is taken and returns operation result.

Change Notice: prior to Tarantool version 1.6.6, the snapshot process caused a fork, which could cause occasional latency spikes. Starting with Tarantool version 1.6.6, the snapshot process creates a consistent read view and writes this view to the snapshot file from a separate thread.

Although `box.snapshot()` does not cause a fork, there is a separate fiber which may produce snapshots at regular intervals – see the discussion of the *snapshot daemon*.

Example:

```
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)
...
```

Taking a snapshot does not cause the server to start a new write-ahead log. Once a snapshot is taken, old WALs can be deleted as long as all replicas are up to date. But the WAL which was current at the time `box.snapshot()` started must be kept for recovery, since it still contains log records written after the start of `box.snapshot()`.

An alternative way to save a snapshot is to send the server SIGUSR1 UNIX signal. While this approach could be handy, it is not recommended for use in automation: a signal provides no way to find out whether the snapshot was taken successfully or not.

`coredump()`

Fork and dump a core. Since Tarantool stores all tuples in memory, it can take some time. Mainly useful for debugging.

Атомарные операции

In several places in this manual it's been noted that Lua processes occur in fibers on a single thread. That is why there can be a guarantee of execution atomicity. That requires emphasis.

Среда взаимной многозадачности

Tarantool uses cooperative multitasking: unless a running fiber deliberately yields control, it is not preempted by some other fiber. But a running fiber will deliberately yield when it encounters a "yield point": an explicit `yield()` request, or an implicit yield due to an operating-system call. Any system call which can block will be performed asynchronously, and any running fiber which must wait for a system call will be preempted so that another ready-to-run fiber takes its place and becomes the new running fiber. This model makes all programmatic locks unnecessary: cooperative multitasking ensures that there will be no concurrency around a resource, no race conditions, and no memory consistency issues.

When requests are small, for example simple UPDATE or INSERT or DELETE or SELECT, fiber scheduling is fair: it takes only a little time to process the request, schedule a disk write, and yield to a fiber serving the next client.

However, a function might perform complex computations or might be written in such a way that yields do not occur for a long time. This can lead to unfair scheduling, when a single client throttles the rest of the system, or to apparent stalls in request processing. Avoiding this situation is the responsibility of the function's author. For the default memtx storage engine some of the box calls, including the data-change requests `box.space...insert` or `box.space...update` or `box.space...delete`, will usually cause yielding; however, `box.space...select` will not. A fuller description will appear in section *The Implicit Yield Rules*.

Note re storage engine: vinyl has different rules: insert or update or delete will very rarely cause a yield, but select can cause a yield.

In the absence of transactions, any function that contains yield points may see changes in the database state caused by fibers that preempt. Then the only safe atomic functions for memtx databases would be functions which contain only one database request, or functions which contain a select request followed by a data-change request.

At this point an objection could arise: "It's good that a single data-change request will commit and yield, but surely there are times when multiple data-change requests must happen without yielding." The standard example is the money-transfer, where \$1 is withdrawn from account #1 and deposited into account #2. If something interrupted after the withdrawal, then the institution would be out of balance. For such cases, the `begin ... commit|rollback` block was designed.

`box.begin()`

Begin the transaction. Disable implicit yields until the transaction ends. Signal that writes to the write-ahead log will be deferred until the transaction ends. In effect the fiber which executes `box.begin()` is starting an "active multi-request transaction blocking all other fibers.

`box.commit()`

End the transaction, and make all its data-change operations permanent.

`box.rollback()`

End the transaction, but cancel all its data-change operations. An explicit call to functions outside `box.space` that always yield, such as `fiber.yield` or `fiber.sleep`, will have the same effect.

The requests in a transaction must be sent to the server as a single block. It is not enough to enclose them between `begin` and `commit` or `rollback`. To ensure they are sent as a single block: put them in a function, or put them all on one line, or use a delimiter so that multi-line requests are handled together.

All database operations in a transaction should use the same storage engine. It is not safe to access tuple sets that are defined with `{engine='vinyl'}` and also access tuple sets that are defined with `{engine='memtx'}`, in the same transaction.

Пример

Assuming that in tuple set 'tester' there are tuples in which the third field represents a positive dollar amount ... Start a transaction, withdraw from tuple#1, deposit in tuple#2, and end the transaction, making its effects permanent.

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



```
- "ok"  
...
```

Правила неявной передачи управления

The only explicit yield requests are *fiber.sleep()* and *fiber.yield()*, but many other requests "imply" yields because Tarantool is designed to avoid blocking.

The implicit yield requests are: *insert replace update upsert delete* (the "data-change" requests), and functions in module *fib*, *net_box*, *console*, or *socket* (the "os" and "network" requests).

Note re storage engine: vinyl causes *select* to be an implicit yield request, but data-change requests may not be.

The yield occurs just before a blocking syscall, such as a write to the Write-Ahead Log (WAL) or a network message reception.

Implicit yield requests are disabled by *box.begin*, and enabled again by *commit*. Therefore the sequence

```
begin  
implicit yield request #1  
implicit yield request #2  
implicit yield request #3  
commit
```

will not cause implicit yield until the commit occurs (specifically: just before the writes to the WAL, which are delayed until commit time). The commit request is not itself an implicit yield request, it only enables yields caused by earlier implicit yield requests.

Despite their resemblance to implicit yield requests, *truncate* and *drop* do not cause implicit yield. Despite their resemblance to functions of the *fib* module, functions of the *os* module do not cause implicit yield. Despite its resemblance to commit, *rollback* does not enable yields.

If *wal_mode* = 'none', then implicit yielding is disabled, because there are no writes to the WAL.

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

```
select  
select  
select
```

causes blocking if it is inside a function or Lua program being executed on the server, but causes yielding if it is done as a series of transmissions from a client, including a client which operates via telnet, via one of the connectors, or via the MySQL and PostgreSQL rocks, or via the interactive mode when "*Using tarantool as a client*"@.

After a fiber has yielded and then has regained control, it immediately issues *testcancel*.

Ограничение доступа

Understanding the details of security is primarily an issue for administrators, but ordinary users should at least skim this section so that they will have an idea of how Tarantool makes it possible for administrators to prevent unauthorized access to the database and to certain functions.

Briefly: there is a method to guarantee with password checks that users really are who they say they are ("authentication"). There is a `_user` space where user names and password-hashes are stored. There are functions for saying that certain users are allowed to do certain things ("privileges"). There is a `_priv` space where privileges are stored. Whenever a user tries to do an operation, there is a check whether the user has the privilege to do the operation ("access control").

Passwords

Each user may have a password. The password is any alphanumeric string. Administrators should advise users to choose long unobvious passwords, but it is ultimately up to the users to choose or change their own passwords.

Tarantool passwords are stored in the `_user` space with a [Cryptographic hash function](#) so that, if the password is 'x', the stored hashed-password is a long string like '1L30vhkIP0Kh+Vn9Av1kx69M/Ck='. When a client connects to a Tarantool server, the server sends a random [Salt Value](#) which the client must mix with the hashed-password before sending to the server. Thus the original value 'x' is never stored anywhere except in the user's head, and the hashed value is never passed down a network wire except when mixed with a random salt. This system prevents malicious onlookers from finding passwords by snooping in the log files or snooping on the wire. It is the same system that [MySQL introduced several years ago](#) which has proved adequate for medium-security installations. Nevertheless administrators should warn users that no system is foolproof against determined long-term attacks, so passwords should be guarded and changed occasionally.

Примечание: To get the hash-password of a string 'X', say `box.schema.user.password('X')`. To see more about the details of the algorithm for the purpose of writing a new client application, read the [scramble.h](#) header file.

Users and the `_user` space

The fields in the `_user` space are:

- the numeric id of the tuple
- the numeric id of the tuple's creator
- the user name
- the type
- optional password

There are four special tuples in the `_user` space: 'guest', 'admin', 'public', and 'replication'.

Name	ID	Type	Description
guest	0	user	Default when connecting remotely. Usually an untrusted user with few privileges.
admin	1	user	Default when using <code>tarantool</code> as a console. Usually an administrative user with all privileges.
public	2	role	Not a user in the usual sense. Described later in section Roles .
replication	3	role	Not a user in the usual sense. Described later in section Roles .

To select a row from the `_user` space, use `box.space._user:select`. For example, here is what happens with a select for user id = 0, which is the 'guest' user, which by default has no password:

```
tarantool> box.space._user:select{0}
---
- - [0, 1, 'guest', 'user']
...

```

To change tuples in the `_user` space, do not use ordinary `box.space` functions for insert or update or delete - the `_user` space is special so there are special functions which have appropriate error checking.

To create a new user, say:

```
box.schema.user.create(user-name)
box.schema.user.create(user-name, {if_not_exists = true})
box.schema.user.create(user-name, {password = password}).
```

The `password=password` specification is good because in a [URI](#) (Uniform Resource Identifier) it is usually illegal to include a user-name without a password.

To change the user's password, say:

```
-- To change the current user's password
box.schema.user.passwd(password)

-- To change a different user's password
box.schema.user.passwd(user-name, password)
```

(Usually it is only the admin user who can change a different user's password.)

To drop a user, say:

```
box.schema.user.drop(user-name).
```

To check whether a user exists, say:

```
box.schema.user.exists(user-name)
```

which returns true or false.

To find what privileges a user has, say:

```
box.schema.user.info(user-name)
```

Example:

Here is a session which creates a new user with a strong password, selects a tuple in the `_user` space, and then drops the user.

```
tarantool> box.schema.user.create('JeanMartin', {password = 'Iwtso_6_os$$'})
---
...
tarantool> box.space._user.index.name:select{'JeanMartin'}
---
- - [17, 1, 'JeanMartin', 'user', {'chap-sha1': 't3xjUpQdrt8570+YRvGbMY5py8Q='}]
...
tarantool> box.schema.user.drop('JeanMartin')
---
...

```

Примечание: The maximum number of users is 32.

Privileges and the `_priv` space

The fields in the `_priv` space are:

- the numeric id of the user who gave the privilege (`"grantor_id"`),
- the numeric id of the user who received the privilege (`"grantee_id"`),
- the type of object - `"space"` or `"function"` or `"universe"`
- the numeric id of the object,
- the type of operation - `"read- 1, or "write- 2, or "execute- 4, or a combination such as "read,write,execute"`.

The function for granting a privilege is:

```
box.schema.user.grant(grantee, operation, object-type, object-name* [, *options])
-- OR
box.schema.user.grant(grantee, operation, 'universe' [, nil, options])
```

where `'universe'` means `'all objects'`, and the optional grant-option can be:

- `grantor=grantor_name_or_id` - string or number, for custom grantor
- `if_not_exists=true|false` - bool, do not throw error if user already has the privilege

The function for revoking a privilege is:

```
box.schema.user.revoke(grantee, operation, object-type, object-name* [, *options])
box.schema.user.revoke(grantee, operation, 'universe' [, nil, options])
```

where `'universe'` means `'all objects'`, and the optional grant-option can be:

- `if_not_exists=true|false` - bool, do not throw error if user already lacks the privilege

For example, here is a session where the admin user gave the guest user the privilege to read from a space named `space55`, and then took the privilege away:

```
tarantool> box.schema.user.grant('guest', 'read', 'space', 'space55')
---
...
tarantool> box.schema.user.revoke('guest', 'read', 'space', 'space55')
---
...
```

Примечание: Generally privileges are granted or revoked by the owner of the object (the user who created it), or by the `'admin'` user. Before dropping any objects or users, steps should be taken to ensure that all their associated privileges have been revoked.

Примечание: Only the `'admin'` user can grant privileges for the `'universe'`.

Примечание: Only the creator of a space can drop, alter, or truncate the space. Only the creator of a user can change a different user's password.

Functions and the `_func` space

The fields in the `_func` space are:

- the numeric function id, a number,
- the function name
- flag
- possibly a language name.

The `_func` space does not include the function's body. One continues to create Lua functions in the usual way, by saying `function function_name () ... end` without adding anything in the `_func` space. The `_func` space only exists for storing function tuples so that their names can be used within `grant/revoke` functions.

The function for creating a `_func` tuple is:

```
box.schema.func.create(function-name [, options])
```

The possible options are:

- `if_not_exists = true/false` - default = false,
- `setuid = true/false` - default = false,
- `language = 'LUA'/'C'` - default = 'LUA'.

Example:

```
box.schema.func.create('f', {language = 'C', setuid = false})
```

Specifying `if_not_exists=false` would cause error: Function `'...'` already exists if the `_func` tuple already exists.

Specifying `setuid=true` would cause the `setuid` flag (the fourth field in the `_func` tuple) to have a value meaning "true and the effect of that is that the function's caller is treated as the function's creator, with full privileges. The `setuid` behavior does not apply for users who connect via `console.connect`.

Specifying `language='C'` would cause the `language` field (the fifth field in the `_func` tuple) to have a value `'C'`, which means the function was written in C. Tarantool functions are normally written in Lua but can be written in C as well.

The function for dropping a `_func` tuple is:

```
box.schema.func.drop(function-name)
```

The function for checking whether a `_func` tuple exists is:

```
box.schema.func.exists(function-name)
```

In the following example, a function named `'f7'` is created, then it is put in the `_func` space, then it is used in a `box.schema.user.grant` function, then it is dropped:

```
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')
---
...
tarantool> box.schema.func.drop('f7')
---
...

```

box.session and security

After a connection has taken place, the user has access to a "session" object which has several functions. The ones which are of interest for security purposes are:

```

box.session.uid()          -- returns the id of the current user
box.session.user()        -- returns the name of the current user
box.session.su(user-name) -- allows changing current user to 'user-name'

```

If a user types requests directly on the Tarantool server in its *interactive mode*, or if a user connects to the *admin port*, then the user by default is 'admin' and has many privileges. If a user connects from an application program via one of the *connectors*, then the user by default is 'guest' and has few privileges. Typically an admin user will set up and configure objects, then grant privileges to appropriate non-admin users. Typically a guest user will use `box.session.su()` to change into a non-generic user to whom admin has granted more than the default privileges. For example, admin might say:

```

box.space._user:insert{123456,0,'manager','user'}
box.schema.user.grant('manager', 'read', 'space', '_space')
box.schema.user.grant('manager', 'read', 'space', 'payroll')

```

and later a guest user, who wishes to see the payroll, might say:

```

box.session.su('manager')
box.space.payroll:select{'Jones'}

```

Roles

A role is a container for privileges which can be granted to regular users. Instead of granting and revoking individual privileges, one can put all the privileges in a role and then grant or revoke the role. Role information is in the `_user` space but the third field - the type field - is 'role' rather than 'user'.

If a role R1 is granted a privilege X, and user U1 is granted a privilege "role R1 then user U1 in effect has privilege X. Then if a role R2 is granted a privilege Y, and role R1 is granted a privilege "role R2 then user U1 in effect has both privilege X and privilege Y. In other words, a user gets all the privileges that are granted to a user's roles, directly or indirectly.

```

box.schema.role.create(role-name [, {if_not_exists=true}])

```

Create a new role.

```

box.schema.role.grant(role-name, privilege)

```

Put a privilege in a role.

```

box.schema.role.revoke(role-name, privilege)

```

Take a privilege out of a role.

```
box.schema.role.drop(role-name)
```

Drop a role.

```
box.schema.role.grant(role-name, 'execute', 'role', role-name)
```

Grant a role to a role.

```
box.schema.role.revoke(role-name, 'execute', 'role', role-name)
```

Revoke a role from a role.

```
box.schema.role.exists(role-name)
```

Check whether a role exists. Returns (type = boolean) true if role-name identifies a role, otherwise false.

```
box.schema.user.grant(user-name, 'execute', 'role', role-name)
```

Grant a role to a user.

```
box.schema.user.revoke(user-name, 'execute', 'role', role-name)
```

Revoke a role from a user.

There are two predefined roles. The first predefined role, named 'public', is automatically assigned to new users when they are created with `box.schema.user.create(user-name)` - Therefore a convenient way to grant 'read' on space 't' to every user that will ever exist is: `box.schema.role.grant('public', 'read', 'space', 't')`. The second predefined role, named 'replication', can be assigned by the 'admin' user to users who need to use replication features.

Example showing a role within a role

In this example, a new user named U1 will insert a new tuple into a new space named T, and will succeed even though user U1 has no direct privilege to do such an insert – that privilege is inherited from role R1, which in turn inherits from role R2.

```
-- This example will work for a user with many privileges, such as 'admin'
box.schema.space.create('T')
box.space.T:create_index('primary', {})
-- Create a user U1 so that later it's possible to say box.session.su('U1')
box.schema.user.create('U1')
-- Create two roles, R1 and R2
box.schema.role.create('R1')
box.schema.role.create('R2')
-- Grant role R2 to role R1 and role R1 to U1 (order doesn't matter)
box.schema.role.grant('R1', 'execute', 'role', 'R2')
box.schema.user.grant('U1', 'execute', 'role', 'R1')
-- Grant read and execute privileges to R2 (but not to R1 and not to U1)
box.schema.role.grant('R2', 'read,write', 'space', 'T')
box.schema.role.grant('R2', 'execute', 'universe')
-- Use box.session.su to say "now become user U1"
box.session.su('U1')
-- Next insert succeeds because U1 in effect has write privilege on T
box.space.T:insert{1}
```

Триггеры

Triggers, also known as callbacks, are functions which the server executes when certain events happen. Currently the main types of triggers are *connection triggers*, which are executed when a session begins or ends, and *replace triggers* which are for database events.

All triggers have the following characteristics:

- They associate a *function* with an *event*. The request to "define a trigger" consists of passing the name of the trigger's function to one of the "*on_event-name* ()" functions: `on_connect()`, `on_auth()`, `on_disconnect()`, or `on_replace()`.
- They are *defined by any user*. There are no privilege requirements for defining triggers.
- They are called *after* the event. They are not called if the event ends prematurely due to an error. (Exception: `on_auth()` is called before the event.)
- They are in *server memory*. They are not stored in the database. Triggers disappear when the server is shut down. If there is a requirement to make them permanent, then the function definitions and trigger settings should be part of an initialization script.
- They have *low overhead*. If a trigger is not defined, then the overhead is minimal: merely a pointer dereference and check. If a trigger is defined, then its overhead is equivalent to the overhead of calling a stored procedure.
- They can be *multiple* for one event. Triggers are executed in the reverse order that they were defined in.
- They must work *within the event context*. If the function contains requests which normally could not occur immediately after the event but before the return from the event, effects are undefined. For example, putting `os.exit()` or `box.rollback()` in a trigger function would be bringing in requests outside the event context.
- They are *replaceable*. The request to "redefine a trigger" consists of passing the names of a new trigger function and an old trigger function to one of the "*on_event-name ...*" functions.

Connection triggers

`box.session.on_connect(trigger-function[, old-trigger-function-name])`

Define a trigger for execution when a new session is created due to an event such as [console.connect](#). The trigger function will be the first thing executed after a new session is created. If the trigger fails by raising an error, the error is sent to the client and the connection is closed.

Параметры

- `trigger-function` (*function*) – function which will become the trigger function
- `old-trigger-function-name` (*function*) – existing trigger function which will be replaced by `trigger-function`

Return nil or function list

If the parameters are (nil, old-trigger-function-name), then the old trigger is deleted.

Example:

```
tarantool> function f ()
>   x = x + 1
> end
tarantool> box.session.on_connect(f)
```

Предупреждение: If a trigger always results in an error, it may become impossible to connect to the server to reset it.

`box.session.on_disconnect(trigger-function[, old-trigger-function-name])`

Define a trigger for execution after a client has disconnected. If the trigger function causes an error,

the error is logged but otherwise is ignored. The trigger is invoked while the session associated with the client still exists and can access session properties, such as `box.session.id`.

Параметры

- `trigger-function` (*function*) – function which will become the trigger function
- `old-trigger-function-name` (*function*) – existing trigger function which will be replaced by `trigger-function`

Return `nil` or function list

If the parameters are (`nil`, `old-trigger-function-name`), then the old trigger is deleted.

Example:

```
tarantool> function f ()
>   x = x + 1
> end
tarantool> box.session.on_disconnect(f)
```

Пример

After the following series of requests, the server will write a message using the `log` module whenever any user connects or disconnects.

```
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)
```

Here is what might appear in the log file in a typical installation:

```
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
```

Authentication triggers

`box.session.on_auth(trigger-function[, old-trigger-function-name])`
 Define a trigger for execution during authentication.

The `on_auth` trigger function is invoked in these circumstances: (1) The `console.connect` function includes an authentication check for all users except 'guest'; for this case the `on_auth` trigger function is invoked after the `on_connect` trigger function, if and only if the connection has succeeded so far. (2) The binary protocol has a separate `authentication packet` – for this case, connection and authentication are considered to be separate steps.

Unlike other trigger types, `on_auth` trigger functions are invoked *before* the event. Therefore a trigger function like `function auth_function () v = box.session.user(); end` will set `v` to "guest" the user name before the authentication is done. To get the user name after the authentication is done, use the special syntax: `function auth_function (user_name) v = user_name; end`

If the trigger fails by raising an error, the error is sent to the client and the connection is closed.

Параметры

- `trigger-function` (*function*) – function which will become the trigger function
- `old-trigger-function-name` (*function*) – existing trigger function which will be replaced by `trigger-function`

Return nil

If the parameters are (nil, `old-trigger-function-name`), then the old trigger is deleted.

Example:

```
tarantool> function f ()
  >   x = x + 1
  > end
tarantool> box.session.on_auth(f)
```

Replace triggers

object `space_object`

`space_object:on_replace`(*trigger-function*[, *old-trigger-function-name*])

Create a "replace trigger". The `function-name` will be executed whenever a `replace()` or `insert()` or `update()` or `upsert()` or `delete()` happens to a tuple in `<space-name>`.

Параметры

- `trigger-function` (*function*) – function which will become the trigger function
- `old-trigger-function-name` (*function*) – existing trigger function which will be replaced by `trigger-function`

Return nil or function list

If the parameters are (nil, `old-trigger-function-name`), then the old trigger is deleted.

Example:

```
tarantool> function f ()
  >   x = x + 1
  > end
tarantool> box.space.X:on_replace(f)
```

`space_object:run_triggers`(*true/false*)

At the time that a trigger is defined, it is automatically enabled - that is, it will be executed. Replace triggers can be disabled with `box.space.space-name:run_triggers(false)` and re-enabled with `box.space.space-name:run_triggers(true)`.

Return nil

Example:

```
tarantool> box.space.X:run_triggers(false)
```

Пример

The following series of requests will create a space, create an index, create a function which increments a counter, create a trigger, do two inserts, drop the space, and display the counter value - which is 2, because the function is executed once after each insert.

```
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
```

Another Example

The following series of requests will associate an existing function named F with an existing space named T, associate the function a second time with the same space (so it will be called twice), disable all triggers of T, and delete each trigger by replacing with 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)
```

Getting a list of triggers

The code `on_connect()` – with no arguments – returns a table of all connect-trigger functions; `on_auth()` returns all authentication-trigger functions; `on_disconnect()` returns all disconnect-trigger functions; `on_replace()` returns all replace-trigger functions. In the following example a user finds that there are three functions associated with `on_connect` triggers, and executes the third function, which happens to contain the line `print('function #3')`. Then it deletes the third trigger.

```
tarantool> box.session.on_connect()
---
- - 'function: 0x416ab6f8'
- - 'function: 0x416ab6f8'
- - 'function: 0x416ad800'
...

tarantool> box.session.on_connect()[3]()
function #3
---
...

tarantool> box.session.on_connect(nil, box.session.on_connect()[3])
```



Ограничения

Number of parts in an index

For TREE or HASH indexes, the maximum is 255 (`box.schema.INDEX_PART_MAX`). For RTREE indexes, the maximum is 1 but the field is an ARRAY. For BITSET indexes, the maximum is 1.

Number of indexes in a space

128 (`box.schema.INDEX_MAX`).

Number of fields in a tuple

The theoretical maximum is 2147483647 (`box.schema.FIELD_MAX`). The practical maximum is whatever is specified by the space's *field_count* member, or the maximum tuple length.

Number of bytes in a tuple

By default the value of *slab_alloc_maximal* is 1048576, and the maximum tuple length is approximately one quarter of that: approximately 262,000 bytes. To increase it, when starting the server, specify a larger value. For example `box.cfg{slab_alloc_maximal=2*1048576}`.

Number of bytes in an index key

If a field in a tuple can contain a million bytes, then the index key can contain a million bytes, so the maximum is determined by factors such as *Number of bytes in a tuple*, not by the index support.

Number of spaces

The theoretical maximum is 2147483647 (`box.schema.SPACE_MAX`).

Number of connections

The practical limit is the number of file descriptors that one can set with the operating system.

Space size

The total maximum size for all spaces is in effect set by *slab_alloc_arena*, which in turn is limited by the total available memory.

Update operations count

The maximum number of operations that can be in a single update is 4000 (`BOX_UPDATE_OP_CNT_MAX`).

Number of users and roles

32 (`BOX_USER_MAX`).

Length of an index name or space name or user name

32 (`box.schema.NAME_MAX`).

Number of replicas in a cluster

32 (`box.schema.REPLICA_MAX`).

For additional limitations which apply only to the vinyl storage engine, see section *Differences between memtx and vinyl*.

Различия между движками memtx и vinyl

The primary difference between memtx and vinyl is that memtx is an "in-memory" engine while vinyl is an "on-disk" engine. An in-memory storage engine is generally faster, and the memtx engine is justifiably the default for Tarantool, but there are two situations where an on-disk engine such as vinyl would be preferable:

1. when the database is larger than the available memory and adding more memory is not a realistic option;
2. when the server frequently goes down due to errors or a simple desire to save power – bringing the server back up and restoring a memtx database into memory takes time.

Here are behavior differences which affect programmers. All of these differences have been noted elsewhere in sentences that begin with the words "Note re storage engine: vinyl".

With memtx, the index type can be TREE or HASH or RTREE or BITSET. With vinyl, the only index type is TREE.

With memtx, `create_index` can be done at any time. With vinyl, secondary indexes must be created before tuples are inserted.

With memtx, for index searches, `nil` is considered to be equal to any scalar. With vinyl, `nil` or missing parts are not allowed.

With memtx, temporary spaces are supported. With vinyl, they are not.

With memtx, the `alter()` and `len()` and `random()` and `auto_increment()` and `truncate()` functions are supported. With vinyl, they are not.

With memtx, the `count()` function takes a constant amount of time. With vinyl, it takes a variable amount of time depending on index size.

With memtx, `delete` will return deleted tuple, if any. With vinyl, `delete` will always return `nil`.

It was explained *earlier* that memtx does not "yield" on a select request, it yields only on data-change requests. However, vinyl does yield on a select request, or on an equivalent such as `get()` or `pairs()`. This has significance for *cooperative multitasking*.

For more about vinyl, see Appendix D *vinyl*.

3.4 Репликация

Механизм репликации позволяет сразу многим Tarantool-серверам работать с копиями одних и тех же баз данных. При этом все базы остаются в синхронизированном состоянии благодаря тому, что каждый сервер может сообщать другим серверам о совершенных им изменениях. Сервера, которые работают над одними и теми же базами, представляют собой "кластер". У каждого сервера в кластере есть числовой идентификатор (`server id`), уникальный в рамках кластера.

Чтобы настроить репликацию, необходимо настроить главные сервера (`master`), которые первыми обрабатывают запросы на изменение данных, затем настроить сервера-реплики (`replica`), которые копируют к себе запросы на изменение данных с главных серверов, и прописать процедуры для восстановления после сбоя.

3.4.1 Архитектура механизма репликации

Чтобы знать о всех изменениях на стороне главного сервера, каждая реплика непрерывно опрашивает главный сервер на предмет обновлений в его WAL-файле (`write ahead log`) и применяет эти обновления на своей стороне. Каждая запись в WAL-файле представляет собой один запрос на изменение данных

(например, INSERT, UPDATE или DELETE) и присвоенный данной записи номер (LSN = log sequence number). Номера присваиваются в порядке возрастания. По сути, репликация в Tarantool'е является построчной: все команды на изменение данных полностью детерминированы, и каждая такая команда относится только к одному таплу.

Вызовы хранимых Lua-процедур фиксируются не в WAL-файле, а в журнале событий (event log). Таким образом гарантируется, что не детерминированное поведение логики на Lua не приведет к рассинхронизации реплицированных данных.

3.4.2 Настройка главного сервера

Чтобы настроить возможность установки соединения для реплик, на стороне главного сервера требуется лишь указать значение для параметра `listen@"listen"` в init-запросе `box.cfg`. Например, `box.cfg{listen=3301}`. Когда URI для прослушивания задан, главный сервер готов принимать запросы на соединение от любого количества реплик. Каждая реплика при этом находится в некотором *etamuse репликации*.

3.4.3 Настройка сервера-реплики

Каждому Tarantool-серверу необходим корректный файл со статическим снимком данных (.snap-файл). Файл-снимок создается на сервере при первом запросе `box.cfg`. Если при первом таком запросе на сервере не определен источник репликации (replication source), то сервер стартует в режиме главного сервера и создает для себя новый кластер с новым уникальными UUID. Если же источник репликации при первом `box.cfg`-запросе определен, то сервер стартует в режиме реплики, а файл-снимок и информация о кластере берутся из WAL-файлов на главном сервере. Поэтому при настройке репликации нужно указать параметр `replication_source` в запросе `box.cfg`. При первом соединении с главным сервером сервер-реплика включается в состав кластера. В дальнейшем такая реплика общается только с главным сервером из данного кластера.

После установки соединения с главным сервером реплика запрашивает у него все изменения, чьи LSN-номера в WAL-файле больше номера последнего локального изменения на реплике. Поэтому WAL-файлы на главном сервере нужно хранить до тех пор, пока все реплики не применят изменения из этих WAL-файлов на своей стороне. Состояние реплики можно "обнулить" удалив все файлы репликации (.snap-файл со снимком и .xlog-файлы с записями WAL) и запустив сервер снова. Реплика при этом возьмет все таплы с главного сервера и придет в синхронизированное состояние. Обратите внимание, что такая процедура "обнуления" работает, только если на главном сервере будут доступны все нужные WAL-файлы.

Примечание: Параметры репликации можно менять на лету, что позволяет назначать реплику на роль главного сервера и наоборот. Для этого используется запрос `box.cfg`.

Примечание: Реплика не берет настройки конфигурации с главного сервера, например настройки запуска *фоновой программы для работы со снимками* на главном сервере. Чтобы получить те же настройки на реплике, нужно задать их явным образом.

Примечание: Репликация требует настройки привилегий. Привилегии для доступа к спейсам можно задать напрямую для пользователя, под чьим именем запущен сервер-реплика. Но обычно привилегии на доступ к спейсам задаются с помощью *роли*, которая затем присваивается пользователю, под чьим именем запущен сервер-реплика.

3.4.4 Восстановление после сбоя

"Сбойто ситуация, когда главный сервер становится недоступен вследствие проблем с оборудованием, сетевых неполадок или программной ошибки. У реплики нет способа автоматически обнаружить, что связь с главным сервером утеряна насовсем, поскольку причины сбоя и окружение, в котором развернута репликация, могут быть очень разными. Поэтому обнаруживать сбой должен человек.

Но когда сбой уже обнаружен, процедура восстановления проста. Сначала нужно назначить одну из реплик на роль нового главного сервера, задав параметр `box.cfg{... listen=URI <index-uri>}`. Затем, если на прежнем главном сервере остались изменения, которые не успели примениться на выбранной реплике перед сбоем, нужно применить эти изменения вручную.

3.4.5 Инструкции по быстрому запуску простого кластера из двух серверов с нуля

Шаг 1. Запустите первый сервер со следующими настройками:

```
box.cfg{listen = *uri#1*}
-- в этом запросе можно задать больше ограничений
box.schema.user.grant('guest', 'read,write,execute', 'universe')
box.snapshot()
```

... Итак, созданся новый кластер.

Шаг 2. На втором сервере проверьте пути, по которым будут храниться файлы репликации. Эти пути задаются в параметрах `snap_dir` (для `.snap`-файлов) и `wal_dir` (для `.xlog`-файлов). В указанных директориях должно быть пусто, чтобы не случилось конфликта с теми начальными данными, что придут с первого сервера, когда второй сервер присоединится к кластеру.

Step 3. Запустите второй сервер со следующими настройками:

```
box.cfg{
  listen = uri#2,
  replication_source = uri#1
}
```

... где `uri#1` = `URI`, на котором включено прослушивание у первого сервера.

Вот и всё.

В описанной выше конфигурации первый сервер выполняет роль "главного а второй служит "репликой". Далее все изменения, происходящие на стороне главного сервера, будут доступны с реплики. Простой кластер из двух серверов, где главный сервер запущен на одном компьютере, а сервер-реплика – на другом, встречается очень часто и обладает двумя важными преимуществами: `FAILOVER` (т.е. отказоустойчивость, поскольку в случае отключения главного сервера его место может занять сервер-реплика) и `LOAD BALANCING` (т.е. балансировка нагрузки, поскольку клиенты могут обращаться с `SELECT`-запросами как к главному серверу, так и к реплике). При необходимости в настройках реплики можно задать параметр `read_only = true`.

3.4.6 Мониторинг действий реплики

В `box.info` есть поле `box.info.replication.status`, которе может принимать следующие значения: "off" "stopped" "connecting" "auth" "follow или "disconnected". Если реплика имеет статус "follow то для нее выставляются значение еще двух полей: `box.info.replication.idle` = количество секунд, которые реплика провела в состоянии бездействия, `box.info.replication.lag` = количество секунд, на которое реплика отстает от главного сервера.

В *журнале* ведется запись о действиях, связанных с репликацией. Если главный сервер запущен со следующими настройками:

```
box.cfg{
  <...>,
  logger = *имя_файла_для_ведения_журнала*,
  <...>
}
```

то на каждую установку/потерю соединения реплики с главным сервером в журнале будут появляться строчки со словом "relay".

3.4.7 Предотвращение дублирующихся действий

Предположим, что реплика пытается сделать нечто, что уже было сделано на главном сервере. Например: `box.schema.space.create('X')` Это приведет к ошибке "Space X exists" ("Спейс X уже существует"). В данном частном случае можно скорректировать инструкцию следующим образом: `box.schema.space.create('X', {if_not_exists=true})` Но существует и более универсальное решение: использовать метод `box.once(key, function)`. Если `box.once()` был вызван ранее с тем же значением параметра *key*, то функция *function* игнорируется; в противном случае функция *function* будет выполнена. Поэтому действия, которые должны совершаться только один раз за время текущей сессии репликации, нужно помещать в функцию и вызывать ее с помощью метода `box.once()`. Например:

```
function f()
  box.schema.space.create('X')
end
box.once('space_creator', f)
```

3.4.8 Репликация по схеме master-master

В случае настройки репликации по схеме master-replica изменения на главном сервере доступны для просмотра с реплики, но не наоборот, потому как главный сервер в такой схеме указан в качестве единственного источника репликации. В случае схемы master-master (иногда ее также называют multi-master) просмотр изменений возможен в любом направлении. В простом случае (master-master с двумя серверами) на первом сервере нужно задать следующие настройки:

```
box.cfg{ replication_source = uri#2 }
```

Этот запрос можно выполнить в любой момент, т.к. параметр `replication_source` можно задавать на ходу.

В данном примере оба сервера являются одновременно и "главными" и "репликами". Поэтому каждое изменение, которое случается на одном сервере, становится доступно для просмотра с другого сервера. Отказоустойчивость в такой конфигурации сохраняется, а возможности по балансировке нагрузки становятся еще шире (теперь клиенты могут обращаться к обоим серверам со всеми типами запросов – как на чтение данных, так и на изменение).

Если две операции над одним и тем же таплом производятся "параллельно" (а это может потребовать много времени, поскольку репликация – это асинхронный процесс), причем одна из операций – это `delete`, а вторая – `replace`, то существует вероятность, что данные на серверах станут различаться.

3.4.9 Ответы на вопросы "Что если?"

Вопрос: Что если в кластере вида master-master более двух серверов? Ответ: На каждом сервере нужно задать параметр `replication_source` и указать в нем все остальные сервера. Например, для сервера #3 настройки будут следующими: `box.cfg{ replication_source = {uri#1, uri#2} }`

Вопрос: Что если какой-то сервер нужно убрать из кластера? Ответ: Для реплики – выполните запрос `box.cfg{}`, указав пустой источник репликации: `box.cfg{replication_source=''}`

Вопрос: Что если какой-то сервер вдруг выбывает из кластера? Ответ: Остальные сервера продолжают работать. Если выбывший сервер снова возвращается в кластер, то он получит информацию о всех изменениях, которые произошли на остальных серверах за время его отсутствия.

Вопрос: Что если два сервера совершают изменения, связанные с одним и тем же таплом? Ответ: Применяются последние из совершенных изменений. Для примера предположим, что сервер #1 меняет некоторый тапл, а затем сервер #2 меняет тот же тапл. В данном случае изменения сервера #2 затрут изменения сервера #1. Чтобы отслеживать, кто пришел последним, в Tarantool'e используются [векторные часы](#).

Вопрос: Что если оба сервера выполняют вставку одного и того же тапла? Ответ: Если главный сервер попытается вставить тапл, который уже был вставлен на реплике, то мы получим пример серьезной ошибки. Репликация остановится, и ее придется перезапустить в ручную.

Вопрос: Что если главный сервер становится недоступен и пользователям приходится переключаться реплику? Ответ: Реплика получает сообщение, что связь потеряна. Теперь реплика должна начать работать независимо. Для этого ей нужно задать пустой источник репликации, выполнив на стороне реплики запрос `box.cfg{replication_source=''}`.

Вопрос: Что если нужно посмотреть, к какому кластеру принадлежит данный сервер? Ответ: Идентификатором кластера является UUID, который генерируется при первом запуске главного сервера. Данный UUID хранится в системном спейсе `box.space._schema`. Чтобы посмотреть UUID кластера, введите запрос `box.space._schema:select{'cluster'}`

Вопрос: Что если нужно посмотреть, какие сервера входят в кластер? Ответ: У каждого сервера есть универсальный идентификатор – это его UUID в поле `box.info.server.uuid`. Также у сервера есть его порядковый идентификатор в кластере – это номер в поле `box.info.server.id`. Чтобы увидеть номера всех серверов в кластере, введите запрос: `box.space._cluster:select{}`. Данный запрос возвращает таблицу со всеми таплами вида `{server.id, server.uuid}` для всех серверов, что когда-либо входили в данный кластер.

Вопрос: Что если какой-то из файлов репликации на реплике поврежден или удален? Ответ: Нужно остановить сервер, удалить все файлы, относящиеся к базе данных (это файлы с расширениями "snap", "xlog" и ".inprogress"), снова запустить сервер и ввести запрос `box.cfg{...replication_source=...}`, чтобы восстановить соединение с главным сервером и загрузить данные с него.

Вопрос: Что если при репликации возникают вопросы, связанные с безопасностью? Ответ: Чтобы предотвратить появление несанкционированных источников репликации, нужно задать пароль для каждого пользователя, у которого есть привилегии доступа к соответствующим спейсам, а также для каждого пользователя, у которого настроена [репликационная роль](#). Заметьте, что `URI` для параметра `replication_source` теперь нужно всегда указывать в полном виде: `replication_source='username:password@host:port'`

Вопрос: Что если продвинутые пользователи хотят глубже разобраться с тем, как работает репликация? Ответ: См. информацию о запуске сервера с репликацией в Приложении В. [Детали реализации](#).

3.4.10 Практическое руководство по репликации

Ниже приводятся пошаговые инструкции, которые помогут вам получить практический опыт администрирования кластера, а именно опыт создания кластера и добавления реплики.

Запустите два терминала, каждый в своем окне, и расположите их рядом на экране. (Далее в примерах оба терминала показаны в виде закладок. Щелкните на заголовке закладки – "Terminal #1" или "Terminal #2 – чтобы увидеть вывод на соответствующем терминале.)

```
$
```

```
$
```

В первом терминале (Terminal #1) выполните следующие команды:

```
$ # Terminal 1
$ mkdir -p ~/tarantool_test_node_1
$ cd ~/tarantool_test_node_1
$ rm -R ~/tarantool_test_node_1/*
$ ~/tarantool/src/tarantool
tarantool> box.cfg{listen = 3301}
tarantool> box.schema.user.create('replicator', {password = 'password'})
tarantool> box.schema.user.grant('replicator','execute','role','replication')
tarantool> box.space._cluster:select({0}, {iterator = 'GE'})
```

В результате были заданы настройки нового кластера, а на экране был выведен UUID текущего сервера. Теперь вывод на экране выглядит следующим образом (за тем исключением, что UUID у вас будут другие):

```
$ # Terminal 1
$ mkdir -p ~/tarantool_test_node_1
$ cd ~/tarantool_test_node_1
$ rm -R ./*
$ ~/tarantool/src/tarantool
$ ~/tarantool/src/tarantool: version 1.7.0-1724-g033ed69
type 'help' for interactive help
tarantool> box.cfg{listen = 3301}
<... ...>
tarantool> box.schema.user.create('replicator', {password = 'password'})
<...> I> creating ./00000000000000000000000000000000.xlog.inprogress'
---
...
tarantool> box.schema.user.grant('replicator','execute','role','replication')
---
...
tarantool> box.space._cluster:select({0}, {iterator = 'GE'})
---
- - [1, '6190d919-1133-4452-b123-beca0b178b32']
...

```

```
$
```

Во втором терминале (Terminal #2) выполните следующие команды:

```
$ # Terminal 2
$ mkdir -p ~/tarantool_test_node_2
$ cd ~/tarantool_test_node_2
```

```

$ rm -R ~/tarantool_test_node_2/*
$ ~/tarantool/src/tarantool
tarantool> box.cfg{
    > listen = 3302,
    > replication_source = 'replicator:password@localhost:3301'
    > }
tarantool> box.space._cluster:select({0}, {iterator = 'GE'})

```

В результате были заданы настройки сервера-реплики. На экране первого терминала (Terminal #1) появились сообщения с подтверждениями, что реплика установила соединение с главным сервером и что содержимое WAL-файла было отправлено на реплику. На экране второго терминала (Terminal #2) появились сообщения о том, что репликация начинается, а также там были выведены UUID из системного спейса `_cluster` (один из них совпадает с UUID в первом терминале, поскольку оба сервера входят в общий кластер).

```

$ # Terminal 1
$ mkdir -p ~/tarantool_test_node_1
<... ...>
tarantool> box.space._cluster:select({0}, {iterator = 'GE'})
---
- - [1, '6190d919-1133-4452-b123-beca0b178b32']
...
tarantool>
<...> [11031] relay/127.0.0.1:58734/101/main I> recovery start
<...> [11031] relay/127.0.0.1:58734/101/main I> recovering from `./00000000000000000000.snap'
<...> [11031] relay/127.0.0.1:58734/101/main I> snapshot sent
<...> [11031] relay/127.0.0.1:58734/101/main I> recover from `./00000000000000000000.xlog'
<...> [11031] relay/127.0.0.1:58734/101/main recovery.cc:211 W> file
`./00000000000000000000.xlog` wasn't correctly closed
<...> [11031] relay/127.0.0.1:58734/102/main I> recover from `./00000000000000000000.xlog'

```

```

$ # Terminal 2
$ mkdir -p ~/tarantool_test_node_2
$ cd ~/tarantool_test_node_2
$ rm -R ./*
$ ~/tarantool/src/tarantool
/home/username/tarantool/src/tarantool: version 1.7.0-1724-g033ed69
type 'help' for interactive help
tarantool> box.cfg{
    > listen = 3302,
    > replication_source = 'replicator:password@localhost:3301'
    > }
<...>
<...> [11243] main/101/interactive I> mapping 1073741824 bytes for tuple arena...
<...> [11243] main/101/interactive C> starting replication from localhost:3301
<...> [11243] main/102/applier/localhost:3301 I> connected to 1.7.0 at 127.0.0.1:3301
<...> [11243] main/102/applier/localhost:3301 I> authenticated
<...> [11243] main/102/applier/localhost:3301 I> downloading a snapshot from 127.0.0.1:3301
<...> [11243] main/102/applier/localhost:3301 I> done
<...> [11243] snapshot/101/main I> creating `./00000000000000000000.snap.inprogress'
<...> [11243] snapshot/101/main I> saving snapshot `./00000000000000000000.snap.inprogress'
<...> [11243] snapshot/101/main I> done
<...> [11243] iproto I> binary: started
<...> [11243] iproto I> binary: bound to 0.0.0.0:3302
<...> [11243] wal/101/main I> creating `./00000000000000000000.xlog.inprogress'
<...> [11243] main/101/interactive I> ready to accept requests
tarantool> box.space._cluster:select({0}, {iterator = 'GE'})

```

```
- - [1, '6190d919-1133-4452-b123-beca0b178b32']
- - [2, '236230b8-af3e-406b-b709-15a60b44c20c']
...

```

В первом терминале (Terminal #1) выполните следующие запросы:

```
tarantool> s = box.schema.space.create('tester')
tarantool> i = s:create_index('primary', {})
tarantool> s:insert{1, 'Tuple inserted on Terminal #1'}
```

Теперь вывод на экране выглядит следующим образом:

```
<... ...>
tarantool>
tarantool>
<...> [11031] relay/127.0.0.1:58734/101/main I> recovery start
<...> [11031] relay/127.0.0.1:58734/101/main I> recovering from `./00000000000000000000.snap'
<...> [11031] relay/127.0.0.1:58734/101/main I> snapshot sent
<...> [11031] relay/127.0.0.1:58734/101/main I> recover from `./00000000000000000000.xlog'
<...> [11031] relay/127.0.0.1:58734/101/main recovery.cc:211 W> file
    `./00000000000000000000.xlog` wasn't correctly closed
<...> [11031] relay/127.0.0.1:58734/102/main I> recover from `./00000000000000000000.xlog'
tarantool> s = box.schema.space.create('tester')
---
...
tarantool> i = s:create_index('primary', {})
---
...
tarantool> s:insert{1, 'Tuple inserted on Terminal #1'}
---
- [1, 'Tuple inserted on Terminal #1']
...

```

```
$ # Terminal 2
$ mkdir -p ~/tarantool_test_node_2
$ cd ~/tarantool_test_node_2
$ rm -R ./ *
$ ~/tarantool/src/tarantool
/home/username/tarantool/src/tarantool: version 1.7.0-1724-g033ed69
type 'help' for interactive help
tarantool> box.cfg{
  > listen = 3302,
  > replication_source = 'replicator:password@localhost:3301'
  > }
<...>
<...> [11243] main/101/interactive I> mapping 1073741824 bytes for tuple arena...
<...> [11243] main/101/interactive C> starting replication from localhost:3301
<...> [11243] main/102/applier/localhost:3301 I> connected to 1.7.0 at 127.0.0.1:3301
<...> [11243] main/102/applier/localhost:3301 I> authenticated
<...> [11243] main/102/applier/localhost:3301 I> downloading a snapshot from 127.0.0.1:3301
<...> [11243] main/102/applier/localhost:3301 I> done
<...> [11243] snapshot/101/main I> creating `./00000000000000000000.snap.inprogress'
<...> [11243] snapshot/101/main I> saving snapshot `./00000000000000000000.snap.inprogress'
<...> [11243] snapshot/101/main I> done
<...> [11243] iproto I> binary: started
<...> [11243] iproto I> binary: bound to 0.0.0.0:3302
<...> [11243] wal/101/main I> creating `./00000000000000000000.xlog.inprogress'
<...> [11243] main/101/interactive I> ready to accept requests

```

```
tarantool> box.space._cluster:select({0}, {iterator = 'GE'})
- - [1, '6190d919-1133-4452-b123-beca0b178b32']
- - [2, '236230b8-af3e-406b-b709-15a60b44c20c']
...
```

В первом терминале успешно отработали операции CREATE и INSERT. Но во втором терминале ничего не произошло.

Во втором терминале (Terminal #2) выполните следующие запросы:

```
tarantool> s = box.space.testner
tarantool> s:select({1}, {iterator = 'GE'})
tarantool> s:insert{2, 'Tuple inserted on Terminal #2'}
```

Теперь вывод на экране выглядит следующим образом:

```
<... ...>
tarantool>
tarantool>
<...> [11031] relay/127.0.0.1:58734/101/main I> recovery start
<...> [11031] relay/127.0.0.1:58734/101/main I> recovering from `./00000000000000000000.snap'
<...> [11031] relay/127.0.0.1:58734/101/main I> snapshot sent
<...> [11031] relay/127.0.0.1:58734/101/main I> recover from `./00000000000000000000.xlog'
<...> [11031] relay/127.0.0.1:58734/101/main recovery.cc:211 W> file
`./00000000000000000000.xlog` wasn't correctly closed
<...> [11031] relay/127.0.0.1:58734/102/main I> recover from `./00000000000000000000.xlog'
tarantool> s = box.schema.space.create('tester')
---
...
tarantool> i = s:create_index('primary', {})
---
...
tarantool> s:insert{1, 'Tuple inserted on Terminal #1'}
---
- [1, 'Tuple inserted on Terminal #1']
...
```

```
<... ...>
tarantool> box.space._cluster:select({0}, {iterator = 'GE'})
- - [1, '6190d919-1133-4452-b123-beca0b178b32']
- - [2, '236230b8-af3e-406b-b709-15a60b44c20c']
...
tarantool> s = box.space.testner
---
...
tarantool> s:select({1}, {iterator = 'GE'})
---
- - [1, 'Tuple inserted on Terminal #1']
...
tarantool> s:insert{2, 'Tuple inserted on Terminal #2'}
---
- [2, 'Tuple inserted on Terminal #2']
...
```

Во втором терминале успешно отработали операции SELECT и INSERT. Но в первом терминале ничего не произошло.

В первом терминале (Terminal #1) выполните следующие запросы и команды:

```
$ os.exit()
$ ls -l ~/tarantool_test_node_1
$ ls -l ~/tarantool_test_node_2
```

Теперь Tarantool-сервер в первом терминале остановлен. В окне второго терминала появились сообщения об этом событии. С помощью команд `ls -l` мы убедились, что на обоих серверах создались файлы-снимки с одинаковыми размерами, поскольку там содержатся одни и те же таплы.

```
<... ...>
tarantool> s:insert{1, 'Tuple inserted on Terminal #1'}
---
- [1, 'Tuple inserted on Terminal #1']
...
$ ls -l ~/tarantool_test_node_1
tarantool> os.exit()
total 8
-rw-rw-r-- 1 4925 May  5 11:12 00000000000000000000.snap
-rw-rw-r-- 1  634 May  5 11:45 00000000000000000000.xlog
$ ls -l ~/tarantool_test_node_2/
total 8
-rw-rw-r-- 1 4925 May  5 11:20 00000000000000000000.snap
-rw-rw-r-- 1  704 May  5 11:38 00000000000000000000.xlog
$
```

```
<... ...>
tarantool> s:select({1}, {iterator = 'GE'})
---
- - [1, 'Tuple inserted on Terminal #1']
...
tarantool> s:insert{2, 'Tuple inserted on Terminal #2'}
---
- [2, 'Tuple inserted on Terminal #2']
...
tarantool>
<...> [25579] main/103/replica/localhost:3301 I> can't read row
<...> [25579] main/103/replica/localhost:3301 !> SystemError
  unexpected EOF when reading from socket,
  called on fd 10, aka 127.0.0.1:50884, peer of 127.0.0.1:3301: Broken pipe
<...> [25579] main/103/replica/localhost:3301 I> will retry every 1 second
```

Во втором терминале (Terminal #2) проигнорируйте сообщения об ошибках и выполните следующие запросы:

```
tarantool> box.space.tester:select({0}, {iterator = 'GE'})
tarantool> box.space.tester:insert{3, 'Another'}
```

Теперь вывод на экране выглядит следующим образом (сообщения об ошибках мы не приводим):

```
<... ...>
tarantool> s:insert{1, 'Tuple inserted on Terminal #1'}
---
- [1, 'Tuple inserted on Terminal #1']
...
$ ls -l ~/tarantool_test_node_1
tarantool> os.exit()
total 8
-rw-rw-r-- 1 4925 May  5 11:12 00000000000000000000.snap
```

```
-rw-rw-r-- 1 634 May 5 11:45 00000000000000000000.xlog
$ ls -l ~/tarantool_test_node_2/
total 8
-rw-rw-r-- 1 4925 May 5 11:20 00000000000000000000.snap
-rw-rw-r-- 1 704 May 5 11:38 00000000000000000000.xlog
$
```

```
<... ..>
tarantool> s:insert{2, 'Tuple inserted on Terminal #2'}
---
- [2, 'Tuple inserted on Terminal #2']
...
tarantool>
<...> [11243] main/105/applier/localhost:3301 I> can't read row
<...> [11243] main/105/applier/localhost:3301 coio.cc:352 !> SystemError
unexpected EOF when reading from socket,
called on fd 6, aka 127.0.0.1:58734, peer of 127.0.0.1:3301: Broken pipe
<...> [11243] main/105/applier/localhost:3301 I> will retry every 1 second
tarantool> box.space.testers:select({0}, {iterator = 'GE'})
---
- - [1, 'Tuple inserted on Terminal #1']
- [2, 'Tuple inserted on Terminal #2']
...
tarantool> box.space.testers:insert{3, 'Another'}
---
- [3, 'Another']
...

```

Запросы SELECT и INSERT во втором терминале отработали несмотря на то, что сервер в первом терминале остановлен.

В первом терминале (Terminal #1) выполните следующие запросы:

```
$ ~/tarantool/src/tarantool
tarantool> box.cfg{listen = 3301}
tarantool> box.space.testers:select({0}, {iterator = 'GE'})
```

Теперь вывод на экране выглядит следующим образом:

```
<... ..>
tarantool> s:insert{1, 'Tuple inserted on Terminal #1'}
---
- [1, 'Tuple inserted on Terminal #1']
...
$ ls -l ~/tarantool_test_node_1
tarantool> os.exit()
total 8
-rw-rw-r-- 1 4925 May 5 11:12 00000000000000000000.snap
-rw-rw-r-- 1 634 May 5 11:45 00000000000000000000.xlog
$ ls -l ~/tarantool_test_node_2/
total 8
-rw-rw-r-- 1 4925 May 5 11:20 00000000000000000000.snap
-rw-rw-r-- 1 704 May 5 11:38 00000000000000000000.xlog
$ ~/tarantool/src/tarantool
<...>
tarantool> box.cfg{listen = 3301}
<...> [22612] main/101/interactive I> ready to accept requests
<... ..>
```

```
tarantool> box.space.testers:select({0}, {iterator = 'GE'})
- - [1, 'Tuple inserted on Terminal #1']
...
```

```
<... ...>
tarantool> s:insert{2, 'Tuple inserted on Terminal #2'}
---
- [2, 'Tuple inserted on Terminal #2']
...
tarantool>
<...> [25579] main/103/replica/localhost:3301 I> can't read row
<...> [25579] main/103/replica/localhost:3301 !> SystemError
unexpected EOF when reading from socket,
called on fd 10, aka 127.0.0.1:50884, peer of 127.0.0.1:3301: Broken pipe
<...> [25579] main/103/replica/localhost:3301 I> will retry every 1 second
tarantool> box.space.testers:select({0}, {iterator = 'GE'})
---
- - [1, 'Tuple inserted on Terminal #1']
- [2, 'Tuple inserted on Terminal #2']
...
tarantool> box.space.testers:insert{3, 'Another'}
---
- [3, 'Another']
...
<...> [11243] main/105/applier/localhost:3301 I> connected to 1.7.0 at 127.0.0.1:3301
<...> [11243] main/105/applier/localhost:3301 I> authenticated
```

Главный сервер снова установил соединение с кластером и НЕ обнаружил изменения, сделанные репликой за время его недоступности. Это и не удивительно: мы же не просили реплику выступать в качестве источника репликации.

В первом терминале (Terminal #1) введите:

```
tarantool> box.cfg{
  > replication_source = 'replicator:password@localhost:3302'
  > }
tarantool> box.space.testers:select({0}, {iterator = 'GE'})
```

Теперь вывод на экране выглядит следующим образом:

```
<... ...>
$ ~/tarantool/src/tarantool
<...>
tarantool> box.cfg{listen = 3301}
<...> [22612] main/101/interactive I> ready to accept requests
<... ...>
tarantool> box.space.testers:select({0}, {iterator = 'GE'})
---
- - [1, 'Tuple inserted on Terminal #1']
...
tarantool> box.cfg{
  > replication_source='replicator:password@localhost:3302'
  > }
[28987] main/101/interactive C> starting replication from localhost:3302
---
...
[22612] main/101/interactive C> starting replication from localhost:3302
[22612] main/101/interactive I> set 'replication_source' configuration
```



```

option to "replicator:password@localhost:3302"
[22612] main/104/applier/localhost:3302 I> connected to 1.7.0 at 127.0.0.1:3302
[22612] main/104/applier/localhost:3302 I> authenticated
[22612] wal/101/main I> creating `./00000000000000000000.xlog.inprogress'
[22612] relay/127.0.0.1:33510/102/main I> done `./00000000000000000000.xlog'
[22612] relay/127.0.0.1:33510/102/main I> recover from `./00000000000000000000.xlog'
tarantool> box.space.testers:select({0}, {iterator = 'GE'})
---
- - [1, 'Tuple inserted on Terminal #1']
- - [2, 'Tuple inserted on Terminal #2']
- - [3, 'Another']
...

```

```

<... ...>
tarantool> s:insert{2, 'Tuple inserted on Terminal #2'}
---
- [2, 'Tuple inserted on Terminal #2']
...
tarantool>
<...> [25579] main/103/replica/localhost:3301 I> can't read row
<...> [25579] main/103/replica/localhost:3301 !> SystemError
unexpected EOF when reading from socket,
called on fd 10, aka 127.0.0.1:50884, peer of 127.0.0.1:3301: Broken pipe
<...> [25579] main/103/replica/localhost:3301 I> will retry every 1 second
tarantool> box.space.testers:select({0}, {iterator = 'GE'})
---
- - [1, 'Tuple inserted on Terminal #1']
- - [2, 'Tuple inserted on Terminal #2']
...
tarantool> box.space.testers:insert{3, 'Another'}
---
- [3, 'Another']
...
<...> [11243] main/105/applier/localhost:3301 I> connected to 1.7.0 at 127.0.0.1:3301
<...> [11243] main/105/applier/localhost:3301 I> authenticated
tarantool>
<...> [11243] relay/127.0.0.1:36150/101/main I> recover from `./00000000000000000000.xlog'
<...> [11243] relay/127.0.0.1:36150/101/main recovery.cc:211 W> file
`./00000000000000000000.xlog` wasn't correctly closed
<...> [11243] relay/127.0.0.1:36150/102/main I> recover from `./00000000000000000000.xlog'

```

Тут мы видим, что оба сервера снова синхронизовались и что каждый из них видит те записи, которые сделал другой.

Чтобы удалить все тестовые данные, выполните "os.exit()" на обоих терминалах, а затем на каждом из них выполните следующие команды:

```

$ cd ~
$ rm -R ~/tarantool_test_node_1
$ rm -R ~/tarantool_test_node_2

```

3.5 Справочник по конфигурированию

В этой главе приводится справочная информация о конфигурационных параметрах, которые можно передавать в командной строке или указывать в файле инициализации.

- *Опции командной строки*
- *Универсальный код ресурса (URI)*
- *Файл инициализации*
- *Параметры конфигурации*
 - *Основные параметры*
 - *Настройка хранилища данных*
 - *Фоновая программа для работы со снимками*
 - *Журналирование и снимки в бинарном формате*
 - *Репликация*
 - *Работа с сетевыми соединениями*
 - *Запись в журнал*
- *Режим горячего резервирования*

Для запуска Tarantool-сервера нужно ввести одну из следующих команд:

```
$ tarantool
# ИЛИ
$ tarantool опции
# ИЛИ
$ tarantool файл-инициализации-на-lua [ аргументы ]
```

3.5.1 Опции командной строки

`-h, --help`

Вывести на экран список всех опций командной строки с комментариями.

`-V, --version`

Вывести на экран имя продукта и номер версии, например:

```
$ ./tarantool --version
Tarantool 1.7.0-1216-g73f7154
Target: Linux-x86_64-Debug
...
```

В данном примере:

“Tarantool” – это имя повторно используемой асинхронной сетевой программной среды.

Номер версии составлен по стандартной схеме из 3 частей: `<major>-<minor>-<patch>`, где `<major>` изменяется очень редко, `<minor>` увеличивается на каждый milestone и свидетельствует о возможных несовместимостях с предыдущими версиями, а `<patch>` указывает количество релизов с доработками с момента текущего milestone’a. Для сборок, не предназначенных для релиза, здесь могут быть добавлены также номер и SHA1-код коммита, которые указывают, насколько сильно текущая сборка отличается от предыдущего релиза.

“Target” – это платформа, на которой была сделана данная сборка Tarantool’a. За этой строчкой может следовать информация для данной платформы.

Примечание: Идентификатор версии (version id) в Tarantool'е получают с помощью команды [git describe](#). По этому идентификатору можно в любой момент извлечь соответствующие исходные файлы из [git-репозитория](#).

3.5.2 Универсальный код ресурса (URI)

Некоторые функции и параметры конфигурации в Tarantool'е зависят от универсального кода доступа (URI = Universal Resource Identifier). Формат URI-строки аналогичен [общепринятому синтаксису URI-схемы](#) и может содержать (перечисляем по порядку) имя пользователя для входа в систему, пароль, имя или IP-адрес хоста, номер порта. Обязательным параметром является только номер порта. Пароль обязателен, если указано имя пользователя (за исключением имени 'guest'). Таким образом, синтаксис URI формально выглядит так: `[host:]port` или `[username:password@]host:port`. Если не указан `host`, то значением по умолчанию является '0.0.0.0' (т.е. любой адрес формата IPv4) или ':::' (т.е. любой адрес формата IPv6) на локальной машине. Если не указаны `username:password`, то значением по умолчанию является 'guest'. Вот несколько примеров:

Часть URI	Пример
port	3301
host:port	127.0.0.1:3301
username:password@host:port	notguest:sesame@mail.ru:3301

В некоторых случаях вместо URI можно указывать сокет домена Unix, например `"unix:/tmp/unix_domain_socket.sock"` или просто `"/tmp/unix_domain_socket.sock"`.

См. также [Appendix E: Cookbook](#), где приводится вариант реализации для парсинга URI-адресов.

3.5.3 Файл инициализации

Если в команде запуска Tarantool'а указан *файл инициализации на языке Lua*, то запуск начинается с вызова Lua-программы из файла с именем `"script.lua"`. Эта Lua-программа может принимать дальнейшие аргументы, указанные в командной строке, или использовать функции операционной системы, такие как `getenv()`. Если после запуска Tarantool'а предполагается использовать сервер баз данных или требуется открыть порты на прослушивание, то Lua-программа почти всегда начинается с вызова `box.cfg()`. Для примера возьмем файл `script.lua`, который содержит следующие строки:

```
#!/usr/bin/env tarantool
box.cfg{
  listen           = os.getenv("LISTEN_URI"),
  slab_alloc_arena = 0.1,
  pid_file         = "tarantool.pid",
  rows_per_wal    = 50
}
print('Starting ', arg[1])
```

Далее предположим, что системная переменная `LISTEN_URI` содержит значение 3301, а вызов командной строки таков: `~/tarantool/src/tarantool script.lua ARG`. Вывод на экране в таком случае будет 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 после инициализации вам понадобится запустить интерактивную Tarantool-сессию на том же терминале, то это можно сделать с помощью команды `console.start()`.

3.5.4 Параметры конфигурации

Параметры конфигурации задаются следующим образом:

```
box.cfg{ key = value [, key = value ...]}
```

Поскольку вызов `box.cfg` может содержать много параметров конфигурации, причем некоторые параметры (например, адреса директорий) могут со временем меняться, то имеет смысл хранить `box.cfg` в Lua-файле. Обычно это Lua-файл для инициализации, который указывается в командной строке при запуске Tarantool'a.

Большинство параметров конфигурации служат для настройки выделения ресурсов и открытия портов, а также для задания поведения базы данных. Все параметры являются необязательными. Некоторые параметры являются динамическими, т.е. их можно менять во время исполнения с помощью повторного вызова `box.cfg{}`.

Для просмотра все не нулевых значений параметров, введите `box.cfg` (без фигурных скобок). Для просмотра значения определенного параметра, например адреса для прослушивания, введите `box.cfg.listen`.

В следующих разделах описаны все параметры, которые служат для настройки основных операций, хранения, создания статических снимков данных, репликации, работы с сетевыми соединениями и журналирования.

Основные параметры

background

Запустить Tarantool-сервер в фоновом режиме. При этом нужно задать не нулевые значения для параметров `logger` и `pid_file`.

Тип значения: boolean По умолчанию: false Динамический: нет

coredump

Параметр устарел. Не используйте его.

Тип значения: boolean По умолчанию: false Динамический: нет

custom_proc_title

Добавить указанную строку к *заголовку процесса* Tarantool-сервера (заголовок процесса выводится в колонке COMMAND в ответ на команды `ps -ef` и `top -c`).

Например, информация о процессе Tarantool-сервера, которая выводится по команде `ps -ef`, обычно имеет следующий вид:

```

$ ps -ef | grep tarantool
1000      14939 14188  1 10:53 pts/2    00:00:13 tarantool <running>

```

Однако, если в конфигурации указан параметр `custom_proc_title='sessions'`, то информация в выводе `ps -ef` будет другой:

```
$ ps -ef | grep tarantool
1000      14939 14188  1 10:53 pts/2    00:00:16 tarantool <running>: sessions
```

Тип значения: string По умолчанию: null Динамический: да

listen

Номер порта для чтения/записи данных, либо *URI-строка* (URI = Universal Resource Identifier). Значения по умолчанию у данного параметра нет, поэтому он **обязателен**, если у клиентов должна быть возможность устанавливать соединение с Tarantool-сервером по портам, не являющимся *“портами для администрирования”*. Соединения через `listen=URI` иногда называют соединениями по "бинарному протоколу" или по "главному порту".

Стандартное значение – это 3301. Порт для прослушивания можно задавать также для *режима горячего резервирования*.

Примечание: Сервер-реплика принимает запросы на соединение на этом же порту. Реплика принимает только запросы на чтение, пока ей не будет назначена роль главного сервера.

Тип значения: integer или string По умолчанию: null Динамический: да

pid_file

Сохранить идентификатор процесса в указанный файл. Путь можно указывать относительно рабочей директории (*work_dir*). Стандартное значение – это `“tarantool.pid”`.

Тип значения: string По умолчанию: null Динамический: нет

read_only

Перевести сервер в режим "доступен только на чтение". После установки этого параметра Tarantool-сервер будет возвращать ошибку ER_READONLY на все запросы на изменение данных.

Тип значения: boolean По умолчанию: false Динамический: да

snap_dir

Директория для хранения статических снимков данных (.snap-файлов). Путь можно указывать относительно рабочей директории (*work_dir*). Если этот параметр не задан, то по умолчанию используется рабочая директория (*work_dir*). См. также *wal_dir*.

Тип значения: string По умолчанию: "." Динамический: нет

vinyl_dir

Директория для хранения файлов и поддиректорий движка vinyl. Путь можно указывать относительно рабочей директории (*work_dir*). Если этот параметр не задан, то по умолчанию используется рабочая директория (*work_dir*).

Тип значения: string По умолчанию: "." Динамический: нет

username

UNIX-имя пользователя, на которое нужно переключиться после запуска.

Тип значения: string По умолчанию: null Динамический: нет

wal_dir

Директория для хранения журналов упреждающей записи (.xlog-файлов). Путь можно указывать относительно рабочей директории (*work_dir*). Иногда используют разные пути к *wal_dir* и

snap_dir, чтобы .xlog-файлы и .snap-файлы можно было хранить на разных дисках. Если этот параметр не задан, то по умолчанию используется рабочая директория (*work_dir*).

Тип значения: string По умолчанию: "." Динамический: нет

work_dir

Директория для хранения рабочих файлов базы данных. После запуска сервер переключается на *work_dir* с помощью *chdir(2)*. Путь можно указывать относительно текущей директории. Если этот параметр не задан, то по умолчанию используется текущая директория. Относительно *work_dir* можно указывать другие параметры. Например, если задать `box.cfg{work_dir='/home/user/A',wal_dir='B',snap_dir='C'}` то .xlog-файлы будут храниться в директории `/home/user/A/B`, .snap-файлы – в директории `/home/user/A/C`, а все прочие файлы и поддиректории – в рабочей директории `/home/user/A`.

Тип значения: string По умолчанию: null Динамический: нет

Настройка хранилища данных

slab_alloc_arena

Объем памяти (в гигабайтах), который Tarantool выделяет для хранения таплов. Когда этот объем памяти оказывается занят, Tarantool-сервер начинает выдавать ошибку `ER_MEMORY_ISSUE` на запросы `INSERT` и `UPDATE`. Данное ограничение относится только к хранению таплов: для хранения индексов и информации об установленных соединениях Tarantool-сервер использует дополнительную память. Таким образом, в зависимости от конкретной конфигурации и нагрузки, фактическое превышение расхода памяти у Tarantool-сервера может составлять до 20% от указанного здесь значения.

Тип значения: float По умолчанию: 1.0 Динамический: нет

slab_alloc_factor

Данный параметр используется в качестве множителя при вычислении размера фрагментов памяти, выделяемых для хранения таплов. В некоторых случаях (в зависимости от общего объема памяти и разброса по размеру таплов) вы можете снизить объем выделенной памяти, задав меньшее значение множителя.

Тип значения: float По умолчанию: 1.1 Динамический: нет

slab_alloc_maximal

Максимально возможный размер фрагмента памяти, выделяемого для одного тапла. Можно задать больший размер, если необходимо хранить таплы большого размера.

Тип значения: integer По умолчанию: 1048576 Динамический: нет

slab_alloc_minimal

Минимально возможный размер фрагмента памяти, выделяемого для одного тапла. Можно задать меньший размер, если большинство таплов в базе имеют очень маленький размер. Значение может лежать в интервале от 8 до 1048280 включительно.

Тип значения: integer По умолчанию: 16 Динамический: нет

vinyl

Пользовательские значения параметров для движка `vinyl` можно задать с помощью запроса следующего вида:

```
vinyl = {
  run_age_wm = number,
  run_age_period = number of seconds,
  memory_limit = number of gigabytes,
  compact_wm = number,
  threads = number,
```

```

    run_age = number,
    run_prio = number,
}

```

В будущем этот способ может быть пересмотрен.

Значения по умолчанию:

```

vinyl = {
  run_age_wm = 0,
  run_age_period = 0,
  memory_limit = 1,
  compact_wm = 2,
  threads = 5,
  run_age = 0,
  run_prio = 2,
}

```

Фоновая программа для работы со снимками

Фоновая программа для работы со снимками (`snapshot daemon`) – это постоянно запущенный файбер. Он может срабатывать через заданные промежутки времени, создавая новые файлы-снимки (`.snap`-файлы) и удаляя старые. При удалении файла-снимка удаляются также все WAL-файлы (`.xlog`), которые старше удаленного файла-снимка и содержат информацию, которая была в удаленном снимке.

Параметры `snapshot_period` и `snapshot_count` указывают длину интервалов для срабатывания фоновой программы и количество сделанных файлов-снимков, после которого можно удалять самый старый снимок.

`snapshot_period`

Интервал (в секундах) между срабатываниями фоновой программы. Если значение `snapshot_period` больше нуля, а также имели место какие-либо изменения в базе, то фоновая программа будет каждые `snapshot_period` секунд вызывать метод `box.snapshot`, и при каждом таком вызове будет создаваться новый снимок.

Например, в результате запроса `box.cfg{snapshot_period=3600}` фоновая программа будет раз в час создавать новый снимок.

Тип значения: `integer` По умолчанию: 0 Динамический: да

`snapshot_count`

Максимальное количество снимков в директории `snap_dir`, после которого фоновая программа может удалять самые старые снимки. Если значение `snapshot_count` равно нулю, то фоновая программа не занимается удалением старых снимков. Например, в результате следующего запроса:

```

box.cfg{
  snapshot_period = 3600,
  snapshot_count  = 10
}

```

фоновая программа будет раз в час создавать новый снимок до тех пор, пока не создаст 10 снимков. После создания 11-го снимка фоновая программа удалит самый старый снимок и все связанные с ним WAL-файлы.

Тип значения: `integer` По умолчанию: 6 Динамический: да

Журналирование и снимки в бинарном формате

panic_on_snap_error, *panic_on_wal_error*, *rows_per_wal*, *snap_io_rate_limit*, *wal_mode*,
wal_dir_rescan_delay

`panic_on_snap_error`

Прервать исполнение, если при чтении файла-снимка возникает ошибка (на этапе запуска Tarantool-сервера).

Тип значения: boolean По умолчанию: true Динамический: нет

`panic_on_wal_error`

Прервать исполнение, если при чтении WAL-файла возникает ошибка (на этапе запуска Tarantool-сервера, либо на этапе первого запроса данных со стороны реплики).

Тип значения: boolean По умолчанию: true Динамический: да

`rows_per_wal`

Максимальное количество записей, которые можно хранить в одном WAL-файле. Когда это ограничение достигнуто, то Tarantool создает новый WAL-файл и называет его по LSN-номеру первой записи в новом файле (*<lsn-номер-первой-записи>.xlog*). Это может пригодиться при простой организации создания резервных копий с помощью rsync.

Тип значения: integer По умолчанию: 500000 Динамический: нет

`snap_io_rate_limit`

Уменьшить падение производительности на запросах INSERT/UPDATE/DELETE во время отработки *box.snapshot* с помощью ограничения на максимальное количество мегабайт, которое может записываться на диск в секунду. С той же целью можно развести директории *wal_dir* и *snap_dir* по разным адресам и сохранять снимки на другом диске.

Тип значения: float По умолчанию: null Динамический: да

`wal_mode`

Задать режим синхронизации между файберами, WAL-файлом и диском:

- **none**: запись в WAL-файл не ведется;
- **write**: файберы ждут, пока их данные будут записаны в WAL-файл (без *fsync(2)*);
- **fsync**: файберы ожидают свои данные, *fsync(2)* происходит после каждого вызова *write(2)*;

Тип значения: string По умолчанию: "write" Динамический: да

`wal_dir_rescan_delay`

Количество секунд между периодическим сканированием директории с WAL-файлами на предмет наличия изменений в WAL-файлах, которые следует синхронизировать в рамках репликации или режима горячей готовности.

Тип значения: float По умолчанию: 2 Динамический: нет

Репликация

`replication_source`

Источник репликации. Если в параметре *replication_source* указана не пустая строка, то такой Tarantool-сервер считается *репликой*. Сервер-реплика будет пытаться установить соединение с главным сервером, чей адрес указан в параметре *replication_source* в виде *URI* (Universal Resource Identifier), например *konstantin:secret_password@tarantool.org:3301*.

Если в кластере более одного источника репликации, то следует указать массив из URI, например `box.cfg{replication_source = {uri#1,uri#2}}`

Если один из URI соответствует текущему серверу – т.е. тому, на котором сейчас выполняется запрос `box.cfg{}`, – то такой источник игнорируется. Таким образом, можно использовать одни и те же значения `replication_source` на многих серверах.

Имя пользователя по умолчанию – ‘guest’. Сервер-реплика не принимает запросы на изменение данных на порту для прослушивания (параметр `listen`). Параметр `replication_source` является динамическим: чтобы перевести реплику в режим главного сервера, нужно только задать пустую строку в `replication_source` и выполнить запрос `box.cfg{replication_source=new-value}`.

Тип значения: string По умолчанию: null Динамический: yes

Работа с сетевыми соединениями

`io_collect_interval`, `readahead`

`io_collect_interval`

Между итерациями основного цикла Tarantool-сервер уходит в спящий режим на `io_collect_interval` секунд. Этот параметр можно использовать для ограничения нагрузки на процессор в том случае, когда количество клиентских соединений очень велико, но запросы от них приходят нечасто (например, от каждого соединения приходит всего несколько запросов в секунду).

Тип значения: float По умолчанию: null Динамический: да

`readahead`

Размер буфера предварительного чтения на каждое клиентское соединение. Чем больше размер буфера, тем больше памяти будет потреблять каждое активное соединение и тем больше запросов можно будет зачитать из системного буфера за один системный вызов. Общее правило для определения оптимального размера буфера таково: буфер должен вмещать по крайней мере несколько десятков запросов. Поэтому, если средний размер тапла в запросе у вас большой, например несколько кило- или даже мегабайт, то размер буфера предварительного чтения следует увеличить. Если вы не используете пакетную обработку запросов, то лучше оставить размер буфера по умолчанию.

Тип значения: integer По умолчанию: 16320 Динамический: да

Запись в журнал

`log_level`

Уровень детализации при записи в журнал. Всего есть шесть уровней:

- 1 – SYSERROR
- 2 – ERROR
- 3 – CRITICAL
- 4 – WARNING
- 5 – INFO
- 6 – DEBUG

С помощью настройки уровня детализации можно включать журналирование всех классов событий выше или ниже указанного уровня. По умолчанию Tarantool выводит свой журнал в стандартный поток ошибок (`stderr`), но вывод ошибок можно настроить с помощью параметра `logger`.

Тип значения: integer По умолчанию: 5 Динамический: да

logger

По умолчанию Tarantool выводит свой журнал в стандартный поток ошибок (`stderr`). С помощью данного параметра вывод ошибок можно перенастроить на вывод в файл, в программный канал (pipe) или в системный регистратор (system logger).

Пример настроек:

```
box.cfg{logger = 'tarantool.log'}
-- или
box.cfg{logger = 'file: tarantool.log'}
```

В результате в Tarantool-директории, используемой по умолчанию, будет открыт на запись файл `tarantool.log`. Если префикс в значении параметра `logger` не указан или же указан префикс "file: то дальнейшая строка интерпретируется как путь к файлу.

Пример настроек:

```
box.cfg{logger = '| cronolog tarantool.log'}
-- или
box.cfg{logger = 'pipe: cronolog tarantool.log'}
```

В результате при запуске Tarantool-сервера будет также запущена программа `cronolog`, и все сообщения для журнала будут пересылаться на стандартный ввод (`stdin`) программы `cronolog`. Если значение параметра `logger` начинается с символа '|' или с префикса "pipe: то дальнейшая строка интерпретируется как [Unix-конвейер](#).

Пример настроек:

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

Если значение параметра `logger` начинается с префикса "syslog: то дальнейшая строка интерпретируется как сообщение для программы `syslogd`, которая обычно запущена в фоновом режиме на любой Unix-платформе. В этой строке можно задать значения `identity`, `facility` или оба эти значения сразу. В качестве `identity` может быть задана любая строка (значение по умолчанию – `tarantool`), которая будет использоваться как префикс перед всеми сообщениями в журнале. В качестве `facility` задается аббревиатура имени одного из типов ПО в программе `syslog` (значение по умолчанию – `user`), которая указывает программе `syslogd`, куда/кому передавать сообщения.

Возможные значения `facility`: `auth`, `authpriv`, `cron`, `daemon`, `ftp`, `kern`, `lpr`, `mail`, `news`, `security`, `syslog`, `user`, `uucp`, `local0`, `local1`, `local2`, `local3`, `local4`, `local5`, `local6`, `local7`.

На данный момент значение "facility" игнорируется, но в будущем оно будет использоваться.

При записи сообщений в файл Tarantool переоткрывает файл с журналом на каждый `SIGHUP`. Если журнал является программой, то ее `pid`-идентификатор сохраняется в переменной `log.logger_pid`. Для ротации журнала нужно послать сигнал программе с этим `pid`-идентификатором.

Тип значения: string По умолчанию: null Динамический: нет

logger_nonblock

Если в значении параметра `logger_nonblock` указано `true`, то Tarantool не блокирует файл с журналом на запись, и если очередь на запись в ядре оказывается переполнена, то последующие сообщения от Tarantool'a не попадают в очередь и теряются, пока очередь не очистится. Если

в параметре `log_level` указан высокий уровень детализации и Tarantool отправляет в очередь на запись много сообщений для журнала, то можно увеличить скорость записи в журнал, выставив значение параметра `logger_nonblock` в `true` и пожертвовав некоторым количеством сообщений, которые при этом могут быть потеряны.

Тип значения: boolean По умолчанию: true Динамический: нет

`too_long_threshold`

Если обработка запроса занимает больше времени, чем указано в этом параметре (в секундах), то в журнал добавляется предупредительная запись. Это происходит, только если уровень детализации, указанный в параметре `log_level`, выше или равен 4 (WARNING).

Тип значения: float По умолчанию: 0.5 Динамический: да

Пример работы с журналом:

В этом примере мы покажем, как работает ротация журнала, т.е. что происходит, когда настроена запись в журнал и используются сигналы для архивирования записей.

Запустите два отдельных терминала, Terminal #1 и Terminal #2.

В первом терминале (Terminal #1): запустим интерактивную Tarantool-сессию, настроим запись сообщений в журнал `Log_file`, а затем запишем в журнал сообщение "Log Line #1":

```
box.cfg{logger='Log_file'}
log = require('log')
log.info('Log Line #1')
```

Во втором терминале (Terminal #2): с помощью команды `mv` переименуем файл журнала в `Log_file.bak`. В результате следующее сообщение будет записано уже в файл `Log_file.bak`.

```
mv Log_file Log_file.bak
```

В первом терминале (Terminal #1): запишем в журнал сообщение "Log Line #2".

```
log.info('Log Line #2')
```

Во втором терминале (Terminal #2): с помощью команды `ps` выясним идентификатор процесса Tarantool-сервера.

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

Во втором терминале (Terminal #2): с помощью команды `kill -HUP` пошлем Tarantool-серверу SIGHUP-сигнал.

```
kill -HUP process_id
```

В первом терминале (Terminal #1): запишем в журнал сообщение "Log Line #3".

```
log.info('Log Line #3')
```

Во втором терминале (Terminal #2): с помощью команды `less` просмотрим содержимое файлов. Для файла `Log_file.bak` будут выведены следующие строки (если вы выполняете этот пример у себя, то отличаться должны только дата и время):

```
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 для журнала `Log_file` вывод будет следующим:

```
log file has been reopened
2015-11-30 15:15:32.629 [27469] main/101/interactive I> Log Line #3
```

3.5.5 Режим горячего резервирования

Режим горячего резервирования позволяет организовать восстановление после сбоя простым образом, без настройки репликации. Для этого запустите на том же компьютере второй экземпляр Tarantool-сервера с теми же настройками *box.cfg*, включая директории и не нулевые URI-адреса. В случае успеха должно появиться предупреждение:

```
W> primary: [URI] is already in use, will retry binding after [n] seconds
```

Это означает, что второй экземпляр готов заменить первый, если тот выйдет из строя.

Режим горячего резервирования подразумевает, что на одном компьютере запущены два экземпляра Tarantool-сервера с одинаковой конфигурацией. Первый запущенный экземпляр является "главным" (primary), а второй – "резервным" (standby). Резервный экземпляр при запуске попытается установить соединение на порту для прослушивания, но у него не получится это сделать, поскольку адрес уже занят главным экземпляром. Таким образом, резервный экземпляр повторяет в цикле следующие действия: зачитывает данные из WAL-файла, который обновляется главным экземпляром (и благодаря этому оба экземпляра всегда синхронизированы между собой) и пытается установить соединение на порту для прослушивания. Если по какой-либо причине главный сервер выходит из строя, то порт для прослушивания освобождается, резервный экземпляр устанавливает на нем соединение и становится главным. Таким образом, время недоступности в случае сбоя оказывается почти незаметным.

При использовании горячего резервирования (значение параметра `local_hot_standby` равно "true") значение параметра `wal_mode` не должно быть равно "none".

3.6 Администрирование серверной части

Типичные задачи администрирования серверной части включают в себя: запуск и остановку сервера, перезагрузку настроек, создание снимков, ротацию логов.

3.6.1 Обработка сигналов от сервера

Во время основного цикла Tarantool-сервер обрабатывает следующие сигналы:

SIGHUP может привести к ротации лога, см. *пример в разделе "Запись в журнал"*@.

SIGUSR1 может привести к сохранению снимка, см. описание функции *box.snapshot*.

SIGTERM может привести к корректному завершению работы (с предварительным сохранением всех данных).

SIGINT (или "прерывание с клавиатуры") может привести к корректному завершению работы (с предварительным сохранением всех данных).

SIGKILL приводит к аварийному завершению работы (с возможной потерей данных).

Действие других сигналов определяется операционной системой. Все сигналы, кроме SIGKILL, могут быть проигнорированы, особенно если Tarantool-сервер выполняет длительную процедуру, которая позволяет вернуться к главному циклу.

3.6.2 Название процесса

Операционные системы Linux и FreeBSD позволяют запущенному процессу менять его название, в котором изначально содержится имя программы. Tarantool использует эту возможность, чтобы упростить работу системного администратора, например посмотреть, какие службы запущены на хосте, их статус и т.д.

Название процесса Tarantool-сервера состоит из следующих частей:

```
имя_программы [имя_файла_инициализации] <имя_роли> [название_процесса]
```

- **имя_программы** – это, как правило, "tarantool".
- **имя_файла_инициализации** – это имя *файла инициализации на Lua*, если этот файл был указан при запуске.
- **имя_роли** – это может быть один из следующих вариантов: - "running" (узел находится в режиме "готов к принятию запросов"), - "loading" (узел, который загружает данные из ранее сохраненного снимка и WAL-файла), - "orphan" (узел не входит в состав кластера), - "hot_standby" (см. раздел *локальная горячая готовность (local hot standby)*), либо - "dumper"+ process-id (идет сохранение снимка).
- **название_процесса** – это необязательное название Tarantool-процесса в системе, которое берется из конфигурационного параметра *custom_proc_title*, если он указан.

Например:

```
$ ps -AF | grep tarantool
1000      17337 16716  1 91362  6916   0 11:07 pts/5    00:00:13 tarantool script.lua <running>
```

3.6.3 Использование tarantool в качестве клиента

Если tarantool запущен без *файла инициализации*, или же в файле инициализации указана функция *console.start()*, то tarantool стартует в интерактивном режиме: он выводит приглашение командной строки ("tarantool>"), и запросы можно вводить прямо в консоли. В таком режиме "tarantool" может быть использован в качестве клиента для удаленного сервера.

В этом разделе описаны синтаксические правила для ввода запросов в консоли Tarantool'a, с примечаниями и примерами. Другие клиентские программы могут иметь схожие параметры и синтаксис запросов. Некоторые сведения из этого раздела дублируются в главе *book-cfg*.

Условные обозначения, используемые в этом разделе

Токены – это последовательности символов, которые рассматриваются как синтаксические единицы в рамках запроса. Квадратные скобки [и] используются для обозначения опциональных токенов. Три точки в строке ... означают, что предыдущие токены могут повторяться. Вертикальная черта | означает, что предыдущие и последующие токены являются взаимоисключающими альтернативами.

Параметры запуска клиента из командной строки

Общий вид:

```
$ tarantool
ИЛИ
$ tarantool опции
```

```
ИЛИ
$ tarantool файл-инициализации-на-lua [ аргументы ]
```

файл-инициализации-на-lua – это любой скрипт, содержащий логику инициализации. Код из этого файла выполняется при запуске Tarantool'a. Например: `init.lua`. Примечания: При использовании скрипта, Tarantool не выводит приглашение командной строки. Скрипт должен содержать конфигурационные настройки, в т.ч. `box.cfg{...listen=...}` или `box.listen(...)`, чтобы внешние программы могли установить соединение с Tarantool-сервером на одном из указанных портов.

Опция – это одно из следующих значений (указаны в алфавитном порядке, по полному имени опции):

`-, -h, --help`

Tarantool-клиент выводит краткую справку, включая список всех параметров. Например: `tarantool --help`. Вывод останавливается после показа справки.

`-V, --version`

Tarantool-клиент выводит свой номер версии. Например: `tarantool --version`. Вывод останавливается после показа номера версии.

Токены, запросы и специальные комбинации клавиш

Идентификатором процедуры может быть любая последовательность букв, цифр и/или подчеркиваний, которая отвечает правилам именования идентификаторов в Lua. Термин "идентификаторы процедур" также применяют к именам функций. Примечание: в случае с именами функций регистр имеет значение, поэтому `insert` и `Insert` – это не одно и то же.

Строковым литералом может быть любая последовательность из нуля и более символов, которая заключена в *одинарные кавычки*. *Двойные кавычки* также допустимы, но предпочтительным вариантом являются одинарные кавычки. А *двойные квадратные скобки* подходят для многострочных литералов (см. [Lua documentation](#)). Например: `'Hello, world'`, `'A'`, `[[A\B!]]`.

Числовым литералом может быть любая последовательность символов, состоящая из одной и более цифр с необязательным знаком `+` или `-` в начале. В состав больших числовых литералов, а также числовых литералов с плавающей точкой может входить десятичный разделитель (запятая или точка), символы для экспоненциального представления и суффиксы. Например: `500`, `-500`, `5e2`, `500.1`, `5LL`, `5ULL`.

Однобайтовым символом может быть запятая, открывающая или закрывающая круглая скобка, а также арифметический оператор. Например: `*`, `(`, `)`.

Токены должны разделяться одним или большим количеством пробелов. Исключением являются однобайтовые токены и строковые литералы – вокруг них пробелы не нужны.

Запросы

Generally requests are entered following the prompt in interactive mode while `tarantool` is running. (A prompt will be the word `tarantool` and a greater-than sign, for example `tarantool>`). The end-of-request marker is by default a newline (line feed).

For multi-line requests, it is possible to change the end-of-request marker. Syntax: `console = require('console'); console.delimiter(string-literal)`. The string-literal must be a value in single quotes. Effect: string becomes end-of-request delimiter, so newline alone is not treated as end of request. To go back to normal mode: `console.delimiter('')string-literal`. Delimiters are usually not necessary because Tarantool can tell when a multi-line request has not ended (for example, if it sees that a function declaration does not have an `end` keyword). Example:

```
console = require('console'); console.delimiter('!')
function f ()
  statement_1 = 'a'
  statement_2 = 'b'
end!
console.delimiter('')!
```

For a condensed Backus-Naur Form [BNF] description of the suggested form of client requests, see http://tarantool.org/doc/dev_guide/box-protocol.html.

In *interactive* mode, one types requests and gets results. Typically the requests are typed in by the user following prompts. Here is an example of an interactive-mode tarantool client session:

```
$ tarantool
[ tarantool will display an introductory message
  including version number here ]
tarantool> box.cfg{listen = 3301}
[ tarantool will display configuration information
  here ]
tarantool> s = box.schema.space.create('tester')
[ tarantool may display an in-progress message here ]
---
...
tarantool> s:create_index('primary', {type = 'hash', parts = {1, 'unsigned'}})
---
...
tarantool> box.space.tester:insert{1, 'My first tuple'}
---
- [1, 'My first tuple']
...
tarantool> box.space.tester:select(1)
---
- - [1, 'My first tuple']
...
tarantool> box.space.tester:drop()
---
...
tarantool> os.exit()
2014-04-30 10:28:00.886 [20436] main/101/spawner I> Exiting: master shutdown
$
```

Explanatory notes about what tarantool displayed in the above example:

- Many requests return typed objects. In the case of "box.cfg{listen=3301}" this result is displayed on the screen. If the request had assigned the result to a variable, for example "c = box.cfg{listen=3301}" then the result would not have been displayed on the screen.
- A display of an object always begins with "---" and ends with "...".
- The insert request returns an object of type = tuple, so the object display line begins with a single dash ('-'). However, the select request returns an object of type = table of tuples, so the object display line begins with two dashes ('- -').

3.6.4 Утилита tarantoolctl

With `tarantoolctl` one can say: "start an instance of the Tarantool server which runs a single user-written Lua program, allocating disk resources specifically for that program, via a standardized deployment

method."If Tarantool was installed with Debian or Red Hat installation packages, the script is in `/usr/bin/tarantoolctl` or `/usr/local/bin/tarantoolctl`. The script handles such things as: starting, stopping, rotating logs, logging in to the application's console, and checking status.

The use of `tarantoolctl` as a client is described in a separate section, [tarantoolctl connect](#).

configuring for tarantoolctl

"The `tarantoolctl` script will look for a configuration file in the current directory (`$PWD/.tarantoolctl`). If that fails, it looks in the current user's home directory (`$HOME/.config/tarantool/tarantool`). If that fails, it looks in the `SYSCONFDIR` directory (usually `/etc/sysconfig/tarantool` but it may be different on some platforms). Most of the settings are similar to the settings used by `box.cfg{...}`; however, `tarantoolctl` adjusts some of them by adding an application name. A copy of `usr/local/etc/default/tarantool`, with defaults for all settings, would look like this:

```
default_cfg = {
  pid_file   = "/var/run/tarantool",
  wal_dir    = "/var/lib/tarantool",
  snap_dir   = "/var/lib/tarantool",
  vinyl_dir  = "/var/lib/tarantool",
  logger     = "/var/log/tarantool",
  username   = "tarantool",
}
instance_dir = "/etc/tarantool/instances.enabled"
```

The settings in the above script are:

`pid_file` The directory for the pid file and control-socket file. The script will add `"/instance-name"` to the directory name.

`wal_dir` The directory for the write-ahead `*.xlog` files. The script will add `"/instance-name"` to the directory-name.

`snap_dir` The directory for the snapshot `*.snap` files. The script will add `"/instance-name"` to the directory-name.

`vinyl_dir` The directory for the vinyl-storage-engine files. The script will add `"/vinyl/instance-name"` to the directory-name.

`logger` The place where the application log will go. The script will add `"/instance-name.log"` to the name.

`username` The user that runs the tarantool server. This is the operating-system user name rather than the Tarantool-client user name.

`instance_dir` The directory where all applications for this host are stored. The user who writes an application for `tarantoolctl` must put the application's source code in this directory, or a symbolic link. For examples in this section the application name `my_app` will be used, and its source will have to be in `instance_dir/my_app.lua`.

commands for tarantoolctl

The command format is `tarantoolctl operation application_name`, where `operation` is one of: `start`, `stop`, `enter`, `logrotate`, `status`, `eval`. Thus ...

```
start <application>
    Start application <application>

stop <application>
    Stop application
```



```
enter <application>
    Show application's admin console

logrotate <application>
    Rotate application's log files (make new, remove old)

status <application>
    Check application's status

eval <application> <scriptname>
    Execute code from <scriptname> on an instance of application
```

typical code snippets for tarantoolctl

A user can check whether my_app is running with these lines:

```
if tarantoolctl status my_app; then
...
fi
```

A user can initiate, for boot time, an init.d set of instructions:

```
for (each file mentioned in the instance_dir directory):
    tarantoolctl start `basename $ file .lua`
```

A user can set up a further configuration file for log rotation, like this:

```
/path/to/tarantool/*.log {
    daily
    size 512k
    missingok
    rotate 10
    compress
    delaycompress
    create 0640 tarantool adm
    postrotate
        /path/to/tarantoolctl logrotate basename $ 1 .log
    endscript
}
```

A detailed example for tarantoolctl

The example's objective is: make a temporary directory where tarantoolctl can start a long-running application and monitor it.

The assumptions are: the root password is known, the computer is only being used for tests, the Tarantool server is ready to run but is not currently running, tarantoolctl is installed along the user's path, and there currently is no directory named `tarantool_test`.

Create a directory named `/tarantool_test`:

```
$ sudo mkdir /tarantool_test
```

Edit `/usr/local/etc/default/tarantool`. It might be necessary to say `sudo mkdir /usr/local/etc/default` first. Let the new file contents be:

```

default_cfg = {
  pid_file = "/tarantool_test/my_app.pid",
  wal_dir = "/tarantool_test",
  snap_dir = "/tarantool_test",
  vinyl_dir = "/tarantool_test",
  logger = "/tarantool_test/log",
  username = "tarantool",
}
instance_dir = "/tarantool_test"

```

Make the my_app application file, that is, /tarantool_test/my_app.lua. Let the file contents be:

```

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

```

Tell tarantoolctl to start the application ...

```

$ cd /tarantool_test
$ sudo tarantoolctl start my_app

```

... expect to see messages indicating that the instance has started. Then ...

```

$ ls -l /tarantool_test/my_app

```

... expect to see the .snap file and the .xlog file. Then ...

```

$ sudo less /tarantool_test/log/my_app.log

```

... expect to see the contents of my_app's log, including error messages, if any. Then ...

```

$ cd /tarantool_test
$ # assume that 'tarantool' invokes the tarantool server
$ sudo tarantool
tarantool> box.cfg{}
tarantool> console = require('console')
tarantool> console.connect('localhost:3301')
tarantool> box.space.tester:select({0}, {iterator = 'GE'})

```

... expect to see several tuples that my_app has created.

Stop. The only clean way to stop my_app is with tarantoolctl, thus:

```

$ sudo tarantoolctl stop my_app

```

Clean up. Restore the original contents of /usr/local/etc/default/tarantool, and ...

```
$ cd /
$ sudo rm -R tarantool_test
```

3.6.5 Утилита `tarantoolctl connect`

The `tarantoolctl connect` utility is a client program. Use it to connect to a tarantool server and pass requests.

To invoke the utility one says: `tarantoolctl connect URI` and the format of a URI is described *in the [URI section](#)*.

Например:

```
$ tarantoolctl connect username:password@127.0.0.1:3306
```

There are alternatives to `tarantoolctl connect` – one can use the *[console module](#)* or the *[net.box module](#)* from a Tarantool server. Also one can write one's own client programs with any of the Connectors. However, most of the examples in this manual illustrate usage with either `tarantoolctl connect` or with *[using the tarantool server as a client](#)*.

Statements about *[console.connect\(\)](#)* behavior are usually applicable for `tarantoolctl connect` as well. For example, an *[authentication trigger](#)* will be activated whenever `tarantoolctl connect` starts or ends.

3.6.6 Admin ports

"Admin port "admin console and "text protocol" all refer to the same thing: a connection which is set up with *[console.listen\(...\)](#)* for entry of requests by administrators.

"Binary port "binary protocol and "primary port" all refer to a different thing: a connection which is set up with *[box.cfg{listen=...}](#)* for entry of requests by anyone.

Ordinary connections to the Tarantool server should go via a binary port. But admin ports are useful for special cases involving security.

When one connects to an admin port: * No password is necessary * The user is automatically 'admin', a user with many privileges. Therefore one must set up admin ports very cautiously. If it is a TCP port, it should only be opened for a specific IP. Ideally it should not be a TCP port at all, it should be a Unix domain socket, so that access to the server machine is required. Thus a typical setup for an admin port is: `console.listen('/var/lib/tarantool/socket_name.sock')` and a typical connection *URI* is: `admin:any_string@/var/lib/tarantool/socket_name.sock` if the listener has the privilege to write on `/var/lib/tarantool` and the connector has the privilege to read on `/var/lib/tarantool`. Alternatively both setup and connection can be done with *[tarantoolctl](#)*.

If no administrator password exists which could be given out to users, and admin ports are restricted or are sockets, then requests which require 'admin' privileges can only occur locally, and are subject to Unix security and monitoring.

For additional security, some requests are illegal. For example, `"conn:eval"@` will result in the error message `error: console does not support this request type` because `conn:eval` requires the binary protocol.

If security via admin ports is not necessary, it is still possible to be an admin user by *[using the tarantool server as a client](#)*, or by connecting to a binary port with a valid password.

To find out whether a TCP port is an admin port, use `telnet`. For example:

```
$ telnet 0 3303
Trying 0.0.0.0...
Connected to 0.
Escape character is '^]'.
Tarantool 1.7.1-70-gbc479ad (Lua console)
type 'help' for interactive help
```

In this example the response does not include the word "binary" and does include the words "Lua console". Therefore it is clear that this is a successful connection to an admin port, and admin requests can now be entered on this terminal.

3.6.7 Заметки по администрированию на разных платформах

This section will contain information about issues or features which exist on some platforms but not others - for example, on certain versions of a particular Linux distribution.

Administrating with Debian GNU/Linux and Ubuntu

Setting up an instance:

```
$ ln -s /etc/tarantool/instances.available/instance-name.cfg /etc/tarantool/instances.enabled/
```

Starting all instances:

```
$ service tarantool start
```

Stopping all instances:

```
$ service tarantool stop
```

Starting/stopping one instance:

```
$ service tarantool-instance-name start/stop
```

Fedora, RHEL, CentOS

There are no known permanent issues. For transient issues, go to <http://github.com/tarantool/tarantool/issues> and enter "RHEL" or "CentOS" or "Fedora" or "Red Hat" in the search box.

FreeBSD

There are no known permanent issues. For transient issues, go to <http://github.com/tarantool/tarantool/issues> and enter "FreeBSD" in the search box.

Mac OS X

There are no known permanent issues. For transient issues, go to <http://github.com/tarantool/tarantool/issues> and enter "OS X" in the search box.

3.6.8 Заметки для пользователей systemd

Tarantool fully supports `systemd` for managing instances and supervising database daemons.

Instance management

Tarantool was designed to have multiple running instances of Tarantool on the same machine. Use `systemctl start/stop/restart/status tarantool@MYAPP` to manage your databases and Lua applications.

creating instances

Simply put your Lua configuration to `/etc/tarantool/instances.available/MYAPP.lua`:

```
box.cfg{listen = 3313}
require('myappcode').start()
```

(this minimal example is sufficient).

Another starting point could be the `example.lua` script that ships with Tarantool and defines all options.

starting instances

Use `systemctl start tarantool@MYAPP` to start `{MYAPP}` instance:

```
$ systemctl start tarantool@example
$ ps axuf|grep exampl[e]
taranto+ 5350 1.3 0.3 1448872 7736 ?        Ssl  20:05   0:28 tarantool example.lua <running>
```

(console examples here and further on are for Fedora).

Use `systemctl enable tarantool@MYAPP` to enable `{MYAPP}` instance for auto-load during system startup.

monitoring instances

Use `systemctl status tarantool@MYAPP` to check information about `{MYAPP}` instance:

```
$ systemctl status tarantool@example
tarantool@example.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@example.service
+ 5350 tarantool example.lua <running>
```

Use `journalctl -u tarantool@MYAPP` to check the boot log:

```
$ journalctl -u tarantool@example -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 example.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

```

attaching to instances

You can attach to a running Tarantool instance and evaluate some Lua code using the `tarantoolctl` utility:

```

$ tarantoolctl enter example
/bin/tarantoolctl: Found example.lua in /etc/tarantool/instances.available
/bin/tarantoolctl: Connecting to /var/run/tarantool/example.control
/bin/tarantoolctl: connected to unix:/var/run/tarantool/example.control
unix:/var/run/tarantool/example.control> 1 + 1
---
- 2
...
unix:/var/run/tarantool/example.control>

```

checking logs

Tarantool logs important events to `/var/log/tarantool/$MYAPP.log`.

Let's write something to the log file:

```

$ tarantoolctl enter example
/bin/tarantoolctl: Found example.lua in /etc/tarantool/instances.available
/bin/tarantoolctl: Connecting to /var/run/tarantool/example.control
/bin/tarantoolctl: connected to unix:/var/run/tarantool/example.control
unix:/var/run/tarantool/example.control> require('log').info("Hello for README.systemd readers")
---
...

```

Then check the logs:

```

$ tail /var/log/tarantool/example.log
2016-01-21 21:09:45.982 [5914] iproto I> binary: started
2016-01-21 21:09:45.982 [5914] iproto I> binary: bound to 0.0.0.0:3301
2016-01-21 21:09:45.983 [5914] main/101/tarantoolctl I> ready to accept requests
2016-01-21 21:09:45.983 [5914] main/101/example I> Run console at /var/run/tarantool/example.
↳ control
2016-01-21 21:09:45.984 [5914] main/101/example I> tcp_server: remove dead UNIX socket: /var/run/
↳ tarantool/example.control
2016-01-21 21:09:45.984 [5914] main/104/console/unix:/var/run/tarantool I> started
2016-01-21 21:09:45.985 [5914] main C> entering the event loop
2016-01-21 21:14:43.320 [5914] main/105/console/unix/: I> client unix/: connected
2016-01-21 21:15:07.115 [5914] main/105/console/unix/: I> Hello for README.systemd readers
2016-01-21 21:15:09.250 [5914] main/105/console/unix/: I> client unix/: disconnected

```

Log rotation is enabled by default if you have `logrotate` installed. Please configure `/etc/logrotate.d/tarantool` to change the default behavior.

stopping instances

Use `systemctl stop tarantool@MYAPP` to see information about the running `#{MYAPP}` instance.

```
$ systemctl stop tarantool@example
```

Daemon supervision

All instances are automatically restarted by `systemd` in case of failure.

Let's try to destroy an instance:

```
$ systemctl status tarantool@example|grep PID
Main PID: 5885 (tarantool)
$ tarantoolctl enter example
/bin/tarantoolctl: Found example.lua in /etc/tarantool/instances.available
/bin/tarantoolctl: Connecting to /var/run/tarantool/example.control
/bin/tarantoolctl: connected to unix:/var/run/tarantool/example.control
unix:/var/run/tarantool/example.control> os.exit(-1)
/bin/tarantoolctl: unix:/var/run/tarantool/example.control: Remote host closed connection
```

Now let's make sure that `systemd` has revived our Tarantool instance:

```
$ systemctl status tarantool@example|grep PID
Main PID: 5914 (tarantool)
```

Finally, let's check the boot logs:

```
$ journalctl -u tarantool@example -n 8
-- Logs begin at Fri 2016-01-08 12:21:53 MSK, end at Thu 2016-01-21 21:09:45 MSK. --
Jan 21 21:09:45 localhost.localdomain systemd[1]: tarantool@example.service: Unit entered failed_
↳state.
Jan 21 21:09:45 localhost.localdomain systemd[1]: tarantool@example.service: Failed with result
↳'exit-code'.
Jan 21 21:09:45 localhost.localdomain systemd[1]: tarantool@example.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 example.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.
```

Customizing the service file

Please don't modify the `tarantool@.service` file in-place, because it will be overwritten during package upgrades. It is recommended to copy this file to `/etc/systemd/system` and then modify the required settings. Alternatively, you can create a directory named `unit.d/` within `/etc/systemd/system` and put there a drop-in file `name.conf` that only changes the required settings. Please see `systemd.unit(5)` manual page for additional information.

Debugging

coredumpctl automatically saves core dumps and stack traces in case of a crash. Here is how it works:

```
$ # !!! please never do this on the production system !!!
$ tarantoolctl enter example
/bin/tarantoolctl: Found example.lua in /etc/tarantool/instances.available
/bin/tarantoolctl: Connecting to /var/run/tarantool/example.control
/bin/tarantoolctl: connected to unix:///var/run/tarantool/example.control
unix:///var/run/tarantool/example.control> require('ffi').cast('char *', 0)[0] = 48
/bin/tarantoolctl: unix:///var/run/tarantool/example.control: Remote host closed connection
```

coredumpctl list /usr/bin/tarantool displays the latest crashes of the Tarantool daemon:

```
$ 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
```

coredumpctl info <pid> shows the stack trace and other useful information:

```
$ 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 example.lua <running>
  Executable: /usr/bin/tarantool
  Control Group: /system.slice/system-tarantool.slice/tarantool@example.service
  Unit: tarantool@example.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)
...
```

coredumpctl -o filename.core info <pid> saves the core dump into a file.

coredumpctl gdb <pid> starts gdb on the core dump.

It is highly recommended to install the tarantool-debuginfo package to improve gdb experience. Example:


```
$ dnf debuginfo-install tarantool
```

`gdb` also provides information about the `debuginfo` packages you need to install:

```
$ # 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
```

Symbol names are present in stack traces even if you don't have the `tarantool-debuginfo` package installed.

For additional information, please refer to the documentation provided with your Linux distribution.

Precautions

- Please don't use `tarantoolctl` {start,stop,restart} to control instances started by `systemd`. It is still possible to use `tarantoolctl` to start and stop instances from your local directories (e.g. `$HOME`) without obtaining `ROOT` access.
- `tarantoolctl` is configured to work properly with ;program:`systemd`. Please don't modify system-wide settings of `tarantoolctl`, such as paths, directory permissions and usernames. Otherwise, you have a chance to shoot yourself in the foot.
- `systemd` scripts are maintained by the Tarantool Team (<http://tarantool.org>). Please file tickets directly to the upstream's bug tracker rather than to your Linux distribution.

3.6.9 Обновление Tarantool'а в условиях эксплуатации

First, put your application's business logic in a Tarantool-Lua module that exports its functions for `CALL`.

For example, `/usr/share/tarantool/myapp.lua`:

```
local function start()
-- Initial version
box.once("myapp:1.0", function()
box.schema.space.create("somedata")
box.space.somedata:create_index("primary")
...
end

-- migration code from 1.0 to 1.1
box.once("myapp:v1.1", function()
box.space.somedata.index.primary:alter(...)
...
end

-- migration code from 1.1 to 1.2
box.once("myapp:v1.2", function()
box.space.somedata.space:alter(...)
box.space.somedata:insert(...)
...
end
```

```

-- start some background fibers if you need

local function stop()
-- stop all background fibers and cleanup resources
end

local function api_for_call(xxx)
-- do some business
end

return {
start = start;
stop = stop;
api_for_call = api_for_call;
}

```

This file is maintained by the application's developers. On its side, Tarantool Team provides templates for you to [assemble deb/rpm packages](#) and utilities to quickly [assemble packages for specific platforms](#). If needed, you can split applications into standalone files and/or modules.

Second, put an initialization script to the `/etc/tarantool/instances.available` directory.

For example, `/etc/tarantool/instances.available/myappcfg.lua`:

```

#!/usr/bin/env tarantool

box.cfg {
listen = 3301;
}

if myapp ~= nil then
-- hot code reload using tarantoolctl or dofile()

-- unload old application
myapp.stop()
-- clear cache for loaded modules and dependencies
package.loaded['myapp'] = nil
package.loaded['somedep'] = nil; -- dependency of 'myapp'
end

-- load a new version of app and all dependencies
myapp = require('myapp').start({some app options controlled by sysadmins})

```

As a more detailed example, you can take the `example.lua` script that ships with Tarantool and defines all configuration options.

This initialization script is actually a configuration file and should be maintained by system administrators, while developers only provide a template.

Now update your app file in `/usr/share/tarantool`. Replace your application file (for example, `/usr/share/tarantool/myapp.lua`) and manually reload the `myappcfg.lua` initialization script using `tarantoolctl`:

```
$ tarantoolctl eval /etc/tarantool/instance.enabled/myappcfg.lua
```

After that, you need to manually flush the cache of `package.loaded` modules.

For `deb/rpm` packages, you can add the `tarantoolctl eval` instruction directly into Tarantool's

specification in `RPM.spec` and the `/debian` directory.

Finally, clients make a `CALL` to `myapp.api_for_call` and other API functions.

In the case of `tarantool-http`, there is no need to start the binary protocol at all.

3.6.10 Резервное копирование

The exact procedure for backing up a database depends on: how up-to-date the database must be, how frequently backups must be taken, whether it is okay to disrupt other users, and whether the procedure should be optimized for size (saving disk space) or for speed (saving time). So there is a spectrum of possible policies, ranging from cold-and-simple to hot-and-difficult.

Cold Backup

In essence: The last snapshot file is a backup of the entire database; and the WAL files that are made after the last snapshot are incremental backups. Therefore taking a backup is a matter of copying the snapshot and WAL files. (1) Prevent all users from writing to the database. This can be done by shutting down the server, or by saying `box.cfg{read_only=true}` and then ensuring that all earlier writes are complete (`fsync` can be used for this purpose). (2) If this is a backup of the whole database, say `box.snapshot()`. (3) Use `tar` to make a (possibly compressed) copy of the latest `.snap` and `.xlog` files on the `snap_dir` and `wal_dir` directories. (4) If there is a security policy, encrypt the tar file. (5) Copy the tar file to a safe place. ... Later, restoring the database is a matter of taking the tar file and putting its contents back in the `snap_dir` and `wal_dir` directories.

Continuous remote backup

In essence: *replication* is useful for backup as well as for load balancing. Therefore taking a backup is a matter of ensuring that any given replica is up to date, and doing a cold backup on it. Since all the other replicas continue to operate, this is not a cold backup from the end user's point of view. This could be done on a regular basis, with a cron job or with a Tarantool fiber.

Hot backup

In essence: The logged changes done since the last cold backup must be secured, while the system is running.

For this purpose one needs a "file copy" utility that will do the copying remotely and continuously, copying only the parts of a file that are changing. One such utility is `rsync`.

Alternatively, one needs an ordinary file copy utility, but there should be frequent production of new snapshot files or new WAL files as changes occur, so that only the new files need to be copied.

Note re storage engine: vinyl databases require additional steps.

3.6.11 Обновление Tarantool'a

This information applies for users who created databases with older versions of the Tarantool server, and have now installed a newer version. The request to make in this case is: `box.schema.upgrade()`.

For example, here is what happens when one runs `box.schema.upgrade()` with a database that was created in early 2015. Only a small part of the output is shown. `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 — ...`

3.7 Коннекторы

В этой главе описаны API для различных языков программирования.

3.7.1 Протокол

Tarantool protocol was designed with a focus on asynchronous I/O and easy integration with proxies. Each client request starts with a variable-length binary header, containing request id, request type, server id, log sequence number, and so on.

The mandatory length, present in request header simplifies client or proxy I/O. A response to a request is sent to the client as soon as it is ready. It always carries in its header the same type and id as in the request. The id makes it possible to match a request to a response, even if the latter arrived out of order.

Unless implementing a client driver, one needn't concern oneself with the complications of the binary protocol. Language-specific drivers provide a friendly way to store domain language data structures in Tarantool. A complete description of the binary protocol is maintained in annotated Backus-Naur form in the source tree: please see [iproto protocol](#).

3.7.2 Пример пакета

The Tarantool API exists so that a client program can send a request packet to the server, and receive a response. Here is an example of a what the client would send for `box.space[513]:insert{'A', 'BB'}`. The BNF description of the components is in file [iproto protocol](#).

Component	Byte #0	Byte #1	Byte #2	Byte #3
code for insert	02			
rest of header
2-digit number: space id	cd	02	01	
code for tuple	21			
1-digit number: field count = 2	92			
1-character string: field[1]	a1	41		
2-character string: field[2]	a2	42	42	

Now, one could send that packet to the tarantool server, and interpret the response ([iproto protocol](#) has a description of the packet format for responses as well as requests). But it would be easier, and less error-prone, if one could invoke a routine that formats the packet according to typed parameters. Something like `response=tarantool_routine("insert 513,"A "B");`. And that is why APIs exist for drivers for Perl, Python, PHP, and so on.

3.7.3 Примеры настроек для различных коннекторов

This chapter has examples that show how to connect to the Tarantool server via the Perl, PHP, Python, and C connectors. The examples contain hard code that will work if and only if the server (tarantool) is running on localhost (127.0.0.1) and is listening on port 3301 (`box.cfg.listen = '3301'`) and space 'examples' has id = 999 (`box.space.examples.id = 999`), and space 'examples' has a primary-key index for a numeric field (`box.space[999].index[0].parts[1].type = "unsigned"`) and user 'guest' has privileges for reading and writing.

It is easy to meet all the conditions by starting the server and executing this script:

```
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

See <http://github.com/tarantool/tarantool-java/>.

3.7.5 Go

Please see <https://github.com/mialinx/go-tarantool-1.6>.

3.7.6 R

See <https://github.com/thekvs/tarantoolr>.

3.7.7 Perl

The most commonly used Perl driver is `DR::Tarantool`. It is not supplied as part of the Tarantool repository; it must be installed separately. The most common way to install it is with [CPAN, the Comprehensive Perl Archive Network](#). `DR::Tarantool` requires other modules which should be installed first. For example, on Ubuntu, the installation could look like this:

```
$ sudo cpan install AnyEvent
$ sudo cpan install Devel::GlobalDestruction
$ sudo cpan install Coro
$ sudo cpan install Test::Pod
$ sudo cpan install Test::Spelling
$ sudo cpan install PAR::Dist
$ sudo cpan install List::MoreUtils
$ sudo cpan install DR::Tarantool
```

Here is a complete Perl program that inserts `[99999,'BB']` into `space[999]` via the Perl API. Before trying to run, check that the server is listening and that `examples` exists, as [described earlier](#). To run, paste the code into a file named `example.pl` and say `perl example.pl`. The program will connect using an application-specific definition of the space. The program will open a socket connection with the tarantool server at `localhost:3301`, then send an INSERT request, then — if all is well — end without displaying any messages. If tarantool is not running on localhost with listen address = 3301, the program will print “Connection refused”.

```
#!/usr/bin/perl
use DR::Tarantool ':constant', 'tarantool';
use DR::Tarantool ':all';
use DR::Tarantool::MsgPack::SyncClient;

my $tnt = DR::Tarantool::MsgPack::SyncClient->connect(
  host   => '127.0.0.1',           # look for tarantool on localhost
  port   => 3301,                 # on port 3301
  user   => 'guest',              # username. one could also say 'password=>...'

  spaces => {
    999 => {                       # definition of space[999] ...
      name => 'examples',          # space[999] name = 'examples'
      default_type => 'STR',      # space[999] field type is 'STR' if undefined
      fields => [ {               # definition of space[999].fields ...
        name => 'field1', type => 'NUM' } ], # space[999].field[1] name='field1',type='NUM'
      indexes => {               # definition of space[999] indexes ...
        0 => {
          name => 'primary', fields => [ 'field1' ] } } } } );
```

```
$tnt->insert('examples' => [ 99999, 'BB' ]);
```

The example program uses field type names 'STR' and 'NUM' instead of 'string' and 'unsigned', due to a temporary Perl limitation.

The example program only shows one command and does not show all that's necessary for good practice. For that, please see [DR::Tarantool](#) CPAN repository.

3.7.8 PHP

The PHP driver is [tarantool-php](#). It is not supplied as part of the Tarantool repository; it must be installed separately. It can be installed with git. It requires other modules which should be installed first. For example, on Ubuntu, the installation could look like this:

```
$ sudo apt-get install php5-cli
$ sudo apt-get install php5-dev
$ sudo apt-get install php-pear
$ cd ~
$ git clone https://github.com/tarantool/tarantool-php.git
$ cd tarantool-php
$ phpize
$ ./configure
$ make
$ # make install is optional
```

At this point there is a file named `~/tarantool-php/modules/tarantool.so`. PHP will only find it if the PHP initialization file `php.ini` contains a line like `extension=~/tarantool.so`, or if PHP is started with the option `-d extension=~/tarantool-php/modules/tarantool.so`.

Here is a complete PHP program that inserts `[99999,'BB']` into a space named 'examples' via the PHP API. Before trying to run, check that the server is listening and that `examples` exists, as [described earlier](#). To run, paste the code into a file named `example.php` and say `php -d extension=~/tarantool-php/modules/tarantool.so example.php`. The program will open a socket connection with the tarantool server at `localhost:3301`, then send an INSERT request, then — if all is well — print "Insert succeeded". If the tuple already exists, the program will print "Duplicate key exists in unique index 'primary' in space 'examples'".

```
<?php
$tarantool = new Tarantool('localhost', 3301);

try {
    $tarantool->insert('examples', array(99999, 'BB'));
    echo "Insert succeeded\n";
} catch (Exception $e) {
    echo "Exception: ", $e->getMessage(), "\n";
}
```

The example program only shows one command and does not show all that's necessary for good practice. For that, please see [tarantool-php](#) project at GitHub.

3.7.9 Python

Here is a complete Python program that inserts `[99999,'Value','Value']` into space `examples` via the high-level Python API.

```
#!/usr/bin/python
from tarantool import Connection

c = Connection("127.0.0.1", 3301)
result = c.insert("examples", (99999, 'Value', 'Value'))
print result
```

To prepare, paste the code into a file named `example.py` and install `tarantool-python` with either `pip install tarantool\>0.4` to install in `/usr` (requires **root** privilege) or `pip install tarantool\>0.4 --user` to install in `~` i.e. user's default directory. Before trying to run, check that the server is listening and that `examples` exists, as *described earlier*. To run the program, say `python example.py`. The program will connect to the server, will send the request, and will not throw an exception if all went well. If the tuple already exists, the program will throw `tarantool.error.DatabaseError: (3, "Duplicate key exists in unique index 'primary' in space 'examples'")`.

The example program only shows one request and does not show all that's necessary for good practice. For that, see <http://github.com/tarantool/tarantool-python>. For an example of a Python API for `Queue managers on Tarantool`, see <https://github.com/tarantool/tarantool-queue-python>.

3.7.10 C

`tnt_connect()`

Here follow two examples of using Tarantool's high-level C API.

Пример 1

Here is a complete C program that inserts `[99999, 'B']` into space `examples` via the high-level C API.

```
#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);           /* See note = SETUP */
    tnt_set(tnt, TNT_OPT_URI, "localhost:3301");
    if (tnt_connect(tnt) < 0) {                     /* See note = CONNECT */
        printf("Connection refused\n");
        exit(-1);
    }
    struct tnt_stream *tuple = tnt_object(NULL);     /* See note = MAKE REQUEST */
    tnt_object_format(tuple, "[%d%s]", 99999, "B");
    tnt_insert(tnt, 999, tuple);                     /* See note = SEND REQUEST */
    tnt_flush(tnt);
    struct tnt_reply reply; tnt_reply_init(&reply); /* See note = GET REPLY */
    tnt->read_reply(tnt, &reply);
    if (reply.code != 0) {
        printf("Insert failed %lu.\n", reply.code);
    }
    tnt_close(tnt);                                 /* See below = TEARDOWN */
    tnt_stream_free(tuple);
    tnt_stream_free(tnt);
}
```

Paste the code into a file named `example.c` and install `tarantool-c`. One way to install `tarantool-c` (using Ubuntu) is:

```
$ git clone git://github.com/tarantool/tarantool-c.git ~/tarantool-c
$ cd ~/tarantool-c
$ git submodule init
$ git submodule update
$ cmake .
$ make
$ make install
```

To compile and link the program, say:

```
$ # sometimes this is necessary:
$ export LD_LIBRARY_PATH=/usr/local/lib
$ gcc -o example example.c -ltarantool
```

Before trying to run, check that the server is listening and that the space `examples` exists, as [described earlier](#). To run the program, say `./example`. The program will connect to the server, and will send the request. If Tarantool is not running on localhost with listen address = 3301, the program will print "Connection refused". If the insert fails, the program will print "Insert failed" and an error number (see all error codes in the file [/src/box/errcode.h](#)).

Here are notes corresponding to comments in the example program.

SETUP: The setup begins by creating a stream.

```
struct tnt_stream *tnt = tnt_net(NULL);
tnt_set(tnt, TNT_OPT_URI, "localhost:3301");
```

In this program, the stream will be named `tnt`. Before connecting on the `tnt` stream, some options may have to be set. The most important option is `TNT_OPT_URI`. In this program, the URI is `localhost:3301`, since that is where the Tarantool server is supposed to be listening.

Function description:

```
struct tnt_stream *tnt_net(struct tnt_stream *s)
int tnt_set(struct tnt_stream *s, int option, variant option-value)
```

CONNECT: Now that the stream named `tnt` exists and is associated with a URI, this example program can connect to the server.

```
if (tnt_connect(tnt) < 0)
    { printf("Connection refused\n"); exit(-1); }
```

Function description:

```
int tnt_connect(struct tnt_stream *s)
```

The connection might fail for a variety of reasons, such as: the server is not running, or the URI contains an invalid password. If the connect fails, the return value will be `-1`.

MAKE REQUEST: Most requests require passing a structured value, such as the contents of a tuple.

```
struct tnt_stream *tuple = tnt_object(NULL);
tnt_object_format(tuple, "[%d%s]", 99999, "B");
```

In this program, the request will be an INSERT, and the tuple contents will be an integer and a string. This is a simple serial set of values, that is, there are no sub-structures or arrays. Therefore it is easy in this

case to format what will be passed using the same sort of arguments that one would use with a C `printf()` function: `%d` for the integer, `%s` for the string, then the integer value, then a pointer to the string value.

Function description:

```
ssize_t tnt_object_format(struct tnt_stream *s, const char *fmt, ...)
```

SEND REQUEST: The database-manipulation requests are analogous to the requests in the box library.

```
tnt_insert(tnt, 999, tuple);
tnt_flush(tnt);
```

In this program, the choice is to do an INSERT request, so the program passes the `tnt_stream` that was used for connection (`tnt`) and the stream that was set up with `tnt_object_format()` (`tuple`).

Function description:

```
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)
```

GET REPLY: For most requests, the client will receive a reply containing some indication whether the result was successful, and a set of tuples.

```
struct tnt_reply reply; tnt_reply_init(&reply);
tnt->read_reply(tnt, &reply);
if (reply.code != 0)
    { printf("Insert failed %lu.\n", reply.code); }
```

This program checks for success but does not decode the rest of the reply.

Function description:

```
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)
```

TEARDOWN: When a session ends, the connection that was made with `tnt_connect()` should be closed, and the objects that were made in the setup should be destroyed.

```
tnt_close(tnt);
tnt_stream_free(tuple);
tnt_stream_free(tnt);
```

Function description:

```
void tnt_close(struct tnt_stream *s)
void tnt_stream_free(struct tnt_stream *s)
```

Пример 2

Here is a complete C program that selects, using index key [99999], from space `examples` via the high-level C API. To display the results, the program uses functions in the `MsgPuck` library which allow decoding of `MessagePack` arrays.

```

#include <stdio.h>
#include <stdlib.h>
#include <tarantool/tarantool.h>
#include <tarantool/tnt_net.h>
#include <tarantool/tnt_opt.h>

#define MP_SOURCE 1
#include <msgpuck.h>

void main() {
    struct tnt_stream *tnt = tnt_net(NULL);
    tnt_set(tnt, TNT_OPT_URI, "localhost:3301");
    if (tnt_connect(tnt) < 0) {
        printf("Connection refused\n");
        exit(1);
    }
    struct tnt_stream *tuple = tnt_object(NULL);
    tnt_object_format(tuple, "[%d]", 99999); /* tuple = search key */
    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);
}

```

Similarly to the first example, paste the code into a file named `example2.c`.

To compile and link the program, say:

```
$ gcc -o example2 example2.c -ltarantool
```

To run the program, say `./example2`.

The two example programs only show a few requests and do not show all that's necessary for good practice. See more in the `tarantool-c` documentation at <http://github.com/tarantool/tarantool-c>.

3.7.11 Интерпретация возвращаемых значений

For all Connectors, calling a function via Tarantool causes a return in MsgPack format. If the function is called using the connector's API, some conversions may occur. All scalar values are returned as tuples (with a MsgPack type-identifier followed by a value); all non-scalar values are returned as a group of tuples (with a MsgPack array-identifier followed by the scalar values). If the function is called via the binary protocol command layer – "eval" rather than via the connector's API, no conversions occur.

In the following example, a Lua function will be created. Since it will be accessed externally by a 'guest' user, a `grant` of an execute privilege will be necessary. The function returns an empty array, a scalar string, two booleans, and a short integer. The values are the ones described in the msgpack section in the table *Common Types and MsgPack Encodings*.

```

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')
---
...

```

Here is a C program which calls the function. Although C is being used for the example, the result would be precisely the same if the calling program was written in Perl, PHP, Python, Go, or 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);           /* SETUP */
    tnt_set(tnt, TNT_OPT_URI, "localhost:3301");
    if (tnt_connect(tnt) < 0) {                       /* CONNECT */

```

```

    printf("Connection refused\n");
    exit(-1);
}
struct tnt_stream *tuple = tnt_object(NULL);      /* MAKE REQUEST */
struct tnt_stream *arg; arg = tnt_object(NULL);
tnt_object_add_array(arg, 0);
struct tnt_request *req1 = tnt_request_call(NULL); /* CALL function f() */
tnt_request_set_funcz(req1, "f");
tnt_request_set_tuple(req1, arg);
uint64_t sync1 = tnt_request_compile(tnt, req1);
tnt_flush(tnt);                                  /* SEND REQUEST */
struct tnt_reply reply; tnt_reply_init(&reply);  /* GET REPLY */
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; /* PRINT REPLY */
while (p < (unsigned char *) reply.data_end)
{
    printf("%x ", *p);
    ++p;
}
printf("\n");
tnt_close(tnt);                                  /* TEARDOWN */
tnt_stream_free(tuple);
tnt_stream_free(tnt);
}

```

When this program is executed, it will print: dd 0 0 0 5 90 91 a1 61 91 c2 91 c3 91 7f The first five bytes – dd 0 0 0 5 – are the msgpack encoding for "32-bit array header with value 5"(see the msgpack [Specification](#) page). The rest are as described in the Common Types and MsgPack Encodings table.

3.8 Модули

3.8.1 Модули, Lua Rocks и модули, подключаемые с помощью require

Возможности Tarantool'a можно расширять с помощью модулей. В языке Lua их называют "rocks". Тем, кто не знаком с языком Lua, мы рекомендуем пройти краткий обучающий курс по Lua ([Lua-Modules-Tutorial](#)) прежде, чем читать эту главу.

Install a module

The modules that come from Tarantool developers and community contributors are on rocks.tarantool.org. Some of them – *expirationd*, *mysql*, *postgresql*, *shard* – are discussed elsewhere in this manual.

Step 1: Install LuaRocks. A general description for installing LuaRocks on a Unix system is in the [LuaRocks-Quick-Start-Guide](#). For example on Ubuntu one could say: \$ sudo apt-get install luarocks

Step 2: Add the Tarantool repository to the list of rocks servers. This is done by putting rocks.tarantool.org in the `~/.luarocks/config.lua` file: \$ mkdir ~/.luarocks \$ echo "rocks_servers = {[[http://rocks.tarantool.org/]]}">> ~/.luarocks/config.lua

Once these steps are complete, the repositories can be searched with \$ luarocks search module-name and new modules can be added to the local repository with \$ luarocks install module-name -local and any module can be loaded for Tarantool with tarantool> local-name = require('module-name') ... and that

is why the examples in the manual's Modules section often begin with *require* requests. See [rocks on github.com/tarantool](https://github.com/tarantool) for more examples and information about contributing.

3.8.2 Пример: как создать у себя новый модуль на языке Lua

In this example, create a new Lua file named *mymodule.lua*, containing a named function which will be exported. Then, in Tarantool: load, examine, and call.

The Lua file should look like this:

```
-- mymodule - a simple Tarantool module
local exports = {}
exports.myfun = function(input_string)
    print('Hello', input_string)
end
return exports
```

The requests to load and examine and call look like this: tarantool>mymodule = require('mymodule') >- >...

```
tarantool>mymodule -->- myfun: 'function: 0x405edf20' >... tarantool>mymodule.myfun(os.getenv('USER'))
Hello world >- >...
```

3.8.3 Пример: как создать у себя новый модуль на языке C/C++

In this example, create a new C file named *mycmodule.c*, containing a named function which will be exported. Then, in Tarantool: load, examine, and call.

Prerequisite: install *tarantool-dev* first.

The C file should look like this:

```
/* mycmodule - a simple Tarantool module */
#include <lua.h>
#include <luaXlib.h>
#include <lualib.h>
#include <tarantool.h>
static int
myfun(lua_State *L)
{
    if (lua_gettop(L) < 1)
        return luaL_error(L, "Usage: myfun(name)");

    /* Get first argument */
    const char *name = lua_tostring(L, 1);

    /* Push one result to Lua stack */
    lua_pushfstring(L, "Hello, %s", name);
    return 1; /* the function returns one result */
}

LUA_API int
luaopen_mycmodule(lua_State *L)
{
    static const struct luaL_reg reg[] = {
        { "myfun", myfun },
        { NULL, NULL }
    };
```

```

};
luaL_register(L, "mycmodule", reg);
return 1;
}

```

Use gcc to compile the code for a shared library (without a "lib" prefix), then use ls to examine it:

```
$ gcc mycmodule.c -shared -fPIC -I/usr/include/tarantool -o mycmodule.so $ ls mycmodule.so -l -rwxr-xr-x
1 roman roman 7272 Jun 3 16:51 mycmodule.so
```

Tarantool's developers recommend use of Tarantool's [CMake-scripts](#) which will handle some of the build steps automatically.

```

The requests to load and examine and call look like this:
tarantool>mycmodule = require('mycmodule') -- ...
tarantool>mycmodule -- - myfun: 'function: 0x4100ec98' ...
tarantool>mycmodule.myfun(os.getenv('USER')) -- - Hello, world ...

```

One can also make modules with C++, provided that the code does not throw exceptions.

Tips for special situations

Lua caches all loaded modules in the `package.loaded` table. To reload a module from disk, set its key to `nil`: `tarantool> package.loaded['modulename'] = nil`

```

Use package.path to search for .lua modules, and use package.
cpath to search for C binary modules.
tarantool>package.path -- -
./?.lua;./?/init.lua;/home/roman/.luarocks/share/lua/5.1/?.lua;/home/roman/.luarocks/share/lua/5.1/?/init.lua;/home/ron
2.0.3/?.lua;/usr/local/share/lua/5.1/?.lua;/usr/local/share/lua/5.1/?/init.lua ...
tarantool>package.cpath -- -
./?.so;/home/roman/.luarocks/lib/lua/5.1/?.so;/home/roman/.luarocks/lib/lua/?/?.so;/usr/lib/tarantool/?/?.so;/usr/
... Substitute question-mark with modulename when calling require('modulename').

```

To see the internal state from within a Lua module, use `state` and create a local variable inside the scope of the file:

```

-- mymodule
local exports = {}
local state = {}
exports.myfun = function()
    state.x = 42 -- use state
end
return exports

```

Notice that the Lua examples use local variables. Use global variables with caution, since the module's users may be unaware of them.

To see a sample Lua + C module, go to [http](http://github.com/tarantool) on github.com/tarantool.

3.8.4 Модули на смеси языков Lua/C

- Create a Lua module, say `myfunmodule`.
- Create C module and name it `myfunmodule.internal` or something like that (submodule).
- Load C module from Lua code using `require('myfunmodule.internal')` and then wrap or use it.

3.9 Приложение А. Коды ошибок

In the current version of the binary protocol, error message, which is normally more descriptive than error code, is not present in server response. The actual message may contain a file name, a detailed reason or operating system error code. All such messages, however, are logged in the error log. Below follow only general descriptions of some popular codes. A complete list of errors can be found in file [errcode.h](#) in the source tree.

List of error codes

ER_NONMASTER	Can't modify data on a replication slave.
ER_ILLEGAL_PARAMS	Illegal parameters. Malformed protocol message.
ER_MEMORY_ISSUE	Out of memory: <i>slab_alloc_arena</i> limit has been reached.
ER_WAL_IO	Failed to write to disk. May mean: failed to record a change in the write-ahead log. Some sort of disk error.
ER_KEY_PART_COUNT	Key part count is not the same as index part count
ER_NO_SUCH_SPACE	The specified space does not exist.
ER_NO_SUCH_INDEX	The specified index in the specified space does not exist.
ER_PROC_LUA	An error occurred inside a Lua procedure.
ER_FIBER_STACK	Stack recursion limit was reached when creating a new fiber. This usually indicates that a stored procedure is recursively invoking itself too often.
ER_UPDATE_FIELD	Error occurred during update of a field.
ER_TUPLE_FOUND	Duplicate key exists in a unique index.

3.10 Приложение В. Детали реализации

This section is for advanced users or users who wish to know more about how the Tarantool program works. The discussion here is technical but generalized. The ultimate authority is the code.

3.10.1 Сохранность данных и формат WAL-файла

To maintain data persistence, Tarantool writes each data change request (INSERT, UPDATE, DELETE, REPLACE) into a write-ahead log (WAL) file in the *wal_dir* directory. A new WAL file is created for every *rows_per_wal* records. Each data change request gets assigned a continuously growing 64-bit log sequence number. The name of the WAL file is based on the log sequence number of the first record in the file, plus an extension *.xlog*.

Apart from a log sequence number and the data change request (its format is the same as in the binary protocol and is described in [doc/dev_guide/box-protocol.html](#)), each WAL record contains a header, some metadata, and then the data formatted according to *msgpack* rules. For example this is what the WAL file looks like after the first INSERT request ("s:insert({1})") for the introductory sandbox exercise *Starting Tarantool and making your first database*@"*Starting Tarantool and making your first database*". On the left are the hexadecimal bytes that one would see with:

```
$ hexdump 00000000000000000000.xlog
```

and on the right are comments.

Hex dump of WAL file	Comment
-----	-----
58 4c 4f 47 0a	File header: "XLOG\n"

```

30 2e 31 32 0a      File header: "0.12\n" = version
...                (not shown = more header + tuples for system spaces)
d5 ba 0b ab        Magic row marker always = 0xab0bbad5 if version 0.12
19 00              Length, not including length of header, = 25 bytes
ce 16 a4 38 6f     Record header: previous crc32, current crc32,
a7 cc 73 7f 00 00 66 39
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 "Double"
82                msgpack code meaning "map of 2 elements" follows
10 cd 02 00        element#1: tag=space id, value=512, big byte first
21 91 01           element#2: tag=tuple, value=1-element fixed array={1}

```

Tarantool processes requests atomically: a change is either accepted and recorded in the WAL, or discarded completely. Let's clarify how this happens, using the REPLACE request as an example:

1. The server attempts to locate the original tuple by primary key. If found, a reference to the tuple is retained for later use.
2. The new tuple is validated. If for example it does not contain an indexed field, or it has an indexed field whose type does not match the type according to the index definition, the change is aborted.
3. The new tuple replaces the old tuple in all existing indexes.
4. A message is sent to WAL writer running in a separate thread, requesting that the change be recorded in the WAL. The server switches to work on the next request until the write is acknowledged.
5. On success, a confirmation is sent to the client. On failure, a rollback procedure is begun. During the rollback procedure, the transaction processor rolls back all changes to the database which occurred after the first failed change, from latest to oldest, up to the first failed change. All rolled back requests are aborted with `ER_WAL_IO` error. No new change is applied while rollback is in progress. When the rollback procedure is finished, the server restarts the processing pipeline.

One advantage of the described algorithm is that complete request pipelining is achieved, even for requests on the same value of the primary key. As a result, database performance doesn't degrade even if all requests refer to the same key in the same space.

The transaction processor thread communicates with the WAL writer thread using asynchronous (yet reliable) messaging; the transaction processor thread, not being blocked on WAL tasks, continues to handle requests quickly even at high volumes of disk I/O. A response to a request is sent as soon as it is ready, even if there were earlier incomplete requests on the same connection. In particular, SELECT performance, even for SELECTs running on a connection packed with UPDATEs and DELETEs, remains unaffected by disk load.

The WAL writer employs a number of durability modes, as defined in configuration variable `wal_mode`. It is possible to turn the write-ahead log completely off, by setting `wal_mode` to `none`. Even without the write-ahead log it's still possible to take a persistent copy of the entire data set with the `box.snapshot()` request.

An `.xlog` file always contains changes based on the primary key. Even if the client requested an update or delete using a secondary key, the record in the `.xlog` file will contain the primary key.

3.10.2 Формат статического файла-снимка

The format of a snapshot `.snap` file is nearly the same as the format of a WAL `.xlog` file. However, the snapshot header differs: it contains the server's global unique identifier and the snapshot file's position in

history, relative to earlier snapshot files. Also, the content differs: an `.xlog` file may contain records for any data-change requests (inserts, updates, upserts, and deletes), a `.snap` file may only contain records of inserts to memtx spaces.

Primarily, the `.snap` file's records are ordered by space id. Therefore the records of system spaces, such as `_schema` and `_space` and `_index` and `_func` and `_priv` and `_cluster`, will be at the start of the `.snap` file, before the records of any spaces that were created by users.

Secondarily, the `.snap` file's records are ordered by primary key within space id.

3.10.3 Процесс восстановления

The recovery process begins when `box.cfg{}` happens for the first time after the Tarantool server starts.

The recovery process must recover the databases as of the moment when the server was last shut down. For this it may use the latest snapshot file and any WAL files that were written after the snapshot. One complicating factor is that Tarantool has two engines – the memtx data must be reconstructed entirely from the snapshot and the WAL files, while the vinyl data will be on disk but might require updating around the time of a checkpoint. (When a snapshot happens, Tarantool tells the vinyl engine to make a checkpoint, and the snapshot operation is rolled back if anything goes wrong, so vinyl's checkpoint is at least as fresh as the snapshot file.)

Step 1 Read the configuration parameters in the `box.cfg{}` request. Parameters which affect recovery may include `work_dir`, `wal_dir`, `snap_dir`, `vinyl_dir`, `panic_on_snap_error`, and `panic_on_wal_error`.

Step 2 Find the latest snapshot file. Use its data to reconstruct the in-memory databases. Instruct the vinyl engine to recover to the latest checkpoint.

There are actually two variations of the reconstruction procedure for the memtx databases, depending whether the recovery process is "default".

If it is default (`panic_on_snap_error` is `true` and `panic_on_wal_error` is `true`), memtx can read data in the snapshot with all indexes disabled. First, all tuples are read into memory. Then, primary keys are built in bulk, taking advantage of the fact that the data is already sorted by primary key within each space.

If it is not default (`panic_on_snap_error` is `false` or `panic_on_wal_error` is `false`), Tarantool performs additional checking. Indexes are enabled at the start, and tuples are added one by one. This means that any unique-key constraint violations will be caught, and any duplicates will be skipped. Normally there will be no constraint violations or duplicates, so these checks are only made if an error has occurred.

Step 2 Find the WAL file that was made at the time of, or after, the snapshot file. Read its log entries until the log-entry LSN is greater than the LSN of the snapshot, or greater than the LSN of the vinyl checkpoint. This is the recovery process's "start position"; it matches the current state of the engines.

Step 3 Redo the log entries, from the start position to the end of the WAL. The engine skips a redo instruction if it is older than the engine's checkpoint.

Step 4 For the memtx engine, re-create all secondary indexes.

3.10.4 Запуск сервера с репликацией

In addition to the recovery process described above, the server must take additional steps and precautions if `replication` is enabled.

Once again the startup procedure is initiated by the `box.cfg{}` request. One of the `box.cfg` parameters may be `replication_source`. We will refer to this server, which is starting up due to `box.cfg`, as the "local"server to distinguish it from the other servers in a cluster, which we will refer to as "distant"servers.

If there is no snapshot .snap file and replication_source is empty: then the local server assumes it is an unreplicated "standalone" server, or is the first server of a new replication cluster. It will generate new UUIDs for itself and for the cluster. The server UUID is stored in the `_cluster` space; the cluster UUID is stored in the `_schema` space. Since a snapshot contains all the data in all the spaces, that means the local server's snapshot will contain the server UUID and the cluster UUID. Therefore, when the local server restarts on later occasions, it will be able to recover these UUIDs when it reads the .snap file.

If there is no snapshot .snap file and replication_source is not empty and the _cluster space contains no other server UUIDs: then the local server assumes it is not a standalone server, but is not yet part of a cluster. It must now join the cluster. It will send its server UUID to the first distant server which is listed in `replication_source`, which will act as a master. This is called the "join request". When a distant server receives a join request, it will send back:

1. the distant server's cluster UUID,
2. the contents of the distant server's .snap file. When the local server receives this information, it puts the cluster UUID in its `_schema` space, puts the distant server's UUID and connection information in its `_cluster` space, and makes a snapshot containing all the data sent by the distant server. Then, if the local server has data in its WAL .xlog files, it sends that data to the distant server. The distant server will receive this and update its own copy of the data, and add the local server's UUID to its `_cluster` space.

If there is no snapshot .snap file and replication_source is not empty and the _cluster space contains other server UUIDs: then the local server assumes it is not a standalone server, and is already part of a cluster. It will send its server UUID and cluster UUID to all the distant servers which are listed in `replication_source`. This is called the "on-connect handshake". When a distant server receives an on-connect handshake:

1. the distant server compares its own copy of the cluster UUID to the one in the on-connect handshake. If there is no match, then the handshake fails and the local server will display an error.
2. the distant server looks for a record of the connecting instance in its `_cluster` space. If there is none, then the handshake fails. Otherwise the handshake is successful. The distant server will read any new information from its own .snap and .xlog files, and send the new requests to the local server.

In the end ... the local server knows what cluster it belongs to, the distant server knows that the local server is a member of the cluster, and both servers have the same database contents.

If there is a snapshot file and replication source is not empty: first the local server goes through the recovery process described in the previous section, using its own .snap and .xlog files. Then it sends a "subscribe" request to all the other servers of the cluster. The subscribe request contains the server vector clock. The vector clock has a collection of pairs 'server id, lsn' for every server in the `_cluster` system space. Each distant server, upon receiving a subscribe request, will read its .xlog files' requests and send them to the local server if (lsn of .xlog file request) is greater than (lsn of the vector clock in the subscribe request). After all the other servers of the cluster have responded to the local server's subscribe request, the server startup is complete.

The following temporary limitations apply for version 1.7:

- The URIs in `replication_source` should all be in the same order on all servers. This is not mandatory but is an aid to consistency.
- The servers of a cluster should be started up at slightly different times. This is not mandatory but prevents a situation where each server is waiting for the other server to be ready.
- The maximum number of entries in the `_cluster` space is 32. Tuples for out-of-date replicas are not automatically re-used, so if this 32-replica limit is reached, users may have to reorganize the `_cluster` space manually.

3.11 Приложение С. Учебные примеры на языке Lua

There are three tutorials: [Insert one million tuples with a Lua stored procedure](#), [Sum a JSON field for all tuples](#), [Indexed pattern search](#).

3.11.1 Вставка 1 млн таплов с помощью хранимой процедуры на языке Lua

This is an exercise assignment: “Insert one million tuples. Each tuple should have a constantly-increasing numeric primary-key field and a random alphabetic 10-character string field.”

The purpose of the exercise is to show what Lua functions look like inside Tarantool. It will be necessary to employ the Lua math library, the Lua string library, the Tarantool box library, the Tarantool box.tuple library, loops, and concatenations. It should be easy to follow even for a person who has not used either Lua or Tarantool before. The only requirement is a knowledge of how other programming languages work and a memory of the first two chapters of this manual. But for better understanding, follow the comments and the links, which point to the Lua manual or to elsewhere in this Tarantool manual. To further enhance learning, type the statements in with the tarantool client while reading along.

Configure

We are going to use the "tarantool_sandbox" that was created in section [first database](#). So there is a single space, and a numeric primary key, and a running tarantool server which also serves as a client.

Delimiter

In earlier versions of Tarantool, multi-line functions had to be enclosed within "delimiters". They are no longer necessary, and so they will not be used in this tutorial. However, they are still supported. Users who wish to use delimiters, or users of older versions of Tarantool, should check the syntax description for [declaring a delimiter](#) before proceeding.

Create a function that returns a string

We will start by making a function that returns a fixed string, “Hello world”.

```
function string_function()
    return "hello world"
end
```

The word "function" is a Lua keyword – we’re about to go into Lua. The function name is string_function. The function has one executable statement, return "hello world". The string "hello world" is enclosed in double quotes here, although Lua doesn’t care – one could use single quotes instead. The word "end" means “this is the end of the Lua function declaration.” To confirm that the function works, we can say

```
string_function()
```

Sending function-name() means “invoke the Lua function.” The effect is that the string which the function returns will end up on the screen.

For more about Lua strings see Lua manual [chapter 2.4 "Strings"](#) . For more about functions see Lua manual [chapter 5 "Functions"](#).

Теперь вывод на экране выглядит следующим образом:

```

tarantool> function string_function()
>   return "hello world"
> end
---
...
tarantool> string_function()
---
- hello world
...
tarantool>

```

Create a function that calls another function and sets a variable

Now that `string_function` exists, we can invoke it from another function.

```

function main_function()
  local string_value
  string_value = string_function()
  return string_value
end

```

We begin by declaring a variable "string_value". The word "local" means that string_value appears only in `main_function`. If we didn't use "local" then `string_value` would be visible everywhere - even by other users using other clients connected to this server! Sometimes that's a very desirable feature for inter-client communication, but not this time.

Then we assign a value to `string_value`, namely, the result of `string_function()`. Soon we will invoke `main_function()` to check that it got the value.

For more about Lua variables see Lua manual [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>

```

Modify the function so it returns a one-letter random string

Now that it's a bit clearer how to make a variable, we can change `string_function()` so that, instead of returning a fixed literal 'Hello world' it returns a random letter between 'A' and '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
```

It is not necessary to destroy the old `string_function()` contents, they're simply overwritten. The first assignment invokes a random-number function in Lua's math library; the parameters mean "the number must be an integer between 65 and 90." The second assignment invokes an integer-to-character function in Lua's string library; the parameter is the code point of the character. Luckily the ASCII value of 'A' is 65 and the ASCII value of 'Z' is 90 so the result will always be a letter between A and Z.

For more about Lua math-library functions see Lua users "[Math Library Tutorial](#)". For more about Lua string-library functions see Lua users "[String Library Tutorial](#)".

Once again the `string_function()` can be invoked from `main_function()` which can be invoked with `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>
```

... Well, actually it won't always look like this because `math.random()` produces random numbers. But for the illustration purposes it won't matter what the random string values are.

Modify the function so it returns a ten-letter random string

Now that it's clear how to produce one-letter random strings, we can reach our goal of producing a ten-letter string by concatenating ten one-letter strings, in a loop.

```
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
```

The words "for x = 1,10,1" mean "start with x equals 1, loop until x equals 10, increment x by 1 for each iteration." The symbol ".." means "concatenate that is, add the string on the right of the ".." sign to the string on the left of the ".." sign. Since we start by saying that `random_string` is (a blank string), the end result is that `random_string` has 10 random letters. Once again the `string_function()` can be invoked from `main_function()` which can be invoked with `main_function()`.

For more about Lua loops see Lua manual [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>
```

Make a tuple out of a number and a string

Now that it's clear how to make a 10-letter random string, it's possible to make a tuple that contains a number and a 10-letter random string, by invoking a function in Tarantool's library of Lua functions.

```
function main_function()
  local string_value, t
  string_value = string_function()
  t = box.tuple.new({1, string_value})
  return t
end
```

Once this is done, `t` will be the value of a new tuple which has two fields. The first field is numeric: 1. The second field is a random string. Once again the `string_function()` can be invoked from `main_function()` which can be invoked with `main_function()`.

For more about Tarantool tuples see Tarantool manual section [Submodule box.tuple](#).

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>
```

Modify main_function to insert a tuple into the database

Now that it's clear how to make a tuple that contains a number and a 10-letter random string, the only trick remaining is putting that tuple into tester. Remember that tester is the first space that was defined in the sandbox, so it's like a database table.

```
function main_function()
  local string_value, t
  string_value = string_function()
  t = box.tuple.new({1,string_value})
  box.space.tester:replace(t)
end
```

The new line here is `box.space.tester:replace(t)`. The name contains 'tester' because the insertion is going to be to tester. The second parameter is the tuple value. To be perfectly correct we could have said `box.space.tester:insert(t)` here, rather than `box.space.tester:replace(t)`, but "replace" means "insert even if there is already a tuple whose primary-key value is a duplicate", and that makes it easier to re-run the exercise even if the sandbox database isn't empty. Once this is done, tester will contain a tuple with two fields. The first field will be 1. The second field will be a random 10-letter string. Once again the `string_function()` can be invoked from `main_function()` which can be invoked with `main_function()`. But `main_function()` won't tell the whole story, because it does not return t, it only puts t into the database. To confirm that something got inserted, we'll use a SELECT request.

```
main_function()
box.space.tester:select{1}
```

For more about Tarantool insert and replace calls, see Tarantool manual section [Submodule box.space](#).

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>
```

Modify main_function to insert a million tuples into the database

Now that it's clear how to insert one tuple into the database, it's no big deal to figure out how to scale up: instead of inserting with a literal value = 1 for the primary key, insert with a variable value = between 1 and 1 million, in a loop. Since we already saw how to loop, that's a simple thing. The only extra wrinkle that we add here is a timing function.

```
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.testster:replace(t)
end
end
start_time = os.clock()
main_function()
end_time = os.clock()
'insert done in ' .. end_time - start_time .. ' seconds'

```

The `os.clock()` function will return the number of CPU seconds since the start. Therefore, by getting `start_time` = number of seconds just before the inserting, and then getting `end_time` = number of seconds just after the inserting, we can calculate $(end_time - start_time)$ = elapsed time in seconds. We will display that value by putting it in a request without any assignments, which causes Tarantool to send the value to the client, which prints it. (Lua's answer to the C `printf()` function, which is `print()`, will also work.)

For more on Lua `os.clock()` see Lua manual [chapter 22.1 "Date and Time"](#). For more on Lua `print()` see Lua manual [chapter 5 "Functions"](#).

Since this is the grand finale, we will redo the final versions of all the necessary requests: the request that created `string_function()`, the request that created `main_function()`, and the request that invokes `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.testster: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.testers: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>

```

What has been shown is that Lua functions are quite expressive (in fact one can do more with Tarantool's Lua stored procedures than one can do with stored procedures in some SQL DBMSs), and that it's straightforward to combine Lua-library functions and Tarantool-library functions.

What has also been shown is that inserting a million tuples took 37 seconds. The host computer was a Linux laptop. By changing *wal_mode* to 'none' before running the test, one can reduce the elapsed time to 4 seconds.

3.11.2 Подсчет суммы по JSON-полям во всех таплах

This is an exercise assignment: "Assume that inside every tuple there is a string formatted as JSON. Inside that string there is a JSON numeric field. For each tuple, find the numeric field's value and add it to a 'sum' variable. At end, return the 'sum' variable." The purpose of the exercise is to get experience in one way to read and process tuples.

```

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.testers: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

LINE 3: WHY "LOCAL". This line declares all the variables that will be used in the function. Actually it's not necessary to declare all variables at the start, and in a long function it would be better to declare variables just before using them. In fact it's not even necessary to declare variables at all, but an undeclared variable is "global". That's not desirable for any of the variables that are declared in line 1, because all of them are for use only within the function.

LINE 5: WHY "PAIRS()". Our job is to go through all the rows and there are two ways to do it: with `box.space.space_object:pairs()` or with `variable = select(...)` followed by `for i, n, 1 do some-function(variable[i]) end`. We preferred `pairs()` for this example.

LINE 5: START THE MAIN LOOP. Everything inside this "for" loop will be repeated as long as there is another index key. A tuple is fetched and can be referenced with variable `t`.

LINE 6: WHY "PCALL". If we simply said `lua_table = json.decode(t[2])`, then the function would abort with an error if it encountered something wrong with the JSON string - a missing colon, for example. By putting the function inside "pcall"(protected call), we're saying: we want to intercept that sort of error, so if there's a problem just set `is_valid_json = false` and we will know what to do about it later.

LINE 6: MEANING. The function is `json.decode` which means decode a JSON string, and the parameter is `t[2]` which is a reference to a JSON string. There's a bit of hard coding here, we're assuming that the second field in the tuple is where the JSON string was inserted. For example, we're assuming a tuple looks like

```
field[1]: 444
field[2]: '{"Hello": "world", "Quantity": 15}'
```

meaning that the tuple's first field, the primary key field, is a number while the tuple's second field, the JSON string, is a string. Thus the entire statement means "decode `t[2]` (the tuple's second field) as a JSON string; if there's an error set `is_valid_json = false`; if there's no error set `is_valid_json = true` and set `lua_table =` a Lua table which has the decoded string".

LINE 8. At last we are ready to get the JSON field value from the Lua table that came from the JSON string. The value in `field_name`, which is the parameter for the whole function, must be a name of a JSON field. For example, inside the JSON string `'{"Hello": "world" "Quantity": 15}'`, there are two JSON fields: "Hello" and "Quantity". If the whole function is invoked with `sum_json_field("Quantity")`, then `field_value = lua_table[field_name]` is effectively the same as `field_value = lua_table["Quantity"]` or even `field_value = lua_table.Quantity`. Those are just three different ways of saying: for the Quantity field in the Lua table, get the value and put it in variable `field_value`.

LINE 9: WHY "IF". Suppose that the JSON string is well formed but the JSON field is not a number, or is missing. In that case, the function would be aborted when there was an attempt to add it to the sum. By first checking `type(field_value) == "number"`, we avoid that abortion. Anyone who knows that the database is in perfect shape can skip this kind of thing.

And the function is complete. Time to test it. Starting with an empty database, defined the same way as the sandbox database that was introduced in [first database](#),

```
-- if tester is left over from some previous test, destroy it
box.space.tester:drop()
box.schema.space.create('tester')
box.space.tester:create_index('primary', {parts = {1, 'unsigned'}})
```

then add some tuples where the first field is a number and the second field is a string.

```
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}'}
```

Since this is a test, there are deliberate errors. The "golf club" and the "waffle iron" do not have numeric Quantity fields, so must be ignored. Therefore the real sum of the Quantity field in the JSON strings should be: $15 + 7 = 22$.

Invoke the function with `sum_json_field("Quantity")`.

```
tarantool> sum_json_field("Quantity")
---
- 22
...
```

It works. We'll just leave, as exercises for future improvement, the possibility that the "hard coding" assumptions could be removed, that there might have to be an overflow check if some field values are huge, and that the function should contain a "yield" instruction if the count of tuples is huge.

3.11.3 Индексированный поиск по шаблонам

Here is a generic function which takes a field identifier and a search pattern, and returns all tuples that match. * The field must be the first field of a TREE index. * The function will use [Lua pattern matching](#), which allows "magic characters" in regular expressions. * The initial characters in the pattern, as far as the first magic character, will be used as an index search key. For each tuple that is found via the index, there will be a match of the whole pattern. * To be *cooperative*, the function should yield after every 10 tuples, unless there is a reason to delay yielding. With this function, we can take advantage of Tarantool's indexes for speed, and take advantage of Lua's pattern matching for flexibility. It does everything that an SQL "LIKE" search can do, and far more.

Read the following Lua code to see how it works. The comments that begin with "SEE NOTE ..." refer to long explanations that follow the code.

```
function indexed_pattern_search(space_name, field_no, pattern)
  -- SEE NOTE #1 "FIND AN APPROPRIATE INDEX"
  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
  -- SEE NOTE #2 "DERIVE INDEX SEARCH KEY FROM PATTERN"
```

```

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
-- SEE NOTE #3 "OUTER LOOP: INITIATE"
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
  -- SEE NOTE #4 "INNER LOOP: ITERATOR"
  for _,tuple in box.space[space_name].index[index_no]:
  pairs(index_search_key,{iterator = box.index.GE}) do
    -- SEE NOTE #5 "INNER LOOP: BREAK IF INDEX KEY IS TOO GREAT"
    if (string.sub(tuple[field_no], 1, index_search_key_length)
        > index_search_key) then
      break
    end
    -- SEE NOTE #6 "INNER LOOP: BREAK AFTER EVERY 10 TUPLES -- MAYBE"
    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]
    -- SEE NOTE #7 "INNER LOOP: ADD TO RESULT SET IF PATTERN MATCHES"
    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
end
end

```

```

-- SEE NOTE #8 "OUTER LOOP: BREAK, OR YIELD AND CONTINUE"
if (is_time_for_a_yield ~= true) then
    break
end
require('fiber').yield()
end
return result_set
end

```

NOTE #1 "FIND AN APPROPRIATE INDEX" The caller has passed `space_name` (a string) and `field_no` (a number). The requirements are: (a) index type must be "TREE" because for other index types (HASH, BITSET, RTREE) a search with `iterator=GE` will not return strings in order by string value; (b) `field_no` must be the first index part; (c) the field must contain strings, because for other data types (such as "unsigned") pattern searches are not possible; If these requirements are not met by any index, then print an error message and return nil.

NOTE #2 "DERIVE INDEX SEARCH KEY FROM PATTERN" The caller has passed `pattern` (a string). The index search key will be the characters in the pattern as far as the first magic character. Lua's magic characters are `% ^ $ () . [] * + - ?`. For example, if the pattern is "ABC.E" the period is a magic character and therefore the index search key will be "ABC". But there is a complication ... If we see "%" followed by a punctuation character, that punctuation character is "escaped" so remove the "%" when making the index search key. For example, if the pattern is "AB%\$E" the dollar sign is escaped and therefore the index search key will be "AB\$E". Finally there is a check that the index search key length must be at least three – this is an arbitrary number, and in fact zero would be okay, but short index search keys will cause long search times.

NOTE #3 – "OUTER LOOP: INITIATE" The function's job is to return a result set, just as `box.space.select` would. We will fill it within an outer loop that contains an inner loop. The outer loop's job is to execute the inner loop, and possibly yield, until the search ends. The inner loop's job is to find tuples via the index, and put them in the result set if they match the pattern.

NOTE #4 "INNER LOOP: ITERATOR" The for loop here is using `pairs()`, see the [explanation of what index iterators are](#). Within the inner loop, there will be a local variable named "tuple" which contains the latest tuple found via the index search key.

NOTE #5 "INNER LOOP: BREAK IF INDEX KEY IS TOO GREAT" The iterator is GE (Greater or Equal), and we must be more specific: if the search index key has N characters, then the leftmost N characters of the result's index field must not be greater than the search index key. For example, if the search index key is 'ABC', then 'ABCDE' is a potential match, but 'ABD' is a signal that no more matches are possible.

NOTE #6 "INNER LOOP: BREAK AFTER EVERY 10 TUPLES – MAYBE" This chunk of code is for cooperative multitasking. The number 10 is arbitrary, and usually a larger number would be okay. The simple rule would be "after checking 10 tuples, yield, and then resume the search (that is, do the inner loop again) starting after the last value that was found". However, if the index is non-unique or if there is more than one field in the index, then we might have duplicates – for example {"ABC 1"}, {"ABC 2"}, {"ABC 3"} and it would be difficult to decide which "ABC" tuple to resume with. Therefore, if the result's index field is the same as the previous result's index field, there is no break.

NOTE #7 "INNER LOOP: ADD TO RESULT SET IF PATTERN MATCHES" Compare the result's index field to the entire pattern. For example, suppose that the caller passed `pattern "ABC.E"` and there is an indexed field containing "ABCDE". Therefore the initial index search key is "ABC". Therefore a tuple containing an indexed field with "ABCDE" will be found by the iterator, because "ABCDE" > "ABC". In that case `string.match` will return a value which is not nil. Therefore this tuple can be added to the result set.

NOTE #8 "OUTER LOOP: BREAK, OR YIELD AND CONTINUE" There are three conditions which will cause a break from the inner loop: (1) the for loop ends naturally because there are no more index keys

which are greater than or equal to the index search key, (2) the index key is too great as described in NOTE #5, (3) it is time for a yield as described in NOTE #6. If condition (1) or condition (2) is true, then there is nothing more to do, the outer loop ends too. If and only if condition (3) is true, the outer loop must yield and then continue. If it does continue, then the inner loop – the iterator search – will happen again with a new value for the index search key.

Например:

```
Start tarantool, cut and paste the code for function indexed_pattern_search, and try the
following: 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.") The result will be: tarantool> indexed_pattern_search("t 2, "ABC.E.") - - - [7, 'ABCDEF',
'a'] ...
```

3.12 Приложение D. Vinyl

3.12.1 Введение

Vinyl's features are:

- Full ACID compliance
- Multi-Version Concurrency Control (MVCC)
- Pure Append-Only
- Multi-threaded (Client access and Engine scalability)
- Multi-databases support (Single environment and WAL)
- Multi-Statement and Single-Statement Transactions (Snapshot Isolation (SI), multi-databases)
- Asynchronous or synchronous transaction execution (Callback triggered versus blocking)
- Separate storage formats: key-value (Default), or document (Keys are part of value)
- Update without read
- Consistent Cursors
- Prefix search
- Point-in-Time Snapshots
- Versional database creation and asynchronous shutdown/drop
- Asynchronous Online/Hot Backup
- Compression (Per region, both lz4 and zstd are supported)
- Metadata Compression (By default)
- Key Compression (Compress key duplicates, including suffixes)
- Easy to use (Minimalist API)
- Easy to integrate (Native support of using as storage engine)
- Easy to write bindings (Very FFI-friendly, API designed to be stable in future)

- Easy to build in (Amalgamated, compiles into two C files)
- Event loop friendly
- Zero-Configuration (Tuned by default)
- Implemented as a small library **written in C** with zero dependencies
- BSD Licensed

It is appropriate for databases that cannot fit in memory, where access via secondary keys is not required.

In vinyl terminology:

- There is one **Environment**.
- An Environment has N **Databases** - a vinyl database is like a Tarantool *space*.
- A Database has N **Ranges**.
- A Range has one **Range File**.
- A Range File has N **Runs**.
- A Run has N **Regions** - a vinyl Region is like a B-tree *page*.
- A Region has **keys** and **values** - a vinyl key-value is like a Tarantool *tuple*.

A key and its associated value are together, so when one accesses a key one gets the whole tuple. In other words, in vinyl the data is stored in the index. There are up to two in-memory copies of an index, as well as the copy in the Range File.

For operations that insert or update tuples - called Set operations in vinyl - vinyl makes changes to in-memory copies of the index, and writes to Tarantool's Write-ahead Log. A scheduler assigns tasks to multiple background threads for transferring index data from memory to disk, and for reorganizing Runs. To support transactions, Set operations can be delayed until an explicit commit. If multiple users access the same tuples simultaneously, the concurrency control method is [MVCC](#) and the isolation level is [Snapshot](#).

Formally, in terms of disk accesses, vinyl has the following algorithmic complexity:

- **Set** - the worst case is $O(1)$ append-only key writes to the Write-Ahead Log + in-memory Range index searches + in-memory index inserts
- **Delete** - the worst case is $O(1)$ key writes and in-memory index inserts (the same as **Set**)
- **Get** - the worst case is [amortized](#) $O(max_run_count_per_node)$ random Region reads from a single Range file, which itself does in-memory index search + in-memory Region search
- **Range** - queries, the worst case of full Database scan is amortized $O(total_Region_count)$ + in-memory key-index searches for each Range

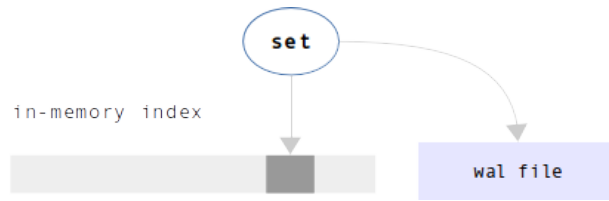
3.12.2 А что там "под капотом"?

In this section, to illustrate internals, we will discuss this example:

1. filling an empty database with one million tuples (we'll call them "keys" to emphasize the indexed nature)
2. reading all stored tuples in the original order.

Inserting the first 200.000 keys

During the first 200,000 Set operations, inserted keys first go to the in-memory index. To maintain persistence, information about each Set operation is written to Tarantool's Write-ahead Log.

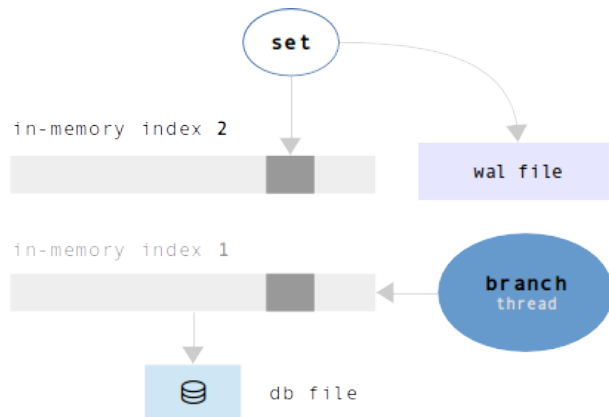


At this point we have keys in an in-memory index and records in the Write-ahead Log.

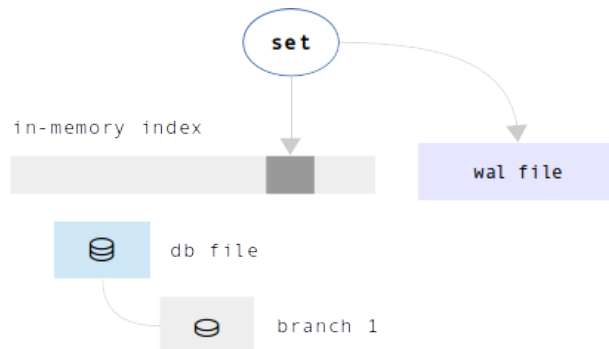
Inserting the next 300.000 keys

As the in-memory index becomes too large for available memory, the index must be copied from memory to disk. The on-disk copy of the in-memory index is called a Run. To save the Run, a new file is created, the Range File. We will call it **db file** for this example.

The scheduler wakes a worker thread in the background, a Run Creation Thread. The thread creates a second in-memory index. If there are Set operations taking place while the thread is working, their contention effect will be small because they will operate on the second in-memory index.

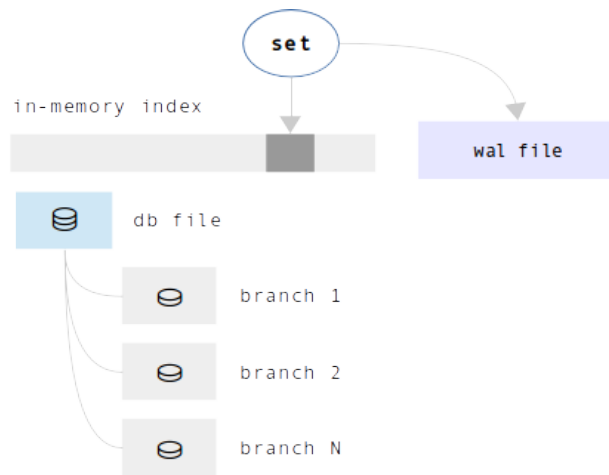


When the Run Creation Thread finishes the task, the first in-memory index is freed.

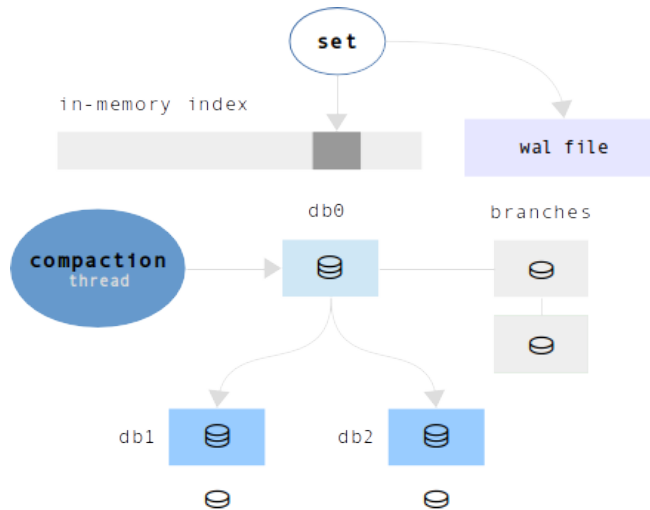


Inserting the next 200.000 keys

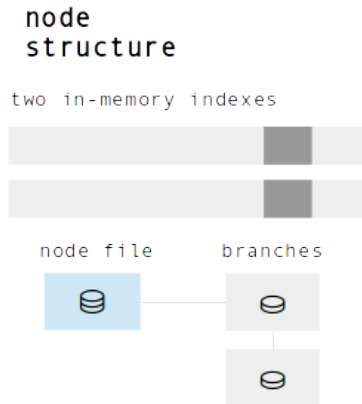
Several times, the in-memory index becomes too large and a Run Creation Thread transfers the keys to a Run. The Runs have been appended to the end of db file. The number of created Runs becomes large.



There is a user-settable maximum number of Runs per Range. When the number of Runs reaches this maximum, the vinyl scheduler wakes a **Compaction Thread** for the db file. The Compaction Thread merges the keys in all the Runs, and creates one or more new db files.



Now there are multiple pairs of in-memory indexes, and each pair has an associated db file. The combination of the in-memory indexes and the db file is called a **Range**, and the db file is called a **Range File**.



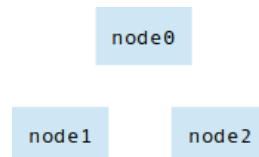
Thus the contents of a Range are: a range of sorted key values, stored in Runs of a Range File and (when necessary) in memory. Since the ranges do not overlap, each Range can be handled independently. Therefore, while one of the background threads is working on Range 1, another background thread can be working on Range 2, without contention. That means that all the background operations (Run Creation, Compaction, Garbage Collection, and Backup) can take place in parallel on multiple threads.

The foregoing explanation will now be repeated with different wording.

Before the Compaction there was one Range, which was created automatically when the Database was initialized. The Range had:

1. an in-memory index with some keys in it,
2. a Range File with several Runs,
3. a Write-Ahead Log file recording the Set operations, in the order they happened.

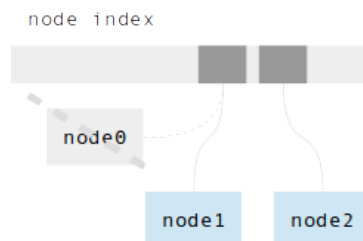
The number of Runs became too big, so the vinyl scheduler starts the Compaction Thread and creates two new Ranges.



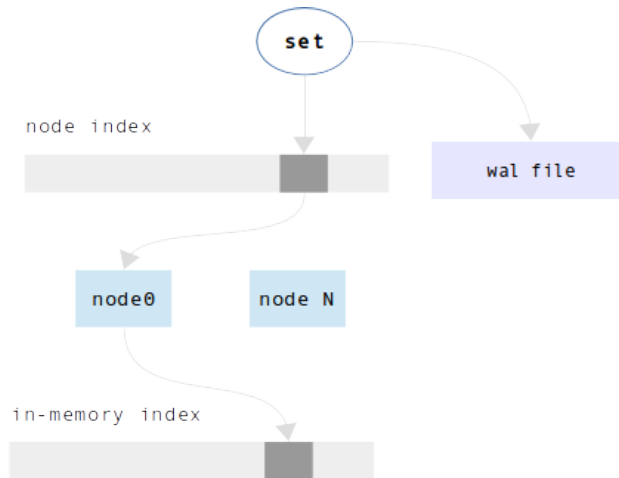
So, each of the two new Range Files contains half of the keys that were in the original Range. The Range's in-memory indexes are split in the same way.

After the splitting, vinyl must take into account that: while the Compaction was going on in the background, there might have been more Set operations taking place in parallel. These Set operations would have changed one of the in-memory indexes, and these changes too will be merged.

When the Compaction Thread finishes, the original Range is deleted, and information about the new Ranges is inserted into an in-memory **Range Index**.

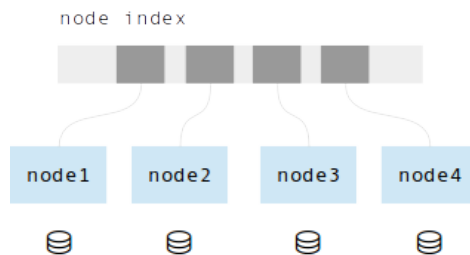


This Range Index is used for all Set operations and all searches. Since the Range Index has the minimum and maximum key values that are in each Range, it is straightforward to scan it to find what Range would contain a particular key value.



Inserting the last 300.000 keys

The final 300,000 Set operations take place; the background threads continue to create new Runs and do more Compactions. After the millionth insertion, the Database has four Ranges.

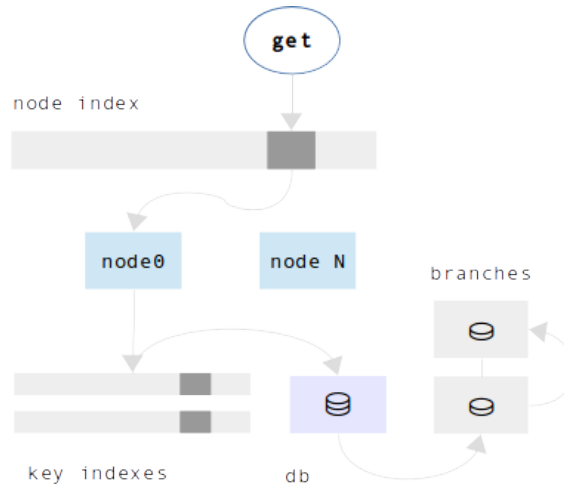


The inserting is done. Now, because the words "memory" and "disk" have appeared in this explanation several times, here are a few words about how vinyl is designed to use these resources most efficiently:

- If there is more memory available, then Run Creation and Compaction will be less frequent, and there will be fewer disk accesses.
- The best vinyl performance will occur if there is no setting of a memory limit, but this must be balanced against other considerations, such as requirements for the memtx storage engine. If there is a setting of a memory limit, the vinyl scheduler will give priority to the Ranges that have the largest in-memory indexes, so that the largest memory blocks are freed first.
- To make the most of hard drives and Flash, vinyl will delay operations that require disk access (except the writing of the Write-ahead Log which is specially tunable), so that the accesses are done in large sequential blocks.
- Overwriting does not occur; vinyl is an "append-only" engine.

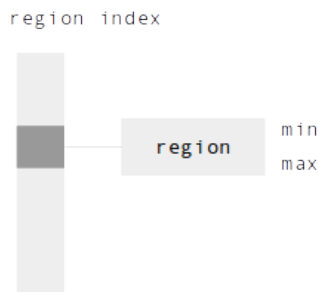
Reading million keys

We will now start to read the million rows in the order that they were inserted, which was random.

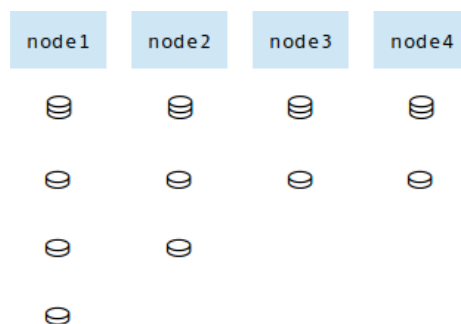


During the Get (search), vinyl first finds the correct Range by looking in the Range Index. Then it searches the Range's first in-memory index, and/or the Range's second in-memory index, and/or each Run of the Range, starting from the end of the Range File.

Remember that a Run is divided into Regions, which are like what would be called "pages" or "blocks" in a B-tree. For each Run, there is a list of the Regions and their minimum/maximum key values - the Region Index - as well as some metadata.



Region Indexes are loaded into memory when the Database is opened. Since the Database's Range Index and the Region Indexes are normally in-memory, searching and retrieving a tuple might require only zero or one disk accesses. However, when memory is limited and there are many Runs, search time may rise. For each additional Run there is a possible additional disk access during a search. Also, it is impossible to maintain memory limits without doing a Run Creation process, because new Set operations might occur more quickly than the Compaction process can run.



Vinyl is read optimized. It is very likely that the most recently created Runs (hot data) will be in the file system cache. The scheduler will give priority to the Ranges which have the largest in-memory indexes and

the most Runs.

The scheduler may also try to arrange that a Range will have only one Run, which will ensure the average number of disk seeks for each search is $O(1)$.

3.13 Приложение Е. Готовые рецепты

These are contributions of Lua programs for some frequent or tricky situations.

Any of the programs can be executed by copying the code into a `.lua` file, and then entering `chmod +x ./program-name.lua` and `./program-name.lua` on the terminal. As is usual for Tarantool/Lua programs, the first line is a "hashbang" `#!/usr/bin/env tarantool`. This runs the Tarantool Lua application server, which should be on the execution path.

Use freely.

hello_world.lua

The standard example of a simple program.

```
#!/usr/bin/env tarantool

print('Hello, World!')
```

console_start.lua

Use `box.once()` to initialize a database (creating spaces) if this is the first time the server has been run. Then use `console.start()` to start interactive mode.

```
#!/usr/bin/env tarantool

-- Configure database
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

Use the `fio module` to open, read, and close a file.

```
#!/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)
```

fiio_write.lua

Use the *fiio module* to open, write, and close a file.

```
#!/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

Use the [LuaJIT ffi library](#) to call a C built-in function: printf(). (For help understanding ffi, see the [FFI tutorial](#).)

```
#!/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

Use the [LuaJIT ffi library](#) to call a C function: gettimeofday(). This delivers time with millisecond precision, unlike the time function in Tarantool's *clock module*.

```
#!/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

Use the [LuaJIT ffi library](#) to call a C library function. (For help understanding ffi, see the [FFI tutorial](#).)

```
#!/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")

-- Lua wrapper for 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

-- Lua wrapper for 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

-- Simple test code.
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

Use the [LuaJIT ffi library](#) to access a C object via a metamethod (a method which is defined with a metatable).

```
#!/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

Create Lua tables, and print them. Notice that for the 'array' table the iterator function is `ipairs()`, while for the 'map' table the iterator function is `pairs()`. (*ipairs()* is faster than *pairs()*, but *pairs()* is recommended for map-like tables or mixed tables.) The display will look like: "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

Use the '#' operator to get the number of items in an array-like Lua table. This operation has $O(\log(N))$ complexity.

```

#!/usr/bin/env tarantool

array = { 1, 2, 3 }
print(#array)

```

count_array_with_nils.lua

Missing elements in arrays, which Lua treats as "nil"s, cause the simple "#" operator to deliver improper results. The "print(#t)" instruction will print "4"; the "print(counter)" instruction will print "3"; the "print(max)" instruction will print "10". Other table functions, such as `table.sort()`, will also misbehave when "nils" are present.

```

#!/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

Use explicit NULL values to avoid the problems caused by Lua's nil == missing value behavior. Although json.NULL == nil is true, all the print instructions in this program will print the correct value: 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

Get the number of elements in a map-like table.

```
#!/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

Use a Lua peculiarity to swap two variables without needing a third variable.

```
#!/usr/bin/env tarantool

local x = 1
local y = 2
x, y = y, x
print(x, y)
```

uri.lua

Use built-in function uri_parse to see what is in a *URI* <configuration-uri>:

```
#!/usr/bin/env tarantool

local uri = require('uri')
local r= uri.parse("scheme://login:password@host:service:/path1/path2/path3?q1=v1&q2=v2#fragment")
print('r.password=',r.password)
print('r.path=',r.path)
print('r.scheme',r.scheme)
print('r.login=',r.login)
print('r.query=',r.query)
print('r.service=',r.service)
print('r.fragment=',r.fragment)
print('r.host=',r.host)
```

class.lua

Create a class, create a metatable for the class, create an instance of the class. Another illustration is at <http://lua-users.org/wiki/LuaClassesWithMetatable>.

```
#!/usr/bin/env tarantool

-- define class objects
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;
    }
}

-- create a new object of myclass
local object = setmetatable({ data = 'data'}, myclass_mt)
print(object:somemethod())
print(object.data)
```

garbage.lua

Force Lua [garbage collection](#) with the [collectgarbage](#) function.

```
#!/usr/bin/env tarantool

collectgarbage('collect')
```

fiber_producer_and_consumer.lua

Start one fiber for producer and one fiber for consumer. Use [fiber.channel\(\)](#) to exchange data and synchronize. One can tweak the channel size (`ch_size` in the program code) to control the number of simultaneous tasks waiting for processing.

```
#!/usr/bin/env tarantool

local fiber = require('fiber')
local function consumer_loop(ch, i)
    -- initialize consumer synchronously or raise an error()
    fiber.sleep(0) -- allow fiber.create() to continue
    while true do
        local data = ch:get()
        if data == nil then
            break
        end
        print('consumed', i, data)
        fiber.sleep(math.random()) -- simulate some work
    end
end
```

```

end

local function producer_loop(ch, i)
  -- initialize consumer synchronously or raise an error()
  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

  -- Create a channel
  local ch_size = math.max(consumer_n, producer_n)
  local ch = fiber.channel(ch_size)

  -- Start consumers
  for i=1, consumer_n,1 do
    fiber.create(consumer_loop, ch, i)
  end

  -- Start producers
  for i=1, producer_n,1 do
    fiber.create(producer_loop, ch, i)
  end
end

start()
print('started')

```

socket_tcpconnect.lua

Use `socket.tcp_connect()` to connect to a remote host via TCP. Display the connection details and the result of a GET request.

```

#!/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

Use `socket.tcp_connect()` to set up a simple TCP server, by creating a function that handles requests and echos them, and passing the function to `socket.tcp_server()`. This program has been used to test with 100,000 clients, with each client getting a separate fiber.

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

local server, addr = require('socket').tcp_server('localhost', 3311, handler)
```

getaddrinfo.lua

Use `socket.getaddrinfo()` to perform non-blocking DNS resolution, getting both the AF_INET6 and AF_INET information for 'google.com'. This technique is not always necessary for tcp connections because `socket.tcp_connect()` performs `socket.getaddrinfo` under the hood, before trying to connect to the first available address.

```
#!/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

Tarantool does not currently have a `udp_server` function, therefore `socket_udp_echo.lua` is more complicated than `socket_tcp_echo.lua`. It can be implemented with sockets and fibers.

```
#!/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
    -- try to read a datagram first
    local msg, peer = s:recvfrom()
    if msg == "" then
      -- socket was closed via s:close()
      break
    elseif msg ~= nil then
```

```

    -- got a new datagram
    handler(s, peer, msg)
else
    if s:errno() == errno.EAGAIN or s:errno() == errno.EINTR then
        -- socket is not ready
        s:readable() -- yield, epoll will wake us when new data arrives
    else
        -- socket error
        local msg = s:error()
        s:close() -- save resources and don't wait GC
        error("Socket error: " .. msg)
    end
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 -- check errno:stderr()
    end
    if not s:bind(host, port) then
        local e = s:errno() -- save errno
        s:close()
        errno(e) -- restore errno
        return nil -- check errno:stderr()
    end

    fiber.create(udp_server_loop, s, handler) -- start a new background fiber
    return s
end
end

```

A function for a client that connects to this server could look something like this ...

```

local function handler(s, peer, msg)
    -- You don't have to wait until socket is ready to send 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.stderr())
end

print('Started')

require('console').start()

```

http_get.lua

Use the [http rock](#) (which must first be installed) to get data via 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

Use the [http rock](#) (which must first be installed) to send data via HTTP.

```

#!/usr/bin/env tarantool

local http_client = require('http.client')
local json = require('json')
local data = json.encode({ Key = 'Value'})
local headers = { Token = 'xxx', ['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

Use the [http rock](#) (which must first be installed) to turn Tarantool into a web server.

```

#!/usr/bin/env tarantool

local function handler(self)
    return self:render{ json = { ['Your-IP-Is'] = self.peer.host } }
end

local server = require('http.server').new(nil, 8080) -- listen *:8080
server:route({ path = '/' }, handler)
server:start()
-- connect to localhost:8080 and see json

```

http_generate_html.lua

Use the [http rock](#) (which must first be installed) to generate HTML pages from templates. The [http rock](#) has a fairly simple template engine which allows execution of regular Lua code inside text blocks (like PHP). Therefore there is no need to learn new languages in order to write templates.

```

#!/usr/bin/env tarantool

local function handler(self)
local fruits = { 'Apple', 'Orange', 'Grapefruit', 'Banana'}
    return self:render{ fruits = fruits }
end

local server = require('http.server').new(nil, 8080) -- nil means '*'
server:route({ path = '/', file = 'index.html.lua' }, handler)
server:start()

```

An "HTML" file for this server, including Lua, could look like this (it would produce "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>

```

3.14 Приложение F. Учебные примеры на языке C

There is one C tutorial: *C stored procedures*.

3.14.1 Работа с хранимыми процедурами на языке C

Tarantool can call C code with *modules*, or with *ffi*, or with C stored procedures. This tutorial only is about the third option, C stored procedures. In fact the routines are always "C functions" but the phrase "stored procedure" is commonly used for historical reasons.

In this tutorial, which can be followed by anyone with a Tarantool development package and a C compiler, there are three tasks. The first – *easy.c* – prints "hello world". The second – *harder.c* – decodes a passed parameter value. The third – *hardest.c* – uses the C API to do DBMS work.

After following the instructions, and seeing that the results are what is described here, users should feel confident about writing their own stored procedures.

Подготовка

Check that these items exist on the computer: * Tarantool 1.7 * A gcc compiler, any modern version should work * "module.h"* "msgpuck.h"

The "module.h" file will exist if Tarantool 1.7 was installed from source. Otherwise Tarantool's "developer" package must be installed. For example on Ubuntu say `sudo apt-get install tarantool-dev` or on Fedora say `dnf -y install tarantool-devel`

The "msgpuck.h" file will exist if Tarantool 1.7 was installed from source. Otherwise the "msgpuck" package must be installed from <https://github.com/rtsisyk/msgpuck>.

Both module.h and msgpuck.h must be on the include path for the C compiler to see them. For example, if module.h address is /usr/local/include/tarantool/module.h, and msgpuck.h address is /usr/local/include/msgpuck/msgpuck.h, and they are not currently on the include path, say `export CPATH=/usr/local/include/tarantool:/usr/local/include/msgpuck`

Requests will be done using tarantool as a *client*. Start tarantool, and enter these requests.

```

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)

```

In plainer language: create a space named `capi_test`, and make a connection to self named `capi_connection`.

Leave the client running. It will be necessary to enter more requests later.

easy.c

Start another shell. Change directory (cd) so that it is the same as the directory that the client is running on.

Create a file. Name it easy.c. Put these six lines in it.

```
#include "module.h"
int easy(box_function_ctx_t *ctx, const char *args, const char *args_end)
{
    printf("hello world\n");
    return 0;
}
```

Compile the program, producing a library file named easy.so: `gcc -shared -o easy.so -fPIC easy.c`

Теперь вернитесь в клиентский терминал и выполните следующие запросы:

```
box.schema.func.create('easy', {language = 'C'})
box.schema.user.grant('guest', 'execute', 'function', 'easy')
capi_connection:call('easy')
```

If these requests appear unfamiliar, re-read the descriptions of [box.schema.func.create](#) and [box.schema.user.grant](#) and [conn:call](#).

The function that matters is `capi_connection:call('easy')`.

Its first job is to find the 'easy' function, which should be easy because by default Tarantool looks on the current directory for a file named easy.so.

Its second job is to call the 'easy' function. Since the `easy()` function in `easy.c` begins with `printf("hello world\n")`, the words "hello world" will appear on the screen.

Its third job is to check that the call was successful. Since the `easy()` function in `easy.c` ends with `return 0`, there is no error message to display and the request is over.

Вывод на экране будет выглядеть следующим образом:

```
tarantool> capi_connection:call('easy')
hello world
---
- []
...
```

Conclusion: calling a C function is easy.

harder.c

Go back to the shell where the `easy.c` program was created.

Create a file. Name it `harder.c`. Put these 17 lines in it:

```
#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;
}

```

Compile the program, producing a library file named `harder.so`: `gcc -shared -o harder.so -fPIC harder.c`

Теперь вернитесь в клиентский терминал 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)

```

This time the call is passing a Lua table (`passable_table`) to the `harder()` function. The `harder()` function will see it, it's in the `char *args` parameter.

At this point the `harder()` function will start using functions defined in `msgpuck.h`, which are documented in <http://rtsisyk.github.io/msgpuck>. The routines that begin with "mp" are msgpuck functions that handle data formatted according to the [MsgPack](#) specification. Passes and returns are always done with this format so one must become acquainted with msgpuck to become proficient with the C API.

For now, though, it's enough to know that `mp_decode_array()` returns the number of elements in an array, and `mp_decode_uint` returns an unsigned integer, from `args`. And there's a side effect: when the decoding finishes, `args` has changed and is now pointing to the next element.

Therefore the first displayed line will be `"arg_count = 1"` because there was only one item passed: `passable_table`. The second displayed line will be `"field_count = 3"` because there are three items in the table. The next three lines will be `"1"` and `"2"` and `"3"` because those are the values in the items in the table.

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.
---
- []
...

```

Conclusion: decoding parameter values passed to a C function is not easy at first, but there are routines to do the job, and they're documented, and there aren't very many of them.

hardest.c

Go back to the shell where the `easy.c` and the `harder.c` programs were created.

Create a file. Name it `hardest.c`. Put these 13 lines in it:

```

#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];
    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;
}

```

Compile the program, producing a library file named `hardest.so`: `gcc -shared -o hardest.so -fPIC hardest.c`

Теперь вернитесь в клиентский терминал 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')

```

This time the C function is doing three things: (1) finding the numeric identifier of the "capi_test" space by calling `box_space_id_by_name()`; (2) formatting a tuple using more `msgpack.h` functions; (3) inserting a row using `box_insert`.

Now, still on the client, execute this request: `box.space.capi_test:select()`

Вывод на экране будет выглядеть следующим образом:

```

tarantool> box.space.capi_test:select()
---
- - [10000, 'String 2']
...

```

This proves that the `hardest()` function succeeded, but where did `box_space_id_by_name()` and `box_insert()` come from? Answer: the C API. The whole C API is documented [here](#). The function `box_space_id_by_name()` is documented [here](#). The function `box_insert()` is documented [here](#).

Conclusion: the long description of the C API is there for a good reason. All of the functions in it can be called from C functions which are called from Lua. So C "stored procedures" have full access to the database.

Cleaning up

Get rid of each of the function tuples with `box.schema.func.drop`, and get rid of the `capi_test` space with `box.schema.capi_test:drop()`, and remove the `.c` and `.so` files that were created for this tutorial.

An example in the test suite

Download the source code of Tarantool. Look in a subdirectory `test/box`. Notice that there is a file named `tuple_bench.test.lua` and another file named `tuple_bench.c`. Examine the Lua file and observe that it is calling a function in the C file, using the same techniques that this tutorial has shown.

Conclusion: parts of the standard test suite use C stored procedures, and they must work, because releases don't happen if Tarantool doesn't pass the tests.

3.15 Appendix G. Some significant changes in version 1.7

The disk-based storage engine, which was called "sophia" or "phia" in earlier versions, is superseded by the "vinyl" storage engine.

There are new types for indexed fields.

The LuaJIT version is updated.

Automatic replication cluster bootstrap (to make it easier to configure a new replication cluster) is supported.

The `space_object:inc()` function is removed.

The `space_object:dec()` function is removed.

Smaller feature changes and bug fixes can be seen by looking at [github.com](https://github.com/tarantool/tarantool/milestones) closed [milestones](#).

4.1 Module *box*

The contents of the `box` library can be inspected at runtime with `box`, with no arguments. The submodules inside the `box` library are: `box.schema`, `box.tuple`, `box.space`, `box.index`, `box.cfg`, `box.info`, `box.slab`, `box.stat`. Every submodule contains one or more Lua functions. A few submodules contain members as well as functions. The functions allow data definition (create alter drop), data manipulation (insert delete update upsert select replace), and introspection (inspecting contents of spaces, accessing server configuration).

4.2 Модуль *clock*

The `clock` module returns time values derived from the Posix / C `CLOCK_GETTIME` function or equivalent. Most functions in the module return a number of seconds; functions whose names end in "64" return a 64-bit number of nanoseconds.

```
clock.time()
clock.time64()
clock.realtime()
clock.realtime64()
```

The wall clock time. Derived from C function `clock_gettime(CLOCK_REALTIME)`. This is the best function for knowing what the official time is, as determined by the system administrator.

Return seconds or nanoseconds since epoch (1970-01-01 00:00:00), adjusted.

Rtype number or number64

Example:

```
-- This will print an approximate number of years since 1970.
clock = require('clock')
print(clock.time() / (365*24*60*60))
```

See also [*fiber.time64*](#) and [*os.clock\(\)*](#).

`clock.monotonic()`
`clock.monotonic64()`

The monotonic time. Derived from C function `clock_gettime(CLOCK_MONOTONIC)`. Monotonic time is similar to wall clock time but is not affected by changes to or from daylight saving time, or by changes done by a user. This is the best function to use with benchmarks that need to calculate elapsed time.

Return seconds or nanoseconds since the last time that the computer was booted.

Rtype number or number64

Example:

```
-- This will print nanoseconds since the start.
clock = require('clock')
print(clock.monotonic64())
```

`clock.proc()`
`clock.proc64()`

The processor time. Derived from C function `clock_gettime(CLOCK_PROCESS_CPUTIME_ID)`. This is the best function to use with benchmarks that need to calculate how much time has been spent within a CPU.

Return seconds or nanoseconds since processor start.

Rtype number or number64

Example:

```
-- This will print nanoseconds in the CPU since the start.
clock = require('clock')
print(clock.proc64())
```

`clock.thread()`
`clock.thread64()`

The thread time. Derived from C function `clock_gettime(CLOCK_THREAD_CPUTIME_ID)`. This is the best function to use with benchmarks that need to calculate how much time has been spent within a thread within a CPU.

Return seconds or nanoseconds since thread start.

Rtype number or number64

Example:

```
-- This will print seconds in the thread since the start.
clock = require('clock')
print(clock.thread64())
```

`clock.bench(function[, function parameters ...])`

The time that a function takes within a processor. This function uses `clock.proc()`, therefore it calculates elapsed CPU time. Therefore it is not useful for showing actual elapsed time.

Parameters:

- *function* = function or function reference;
- *function parameters* = whatever values are required by the function.

Return table. first element = seconds of CPU time; second element = whatever the function returns.

Rtype table

Example:

```
-- Benchmark a function which sleeps 10 seconds.
-- NB: bench() will not calculate sleep time.
-- So the returned value will be {a number less than 10, 88}.
clock = require('clock')
fiber = require('fiber')
function f(param)
  fiber.sleep(param)
  return 88
end
clock.bench(f,10)
```

4.3 Модуль *console*

The console module allows one Tarantool server to access another Tarantool server, and allows one Tarantool server to start listening on an *admin port*.

```
console.connect(uri)
```

Connect to the server at *URI*, change the prompt from 'tarantool>' to '*uri*>', and act henceforth as a client until the user ends the session or types control-D.

The console.connect function allows one Tarantool server, in interactive mode, to access another Tarantool server. Subsequent requests will appear to be handled locally, but in reality the requests are being sent to the remote server and the local server is acting as a client. Once connection is successful, the prompt will change and subsequent requests are sent to, and executed on, the remote server. Results are displayed on the local server. To return to local mode, enter control-D.

If the Tarantool server at *uri* requires authentication, the connection might look something like: console.connect('admin:secretpassword@distanthost.com:3301').

There are no restrictions on the types of requests that can be entered, except those which are due to privilege restrictions – by default the login to the remote server is done with user name = 'guest'. The remote server could allow for this by granting at least one privilege: box.schema.user.grant('guest', 'execute', 'universe').

Параметры

- *uri* (*string*) – the URI of the remote server

Return nil

Possible errors: the connection will fail if the target Tarantool server was not initiated with box.cfg{listen=...}.

Example:

```
tarantool> console = require('console')
---
...
tarantool> console.connect('198.18.44.44:3301')
---
...
198.18.44.44:3301> -- prompt is telling us that server is remote
```

`console.listen(uri)`

Listen on *URI*. The primary way of listening for incoming requests is via the connection-information string, or URI, specified in `box.cfg{listen=...}`. The alternative way of listening is via the URI specified in `console.listen(...)`. This alternative way is called "administrative" or simply "*admin port*"@. The listening is usually over a local host with a Unix domain socket.

Параметры

- `uri` (*string*) – the URI of the local server

The "admin" address is the URI to listen on. It has no default value, so it must be specified if connections will occur via an admin port. The parameter is expressed with URI = Universal Resource Identifier format, for example `"/tmpdir/unix_domain_socket.sock` or a numeric TCP port. Connections are often made with telnet. A typical port value is 3313.

Example:

```
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()`

Start the console on the current interactive terminal.

Example:

A special use of `console.start()` is with *initialization files*. Normally, if one starts the tarantool server with tarantool *initialization file* there is no console. This can be remedied by adding these lines at the end of the initialization file:

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

`console.ac([true/false])`

Set the auto-completion flag. If auto-completion is *true*, and the user is using tarantool as a client, then hitting the TAB key may cause tarantool to complete a word automatically. The default auto-completion value is *true*.

4.4 Модуль *crypto*

"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-0*, *SHA-1*, *SHA-2*). Some of the crypto functionality is also present in the *Модуль digest* module. The functions in crypto are:

```
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)
```

Pass or return a cipher derived from the string, key, and (optionally, sometimes) initialization vector. The four choices of algorithms:

- aes128 - aes-128 (with 192-bit binary strings using AES)
- aes192 - aes-192 (with 192-bit binary strings using AES)
- aes256 - aes-256 (with 256-bit binary strings using AES)
- des - des (with 56-bit binary strings using DES, though DES is not recommended)

Four choices of block cipher modes are also available:

- cbc - Cipher Block Chaining
- cfb - Cipher Feedback
- ecb - Electronic Codebook
- ofb - Output Feedback

For more information on, read article about [Encryption Modes](#)

Example:

```
crypto.cipher.aes192.cbc.encrypt('string', 'key', 'initialization')
crypto.cipher.aes256.ecb.decrypt('string', 'key', 'initialization')
```

```
crypto.digest.{dss|dss1|md4|md5|mdc2|ripemd160}(string)
crypto.digest.{sha|sha1|sha224|sha256|sha384|sha512}(string)
```

Pass or return a digest derived from the string. The twelve choices of algorithms:

- dss - dss (using DSS)
- dss1 - dss (using DSS-1)
- md4 - md4 (with 128-bit binary strings using MD4)
- md5 - md5 (with 128-bit binary strings using MD5)
- mdc2 - mdc2 (using MDC2)
- ripemd160 -
- sha - sha (with 160-bit binary strings using SHA-0)
- sha1 - sha-1 (with 160-bit binary strings using SHA-1)
- sha224 - sha-224 (with 224-bit binary strings using SHA-2)
- sha256 - sha-256 (with 256-bit binary strings using SHA-2)
- sha384 - sha-384 (with 384-bit binary strings using SHA-2)
- sha512 - sha-512 (with 512-bit binary strings using SHA-2).

Example:

```
crypto.digest.md4('string')
crypto.digest.sha512('string')
```


4.4.1 Incremental methods in the crypto module

Suppose that a digest is done for a string 'A', then a new part 'B' is appended to the string, then a new digest is required. The new digest could be recomputed for the whole string 'AB', but it is faster to take what was computed before for 'A' and apply changes based on the new part 'B'. This is called multi-step or "incremental" digesting, which Tarantool supports for all crypto functions..

```
crypto = require('crypto')

-- print aes-192 digest of 'AB', with one step, then incrementally
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()

-- print sha-256 digest of 'AB', with one step, then incrementally
print(crypto.digest.sha256('AB'))
c = crypto.digest.sha256.new()
c:init()
c:update('A')
c:update('B')
print(c:result())
c:free()
```

4.4.2 Getting the same results from digest and crypto modules

The following functions are equivalent. For example, the `digest` function and the `crypto` function will both produce the same result.

```
crypto.cipher.aes256.cbc.encrypt('string', 'key') == digest.aes256cbc.encrypt('string', 'key')
crypto.digest.md4('string') == digest.md4('string')
crypto.digest.md5('string') == digest.md5('string')
crypto.digest.sha('string') == digest.sha('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')
```

4.5 Module `csv`

The `csv` module handles records formatted according to Comma-Separated-Values (CSV) rules.

The default formatting rules are:

- Lua [escape sequences](#) such as `\n` or `\10` are legal within strings but not within files,
- Commas designate end-of-field,
- Line feeds, or line feeds plus carriage returns, designate end-of-record,
- Leading or trailing spaces are ignored,

- Quote marks may enclose fields or parts of fields,
- When enclosed by quote marks, commas and line feeds and spaces are treated as ordinary characters, and a pair of quote marks is treated as a single quote mark.

The possible options which can be passed to csv functions are:

- `delimiter` = *string* – single-byte character to designate end-of-field, default = comma
- `quote_char` = *string* – single-byte character to designate encloser of string, default = quote mark
- `chunk-size` = *number* – number of characters to read at once (usually for file-IO efficiency), default = 4096
- `skip_head_lines` = *number* – number of lines to skip at the start (usually for a header), default 0

`csv.load(readable[, {options}])`

Get CSV-formatted input from `readable` and return a table as output. Usually `readable` is either a string or a file opened for reading. Usually `options` is not specified.

Параметры

- `readable` (*object*) – a string, or any object which has a `read()` method, formatted according to the CSV rules
- `options` (*table*) – see [above](#)

Return loaded_value

Rtype table

Example:

Readable string has 3 fields, field#2 has comma and space so use quote marks:

```
tarantool> csv = require('csv')
---
...
tarantool> csv.load('a,"b,c ",d')
---
- - - a
  - 'b,c '
  - d
...

```

Readable string contains 2-byte character = Cyrillic Letter Palochka: (This displays a palochka if and only if character set = UTF-8.)

```
tarantool> csv.load('a\\211\\128b')
---
- - - a\\211\\128b
...

```

Semicolon instead of comma for the delimiter:

```
tarantool> csv.load('a;b;c;d', {delimiter = ';' })
---
- - - a,b
  - c,d
...

```

Readable file `./file.csv` contains two CSV records. Explanation of `fio` is in section [fio](#). Source CSV file and example respectively:

```

tarantool> -- input in file.csv is:
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(nn)
---
- true
...

```

`csv.dump(csv-table[, options, writable])`

Get table input from `csv-table` and return a CSV-formatted string as output. Or, get table input from `csv-table` and put the output in `writable`. Usually `options` is not specified. Usually `writable`, if specified, is a file opened for writing. `csv.dump()` is the reverse of `csv.load()`.

Параметры

- `csv-table` (*table*) – a table which can be formatted according to the CSV rules.
- `options` (*table*) – optional. see [above](#)
- `writable` (*object*) – any object which has a `write()` method

Return `dumped_value`

Rtype string, which is written to `writable` if specified

Example:

CSV-table has 3 fields, field#2 has "so result has quote marks

```

tarantool> csv = require('csv')
---
...
tarantool> csv.dump({'a', 'b,c ', 'd'})
---
- 'a,"b,c ",d
'
...

```

Round Trip: from string to table and back to string

```

tarantool> csv_table = csv.load('a,b,c')
---
...
tarantool> csv.dump(csv_table)
---
- 'a,b,c

```

```
'
...

```

`csv.iterate(input, {options})`

Form a Lua iterator function for going through CSV records one field at a time.

Параметры

- `csv-table (table)` – a table which can be formatted according to the CSV rules.
- `options (table)` – see [above](#)

Return Lua iterator function

Rtype iterator function

Example:

`csv.iterate()` is the low level of `csv.load()` and `csv.dump()`. To illustrate that, here is a function which is the same as the `csv.load()` function, as seen in [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
...

```

4.6 Модуль *digest*

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-0](#), [SHA-1](#), [SHA-2](#)) 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.

The functions in digest are:

`digest.aes256cbc.encrypt(string, key, iv)`

`digest.aes256cbc.decrypt(string, key, iv)`

Returns 256-bit binary string = digest made with AES.

`digest.md4(string)`

Returns 128-bit binary string = digest made with MD4.

`digest.md4_hex(string)`

Returns 32-byte string = hexadecimal of a digest calculated with md4.

`digest.md5(string)`

Returns 128-bit binary string = digest made with MD5.

`digest.md5_hex(string)`

Returns 32-byte string = hexadecimal of a digest calculated with md5.

`digest.sha(string)`

Returns 160-bit binary string = digest made with SHA-0. |br| Not recommended.

`digest.sha_hex(string)`

Returns 40-byte string = hexadecimal of a digest calculated with sha.

`digest.sha1(string)`

Returns 160-bit binary string = digest made with SHA-1.

`digest.sha1_hex(string)`

Returns 40-byte string = hexadecimal of a digest calculated with sha1.

`digest.sha224(string)`

Returns 224-bit binary string = digest made with SHA-2.

`digest.sha224_hex(string)`

Returns 56-byte string = hexadecimal of a digest calculated with sha224.

`digest.sha256(string)`

Returns 256-bit binary string = digest made with SHA-2.

`digest.sha256_hex(string)`

Returns 64-byte string = hexadecimal of a digest calculated with sha256.

`digest.sha384(string)`

Returns 384-bit binary string = digest made with SHA-2.

`digest.sha384_hex(string)`

Returns 96-byte string = hexadecimal of a digest calculated with sha384.

`digest.sha512(string)`

Returns 512-bit binary string = digest made with SHA-2.

`digest.sha512_hex(string)`

Returns 128-byte string = hexadecimal of a digest calculated with sha512.

`digest.base64_encode(string)`

Returns base64 encoding from a regular string.

`digest.base64_decode(string)`

Returns a regular string from a base64 encoding.

`digest.urandom(integer)`

Returns array of random bytes with length = integer.

`digest.crc32(string)`

Returns 32-bit checksum made with CRC32.

The `crc32` and `crc32_update` functions use the [CRC-32C \(Castagnoli\)](#) polynomial value: 0x1EDC6F41 / 4812730177. If it is necessary to be compatible with other checksum functions in other programming languages, ensure that the other functions use the same polynomial value.

For example, in Python, install the `crcmod` package and say:

```
>>> import crcmod
>>> fun = crcmod.mkCrcFun('4812730177')
>>> fun('string')
3304160206L
```

In Perl, install the `Digest::CRC` module and run the following code:

```
use Digest::CRC;
$d = Digest::CRC->new(width => 32, poly => 0x1EDC6F41, init => 0xFFFFFFFF, refin => 1, refout => 1);
$d->add('string');
print $d->digest;
```

(the expected output is 3304160206).

`digest.crc32.new()`

Initiates incremental crc32. See [incremental methods](#) notes.

`digest.guava(state, bucket)`

Returns a number made with consistent hash.

The guava function uses the [Consistent Hashing](#) algorithm of the Google guava library. The first parameter should be a hash code; the second parameter should be the number of buckets; the returned value will be an integer between 0 and the number of buckets. For example,

```
tarantool> digest.guava(10863919174838991, 11)
---
- 8
...
```

`digest.murmur(string)`

Returns 32-bit binary string = digest made with MurmurHash.

`digest.murmur.new([seed])`

Initiates incremental MurmurHash. See [incremental methods](#) notes.

4.6.1 Incremental methods in the digest module

Suppose that a digest is done for a string 'A', then a new part 'B' is appended to the string, then a new digest is required. The new digest could be recomputed for the whole string 'AB', but it is faster to take what was computed before for 'A' and apply changes based on the new part 'B'. This is called multi-step or "incremental" digesting, which Tarantool supports with `crc32` and with `murmur`...

```
digest = require('digest')

-- print crc32 of 'AB', with one step, then incrementally
print(digest.crc32('AB'))
c = digest.crc32.new()
c:update('A')
c:update('B')
print(c:result())

-- print murmur hash of 'AB', with one step, then incrementally
print(digest.murmur('AB'))
m = digest.murmur.new()
m:update('A')
m:update('B')
print(m:result())
```

4.6.2 Example

In the following example, the user creates two functions, `password_insert()` which inserts a [SHA-1](#) digest of the word "`^S^e^c^ret Wordpass`" into a tuple set, and `password_check()` which requires input of a password.

```
tarantool> digest = require('digest')
---
...
tarantool> function password_insert()
  > box.space.tester:insert{1234, digest.shal('^S^e^c^ret Wordpass')}
  > return 'OK'
  > end
---
...
tarantool> function password_check(password)
  > local t = box.space.tester:select{12345}
  > if digest.shal(password) == t[2] then
  >   return 'Password is valid'
  > else
  >   return 'Password is not valid'
  > end
  > end
---
...
tarantool> password_insert()
---
- 'OK'
...
```

If a later user calls the `password_check()` function and enters the wrong password, the result is an error.

```
tarantool> password_insert('Secret Password')
---
- 'Password is not valid'
...
```

4.7 Вложенный модуль `box.error`

The `box.error` function is for raising an error. The difference between this function and Lua's built-in `error()` function is that when the error reaches the client, its error code is preserved. In contrast, a Lua error would always be presented to the client as `ER_PROC_LUA`.

`box.error(reason=string[, code=number])`

When called with a Lua-table argument, the code and reason have any user-desired values. The result will be those values.

Параметры

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

`box.error()`

When called without arguments, `box.error()` re-throws whatever the last error was.

```
box.error(code, errtext[, errtext ...])
```

Emulate a request error, with text based on one of the pre-defined Tarantool errors defined in the file [errcode.h](#) in the source tree. Lua constants which correspond to those Tarantool errors are defined as members of `box.error`, for example `box.error.NO_SUCH_USER == 45`.

Параметры

- `code` (*number*) – number of a pre-defined error
- `errtext(s)` (*string*) – part of the message which will accompany the error

For example:

the `NO_SUCH_USER` message is "User '%s' is not found" includes one "%s" component which will be replaced with `errtext`. Thus a call to `box.error(box.error.NO_SUCH_USER, 'joe')` or `box.error(45, 'joe')` will result in an error with the accompanying message "User 'joe' is not found".

Except whatever is specified in `errcode-number`.

Example:

```
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()
```

Returns a description of the last error, as a Lua table with five members: "line"(number) Tarantool source file line number, "code"(number) error's number, "type"(string) error's C++ class, "message"(string) error's message, "file"(string) Tarantool source file. Additionally, if the error is a system error (for example due to a failure in socket or file io), there may be a sixth member: "errno"(number) C standard error number.

rtype: table

```
box.error.clear()
```

Clears the record of errors, so functions like `box.error()` or `box.error.last()` will have no effect.

Example:

```
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/build/tarantool-1.7.0.252.g1654e31~precise/src/box/key_def.cc
...
tarantool> box.error.clear()
---
...
tarantool> box.error.last()
---
- null
...

```

4.8 Модуль *fiber*

The *fiber* module allows for creating, running and managing *fibers*.

A fiber is a set of instructions which are executed with cooperative multitasking. Fibers managed by the fiber module are associated with a user-supplied function called the *fiber function*. A fiber has three possible states: **running**, **suspended** or **dead**. When a fiber is created with *fiber.create()*, it is running. When a fiber yields control with *fiber.sleep()*, it is suspended. When a fiber ends (because the fiber function ends), it is dead.

All fibers are part of the fiber registry. This registry can be searched with *fiber.find()* - via fiber id (fid), which is a numeric identifier.

A runaway fiber can be stopped with *fiber_object.cancel*. However, *fiber_object.cancel* is advisory — it works only if the runaway fiber calls *fiber.testcancel()* occasionally. Most *box.** functions, such as *box.space...delete()* or *box.space...update()*, do call *fiber.testcancel()* but *box.space...select{}* does not. In practice, a runaway fiber can only become unresponsive if it does many computations and does not check whether it has been cancelled.

The other potential problem comes from fibers which never get scheduled, because they are not subscribed to any events, or because no relevant events occur. Such morphing fibers can be killed with *fiber.kill()* at any time, since *fiber.kill()* sends an asynchronous wakeup event to the fiber, and *fiber.testcancel()* is checked whenever such a wakeup event occurs.

Like all Lua objects, dead fibers are garbage collected. The garbage collector frees pool allocator memory owned by the fiber, resets all fiber data, and returns the fiber (now called a fiber carcass) to the fiber pool. The carcass can be reused when another fiber is created.

A fiber has all the features of a Lua *coroutine* and all the programming concepts that apply for Lua coroutines will apply for fibers as well. However, Tarantool has made some enhancements for fibers and has used fibers internally. So, although use of coroutines is possible and supported, use of fibers is recommended.

```
fiber.create(function[, function-arguments])
```

Create and start a fiber. The fiber is created and begins to run immediately.

Параметры

- **function** – the function to be associated with the fiber
- **function-arguments** – what will be passed to function

Return created fiber object

Rtype userdata

Example:

```

tarantool> fiber = require('fiber')
---
...
tarantool> function function_name()
  > fiber.sleep(1000)
  > end
---
...
tarantool> fiber_object = fiber.create(function_name)
---
...

```

`fiber.self()`

Return fiber object for the currently scheduled fiber.

Rtype userdata

Example:

```

tarantool> fiber.self()
---
- status: running
  name: interactive
  id: 101
...

```

`fiber.find(id)`

Параметры

- `id` – numeric identifier of the fiber.

Return fiber object for the specified fiber.

Rtype userdata

Example:

```

tarantool> fiber.find(101)
---
- status: running
  name: interactive
  id: 101
...

```

`fiber.sleep(time)`

Yield control to the transaction processor thread and sleep for the specified number of seconds. Only the current fiber can be made to sleep.

Параметры

- `time` – number of seconds to sleep.

Example:

```

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

```

`fiber.yield()`

Yield control to the scheduler. Equivalent to `fiber.sleep(0)`.

Example:

```
tarantool> fiber.yield()
---
...
```

fiber.status()

Return the status of the current fiber.

Return the status of `fiber`. One of: “dead”, “suspended”, or “running”.

Rtype string

Example:

```
tarantool> fiber.status()
---
- running
...
```

fiber.info()

Return information about all fibers.

Return number of context switches, backtrace, id, total memory, used memory, name for each fiber.

Rtype table

Example:

```
tarantool> fiber.info()
---
- 101:
  csw: 7
  backtrace: []
  fid: 101
  memory:
    total: 65776
    used: 0
  name: interactive
...
```

fiber.kill(*id*)

Locate a fiber by its numeric id and cancel it. In other words, *fiber.kill()* combines *fiber.find()* and *fiber_object:cancel()*.

Параметры

- `id` – the id of the fiber to be cancelled.

Exception the specified fiber does not exist or cancel is not permitted.

Example:

```
tarantool> fiber.kill(fiber.id())
---
- error: fiber is cancelled
...
```

fiber.testcancel()

Check if the current fiber has been cancelled and throw an exception if this is the case.

Example:

```
tarantool> fiber.testcancel()
---
- error: fiber is cancelled
...
```

object fiber_object

fiber_object:id()

Параметры

- self – fiber object, for example the fiber object returned by [fiber.create](#)

Return id of the fiber.

Rtype number

Example:

```
tarantool> fiber_object = fiber.self()
---
...
tarantool> fiber_object:id()
---
- 101
...
```

fiber_object:name()

Параметры

- self – fiber object, for example the fiber object returned by [fiber.create](#)

Return name of the fiber.

Rtype string

Example:

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

fiber_object:name(*name*)

Change the fiber name. By default the Tarantool server's interactive-mode fiber is named 'interactive' and new fibers created due to [fiber.create](#) are named 'lua'. Giving fibers distinct names makes it easier to distinguish them when using [fiber.info](#).

Параметры

- self – fiber object, for example the fiber object returned by [fiber.create](#)
- name (*string*) – the new name of the fiber.

Return nil

Example:

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

`fiber_object:status()`

Return the status of the specified fiber.

Параметры

- `self` – fiber object, for example the fiber object returned by `fiber.create`

Return the status of fiber. One of: “dead”, “suspended”, or “running”.

Rtype string

Example:

```
tarantool> fiber.self():status()
---
- running
...
```

`fiber_object:cancel()`

Cancel a fiber. Running and suspended fibers can be cancelled. After a fiber has been cancelled, attempts to operate on it will cause errors, for example `fiber_object:id()` will cause `error: the fiber is dead`.

Параметры

- `self` – fiber object, for example the fiber object returned by `fiber.create`

Return nil

Possible errors: cancel is not permitted for the specified fiber object.

Example:

```
tarantool> fiber.self():cancel()
---
- error: fiber is cancelled
...
```

`fiber_object.storage`

Local storage within the fiber. The storage can contain any number of named values, subject to memory limitations. Naming may be done with `fiber_object.storage.name` or `fiber_object}.storage['name']`. or with a number `fiber_object.storage[number]`. Values may be either numbers or strings. The storage is garbage-collected when `fiber_object:cancel()` happens.

Example:

```
tarantool> fiber = require('fiber')
---
...
tarantool> function f () fiber.sleep(1000); end
---
...
tarantool> fiber_function = fiber:create(f)
---
- error: '[string "fiber_function = fiber:create(f)"]:1: fiber.create(function, ...):
  bad arguments'
```

```

...
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'
...

```

See also [box.session.storage](#).

`fiber.time()`

Return current system time (in seconds since the epoch) as a Lua number. The time is taken from the event loop clock, which makes this call very cheap, but still useful for constructing artificial tuple keys.

Rtype num

Example:

```

tarantool> fiber.time(), fiber.time()
---
- 1448466279.2415
- 1448466279.2415
...

```

`fiber.time64()`

Return current system time (in microseconds since the epoch) as a 64-bit integer. The time is taken from the event loop clock.

Rtype num

Example:

```

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

```

4.8.1 Example Of Fiber Use

Make the function which will be associated with the fiber. This function contains an infinite loop (`while 0 == 0` is always true). Each iteration of the loop adds 1 to a global variable named `gvar`, then goes to sleep for 2 seconds. The sleep causes an implicit [fiber.yield\(\)](#).

```

tarantool> fiber = require('fiber')
tarantool> function function_x()
  >   gvar = 0
  >   while 0 == 0 do
  >     gvar = gvar + 1
  >     fiber.sleep(2)
  >   end
  > end
---
...

```

Make a fiber, associate `function_x` with the fiber, and start `function_x`. It will immediately "detach" so it will be running independently of the caller.

```

tarantool> gvar = 0

tarantool> fiber_of_x = fiber.create(function_x)
---
...

```

Get the id of the fiber (`fid`), to be used in later displays.

```

tarantool> fid = fiber_of_x:id()
---
...

```

Pause for a while, while the detached function runs. Then ... Display the fiber id, the fiber status, and `gvar` (`gvar` will have gone up a bit depending how long the pause lasted). The status is suspended because the fiber spends almost all its time sleeping or yielding.

```

tarantool> print('#', fid, '. ', fiber_of_x:status(), '. gvar=', gvar)
# 102 . suspended . gvar= 399
---
...

```

Pause for a while, while the detached function runs. Then ... Cancel the fiber. Then, once again ... Display the fiber id, the fiber status, and `gvar` (`gvar` will have gone up a bit more depending how long the pause lasted). This time the status is dead because the cancel worked.

```

tarantool> fiber_of_x:cancel()
---
...
tarantool> print('#', fid, '. ', fiber_of_x:status(), '. gvar=', gvar)
# 102 . dead . gvar= 421
---
...

```

4.9 Модуль *fiber-ipc*

The `fiber-ipc` submodule allows sending and receiving messages between different processes. The words "different processes" in this context mean different connections, different sessions, or different fibers.

Call `fiber.channel()` to allocate space and get a channel object, which will be called `channel` for examples in this section. Call the other `fiber-ipc` routines, via `channel`, to send messages, receive messages, or check ipc status. Message exchange is synchronous. The channel is garbage collected when no one is using it, as

with any other Lua object. Use object-oriented syntax, for example `channel:put(message)` rather than `fiber.channel.put(message)`.

```
fiber.channel([capacity])
```

Create a new communication channel.

Параметры

- `capacity` (*int*) – positive integer as great as the maximum number of slots (spaces for `get` or `put` messages) that might be pending at any given time.

Return new channel.

Rtype userdata

object `channel_object`

```
channel_object:put(message[, timeout])
```

Send a message using a channel. If the channel is full, `channel:put()` blocks until there is a free slot in the channel.

Параметры

- `message` (*lua_object*) –
- `timeout` –

Return If `timeout` is provided, and the channel doesn't become empty for the duration of the timeout, `channel:put()` returns false. Otherwise it returns true.

Rtype boolean

```
channel_object:close()
```

Close the channel. All waiters in the channel will be woken up. All following `channel:put()` or `channel:get()` operations will return an error (`nil`).

```
channel_object:get([timeout])
```

Fetch a message from a channel. If the channel is empty, `channel:get()` blocks until there is a message.

Параметры

- `timeout` –

Return the value placed on the channel by an earlier `channel:put()`.

Rtype `lua_object`

```
channel_object:is_empty()
```

Check whether the specified channel is empty (has no messages).

Return true if the specified channel is empty

Rtype boolean

```
channel_object:count()
```

Find out how many messages are on the channel. The answer is 0 if the channel is empty.

Return the number of messages.

Rtype number

```
channel_object:is_full()
```

Check whether the specified channel is full.

Return true if the specified channel is full (has no room for a new message).

Rtype boolean

`channel_object:has_readers()`

Check whether the specified channel is empty and has readers waiting for a message (because they have issued `channel:get()` and then blocked).

Return true if blocked users are waiting. Otherwise false.

Rtype boolean

`channel_object:has_writers()`

Check whether the specified channel is full and has writers waiting (because they have issued `channel:put()` and then blocked due to lack of room).

Return true if blocked users are waiting. Otherwise false.

Rtype boolean

`channel_object:is_closed()`

Return true if the specified channel is already closed. Otherwise false.

Rtype boolean

4.9.1 Example

```

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 seconds
        local task = channel:get(10)
        if task ~= nil then
            ...
        else
            -- timeout
        end
    end
end

function producer_fiber()
    while true do
        task = box.space...:select{...}
        ...
        if channel:is_empty() then
            -- channel is empty
        end

        if channel:is_full() then
            -- channel is full
        end
    end
    ...

```

```

    if channel:has_readers() then
        -- there are some fibers
        -- that are waiting for data
    end
    ...

    if channel:has_writers() then
        -- there are some fibers
        -- that are waiting for readers
    end
    channel:put(task)
end
end

function producer2_fiber()
    while true do
        task = box.space...select{...}
        -- 10 seconds
        if channel:put(task, 10) then
            ...
        else
            -- timeout
        end
    end
end
end

```

4.10 Module *fio*

Tarantool supports file input/output with an API that is similar to POSIX syscalls. All operations are performed asynchronously. Multiple fibers can access the same file simultaneously.

4.10.1 Common pathname manipulations

`fio.pathjoin(partial-string [, partial-string ...])`

Concatenate partial string, separated by '/' to form a path name.

Параметры

- *partial-string* (*string*) – one or more strings to be concatenated.

Return path name

Rtype string

Example:

```

tarantool> fio.pathjoin('/etc', 'default', 'myfile')
---
- /etc/default/myfile
...

```

`fio.basename(path-name [, suffix])`

Given a full path name, remove all but the final part (the file name). Also remove the suffix, if it is passed.

Параметры

- `path-name` (*string*) – path name
- `suffix` (*string*) – suffix

Return file name

Rtype string

Example:

```
tarantool> fio.basename('/path/to/my.lua', '.lua')
---
- my
...
```

`fio.dirname(path-name)`

Given a full path name, remove the final part (the file name).

Параметры

- `path-name` (*string*) – path name

Return directory name, that is, path name except for file name.

Rtype string

Example:

```
tarantool> fio.dirname('path/to/my.lua')
---
- 'path/to/'
...
```

4.10.2 Common file manipulations

`fio.umask(mask-bits)`

Set the mask bits used when creating files or directories. For a detailed description type "man 2 umask".

Параметры

- `mask-bits` (*number*) – mask bits.

Return previous mask bits.

Rtype number

Example:

```
tarantool> fio.umask(tonumber('755', 8))
---
- 493
...
```

`fio.lstat(path-name)`

`fio.stat(path-name)`

Returns information about a file object. For details type "man 2 lstat" or "man 2 stat".

Параметры

- `path-name` (*string*) – path name of file.

Return fields which describe the file's block size, creation time, size, and other attributes.

Rtype table

Example:

```
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)`

Create or delete a directory. For details type "man 2 mkdir" or "man 2 rmdir".

Параметры

- `path-name` (*string*) – path of directory.
- `mode` (*number*) – Mode bits can be passed as a number or as string constants, for example "S_IWUSR". Mode bits can be combined by enclosing them in braces.

Return true if success, false if failure.

Rtype boolean

Example:

```
tarantool> fio.mkdir('/etc')
---
- false
...
```

`fio.glob(path-name)`

Return a list of files that match an input string. The list is constructed with a single flag that controls the behavior of the function: GLOB_NOESCAPE. For details type "man 3 glob".

Параметры

- `path-name` (*string*) – path-name, which may contain wildcard characters.

Return list of files whose names match the input string

Rtype table

Possible errors: nil.

Example:

```
tarantool> fio.glob('/etc/x*')
---
- - /etc/xdg
  - /etc/xml
```

```
- /etc/xul-ext
...
```

`fiio.tmpdir()`
Return the name of a directory that can be used to store temporary files.

Example:

```
tarantool> fio.tmpdir()
---
- /tmp/lG31e7
...
```

`fiio.cwd()`
Return the name of the current working directory.

Example:

```
tarantool> fio.cwd()
---
- /home/username/tarantool_sandbox
...
```

`fiio.link(src, dst)`

`fiio.symlink(src, dst)`

`fiio.readlink(src)`

`fiio.unlink(src)`

Functions to create and delete links. For details type "man readlink "man 2 link "man 2 symlink "man 2 unlink"..

Параметры

- `src` (*string*) – existing file name.
- `dst` (*string*) – linked name.

Return `fiio.link` and `fiio.symlink` and `fiio.unlink` return true if success, false if failure. `fiio.readlink` returns the link value if success, nil if failure.

Example:

```
tarantool> fio.link('/home/username/tmp.txt', '/home/username/tmp.txt2')
---
- true
...
tarantool> fio.unlink('/home/username/tmp.txt2')
---
- true
...
```

`fiio.rename(path-name, new-path-name)`

Rename a file or directory. For details type "man 2 rename".

Параметры

- `path-name` (*string*) – original name.
- `new-path-name` (*string*) – new name.

Return true if success, false if failure.

Rtype boolean

Example:

```
tarantool> fio.rename('/home/username/tmp.txt', '/home/username/tmp.txt2')
----
- true
...
```

`fio.chown(path-name, owner-user, owner-group)`

`fio.chmod(path-name, new-rights)`

Manage the rights to file objects, or ownership of file objects. For details type "man 2 chown" or "man 2 chmod".

Параметры

- `owner-user` (*string*) – new user uid.
- `owner-group` (*string*) – new group uid.
- `new-rights` (*number*) – new permissions

Example:

```
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)`

Reduce file size to a specified value. For details type "man 2 truncate".

Параметры

- `path-name` (*string*) –
- `new-size` (*number*) –

Return true if success, false if failure.

Rtype boolean

Example:

```
tarantool> fio.truncate('/home/username/tmp.txt', 99999)
----
- true
...
```

`fio.sync()`

Ensure that changes are written to disk. For details type "man 2 sync".

Return true if success, false if failure.

Rtype boolean

Example:

```
tarantool> fio.sync()
---
- true
...
```

`fio.open(path-name[, flags[, mode]])`

Open a file in preparation for reading or writing or seeking.

Параметры

- `path-name` (*string*) –
- `flags` (*number*) – Flags can be passed as a number or as string constants, for example `'O_RDONLY'`, `'O_WRONLY'`, `'O_RDWR'`. Flags can be combined by enclosing them in braces.
- `mode` (*number*) – Mode bits can be passed as a number or as string constants, for example `"S_IWUSR"`. Mode bits are significant if flags include `O_CREATE` or `O_TMPFILE`. Mode bits can be combined by enclosing them in braces.

Return file handle (later - `fh`)

Rtype userdata

Possible errors: nil.

Example:

```
tarantool> fh = fio.open('/home/username/tmp.txt', {'O_RDWR', 'O_APPEND'})
---
...
tarantool> fh -- display file handle returned by fio.open
---
- fh: 11
...
```

object file-handle

`file-handle:close()`

Close a file that was opened with `fio.open`. For details type "man 2 close".

Параметры

- `fh` (*userdata*) – file-handle as returned by `fio.open()`.

Return true if success, false on failure.

Rtype boolean

Example:

```
tarantool> fh:close() -- where fh = file-handle
---
- true
...
```

`file-handle:pread(count, offset)`

`file-handle:pwrite(new-string, offset)`

Perform read/write random-access operation on a file, without affecting the current seek position of the file. For details type "man 2 pread" or "man 2 pwrite".

Параметры

- `fh` (*userdata*) – file-handle as returned by `file.open()`.
- `count` (*number*) – number of bytes to read
- `new-string` (*string*) – value to write
- `offset` (*number*) – offset within file where reading or writing begins

Return `fh:write` returns true if success, false if failure. `fh:pread` returns the data that was read, or nil if failure.

Example:

```
tarantool> fh:pread(25, 25)
---
- |
  elete from t8//
  insert in
  ...
```

`file-handle:read(count)`

`file-handle:write(new-string)`

Perform non-random-access read or write on a file. For details type "man 2 read" or "man 2 write".

Примечание: `fh:read` and `fh:write` affect the seek position within the file, and this must be taken into account when working on the same file from multiple fibers. It is possible to limit or prevent file access from other fibers with `fiber.ipc`.

Параметры

- `fh` (*userdata*) – file-handle as returned by `file.open()`.
- `count` (*number*) – number of bytes to read
- `new-string` (*string*) – value to write

Return `fh:write` returns true if success, false if failure. `fh:read` returns the data that was read, or nil if failure.

Example:

```
tarantool> fh:write('new data')
---
- true
  ...
```

`file-handle:truncate(new-size)`

Change the size of an open file. Differs from `file.truncate`, which changes the size of a closed file.

Параметры

- `fh` (*userdata*) – file-handle as returned by `file.open()`.

Return true if success, false if failure.

Rtype boolean

Example:


```
tarantool> fh:truncate(0)
---
- true
...
```

`file-handle:seek(position[, offset-from])`

Shift position in the file to the specified position. For details type "man 2 seek".

Параметры

- `fh` (*userdata*) – file-handle as returned by `file-handle:open()`.
- `position` (*number*) – position to seek to
- `offset-from` (*string*) – 'SEEK_END' = end of file, 'SEEK_CUR' = current position, 'SEEK_SET' = start of file.

Return the new position if success

Rtype number

Possible errors: nil.

Example:

```
tarantool> fh:seek(20, 'SEEK_SET')
---
- 20
...
```

`file-handle:stat()`

Return statistics about an open file. This differs from `file-handle:stat` which return statistics about a closed file. For details type "man 2 stat".

Параметры

- `fh` (*userdata*) – file-handle as returned by `file-handle:open()`.

Return details about the file.

Rtype table

Example:

```
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()`

Ensure that file changes are written to disk, for an open file. Compare `file-handle:sync`, which is for all files. For details type "man 2 fsync" or "man 2 fdatasync".

Параметры

- `fh` (*userdata*) – file-handle as returned by `file-handle:open()`.

Return true if success, false if failure.

Example:

```
tarantool> fh:fsync()
---
- true
...
```

4.11 Module *fun*

Lua *fun*, also known as the Lua Functional Library, takes advantage of the features of LuaJIT to help users create complex functions. Inside the module are "sequence processors" such as `map`, `filter`, `reduce`, `zip` – they take a user-written function as an argument and run it against every element in a sequence, which can be faster or more convenient than a user-written loop. Inside the module are "generators" such as `range`, `tabulate`, and `rands` – they return a bounded or boundless series of values. Within the module are "reducers" "filters" "composers"... or, in short, all the important features found in languages like Standard ML, Haskell, or Erlang.

The full documentation is [On the luafun section of github](#). However, the first chapter can be skipped because installation is already done, it's inside Tarantool. All that is needed is the usual `require` request. After that, all the operations described in the Lua *fun* manual will work, provided they are preceded by the name returned by the `require` request. For example:

```
tarantool> fun = require('fun')
---
...
tarantool> for _k, a in fun.range(3) do
>   print(a)
> end
1
2
3
---
...
```

4.12 Module *jit*

The *jit* module has functions for tracing the LuaJIT Just-In-Time compiler's progress, showing the bytecode or assembler output that the compiler produces, and in general providing information about what LuaJIT does with Lua code.

`jit.bc.dump(function)`

Prints the byte code of a function.

Example:

```
function f()
  print("D")
end
jit.bc.dump(f)
```

For a list of available options, read [the source code of bc.lua](#).

`jit.dis_x86.disass(string)`
Prints the i386 assembler code of a string of bytes

Example:

```
-- Disassemble hexadecimal 97 which is the x86 code for xchg eax, edi
jit.dis_x86.disass('\x97')
```

For a list of available options, read [the source code of dis_x86.lua](#).

`jit.dis_x64.disass(string)`
Prints the x86-64 assembler code of a string of bytes

Example:

```
-- Disassemble hexadecimal 97 which is the x86-64 code for xchg eax, edi
jit.dis_x64.disass('\x97')
```

For a list of available options, read [the source code of dis_x64.lua](#).

`jit.dump.on(option[, output file])`
`jit.dump.off()`
Prints the intermediate or machine code of following Lua code

Example:

```
-- Show the machine code of a Lua "for" loop
jit.dump.on('m')
local x = 0;
for i = 1, 1e6 do
  x = x + i
end
print(x)
jit.dump.off()
```

For a list of available options, read [the source code of dump.lua](#).

`jit.v.on(option[, output file])`
`jit.v.off()`
Prints a trace of LuaJIT's progress compiling and interpreting code

Example:

```
-- Show what LuaJIT is doing for a Lua "for" loop
jit.v.on()
local x = 0
for i = 1, 1e6 do
  x = x + i
end
print(x)
jit.v.off()
```

For a list of available options, read [the source code of v.lua](#).

4.13 Module *json*

The `json` module provides JSON manipulation routines. It is based on the [Lua-CJSON module](#) by Mark Pulford. For a complete manual on Lua-CJSON please read [the official documentation](#).

`json.encode(lua-value)`

Convert a Lua object to a JSON string.

Параметры

- `lua_value` – either a scalar value or a Lua table value.

Return the original value reformatted as a JSON string.

Rtype string

Example:

```
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)`

Convert a JSON string to a Lua object.

Параметры

- `string` (*string*) – a string formatted as JSON.

Return the original contents formatted as a Lua table.

Rtype table

Example:

```
tarantool> json = require('json')
---
...
tarantool> json.decode('123')
---
- 123

```

```

...
tarantool> json.decode('[123, "hello"]')
---
- [123, 'hello']
...
tarantool> json.decode('{"hello": "world"}').hello
---
- world
...

```

json.NULL

A value comparable to Lua "nil" which may be useful as a placeholder in a tuple.

Example:

```

-- When nil is assigned to a Lua-table field, the field is null
tarantool> {nil, 'a', 'b'}
---
- - null
- a
- b
...
-- When json.NULL is assigned to a Lua-table field, the field is json.NULL
tarantool> {json.NULL, 'a', 'b'}
---
- - null
- a
- b
...
-- When json.NULL is assigned to a JSON field, the field is null
tarantool> json.encode({field2 = json.NULL, field1 = 'a', field3 = 'c'})
---
- '{"field2":null,"field1":"a","field3":"c"}'
...

```

The JSON output structure can be specified with `__serialize`:

- `__serialize="seq"` for an array
- `__serialize="map"` for a map

Serializing 'A' and 'B' with different `__serialize` values causes different results:

```

tarantool> json.encode(setmetatable({'A', 'B'}, { __serialize="seq"}))
---
- '["A","B"]'
...
tarantool> json.encode(setmetatable({'A', 'B'}, { __serialize="map"}))
---
- '{"1":"A","2":"B"}'
...
tarantool> json.encode({setmetatable({f1 = 'A', f2 = 'B'}, { __serialize="map"})})
---
- '[{"f2":"B","f1":"A"}]'
...
tarantool> json.encode({setmetatable({f1 = 'A', f2 = 'B'}, { __serialize="seq"})})
---
- '[]'
...

```

4.13.1 Configuration settings

There are configuration settings which affect the way that Tarantool encodes invalid numbers or types. They are all boolean `true/false` values

- `cfg.encode_invalid_numbers` - allow nan and inf (default is true)
- `cfg.encode_use_tostring` - use tostring for unrecognizable types (default is false)
- `cfg.encode_invalid_as_nil` - use null for all unrecognizable types (default is false)
- `cfg.encode_load_metatables` - load metatables (default is false)

For example, the following code will interpret `0/0` (which is "not a number") and `1/0` (which is "infinity") as special values rather than nulls or errors:

```
json = require('json')
json.cfg{encode_invalid_numbers = true}
x = 0/0
y = 1/0
json.encode({1, x, y, 2})
```

The result of the `json.encode` request will look like this:

```
tarantool> json.encode({1, x, y, 2})
---
- '[1,nan,inf,2]
...
```

The same configuration settings exist for `json`, for `MsgPack`, and for `YAML`. >>>>>>> Fix every NOTE to be highlighted on site + JSON cfg rewritten

4.14 Module *log*

The Tarantool server puts all diagnostic messages in a log file specified by the `logger` configuration parameter. Diagnostic messages may be either system-generated by the server's internal code, or user-generated with the `log.log_level_function_name` function.

```
log.error(message)
log.warn(message)
log.info(message)
log.debug(message)
```

Output a user-generated message to the *log file*, given `log_level_function_name = error` or `warn` or `info` or `debug`.

Параметры

- `message` (*string*) - The actual output will be a line containing the current timestamp, a module name, 'E' or 'W' or 'I' or 'D' or 'R' depending on `log_level_function_name`, and `message`. Output will not occur if `log_level_function_name` is for a type greater than `log_level`. Messages may contain C-style format specifiers `%d` or `%s`, so `log.error('...%d...%s', x, y)` will work if `x` is a number and `y` is a string.

Return nil

```
log.logger_pid()
log.rotate()
```

4.14.1 Example

```
$ ~/tarantool/src/tarantool
tarantool> box.cfg{log_level=3, logger='tarantool.txt'}
tarantool> log = require('log')
tarantool> log.error('Error')
tarantool> log.info('Info %s', box.info.version)
tarantool> os.exit()
$ less tarantool.txt
```

```
2...0 [5257] main/101/interactive C> version 1.7.0-355-ga4f762d
2...1 [5257] main/101/interactive C> log level 3
2...1 [5261] main/101/spawner C> initialized
2...0 [5257] main/101/interactive [C]:-1 E> Error
```

The 'Error' line is visible in tarantool.txt preceded by the letter E.

The 'Info' line is not present because the log_level is 3.

4.15 Модуль *msgpack*

The `msgpack` module takes strings in `MsgPack` format and decodes them, or takes a series of non-`MsgPack` values and encodes them.

`msgpack.encode(lua_value)`

Convert a Lua object to a `MsgPack` string.

Параметры

- `lua_value` – either a scalar value or a Lua table value.

Return the original value reformatted as a `MsgPack` string.

Rtype string

`msgpack.decode(string)`

Convert a `MsgPack` string to a Lua object.

Параметры

- `string` – a string formatted as `MsgPack`.

Return

- the original contents formatted as a Lua table;
- the number of bytes that were decoded.

Rtype table

`msgpack.NULL`

A value comparable to Lua "nil" which may be useful as a placeholder in a tuple.

4.15.1 Example

```

tarantool> msgpack = require('msgpack')
---
...
tarantool> y = msgpack.encode({'a',1,'b',2})
---
...
tarantool> z = msgpack.decode(y)
---
...
tarantool> z[1], z[2], z[3], z[4]
---
- a
- 1
- b
- 2
...
tarantool> box.space.testers:insert{20, msgpack.NULL, 20}
---
- [20, null, 20]
...

```

The MsgPack output structure can be specified with `__serialize`:

- `__serialize = "seq"` or "sequence" for an array
- `__serialize = "map"` or "mapping" for a map

Serializing 'A' and 'B' with different `__serialize` values causes different results. To show this, here is a routine which encodes `{'A','B'}` both as an array and as a map, then displays each result in hexadecimal.

```

function hexdump(bytes)
  local result = ''
  for i = 1, #bytes do
    result = result .. string.format("%x", string.byte(bytes, i)) .. ' '
  end
  return result
end

msgpack = require('msgpack')
m1 = msgpack.encode(setmetatable({'A', 'B'}, {
  __serialize = "seq"
}))
m2 = msgpack.encode(setmetatable({'A', 'B'}, {
  __serialize = "map"
}))
print('array encoding: ', hexdump(m1))
print('map encoding: ', hexdump(m2))

```

Result:

```

array encoding: 92 a1 41 a1 42
map encoding:   82 01 a1 41 02 a1 42

```

The MsgPack [Specification](#) page explains that the first encoding means:

```
fixarray(2), fixstr(1), "A", fixstr(1), "B"
```

and the second encoding means:


```
fixmap(2), key(1), fixstr(1), "A", key(2), fixstr(2), "B".
```

Here are examples for all the common types, with the Lua-table representation on the left, with the MsgPack format name and encoding on the right.

Common Types and MsgPack Encodings

<code>{}</code>	'fixmap' if metatable is 'map' = 80 otherwise 'fixarray' = 90
<code>'a'</code>	'fixstr' = a1 61
<code>false</code>	'false' = c2
<code>true</code>	'true' = c3
<code>127</code>	'positive fixint' = 7f
<code>65535</code>	'uint 16' = cd ff ff
<code>4294967295</code>	'uint 32' = ce ff ff ff ff
<code>nil</code>	'nil' = c0
<code>msgpack.NULL</code>	same as nil
<code>[0] = 5</code>	'fixmap(1)' + 'positive fixint' (for the key) + 'positive fixint' (for the value) = 81 00 05
<code>[0] = nil</code>	'fixmap(0)' = 80 – nil is not stored when it is a missing map value
<code>1.5</code>	'float 64' = cb 3f f8 00 00 00 00 00 00

Also, some MsgPack configuration settings for encoding can be changed, in the same way that they can be changed for [JSON](#).

4.16 Модуль *net.box*

The `net.box` module contains connectors to remote database systems. One variant, to be discussed later, is for connecting to MySQL or MariaDB or PostgreSQL — that variant is the subject of the [SQL DBMS modules](#) appendix. In this section the subject is the built-in variant, `net.box`. This is for connecting to tarantool servers via a network.

Call `require('net.box')` to get a `net.box` object, which will be called `net_box` for examples in this section. Call `net_box.new()` to connect and get a connection object, which will be called `conn` for examples in this section. Call the other `net.box()` routines, passing `conn:`, to execute requests on the remote box. Call `conn:close` to disconnect.

All `net.box` methods are fiber-safe, that is, it is safe to share and use the same connection object across multiple concurrent fibers. In fact, it's perhaps the best programming practice with Tarantool. When multiple fibers use the same connection, all requests are pipelined through the same network socket, but each fiber gets back a correct response. Reducing the number of active sockets lowers the overhead of system calls and increases the overall server performance. There are, however, cases when a single connection is not enough — for example when it's necessary to prioritize requests or to use different authentication ids.

```
net_box.new(URI[, {option[s]}])
```

Create a new connection. The connection is established on demand, at the time of the first request. It is re-established automatically after a disconnect. The returned `conn` object supports methods for making remote requests, such as `select`, `update` or `delete`.

For the local tarantool server there is a pre-created always-established connection object named `net_box.self`. Its purpose is to make polymorphic use of the `net_box` API easier. Therefore `conn = net_box.new('localhost:3301')` can be replaced by `conn = net_box.self`. However, there is an important difference between the embedded connection and a remote one. With the embedded connection, requests which do not modify data do not yield. When using a remote connection, due to

the implicit rules any request can yield, and database state may have changed by the time it regains control.

Параметры

- `URI` (*string*) – the *URI* of the target for the connection
- `options` – a possible option is `wait_connect`

Return conn object

Rtype userdata

Example:

```
conn = net_box.new('localhost:3301')
conn = net_box.new('127.0.0.1:3306', {wait_connect = false})
```

object conn

`conn:ping()`

Execute a PING command.

Return true on success, false on error

Rtype boolean

Example:

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

`conn:wait_connected([timeout])`

Wait for connection to be active or closed.

Параметры

- `timeout` (*number*) –

Return true when connected, false on failure.

Rtype boolean

Example:

```
net_box.self:wait_connected()
```

`conn:is_connected()`

Show whether connection is active or closed.

Return true if connected, false on failure.

Rtype boolean

Example:

```
net_box.self:is_connected()
```

`conn:close()`

Close a connection.

Connection objects are garbage collected just like any other objects in Lua, so an explicit destruction is not mandatory. However, since `close()` is a system call, it is good programming practice to close a connection explicitly when it is no longer needed, to avoid lengthy stalls of the garbage collector.

Example:

```
conn:close()
```

`conn.space.<space-name>:select{field-value, ...}`
`conn.space.space-name:select{...}` is the remote-call equivalent of the local call `box.space.space-name:select{...}`.

Примечание: due to *the implicit yield rules* a local `box.space.space-name:select{...}` does not yield, but a remote `conn.space.space-name:select{...}` call does yield, so global variables or database tuples data may change when a remote `conn.space.space-name:select{...}` occurs.

`conn.space.<space-name>:get{field-value, ...}`
`conn.space.space-name:get(...)` is the remote-call equivalent of the local call `box.space.space-name:get(...)`.

`conn.space.<space-name>:insert{field-value, ...}`
`conn.space.space-name:insert(...)` is the remote-call equivalent of the local call `box.space.space-name:insert(...)`.

`conn.space.<space-name>:replace{field-value, ...}`
`conn.space.space-name:replace(...)` is the remote-call equivalent of the local call `box.space.space-name:replace(...)`.

`conn.space.<space-name>:update{field-value, ...}`
`conn.space.space-name:update(...)` is the remote-call equivalent of the local call `box.space.space-name:update(...)`.

`conn.space.<space-name>:upsert{field-value, ...}`
`conn.space.space-name:upsert(...)` is the remote-call equivalent of the local call `box.space.space-name:upsert(...)`.

`conn.space.<space-name>:delete{field-value, ...}`
`conn.space.space-name:delete(...)` is the remote-call equivalent of the local call `box.space.space-name:delete(...)`.

`conn:call(function-name[, arguments])`
`conn:call('func', '1', '2', '3')` is the remote-call equivalent of `func('1', '2', '3')`. That is, `conn:call` is a remote stored-procedure call.

Example:

```
conn:call('function5')
```

`conn:eval(Lua-string)`
`conn:eval(Lua-string)` evaluates and executes the expression in `Lua-string`, which may be any statement or series of statements. An *execute privilege* is required; if the user does not have it, an administrator may grant it with `box.schema.user.grant(username, 'execute', 'universe')`.

Example:

```
conn:eval('return 5+5')
```

`conn:timeout(timeout)`
`timeout(...)` is a wrapper which sets a timeout for the request that follows it.

Example:

```
conn:timeout(0.5).space.testers:update({1}, {{'=', 2, 15}})
```

All remote calls support execution timeouts. Using a wrapper object makes the remote connection API compatible with the local one, removing the need for a separate `timeout` argument, which the local version would ignore. Once a request is sent, it cannot be revoked from the remote server even if a timeout expires: the timeout expiration only aborts the wait for the remote server response, not the request itself.

4.16.1 Example showing use of most of the `net.box` methods

This example will work with the sandbox configuration described in the preface. That is, there is a space named `tester` with a numeric primary key. Assume that the database is nearly empty. Assume that the tarantool server is running on localhost `127.0.0.1:3301`.

```
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.new('127.0.0.1:3301')
  > conn.space.testers:delete{800}
  > table.insert(ta, 'conn delete done on tester.')
  > conn.space.testers:insert{800, 'data'}
  > table.insert(ta, 'conn insert done on tester, index 0')
  > table.insert(ta, ' primary key value = 800.')
  > wtuple = conn.space.testers:select{800}
  > table.insert(ta, 'conn select done on tester, index 0')
  > table.insert(ta, ' number of fields = ' .. #wtuple)
  > conn.space.testers:delete{800}
  > table.insert(ta, 'conn delete done on tester')
  > conn.space.testers:replace{800, 'New data', 'Extra data'}
  > table.insert(ta, 'conn:replace done on tester')
  > conn:timeout(0.5).space.testers: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
...

```

4.17 Функция *box.once*

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

Execute a function, provided it has not been executed before. A passed value is checked to see whether the function has already been executed. If it has been executed before, nothing happens. If it has not been executed before, the function is invoked. For an explanation why `box.once` is useful, see the section [Preventing Duplicate Actions](#).

Параметры

- `key` (*string*) – a value that will be checked
- `function` (*function*) – a function
- ... – arguments, that must be passed to function

4.18 Module *os*

The `os` module contains the functions [execute\(\)](#), [rename\(\)](#), [getenv\(\)](#), [remove\(\)](#), [date\(\)](#), [exit\(\)](#), [time\(\)](#), [clock\(\)](#), [tmpname\(\)](#). Most of these functions are described in the Lua manual Chapter 22 [The Operating System Library](#).

`os.execute(shell-command)`

Execute by passing to the shell.

Параметры

- `shell-command` (*string*) – what to execute.

Example:

```

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)`
Rename a file or directory.

Параметры

- `old-name` (*string*) – name of existing file or directory,
- `new-name` (*string*) – changed name of file or directory.

Example:

```
tarantool> os.rename('local', 'foreign')
---
- null
- 'local: No such file or directory'
- 2
...
```

`os.getenv(variable-name)`
Get environment variable.

Parameters: (string) `variable-name` = environment variable name.

Example:

```
tarantool> os.getenv('PATH')
---
- /usr/local/sbin:/usr/local/bin:/usr/sbin
...
```

`os.remove(name)`
Remove file or directory.

Parameters: (string) `name` = name of file or directory which will be removed.

Example:

```
tarantool> os.remove('file')
---
- true
...
```

`os.date(format-string [, time-since-epoch])`
Return a formatted date.

Parameters: (string) `format-string` = instructions; (string) `time-since-epoch` = number of seconds since 1970-01-01. If `time-since-epoch` is omitted, it is assumed to be the current time.

Example:

```
tarantool> os.date("%A %B %d")
---
- Sunday April 24
...
```

`os.exit()`
Exit the program. If this is done on the server, then the server stops.

Example:

```
tarantool> os.exit()
user@user-shell:~/tarantool_sandbox$
```

`os.time()`

Return the number of seconds since the epoch.

Example:

```
tarantool> os.time()
---
- 1461516945
...
```

`os.clock()`

Return the number of CPU seconds since the program start.

Example:

```
tarantool> os.clock()
---
- 0.05
...
```

`os.tmpname()`

Return a name for a temporary file.

Example:

```
tarantool> os.tmpname()
---
- /tmp/lua_7SW1m2
...
```

4.19 Модуль *pickle*

`pickle.pack(format, argument[, argument ...])`

To use Tarantool binary protocol primitives from Lua, it's necessary to convert Lua variables to binary format. The `pickle.pack()` helper function is prototyped after Perl `'pack'`.

Format specifiers

b, B	converts Lua variable to a 1-byte integer, and stores the integer in the resulting string
s, S	converts Lua variable to a 2-byte integer, and stores the integer in the resulting string, low byte first
i, I	converts Lua variable to a 4-byte integer, and stores the integer in the resulting string, low byte first
l, L	converts Lua variable to an 8-byte integer, and stores the integer in the resulting string, low byte first
n	converts Lua variable to a 2-byte integer, and stores the integer in the resulting string, big endian,
N	converts Lua variable to a 4-byte integer, and stores the integer in the resulting string, big
q, Q	converts Lua variable to an 8-byte integer, and stores the integer in the resulting string, big endian,
f	converts Lua variable to a 4-byte float, and stores the float in the resulting string
d	converts Lua variable to a 8-byte double, and stores the double in the resulting string
a, A	converts Lua variable to a sequence of bytes, and stores the sequence in the resulting string

Параметры

- `format` (*string*) – string containing format specifiers
- `argument(s)` (*scalar-value*) – scalar values to be formatted

Return a binary string containing all arguments, packed according to the format specifiers.

Rtype string

Possible errors: unknown format specifier.

Example:

```
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)`

Counterpart to `pickle.pack()`. Warning: if format specifier 'A' is used, it must be the last item.

Параметры

- `format (string)` –
- `binary-string (string)` –

Return A list of strings or numbers.

Rtype table

Example:

```

tarantool> pickle = require('pickle')
---
...
tarantool> tuple = box.space.testster: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('6666666666666666'), 'string'))
---
...

```

4.20 Модуль *socket*

The `socket` module allows exchanging data via BSD sockets with a local or remote host in connection-oriented (TCP) or datagram-oriented (UDP) mode. Semantics of the calls in the `socket` API closely follow semantics of the corresponding POSIX calls. Function names and signatures are mostly compatible with [luasocket](#).

The functions for setting up and connecting are `socket`, `sysconnect`, `tcp_connect`. The functions for sending data are `send`, `sendto`, `write`, `syswrite`. The functions for receiving data are `recv`, `recvfrom`, `read`. The functions for waiting before sending/receiving data are `wait`, `readable`, `writable`. The functions for setting

flags are `nonblock`, `setsockopt`. The functions for stopping and disconnecting are `shutdown`, `close`. The functions for error checking are `errno`, `error`.

Socket functions

Purposes	Names
setup	<code>socket()</code>
	<code>socket.tcp_connect()</code>
	<code>socket.tcp_server()</code>
	<code>socket_object:sysconnect()</code>
sending	<code>socket_object:send()</code>
	<code>socket_object:sendto()</code>
	<code>socket_object:write()</code>
receiving	<code>socket_object:syswrite()</code>
	<code>socket_object:recv()</code>
	<code>socket_object:recvfrom()</code>
flag setting	<code>socket_object:read()</code>
	<code>socket_object:nonblock()</code>
	<code>socket_object:setsockopt()</code>
client/server	<code>socket_object:linger()</code>
	<code>socket_object:listen()</code>
	<code>socket_object:accept()</code>
teardown	<code>socket_object:shutdown()</code>
	<code>socket_object:close()</code>
error checking	<code>socket_object:error()</code>
	<code>socket_object:errno()</code>
information	<code>socket.getaddrinfo()</code>
	<code>socket_object:getsockopt()</code>
	<code>socket_object:peer()</code>
state checking	<code>socket_object:name()</code>
	<code>socket_object:readable()</code>
	<code>socket_object:writable()</code>
	<code>socket_object:wait()</code>

Typically a socket session will begin with the setup functions, will set one or more flags, will have a loop with sending and receiving functions, will end with the teardown functions – as an example at the end of this section will show. Throughout, there may be error-checking and waiting functions for synchronization. To prevent a fiber containing socket functions from "blocking" other fibers, the *implicit yield rules* will cause a yield so that other processes may take over, as is the norm for cooperative multitasking.

For all examples in this section the socket name will be `sock` and the function invocations will look like `sock:function_name(...)`.

```
socket.__call(domain, type, protocol)
```

Create a new TCP or UDP socket. The argument values are the same as in the [Linux socket\(2\) man page](#).

Return an unconnected socket, or nil.

Rtype userdata

Example:

```
socket('AF_INET', 'SOCK_STREAM', 'tcp')
```

```
socket.tcp_connect(host[, port])
```

Connect a socket to a remote host.

Параметры

- *host* (*string*) – URL or IP address
- *port* (*number*) – port number

Return a connected socket, if no error.

Rtype userdata

Example:

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

```
socket.getaddrinfo(host, type[, {option-list}]
```

The `socket.getaddrinfo()` function is useful for finding information about a remote site so that the correct arguments for `sock:sysconnect()` can be passed.

Return A table containing these fields: "host "family "type "protocol "port".

Rtype table

Example:

`socket.getaddrinfo('tarantool.org', 'http')` will return variable information such as

```
---
- - 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)
```

The `socket.tcp_server()` function makes Tarantool act as a server that can accept connections. Usually the same objective is accomplished with `box.cfg{listen=...}`.

Example:

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

object `socket_object`

```
socket_object:sysconnect(host, port)
```

Connect an existing socket to a remote host. The argument values are the same as in `tcp_connect()`. The host must be an IP address.

Parameters:

- **Either:**
 - *host* - a string representation of an IPv4 address or an IPv6 address;
 - *port* - a number.

- **Or:**
 - host - a string containing "unix/";
 - port - a string containing a path to a unix socket.
- **Or:**
 - host - a number, 0 (zero), meaning "all local interfaces";
 - port - a number. If a port number is 0 (zero), the socket will be bound to a random local port.

Return the socket object value may change if `sysconnect()` succeeds.

Rtype boolean

Example:

```
socket = require('socket')
sock = socket('AF_INET', 'SOCK_STREAM', 'tcp')
sock:sysconnect(0, 3301)
```

`socket_object:send(data)`

`socket_object:write(data)`

Send data over a connected socket.

Параметры

- *data* (*string*) –

Return the number of bytes sent.

Rtype number

Possible errors: nil on error.

`socket_object:syswrite(size)`

Write as much as possible data to the socket buffer if non-blocking. Rarely used. For details see [this description](#).

`socket_object:recv(size)`

Read *size* bytes from a connected socket. An internal read-ahead buffer is used to reduce the cost of this call.

Параметры

- *size* (*integer*) –

Return a string of the requested length on success.

Rtype string

Possible errors: On error, returns an empty string, followed by status, errno, errstr. In case the writing side has closed its end, returns the remainder read from the socket (possibly an empty string), followed by "eof" status.

`socket_object:read(limit[, timeout])`

`socket_object:read(delimiter[, timeout])`

`socket_object:read({limit=limit}[, timeout])`

`socket_object:read({delimiter=delimiter}[, timeout])`

`socket_object:read({limit=limit, delimiter=delimiter}[, 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.

Параметры

- `limit` (*integer*) – maximum number of bytes to read for example 50 means "stop after 50 bytes"
- `delimiter` (*string*) – separator for example '?' means "stop after a question mark"
- `timeout` (*number*) – maximum number of seconds to wait for example 50 means "stop after 50 seconds".

Return an empty string if there is nothing more to read, or a nil value if error, or a string up to `limit` bytes long, which may include the bytes that matched the `delimiter` expression.

Rtype string

`socket_object:sysread(size)`

Return all available data from the socket buffer if non-blocking. Rarely used. For details see [this description](#).

`socket_object:bind(host[, port])`

Bind a socket to the given host/port. A UDP socket after binding can be used to receive data (see [socket_object.recvfrom](#)). A TCP socket can be used to accept new connections, after it has been put in listen mode.

Параметры

- `host` –
- `port` –

Return a socket object on success

Rtype userdata

Possible errors: Returns nil, status, errno, errstr on error.

`socket_object:listen(backlog)`

Start listening for incoming connections.

Параметры

- `backlog` – On Linux the listen backlog backlog may be from `/proc/sys/net/core/somaxconn`, on BSD the backlog may be `SOMAXCONN`.

Return true for success, false for error.

Rtype boolean.

`socket_object:accept()`

Accept a new client connection and create a new connected socket. It is good practice to set the socket's blocking mode explicitly after accepting.

Return new socket if success.

Rtype userdata

Possible errors: nil.

`socket_object:sendto(host, port, data)`

Send a message on a UDP socket to a specified host.

Параметры

- `host` (*string*) –
- `port` (*number*) –
- `data` (*string*) –

Return the number of bytes sent.

Rtype number

Possible errors: on error, returns status, errno, errstr.

`socket_object:recvfrom(limit)`

Receive a message on a UDP socket.

Параметры

- `limit` (*integer*) –

Return message, a table containing "host" "family" and "port" fields.

Rtype string, table

Possible errors: on error, returns status, errno, errstr.

Example:

After `message_content`, `message_sender = recvfrom(1)` the value of `message_content` might be a string containing 'X' and the value of `message_sender` might be a table containing

```
message_sender.host = '18.44.0.1'
message_sender.family = 'AF_INET'
message_sender.port = 43065
```

`socket_object:shutdown(how)`

Shutdown a reading end, a writing end, or both ends of a socket.

Параметры

- `how` – `socket.SHUT_RD`, `socket.SHUT_WR`, or `socket.SHUT_RDWR`.

Return true or false.

Rtype boolean

`socket_object:close()`

Close (destroy) a socket. A closed socket should not be used any more. A socket is closed automatically when its userdata is garbage collected by Lua.

Return true on success, false on error. For example, if sock is already closed, `sock:close()` returns false.

Rtype boolean

`socket_object:error()`

`socket_object:errno()`

Retrieve information about the last error that occurred on a socket, if any. Errors do not cause throwing of exceptions so these functions are usually necessary.

Return result for `sock:errno()`, result for `sock:error()`. If there is no error, then `sock:errno()` will return 0 and `sock:error()`.

Rtype number, string

`socket_object:setsockopt(level, name, value)`

Set socket flags. The argument values are the same as in the [Linux getsockopt\(2\) man page](#). The ones that Tarantool accepts are:

- SO_ACCEPTCONN
- SO_BINDTODEVICE
- SO_BROADCAST
- SO_DEBUG
- SO_DOMAIN
- SO_ERROR
- SO_DONTROUTE
- SO_KEEPALIVE
- SO_MARK
- SO_OOBINLINE
- SO_PASSCRED
- SO_PEERCREC
- 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

Setting SO_LINGER is done with `sock:linger(active)`.

`socket_object:getsockopt(level, name)`

Get socket flags. For a list of possible flags see `sock:setsockopt()`.

`socket_object:linger([active])`

Set or clear the SO_LINGER flag. For a description of the flag, see the [Linux man page](#).

Параметры

- `active` (*boolean*) –

Return new active and timeout values.

`socket_object:nonblock([flag])`

- `sock:nonblock()` returns the current flag value.
- `sock:nonblock(false)` sets the flag to false and returns false.
- `sock:nonblock(true)` sets the flag to true and returns true.

This function may be useful before invoking a function which might otherwise block indefinitely.

`socket_object:readable([timeout])`

Wait until something is readable, or until a timeout value expires.

Return true if the socket is now readable, false if timeout expired;

`socket_object:writable([timeout])`

Wait until something is writable, or until a timeout value expires.

Return true if the socket is now writable, false if timeout expired;

`socket_object:wait([timeout])`

Wait until something is either readable or writable, or until a timeout value expires.

Return 'R' if the socket is now readable, 'W' if the socket is now writable, 'RW' if the socket is now both readable and writable, "" (empty string) if timeout expired;

`socket_object:name()`

The `sock:name()` function is used to get information about the near side of the connection. If a socket was bound to `xyz.com:45`, then `sock:name` will return information about `[host:xyz.com, port:45]`. The equivalent POSIX function is `getsockname()`.

Return A table containing these fields: "host "family "type "protocol "port".

Rtype table

`socket_object:peer()`

The `sock:peer()` function is used to get information about the far side of a connection. If a TCP connection has been made to a distant host `tarantool.org:80`, `sock:peer()` will return information about `[host:tarantool.org, port:80]`. The equivalent POSIX function is `getpeername()`.

Return A table containing these fields: "host "family "type "protocol "port".

Rtype table

4.20.1 Example

Use of a TCP socket over the Internet

In this example a connection is made over the internet between the Tarantool server and `tarantool.org`, then an HTTP "head" message is sent, and a response is received: `"HTTP/1.1 200 OK"`. This is not a useful way to communicate with this particular site, but shows that the system works.

```
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/1.0rnHost: tarantool.orgnrn")
----
- true
...
tarantool> sock:read(17)
----
- "HTTP/1.1 200 OKrn"
...
tarantool> sock:close()
----
- true
...

```

Use of a UDP socket on localhost

Here is an example with datagrams. Set up two connections on 127.0.0.1 (localhost): `sock_1` and `sock_2`. Using `sock_2`, send a message to `sock_1`. Using `sock_1`, receive a message. Display the received message. Close both connections. This is not a useful way for a computer to communicate with itself, but shows that the system works.

```

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')
----
- true
...
tarantool> message = sock_1:recvfrom()
----
...
tarantool> message
----
- X
...
tarantool> sock_1:close()
----
- true
...
tarantool> sock_2:close()
----

```

```
- true
...
```

Use `tcp_server` to accept file contents sent with `socat`

Here is an example of the `tcp_server` function, reading strings from the client and printing them. On the client side, the Linux `socat` utility will be used to ship a whole file for the `tcp_server` function to read.

Start two shells. The first shell will be the server. The second shell will be the client.

On the first shell, start Tarantool and say:

```
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)
```

The above code means: use `tcp_server()` to wait for a connection from any host on port 3302. When it happens, enter a loop that reads on the socket and prints what it reads. The "delimiter" for the read function is `"\n"` so each `read()` will read a string as far as the next line feed, including the line feed.

On the second shell, create a file that contains a few lines. The contents don't matter. Suppose the first line contains A, the second line contains B, the third line contains C. Call this file "tmp.txt".

On the second shell, use the `socat` utility to ship the tmp.txt file to the server's host and port:

```
$ socat TCP:localhost:3302 ./tmp.txt
```

Now watch what happens on the first shell. The strings "A "B "C" are printed.

4.21 Модуль *strict*

The `strict` module has functions for turning "strict mode" on or off. When strict mode is on, an attempt to use an undeclared global variable will cause an error. A global variable is considered "undeclared" if it has never had a value assigned to it. Often this is an indication of a programming error.

By default strict mode is off, unless tarantool was built with the `-DCMAKE_BUILD_TYPE=Debug` option – see the description of build options in section [building-from-source](#).

Example:

```
tarantool> strict = require('strict')
---
...
tarantool> strict.on()
---
...
tarantool> a = b -- strict mode is on so this will cause an error
```

```

---
- error: ... variable 'b' is not declared'
...
tarantool> strict.off()
---
...
tarantool> a = b -- strict mode is off so this will not cause an error
---
...

```

4.22 Module *tap*

The `tap` module streamlines the testing of other modules. It allows writing of tests in the [TAP protocol](#). The results from the tests can be parsed by standard TAP-analyzers so they can be passed to utilities such as [prove](#). Thus one can run tests and then use the results for statistics, decision-making, and so on.

`tap.test(test-name)`

Initialize.

The result of `tap.test` is an object, which will be called `taptest` in the rest of this discussion, which is necessary for `taptest:plan()` and all the other methods.

Параметры

- `test-name` (*string*) – an arbitrary name to give for the test outputs.

Return `taptest`

Rtype `userdata`

```

tap = require('tap')
taptest = tap.test('test-name')

```

object `taptest`

`taptest:plan(count)`

Indicate how many tests will be performed.

Параметры

- `count` (*number*) –

Return `nil`

`taptest:check()`

Checks the number of tests performed. This check should only be done after all planned tests are complete, so ordinarily `taptest:check()` will only appear at the end of a script.

Will display `# bad plan: ...` if the number of completed tests is not equal to the number of tests specified by `taptest:plan(...)`.

Return `nil`

`taptest:diag(message)`

Display a diagnostic message.

Параметры

- `message` (*string*) – the message to be displayed.

Return nil

`taptest:ok(condition, test-name)`

This is a basic function which is used by other functions. Depending on the value of `condition`, print 'ok' or 'not ok' along with debugging information. Displays the message.

Параметры

- `condition` (*boolean*) – an expression which is true or false
- `test-name` (*string*) – name of test

Return true or false.

Rtype boolean

Example:

```
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')` is equivalent to `taptest:ok(false, 'x')`. Displays the message.

Параметры

- `test-name` (*string*) – name of test

Return true or false.

Rtype boolean

`taptest:skip(message)`

`taptest:skip('x')` is equivalent to `taptest:ok(true, 'x' .. '# skip')`. Displays the message.

Параметры

- `test-name` (*string*) – name of test

Return nil

Example:

```
tarantool> taptest:skip('message')
ok - message # skip
---
- true
...
```

`taptest:is(got, expected, test-name)`

Check whether the first argument equals the second argument. Displays extensive message if the result is false.

Параметры

- `got` (*number*) – actual result
- `expected` (*number*) – expected result
- `test-name` (*string*) – name of test

Return true or false.

Rtype boolean

`taptest:isnt(got, expected, test-name)`

This is the negation of `taptest:is(...)`.

Параметры

- `got` (*number*) – actual result
- `expected` (*number*) – expected result
- `test-name` (*string*) – name of test

Return true or false.

Rtype boolean

`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)`

Test whether a value has a particular type. Displays a long message if the value is not of the specified type.

Параметры

- `value` (*lua-value*) –
- `test-name` (*string*) – name of test

Return true or false.

Rtype boolean

`taptest:is_deeply(got, expected, test-name)`

Recursive version of `taptest:is(...)`, which can be used to compare tables as well as scalar values.

Return true or false.

Rtype boolean

Параметры

- `got` (*lua-value*) – actual result
- `expected` (*lua-value*) – expected result
- `test-name` (*string*) – name of test

4.22.1 Example

To run this example: put the script in a file named `./tap.lua`, then make `tap.lua` executable by saying `chmod a+x ./tap.lua`, then execute using Tarantool as a script processor by saying `./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()
```

The output from the above script will look approximately like this:

```
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
```

4.23 Модуль *tarantool*

By saying `require('tarantool')`, one can answer some questions about how the tarantool server was built, such as "what flags were used or "what was the version of the compiler".

Additionally one can see the uptime and the server version and the process id. Those information items can also be accessed with *box.info* but use of the tarantool module is recommended.

Example:

```
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
...
```

4.24 Module *uuid*

A "UUID" is a [Universally unique identifier](#). If an application requires that a value be unique only within a single computer or on a single database, then a simple counter is better than a UUID, because getting a UUID is time-consuming (it requires a [syscall](#)). For clusters of computers, or widely distributed applications, UUIDs are better.

The functions that can return a UUID are:

- [*uuid\(\)*](#)
- [*uuid.bin\(\)*](#)
- [*uuid.str\(\)*](#)

The functions that can convert between different types of UUID are:

- [*uuid_object.bin\(\)*](#)
- [*uuid_object.str\(\)*](#)
- [*uuid.fromstr\(\)*](#)
- [*uuid.frombin\(\)*](#)

The function that can determine whether a UUID is an all-zero value is:

- [*uuid_object.isnil\(\)*](#)

`uuid.nil`

A nil object

`uuid.__call()`

Return a UUID

Rtype cdata

`uuid.bin()`

Return a UUID

Rtype 16-byte string

`uuid.str()`

Return a UUID

Rtype 36-byte binary string

`uuid.fromstr(uuid_str)`

Параметры

- `uuid_str` – UUID in 36-byte hexadecimal string

Return converted UUID

Rtype cdata

```
uuid.frombin(uuid_bin)
```

Параметры

- *uuid_str* – UUID in 16-byte binary string

Return converted UUID

Rtype cdata

```
object uuid_object
```

```
uuid_object:bin([byte-order])
byte-order can be one of next flags:
```

- 'l' - little-endian,
- 'b' - big-endian,
- 'h' - endianness depends on host (default),
- 'n' - endianness depends on network

Параметры

- *byte-order* (*string*) – one of 'l', 'b', 'h' or 'n'.

Return UUID converted from cdata input value.

Rtype 16-byte binary string

```
uuid_object:str()
```

Return UUID converted from cdata input value.

Rtype 36-byte hexadecimal string

```
uuid_object:isnil()
```

The all-zero UUID value can be expressed as `uuid.NULL`, or as `uuid.fromstr('00000000-0000-0000-0000-000000000000')`. The comparison with an all-zero value can also be expressed as `uuid_with_type_cdata == uuid.NULL`.

Return true if the value is all zero, otherwise false.

Rtype bool

4.24.1 Example

```
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
...

```

4.25 Module *yaml*

The `yaml` module takes strings in [YAML](#) format and decodes them, or takes a series of non-YAML values and encodes them.

`yaml.encode(lua_value)`

Convert a Lua object to a YAML string.

Параметры

- `lua_value` – either a scalar value or a Lua table value.

Return the original value reformatted as a YAML string.

Rtype string

`yaml.decode(string)`

Convert a YAML string to a Lua object.

Параметры

- `string` – a string formatted as YAML.

Return the original contents formatted as a Lua table.

Rtype table

`yaml.NULL`

A value comparable to Lua "nil" which may be useful as a placeholder in a tuple.

4.25.1 Example

```

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
---
...

```

The [YAML collection style](#) can be specified with `__serialize`:

- `__serialize="sequence"` for a Block Sequence array,
- `__serialize="seq"` for a Flow Sequence array,
- `__serialize="mapping"` for a Block Mapping map,
- `__serialize="map"` for a Flow Mapping map.

Serializing 'A' and 'B' with different `__serialize` values causes different results:

```

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
  ...
...

```

Also, some YAML configuration settings for encoding can be changed, in the same way that they can be changed for [JSON](#).

4.26 Разное

`tonumber64(value)`

Convert a string or a Lua number to a 64-bit integer. The result can be used in arithmetic, and the arithmetic will be 64-bit integer arithmetic rather than floating-point arithmetic. (Operations on an unconverted Lua number use floating-point arithmetic.) The `tonumber64()` function is added by Tarantool; the name is global.

Example:

```
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 ...])`

Parse and execute an arbitrary chunk of Lua code. This function is mainly useful to define and run Lua code without having to introduce changes to the global Lua environment.

Параметры

- `lua-chunk-string` (*string*) – Lua code
- `lua-chunk-string-argument` (*lua-value*) – zero or more scalar values which will be appended to, or substitute for, items in the Lua chunk.

Return whatever is returned by the Lua code chunk.

Possible errors: If there is a compilation error, it is raised as a Lua error.

Example:

```
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.testers:select{key}
>   if t ~= nil then
>     return t[1]
>   else
>     return nil
>   end
> end
> return f(...)]], 1)
---
- null
...
```

A module is an optional library which enhances Tarantool functionality. For examples of creating one's own module with Lua or C, see [this link](#).

5.1 Модули SQL СУБД

The discussion here in the reference is about incorporating and using two modules that have already been created: the "SQL DBMS rocks" for MySQL and PostgreSQL.

To call another DBMS from Tarantool, the essential requirements are: another DBMS, and Tarantool. The module which connects Tarantool to another DBMS may be called a "connector". Within the module there is a shared library which may be called a "driver".

Tarantool supplies DBMS connector modules with the module manager for Lua, LuaRocks. So the connector modules may be called "rocks".

The Tarantool rocks allow for connecting to an SQL server and executing SQL statements the same way that a MySQL or PostgreSQL client does. The SQL statements are visible as Lua methods. Thus Tarantool can serve as a "MySQL Lua Connector" or "PostgreSQL Lua Connector" which would be useful even if that was all Tarantool could do. But of course Tarantool is also a DBMS, so the module also is useful for any operations, such as database copying and accelerating, which work best when the application can work on both SQL and Tarantool inside the same Lua routine. The methods for connect/select/insert/etc. are similar to the ones in the *net.box* module.

From a user's point of view the MySQL and PostgreSQL rocks are very similar, so the following sections – "MySQL Example" and "PostgreSQL Example" contain some redundancy.

5.1.1 MySQL Example

This example assumes that MySQL 5.5 or MySQL 5.6 or MySQL 5.7 has been installed. Recent MariaDB versions will also work, the MariaDB C connector is used. The package that matters most is the MySQL client developer package, typically named something like libmysqlclient-dev. The file that matters most from

this package is `libmysqlclient.so` or a similar name. One can use `find` or `whereis` to see what directories these files are installed in.

It will be necessary to install Tarantool's MySQL driver shared library, load it, and use it to connect to a MySQL server. After that, one can pass any MySQL statement to the server and receive results, including multiple result sets.

Installation

Check the instructions for [Downloading and installing a binary package](#) that apply for the environment where tarantool was installed. In addition to installing `tarantool`, install `tarantool-dev`. For example, on Ubuntu, add the line

```
sudo apt-get install tarantool-dev
```

Now, for the MySQL driver shared library, there are two ways to install:

With LuaRocks

Begin by installing luarocks and making sure that tarantool is among the upstream servers, as in the instructions on rocks.tarantool.org, the Tarantool luarocks page. Now execute this:

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

For example:

```
luarocks install mysql MYSQL_LIBDIR=/usr/local/mysql/lib
```

See also [Modules, LuaRocks, and requiring modules](#).

With GitHub

Go the site github.com/tarantool/mysql. Follow the instructions there, saying:

```
git clone https://github.com/tarantool/mysql.git
cd mysql && cmake . -DCMAKE_BUILD_TYPE=RelWithDebInfo
make
make install
```

At this point it is a good idea to check that the installation produced a file named `driver.so`, and to check that this file is on a directory that is searched by the `require` request.

Connecting

Begin by making a `require` request for the mysql driver. We will assume that the name is `mysql` in further examples.

```
mysql = require('mysql')
```

Now, say:

```
connection_name = mysql.connect(connection_options)
```

The connection-options parameter is a table. Possible options are:

- `host` = *host-name* - string, default value = 'localhost'
- `port` = *port-number* - number, default value = 3306
- `user` = *user-name* - string, default value is operating-system user name
- `password` = *password* - string, default value is blank
- `db` = *database-name* - string, default value is blank
- `raise` = *true/false* - boolean, default value is false

The option names, except for *raise*, are similar to the names that MySQL's `mysql` client uses, for details see the MySQL manual at dev.mysql.com/doc/refman/5.6/en/connecting.html. The *raise* option should be set to true if errors should be raised when encountered. To connect with a Unix socket rather than with TCP, specify `host = 'unix/'` and `port = socket-name`.

Example, using a table literal enclosed in {braces}:

```
conn = mysql.connect({
  host = '127.0.0.1',
  port = 3306,
  user = 'p',
  password = 'p',
  db = 'test',
  raise = true
})
-- OR
conn = mysql.connect({
  host = 'unix/',
  port = '/var/run/mysqld/mysqld.sock'
})
```

Example, creating a function which sets each option in a separate line:

```
tarantool> -- Connection function. Usage: 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()
---
...
```

We will assume that the name is 'conn' in further examples.

How to ping

To ensure that a connection is working, the request is:

```
connection-name:ping()
```

Example:

```
tarantool> conn:ping()
---
- true
...
```

Executing a statement

For all MySQL statements, the request is:

```
connection-name:execute(sql-statement [, parameters])
```

where *sql-statement* is a string, and the optional *parameters* are extra values that can be plugged in to replace any question marks ("?"s) in the SQL statement.

Example:

```
tarantool> conn:execute('select table_name from information_schema.tables')
---
- - table_name: ALL_PLUGINS
- table_name: APPLICABLE_ROLES
- table_name: CHARACTER_SETS
<...>
- 78
...
```

Closing connection

To end a session that began with `mysql.connect`, the request is:

```
connection-name:close()
```

Example:

```
tarantool> conn:close()
---
...
```

For further information, including examples of rarely-used requests, see the README.md file at github.com/tarantool/mysql.

Example

The example was run on an Ubuntu 12.04 ("precise") machine where tarantool had been installed in a /usr subdirectory, and a copy of MySQL had been installed on ~/mysql-5.5. The mysqld server is already running on the local host 127.0.0.1.

```
$ export TMDIR=~/mysql-5.5
$ # Check that the include subdirectory exists by looking
$ # for ../include/mysql.h. (If this fails, there's a chance
$ # that it's in ../include/mysql/mysql.h instead.)
$ [ -f $TMDIR/include/mysql.h ] && echo "OK" || echo "Error"
OK

$ # Check that the library subdirectory exists and has the
$ # necessary .so file.
```



```

$ [ -f $TMDIR/lib/libmysqlclient.so ] && echo "OK" || echo "Error"
OK

$ # Check that the mysql client can connect using some factory
$ # defaults: port = 3306, user = 'root', user password = '',
$ # database = 'test'. These can be changed, provided one uses
$ # the changed values in all places.
$ $TMDIR/bin/mysql --port=3306 -h 127.0.0.1 --user=root \
  --password= --database=test
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 25
Server version: 5.5.35 MySQL Community Server (GPL)
...
Type 'help;' or '\h' for help. Type '\c' to clear ...

$ # Insert a row in database test, and quit.
mysql> CREATE TABLE IF NOT EXISTS test (s1 INT, s2 VARCHAR(50));
Query OK, 0 rows affected (0.13 sec)
mysql> INSERT INTO test.test VALUES (1, 'MySQL row');
Query OK, 1 row affected (0.02 sec)
mysql> QUIT
Bye

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

$ # Set up the Tarantool rock list in ~/.luarocks,
$ # following instructions at rocks.tarantool.org
$ mkdir ~/.luarocks
$ echo "rocks_servers = {[[http://rocks.tarantool.org/]]}" >> \
  ~/.luarocks/config.lua

$ # Ensure that the next "install" will get files from Tarantool
$ # master repository. The resultant display is normal for Ubuntu
$ # 12.04 precise
$ cat /etc/apt/sources.list.d/tarantool.list
deb http://tarantool.org/dist/1.7/ubuntu/ precise main
deb-src http://tarantool.org/dist/1.7/ubuntu/ precise main

$ # Install tarantool-dev. The displayed line should show version = 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) ...
$

$ # Use luarocks to install locally, that is, relative to $HOME
$ luarocks install mysql MYSQL_LIBDIR=/usr/local/mysql/lib --local
Installing http://rocks.tarantool.org/mysql-scm-1.rockspec...
... (more info about building the Tarantool/MySQL driver appears here)
mysql scm-1 is now built and installed in ~/.luarocks/

$ # Ensure driver.so now has been created in a place
$ # tarantool will look at
$ find ~/.luarocks -name "driver.so"
~/.luarocks/lib/luarocks/5.1/mysql/driver.so

$ # Change directory to a directory which can be used for
$ # temporary tests. For this example we assume that the name

```

```

$ # of this directory is /home/pgulutzan/tarantool_sandbox.
$ # (Change "/home/pgulutzan" to whatever is the user's actual
$ # home directory for the machine that's used for this test.)
$ cd /home/pgulutzan/tarantool_sandbox

$ # Start the Tarantool server. Do not use a Lua initialization file.

$ tarantool
tarantool: version 1.7.0-222-g48b98bb
type 'help' for interactive help
tarantool>

```

Configure tarantool and load mysql module. Make sure that tarantool doesn't reply "error" for the call to "require()".

```

tarantool> box.cfg{}
...
tarantool> mysql = require('mysql')
---
...

```

Create a Lua function that will connect to the MySQL server, (using some factory default values for the port and user and password), retrieve one row, and display the row. For explanations of the statement types used here, read the Lua tutorial earlier in the Tarantool user manual.

```

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 '
...

```

Observe the result. It contains "MySQL row". So this is the row that was inserted into the MySQL database. And now it's been selected with the Tarantool client.

5.1.2 PostgreSQL Example

This example assumes that PostgreSQL 8 or PostgreSQL 9 has been installed. More recent versions should also work. The package that matters most is the PostgreSQL developer package, typically named something like libpq-dev. On Ubuntu this can be installed with:

```
sudo apt-get install libpq-dev
```

However, because not all platforms are alike, for this example the assumption is that the user must check that the appropriate PostgreSQL files are present and must explicitly state where they are when building the Tarantool/PostgreSQL driver. One can use `find` or `whereis` to see what directories PostgreSQL files are installed in.

It will be necessary to install Tarantool's PostgreSQL driver shared library, load it, and use it to connect to a PostgreSQL server. After that, one can pass any PostgreSQL statement to the server and receive results.

Installation

Check the instructions for [Downloading and installing a binary package](#) that apply for the environment where tarantool was installed. In addition to installing `tarantool`, install `tarantool-dev`. For example, on Ubuntu, add the line:

```
sudo apt-get install tarantool-dev
```

Now, for the PostgreSQL driver shared library, there are two ways to install:

With LuaRocks

Begin by installing luarocks and making sure that tarantool is among the upstream servers, as in the instructions on rocks.tarantool.org, the Tarantool luarocks page. Now execute this:

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

For example:

```
luarocks install pg POSTGRESQL_LIBDIR=/usr/local/postgresql/lib
```

See also [Modules, LuaRocks, and requiring modules](#).

With GitHub

Go the site github.com/tarantool/pg. Follow the instructions there, saying:

```
git clone https://github.com/tarantool/pg.git  
cd pg && cmake . -DCMAKE_BUILD_TYPE=RelWithDebInfo  
make  
make install
```

At this point it is a good idea to check that the installation produced a file named `driver.so`, and to check that this file is on a directory that is searched by the `require` request.

Connecting

Begin by making a `require` request for the `pg` driver. We will assume that the name is `pg` in further examples.

```
pg = require('pg')
```

Now, say:

```
connection_name = pg.connect(connection_options)
```

The connection-options parameter is a table. Possible options are:

- `host` = *host-name* - string, default value = 'localhost'
- `port` = *port-number* - number, default value = 3306
- `user` = *user-name* - string, default value is operating-system user name
- `pass` = *password* or `password` = *password* - string, default value is blank
- `db` = *database-name* - string, default value is blank

The names are similar to the names that PostgreSQL itself uses.

Example, using a table literal enclosed in {braces}:

```
conn = pg.connect({
  host = '127.0.0.1',
  port = 5432,
  user = 'p',
  password = 'p',
  db = 'test'
})
```

Example, creating a function which sets each option in a separate line:

```
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()
---
...
```

We will assume that the name is 'conn' in further examples.

How to ping

To ensure that a connection is working, the request is:

```
connection-name:ping()
```

Example:

```
tarantool> conn:ping()
---
- true
...
```

Executing a statement

For all PostgreSQL statements, the request is:

```
connection-name:execute(sql-statement [, parameters])
```

where *sql-statement* is a string, and the optional *parameters* are extra values that can be plugged in to replace any question marks ("?"s) in the SQL statement.

Example:

```
tarantool> conn:execute('select tablename from pg_tables')
---
- - tablename: pg_statistic
- - tablename: pg_type
- - tablename: pg_authid
<...>
...
```

Closing connection

To end a session that began with `pg.connect`, the request is:

```
connection-name:close()
```

Example:

```
tarantool> conn:close()
---
...
```

For further information, including examples of rarely-used requests, see the README.md file at github.com/tarantool/pg.

Example

The example was run on an Ubuntu 12.04 ("precise") machine where tarantool had been installed in a /usr subdirectory, and a copy of PostgreSQL had been installed on /usr. The PostgreSQL server is already running on the local host 127.0.0.1.

```
$ # Check that the include subdirectory exists
$ # by looking for /usr/include/postgresql/libpq-fe.h.
$ [ -f /usr/include/postgresql/libpq-fe.h ] && echo "OK" || echo "Error"
OK

$ # Check that the library subdirectory exists and has the necessary .so file.
$ [ -f /usr/lib/x86_64-linux-gnu/libpq.so ] && echo "OK" || echo "Error"
OK

$ # Check that the psql client can connect using some factory defaults:
$ # port = 5432, user = 'postgres', user password = 'postgres',
$ # database = 'postgres'. These can be changed, provided one changes
$ # them in all places. Insert a row in database postgres, and quit.
$ psql -h 127.0.0.1 -p 5432 -U postgres -d postgres
Password for user postgres:
psql (9.3.10)
SSL connection (cipher: DHE-RSA-AES256-SHA, bits: 256)
```

```

Type "help" for help.

postgres=# CREATE TABLE test (s1 INT, s2 VARCHAR(50));
CREATE TABLE
postgres=# INSERT INTO test VALUES (1,'PostgreSQL row');
INSERT 0 1
postgres=# \q
$

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

$ # Set up the Tarantool rock list in ~/.luarocks,
$ # following instructions at rocks.tarantool.org
$ mkdir ~/.luarocks
$ echo "rocks_servers = {[[http://rocks.tarantool.org/]]}" >> \
  ~/.luarocks/config.lua

$ # Ensure that the next "install" will get files from Tarantool master
$ # repository. The resultant display is normal for Ubuntu 12.04 precise
$ cat /etc/apt/sources.list.d/tarantool.list
deb http://tarantool.org/dist/1.7/ubuntu/ precise main
deb-src http://tarantool.org/dist/1.7/ubuntu/ precise main

$ # Install tarantool-dev. The displayed line should show version = 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) ...
$

$ # Use luarocks to install locally, that is, relative to $HOME
$ luarocks install pg POSTGRESQL_LIBDIR=/usr/lib/x86_64-linux-gnu --local
Installing http://rocks.tarantool.org/pg-scm-1.rockspec...
... (more info about building the Tarantool/PostgreSQL driver appears here)
pg scm-1 is now built and installed in ~/.luarocks/

$ # Ensure driver.so now has been created in a place
$ # tarantool will look at
$ find ~/.luarocks -name "driver.so"
~/.luarocks/lib/luarocks/5.1/pg/driver.so

$ # Change directory to a directory which can be used for
$ # temporary tests. For this example we assume that the
$ # name of this directory is $HOME/tarantool_sandbox.
$ # (Change "$HOME" to whatever is the user's actual
$ # home directory for the machine that's used for this test.)
cd $HOME/tarantool_sandbox

$ # Start the Tarantool server. Do not use a Lua initialization file.

$ tarantool
tarantool: version 1.7.0-412-g803b15c
type 'help' for interactive help
tarantool>

```

Configure tarantool and load pg module. Make sure that tarantool doesn't reply "error" for the call to "require()".

```
tarantool> box.cfg{}
...
tarantool> pg = require('pg')
---
```

Create a Lua function that will connect to the PostgreSQL server, (using some factory default values for the port and user and password), retrieve one row, and display the row. For explanations of the statement types used here, read the Lua tutorial earlier in the Tarantool user manual.

```
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 '
```

Observe the result. It contains "PostgreSQL row". So this is the row that was inserted into the PostgreSQL database. And now it's been selected with the Tarantool client.

5.2 Модуль *expirationd*

For a commercial-grade example of a Lua rock that works with Tarantool, let us look at *expirationd*, which Tarantool supplies on [GitHub](#) with an Artistic license. The *expirationd.lua* program is lengthy (about 500 lines), so here we will only highlight the matters that will be enhanced by studying the full source later.

```
task.worker_fiber = fiber.create(worker_loop, task)
log.info("expiration: task %q restarted", task.name)
...
fiber.sleep(expirationd.constants.check_interval)
...
```

Whenever one hears "daemon" in Tarantool, one should suspect it's being done with a *fiber*. The program is making a fiber and turning control over to it so it runs occasionally, goes to sleep, then comes back for more.

```
for _, tuple in scan_space.index[0]:pairs(nil, {iterator = box.index.ALL}) do
...
  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)
...

```

The "for" instruction can be translated as "iterate through the index of the space that is being scanned and within it, if the tuple is "expired"(for example, if the tuple has a timestamp field which is less than the current time), process the tuple as an expired tuple.

```

-- default process_expired_tuple function
local function default_tuple_drop(space_id, args, tuple)
    local key = fun.map(
        function(x) return tuple[x.fieldno] end,
        box.space[space_id].index[0].parts
    ):totable()
    box.space[space_id]:delete(key)
end

```

Ultimately the tuple-expiry process leads to `default_tuple_drop()` which does a "delete" of a tuple from its original space. First the `fun` module is used, specifically `fun.map`. Remembering that `index[0]` is always the space's primary key, and `index[0].parts[N].fieldno` is always the field number for key part N, `fun.map()` is creating a table from the primary-key values of the tuple. The result of `fun.map()` is passed to `space_object:delete()`.

```

local function expirationd_run_task(name, space_id, is_tuple_expired, options)
...

```

At this point, if the above explanation is worthwhile, it's clear that `expirationd.lua` starts a background routine (fiber) which iterates through all the tuples in a space, sleeps cooperatively so that other fibers can operate at the same time, and - whenever it finds a tuple that has expired - deletes it from this space. Now the "expirationd_run_task()" function can be used in a test which creates sample data, lets the daemon run for a while, and prints results.

For those who like to see things run, here are the exact steps to get `expirationd` through the test.

1. Get `expirationd.lua`. There are standard ways - it is after all part of a [standard rock](#) - but for this purpose just copy the contents of `expirationd.lua` to a default directory.
2. Start the Tarantool server as described before.
3. Execute these requests:

```

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()
```

The database-specific requests (`cfg`, `space.create`, `create_index`) should already be familiar.

The function which will be supplied to `expirationd` is `is_tuple_expired`, which is saying "if the second field of the tuple is less than the *current time*, then return true, otherwise return false".

The key for getting the rock rolling is `expd = require('expirationd')`. The "require" function is what reads in the program; it will appear in many later examples in this manual, when it's necessary to get a module that's not part of the Tarantool kernel. After the Lua variable `expd` has been assigned the value of the `expirationd` module, it's possible to invoke the module's `run_task()` function.

After *sleeping* for two seconds, when the task has had time to do its iterations through the spaces, `expd.task_stats()` will print out a report showing how many tuples have expired – "expired_count: 0". After sleeping for two more seconds, `expd.task_stats()` will print out a report showing how many tuples have expired – "expired_count: 1". This shows that the `is_tuple_expired()` function eventually returned "true" for one of the tuples, because its timestamp field was more than three seconds old.

Of course, `expirationd` can be customized to do different things by passing different parameters, which will be evident after looking in more detail at the source code.

5.3 Модуль *shard*

With sharding, the tuples of a tuple set are distributed to multiple nodes, with a Tarantool database server on each node. With this arrangement, each server is handling only a subset of the total data, so larger loads can be handled by simply adding more computers to a network.

The Tarantool `shard` module has facilities for creating shards, as well as analogues for the data-manipulation functions of the `box` library (`select`, `insert`, `replace`, `update`, `delete`).

First some terminology:

Consistent Hash The `shard` module distributes according to a hash algorithm, that is, it applies a hash function to a tuple's primary-key value in order to decide which shard the tuple belongs to. The hash function is *consistent* so that changing the number of servers will not affect results for many keys. The specific hash function that the `shard` module uses is *digest.guava* in the `digest` module.

Queue A temporary list of recent update requests. Sometimes called "batching". Since updates to a sharded database can be slow, it may speed up throughput to send requests to a queue rather than wait for the update to finish on every node. The `shard` module has functions for adding requests to the queue, which it will process without further intervention. Queuing is optional.

Redundancy The number of replicas in each shard.

Replica A complete copy of the data. The `shard` module handles both sharding and replication. One shard can contain one or more replicas. When a write occurs, the write is attempted on every replica in turn. The `shard` module does not use the built-in replication feature.

Shard A subset of the tuples in the database partitioned according to the value returned by the consistent hash function. Usually each shard is on a separate node, or a separate set of nodes (for example if `redundancy = 3` then the shard will be on three nodes).

Zone A physical location where the nodes are closely connected, with the same security and backup and access points. The simplest example of a zone is a single computer with a single `tarantool-server` instance. A shard's replicas should be in different zones.

The shard package is distributed separately from the main tarantool package. To acquire it, do a separate install. For example on Ubuntu say:

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

Or, download from github tarantool/shard and compile as described in the README. Then, before using the module, say `shard = require('shard')`

The most important function is:

```
shard.init(shard-configuration)
```

This must be called for every shard. The shard-configuration is a table with these fields:

- servers (a list of URIs of nodes and the zones the nodes are in)
- login (the user name which applies for accessing via the shard module)
- password (the password for the login)
- redundancy (a number, minimum 1)
- binary (a port number that this host is listening on, on the current host) (distinguishable from the 'listen' port specified by box.cfg)

Possible Errors: Redundancy should not be greater than the number of servers; the servers must be alive; two replicas of the same shard should not be in the same zone.

5.3.1 Example: shard.init syntax for one shard

The number of replicas per shard (redundancy) is 3. The number of servers is 3. The shard module will conclude that there is only one shard.

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

5.3.2 Example: shard.init syntax for three shards

This describes three shards. Each shard has two replicas. Since the number of servers is 7, and the number of replicas per shard is 2, and dividing $7 / 2$ leaves a remainder of 1, one of the servers will not be used. This is not necessarily an error, because perhaps one of the servers in the list is not alive.

```

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 = 'tester',
  >   password = 'pass',
  >   redundancy = '2',
  >   binary = 33131,
  > }
---
...
tarantool> shard.init(cfg)
---
...

```

```

shard[space-name].insert{...}
shard[space-name].replace{...}
shard[space-name].delete{...}
shard[space-name].select{...}
shard[space-name].update{...}
shard[space-name].auto_increment{...}

```

Every data-access function in the box module has an analogue in the shard module, so (for example) to insert in table T in a sharded database one simply says `shard.T:insert{...}` instead of `box.space.T:insert{...}`. A `shard.T:select{}` request without a primary key will search all shards.

```

shard[space-name].q_insert{...}
shard[space-name].q_replace{...}
shard[space-name].q_delete{...}
shard[space-name].q_select{...}
shard[space-name].q_update{...}
shard[space-name].q_auto_increment{...}

```

Every queued data-access function has an analogue in the shard module. The user must add an `operation_id`. The details of queued data-access functions, and of maintenance-related functions, are on [the shard section of github](#).

5.3.3 Example: Shard, Minimal Configuration

There is only one shard, and that shard contains only one replica. So this isn't illustrating the features of either replication or sharding, it's only illustrating what the syntax is, and what the messages look like, that anyone could duplicate in a minute or two with the magic of cut-and-paste.

```

$ 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.passwd('admin', 'password')
tarantool> cfg = {
  > servers = {
  >   { uri = 'localhost:3301', zone = '1' },
  > },
  > login = 'admin';
  > password = 'password';
  > redundancy = 1;
  > binary = 3301;
  > }
tarantool> shard = require('shard')
tarantool> shard.init(cfg)
tarantool> -- Now put something in ...
tarantool> shard.tester:insert{1,'Tuple #1'}

```

If one cuts and pastes the above, then the result, showing only the requests and responses for `shard.init` and `shard.tester`, should look approximately like this:

```

tarantool> shard.init(cfg)
2015-08-09 ... I> Sharding initialization started...
2015-08-09 ... I> establishing connection to cluster servers...
2015-08-09 ... I> - localhost:3301 - connecting...
2015-08-09 ... I> - localhost:3301 - connected
2015-08-09 ... I> connected to all servers
2015-08-09 ... I> started
2015-08-09 ... I> redundancy = 1
2015-08-09 ... I> Zone len=1 THERE
2015-08-09 ... I> Adding localhost:3301 to shard 1
2015-08-09 ... I> Zone len=1 THERE
2015-08-09 ... I> shards = 1
2015-08-09 ... I> Done
---
- true
...
tarantool> -- Now put something in ...
---
...
tarantool> shard.tester:insert{1,'Tuple #1'}
---
- - [1, 'Tuple #1']
...

```

5.3.4 Example: Shard, Scaling Out

There are two shards, and each shard contains one replica. This requires two nodes. In real life the two nodes would be two computers, but for this illustration the requirement is merely: start two shells, which we'll call Terminal#1 and Terminal #2.

On Terminal #1, say:

```

$ 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.passwd('admin', 'password')
tarantool> console = require('console')
tarantool> cfg = {
  > servers = {
  >   { uri = 'localhost:3301', zone = '1' },
  >   { uri = 'localhost:3302', zone = '2' },
  > },
  > login = 'admin',
  > password = 'password',
  > redundancy = 1,
  > binary = 3301,
  > }
tarantool> shard = require('shard')
tarantool> shard.init(cfg)
tarantool> -- Now put something in ...
tarantool> shard.tester:insert{1, 'Tuple #1'}
```

On Terminal #2, say:

```
$ 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.passwd('admin', 'password')
tarantool> console = require('console')
tarantool> cfg = {
  > servers = {
  >   { uri = 'localhost:3301', zone = '1' };
  >   { uri = 'localhost:3302', zone = '2' };
  > };
  > login = 'admin';
  > password = 'password';
  > redundancy = 1;
  > binary = 3302;
  > }
tarantool> shard = require('shard')
tarantool> shard.init(cfg)
tarantool> -- Now get something out ...
tarantool> shard.tester:select{1}
```

What will appear on Terminal #1 is: a loop of error messages saying "Connection refused" and "server check failure". This is normal. It will go on until Terminal #2 process starts.

What will appear on Terminal #2, at the end, should look like this:

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

```
...
```

This shows that what was inserted by Terminal #1 can be selected by Terminal #2, via the shard module. Details are on [the shard section of github](#).

5.4 Module *tdb*

The Tarantool Debugger (abbreviation = *tdb*) can be used with any Lua program. The operational features include: setting breakpoints, examining variables, going forward one line at a time, backtracing, and showing information about fibers. The display features include: using different colors for different situations, including line numbers, and adding hints.

It is not supplied as part of the Tarantool repository; it must be installed separately. Here is the usual way:

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

To initiate *tdb* within a Lua program and set a breakpoint, edit the program to include these lines:

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

To start the debugging session, execute the Lua program. Execution will stop at the breakpoint, and it will be possible to enter debugging commands.

5.4.1 Debugger Commands

- bt** Backtrace – show the stack (in red), with program/function names and line numbers of whatever has been invoked to reach the current line.
- c** Continue till next breakpoint or till program ends.
- e** Enter evaluation mode. When the program is in evaluation mode, one can execute certain Lua statements that would be valid in the context. This is particularly useful for displaying the values of the program's variables. Other debugger commands will not work until one exits evaluation mode by typing `-e`.
- e** Exit evaluation mode.
- f** Display the fiber id, the program name, and the percentage of memory used, as a table.
- n** Go to the next line, skipping over any function calls.
- globals** Display names of variables or functions which are defined as global.
- h** Display a list of debugger commands.
- locals** Display names and values of variables, for example the control variables of a Lua "for" statement.
- q** Quit immediately.

5.4.2 Example Session

Put the following program in a default directory and call it "example.lua":

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

Now start Tarantool, using example.lua as the initialization file

```
$ tarantool example.lua
```

The screen should now look like this:

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

Debugger prompts are blue, debugger hints and information are green, and the current line – line 3 of example.lua – is the default color. Now enter six debugger commands:

```
n -- go to next line
n -- go to next line
e -- enter evaluation mode
j -- display j
-e -- exit evaluation mode
q -- quit
```

The screen should now look like this:

```
$ 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
```

Another debugger example can be found [here](#).

6.1 Модуль *box*

`box_function_ctx_t`

Opaque structure passed to the stored C procedure

`int box_return_tuple(box_function_ctx_t *ctx, box_tuple_t *tuple)`

Return a tuple from stored C procedure.

Returned tuple is automatically reference counted by Tarantool.

Параметры

- `ctx` (*box_function_ctx_t**) – an opaque structure passed to the stored C procedure by Tarantool
- `tuple` (*box_tuple_t**) – a tuple to return

Результат -1 on error (perhaps, out of memory; check *box_error_last()*)

Результат 0 otherwise

`uint32_t box_space_id_by_name(const char *name, uint32_t len)`

Find space id by name.

This function performs SELECT request to `_vspace` system space.

Параметры

- `char* name` (*const*) – space name
- `len` (*uint32_t*) – length of `name`

Результат `BOX_ID_NIL` on error or if not found (check *box_error_last()*)

Результат `space_id` otherwise

See also: *box_index_id_by_name*

`uint32_t box_index_id_by_name(uint32_t space_id, const char *name, uint32_t len)`
Find index id by name.

Параметры

- `space_id` (*uint32_t*) – space identifier
- `char* name` (*const*) – index name
- `len` (*uint32_t*) – length of `name`

Результат BOX_ID_NIL on error or if not found (check [box_error_last\(\)](#))

Результат `space_id` otherwise

This function performs SELECT request to `_vindex` system space.

See also: [box_space_id_by_name](#)

`int box_insert(uint32_t space_id, const char *tuple, const char *tuple_end, box_tuple_t **result)`
Execute an INSERT/REPLACE request.

Параметры

- `space_id` (*uint32_t*) – space identifier
- `char* tuple` (*const*) – encoded tuple in MsgPack Array format ([field1, field2, ...])
- `char* tuple_end` (*const*) – end of a tuple
- `result` ([box_tuple_t**](#)) – output argument. Resulted tuple. Can be set to NULL to discard result

Результат -1 on error (check [box_error_last\(\)](#))

Результат 0 otherwise

See also [space_object.insert\(\)](#)

`int box_replace(uint32_t space_id, const char *tuple, const char *tuple_end, box_tuple_t **result)`
Execute an REPLACE request.

Параметры

- `space_id` (*uint32_t*) – space identifier
- `char* tuple` (*const*) – encoded tuple in MsgPack Array format ([field1, field2, ...])
- `char* tuple_end` (*const*) – end of a tuple
- `result` ([box_tuple_t**](#)) – output argument. Resulted tuple. Can be set to NULL to discard result

Результат -1 on error (check [box_error_last\(\)](#))

Результат 0 otherwise

See also [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)`
Execute an DELETE request.

Параметры

- `space_id` (*uint32_t*) – space identifier
- `index_id` (*uint32_t*) – index identifier
- `char* key` (*const*) – encoded key in MsgPack Array format ([field1, field2, ...])

- `char* key_end (const)` – end of a key
- `result (box_tuple_t**)` – output argument. Result an old tuple. Can be set to NULL to discard result

Результат -1 on error (check [box_error_last\(\)](#))

Результат 0 otherwise

See also [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)
Execute an UPDATE request.
```

Параметры

- `space_id (uint32_t)` – space identifier
- `index_id (uint32_t)` – index identifier
- `char* key (const)` – encoded key in MsgPack Array format ([field1, field2, ...])
- `char* key_end (const)` – end of a key
- `char* ops (const)` – encoded operations in MsgPack Array format, e.g. [['=', field_id, value], ['!', 2, 'xxx']]
- `char* ops_end (const)` – end of a ops
- `index_base (int)` – 0 if field_ids in update operation are zero-based indexed (like C) or 1 if for one-based indexed field ids (like Lua).
- `result (box_tuple_t**)` – output argument. Result an old tuple. Can be set to NULL to discard result

Результат -1 on error (check [box_error_last\(\)](#))

Результат 0 otherwise

See also [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)
Execute an UPSERT request.
```

Параметры

- `space_id (uint32_t)` – space identifier
- `index_id (uint32_t)` – index identifier
- `char* tuple (const)` – encoded tuple in MsgPack Array format ([field1, field2, ...])
- `char* tuple_end (const)` – end of a tuple
- `char* ops (const)` – encoded operations in MsgPack Array format, e.g. [['=', field_id, value], ['!', 2, 'xxx']]
- `char* ops_end (const)` – end of a ops
- `index_base (int)` – 0 if field_ids in update operation are zero-based indexed (like C) or 1 if for one-based indexed field ids (like Lua).
- `result (box_tuple_t**)` – output argument. Result an old tuple. Can be set to NULL to discard result

Результат -1 on error (check [:box_error_last\(\)](#))

Результат 0 otherwise

See also `space_object.upsert()`

6.2 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)
```

6.3 Модуль *coio*

```
enum COIO_EVENT
```

```
enumerator COIO_READ
    READ event
```

```
enumerator COIO_WRITE
    WRITE event
```

```
int coio_wait(int fd, int event, double timeout)
    Wait until READ or WRITE event on socket (fd). Yields.
```

Параметры

- `fd` (*int*) – non-blocking socket file description
- `event` (*int*) – requested events to wait. Combination of `COIO_READ` | `COIO_WRITE` bit flags.
- `timeout` (*double*) – timeout in seconds.

Результат 0 - timeout

Результат >0 - returned events. Combination of `TNT_IO_READ` | `TNT_IO_WRITE` bit flags.

```
ssize_t coio_call(ssize_t (*func)(va_list), ...)
```

Create new eio task with specified function and arguments. Yield and wait until the task is complete or a timeout occurs.

This function doesn't throw exceptions to avoid double error checking: in most cases it's also necessary to check the return value of the called function and perform necessary actions. If `func` sets `errno`, the `errno` is preserved across the call.

Результат -1 and `errno = ENOMEM` if failed to create a task

Результат the function return (`errno` is preserved).

Example:

```

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)
    // handle errors.
...

```

int coio_getaddrinfo(const char *host, const char *port, const struct addrinfo *hints, struct addrinfo **res, double timeout)
 Fiber-friendly version of *getaddrinfo(3)*.

6.4 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_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_SPACE_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_SOPHIA
enumerator ER_LOCAL_SERVER_IS_NOT_ACTIVE
enumerator ER_UNKNOWN_SERVER
enumerator ER_CLUSTER_ID_MISMATCH
enumerator ER_INVALID_UUID
enumerator ER_CLUSTER_ID_IS_RO
enumerator ER_RESERVED66
enumerator ER_SERVER_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_SLAB_ALLOC_MAX
enumerator ER_WRONG_SPACE_OPTIONS
enumerator ER_UNSUPPORTED_INDEX_FEATURE
enumerator ER_VIEW_IS_RO
enumerator box_error_code_MAX
```

`box_error_t`

Error - contains information about error.

```
const char * box_error_type(const box_error_t *error)
Return the error type, e.g. "ClientError "SocketError etc.
```

Параметры

- `error` (`box_error_t*`) – error

Результат not-null string

```
uint32_t box_error_code(const box_error_t *error)
Return IPPROTO error code
```

Параметры

- `error` (`box_error_t*`) – error

Результат enum `box_error_code`

```
const char * box_error_message(const box_error_t *error)
Return the error message
```

Параметры

- `error` (`box_error_t*`) – error

Результат not-null string

`box_error_t * box_error_last(void)`

Get the information about the last API call error.

The Tarantool error handling works most like libc's `errno`. All API calls return `-1` or `NULL` in the event of error. An internal pointer to `box_error_t` type is set by API functions to indicate what went wrong. This value is only significant if API call failed (returned `-1` or `NULL`).

Successful function can also touch the last error in some cases. You don't have to clear the last error before calling API functions. The returned object is valid only until next call to **any** API function.

You must set the last error using `box_error_set()` in your stored C procedures if you want to return a custom error message. You can re-throw the last API error to IPROTO client by keeping the current value and returning `-1` to Tarantool from your stored procedure.

Результат last error

`void box_error_clear(void)`

Clear the last error.

`int box_error_set(const char *file, unsigned line, uint32_t code, const char *format, ...)`

Set the last error.

Параметры

- `char* file (const)` –
- `line (unsigned)` –
- `code (uint32_t)` – IPROTO *error code*
- `char* format (const)` –
- ... – format arguments

See also: IPROTO *error code*

`box_error_raise(code, format, ...)`

A backward-compatible API define.

6.5 Модуль *fiber*

`struct fiber`

Fiber - contains information about fiber

`struct fiber *fiber_new(const char *name, fiber_func f)`

`typedef int (*fiber_func)(va_list)`

Create a new fiber.

Takes a fiber from fiber cache, if it's not empty. Can fail only if there is not enough memory for the fiber structure or fiber stack.

The created fiber automatically returns itself to the fiber cache when its "main" function completes.

Параметры

- `char* name (const)` – string with fiber name
- `f (fiber_func)` – func for run inside fiber

See also: *fiber_start()*

void `fiber_yield`(void)
Return control to another fiber and wait until it'll be woken.

See also: [`fiber_wakeup\(\)`](#)

void `fiber_start`(struct *fiber* **callee*, ...)
Start execution of created fiber.

Параметры

- `fiber*` *callee* (*struct*) – fiber to start
- ... – arguments to start the fiber with

void `fiber_wakeup`(struct *fiber* **f*)
Interrupt a synchronous wait of a fiber

Параметры

- `fiber*` *f* (*struct*) – fiber to be woken up

void `fiber_cancel`(struct *fiber* **f*)
Cancel the subject fiber (set `FIBER_IS_CANCELLED` flag)

If target fiber's flag `FIBER_IS_CANCELLED` set, then it would be woken up (maybe prematurely). Then current fiber yields until the target fiber is dead (or is woken up by [`fiber_wakeup\(\)`](#)).

Параметры

- `fiber*` *f* (*struct*) – fiber to be cancelled

bool `fiber_set_cancellable`(bool *yesno*)
Make it possible or not possible to wakeup the current fiber immediately when it's cancelled.

Параметры

- `fiber*` *f* (*struct*) – fiber
- *yesno* (*bool*) – status to set

Результат previous state

void `fiber_set_joinable`(struct *fiber* **fiber*, bool *yesno*)
Set fiber to be joinable (`false` by default).

Параметры

- `fiber*` *f* (*struct*) – fiber
- *yesno* (*bool*) – status to set

void `fiber_join`(struct *fiber* **f*)
Wait until the fiber is dead and then move its execution status to the caller. The fiber must not be detached.

Параметры

- `fiber*` *f* (*struct*) – fiber to be woken up

Before: `FIBER_IS_JOINABLE` flag is set.

See also: [`fiber_set_joinable\(\)`](#)

void `fiber_sleep`(double *s*)
Put the current fiber to sleep for at least '*s*' seconds.

Параметры

- `s` (*double*) – time to sleep

Note: this is a cancellation point.

See also: `fiber_is_cancelled()`

bool `fiber_is_cancelled()`

Check current fiber for cancellation (it must be checked manually).

double `fiber_time(void)`

Report loop begin time as double (cheap).

uint64_t `fiber_time64(void)`

Report loop begin time as 64-bit int.

void `fiber_reschedule(void)`

Reschedule fiber to end of event loop cycle.

struct `slab_cache`

struct `slab_cache` *`cord_slab_cache(void)`

Return `slab_cache` suitable to use with `tarantool/small` library

6.6 Модуль *index*

box_iterator_t

A space iterator

enum iterator_type

Controls how to iterate over tuples in an index. Different index types support different iterator types. For example, one can start iteration from a particular value (request key) and then retrieve all tuples where keys are greater or equal (= GE) to this key.

If iterator type is not supported by the selected index type, iterator constructor must fail with `ER_UNSUPPORTED`. To be selectable for primary key, an index must support at least `ITER_EQ` and `ITER_GE` types.

NULL value of request key corresponds to the first or last key in the index, depending on iteration direction. (first key for GE and GT types, and last key for LE and LT). Therefore, to iterate over all tuples in an index, one can use `ITER_GE` or `ITER_LE` iteration types with start key equal to NULL. For `ITER_EQ`, the key must not be NULL.

enumerator `ITER_EQ`

key == x ASC order

enumerator `ITER_REQ`

key == x DESC order

enumerator `ITER_ALL`

all tuples

enumerator `ITER_LT`

key < x

enumerator `ITER_LE`

key <= x

enumerator `ITER_GE`

key >= x

```

enumerator ITER_GT
    key > x

enumerator ITER_BITS_ALL_SET
    all bits from x are set in key

enumerator ITER_BITS_ANY_SET
    at least one x's bit is set

enumerator ITER_BITS_ALL_NOT_SET
    all bits are not set

enumerator ITER_OVERLAPS
    key overlaps x

enumerator ITER_NEIGHBOR
    tuples in distance ascending order from specified point

```

```

box_iterator_t *box_index_iterator(uint32_t space_id, uint32_t index_id, int type, const
                                char *key, const char *key_end)

```

Allocate and initialize iterator for space_id, index_id.

The returned iterator must be destroyed by *box_iterator_free*.

Параметры

- space_id (*uint32_t*) – space identifier
- index_id (*uint32_t*) – index identifier
- type (*int*) – *iterator_type*
- char* key (*const*) – encode key in MsgPack Array format ([part1, part2, ...])
- char* key_end (*const*) – the end of encoded key

Результат NULL on error (check :ref:box_error_last‘c_api-error-box_error_last>‘)

Результат iterator otherwise

See also *box_iterator_next*, *box_iterator_free*

```

int box_iterator_next(box_iterator_t *iterator, box_tuple_t **result)

```

Retrieve the next item from the iterator.

Параметры

- iterator (*box_iterator_t**) – an iterator returned by :ref:box_index_iterator‘c_api-box_index-box_index_iterator>‘
- result (*box_tuple_t***) – output argument. result a tuple or NULL if there is no more data.

Результат -1 on error (check :ref:box_error_last‘c_api-error-box_error_last>‘)

Результат 0 on success. The end of data is not an error.

```

void box_iterator_free(box_iterator_t *iterator)

```

Destroy and deallocate iterator.

Параметры

- iterator (*box_iterator_t**) – an iterator returned by :ref:box_index_iterator‘c_api-box_index-box_index_iterator>‘

```

ssize_t box_index_len(uint32_t space_id, uint32_t index_id)

```

Return the number of element in the index.

Параметры

- `space_id (uint32_t)` – space identifier
- `index_id (uint32_t)` – index identifier

Результат -1 on error (check `:ref:box_error_last`'`c_api-error-box_error_last`>')

Результат `>= 0` otherwise

```
ssize_t box_index_bsize(uint32_t space_id, uint32_t index_id)
```

Return the number of bytes used in memory by the index.

Параметры

- `space_id (uint32_t)` – space identifier
- `index_id (uint32_t)` – index identifier

Результат -1 on error (check `:ref:box_error_last`'`c_api-error-box_error_last`>')

Результат `>= 0` otherwise

```
int box_index_random(uint32_t space_id, uint32_t index_id, uint32_t rnd, box_tuple_t **result)
```

Return a random tuple from the index (useful for statistical analysis).

Параметры

- `space_id (uint32_t)` – space identifier
- `index_id (uint32_t)` – index identifier
- `rnd (uint32_t)` – random seed
- `result (box_tuple_t**)` – output argument. result a tuple or NULL if there is no tuples in space

See also: [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)
```

Get a tuple from index by the key.

Please note that this function works much more faster than [index_object.select](#) or [box_index_iterator](#) + [box_iterator_next](#).

Параметры

- `space_id (uint32_t)` – space identifier
- `index_id (uint32_t)` – index identifier
- `char* key (const)` – encode key in MsgPack Array format ([part1, part2, ...])
- `char* key_end (const)` – the end of encoded key
- `result (box_tuple_t**)` – output argument. result a tuple or NULL if there is no tuples in space

Результат -1 on error (check `:ref:box_error_last`'`c_api-error-box_error_last`>')

Результат 0 on success

See also: `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)
```

Return a first (minimal) tuple matched the provided key.

Параметры

- `space_id` (*uint32_t*) – space identifier
- `index_id` (*uint32_t*) – index identifier
- `char* key` (*const*) – encode key in MsgPack Array format ([part1, part2, ...])
- `char* key_end` (*const*) – the end of encoded key
- `result` (*box_tuple_t***) – output argument. result a tuple or NULL if there is no tuples in space

Результат -1 on error (check `:ref:box_error_last()` ‘`c_api-error-box_error_last`>’)

Результат 0 on success

See also: [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)
```

Return a last (maximal) tuple matched the provided key.

Параметры

- `space_id` (*uint32_t*) – space identifier
- `index_id` (*uint32_t*) – index identifier
- `char* key` (*const*) – encode key in MsgPack Array format ([part1, part2, ...])
- `char* key_end` (*const*) – the end of encoded key
- `result` (*box_tuple_t***) – output argument. result a tuple or NULL if there is no tuples in space

Результат -1 on error (check `:ref:box_error_last()` ‘`c_api-error-box_error_last`>’)

Результат 0 on success

See also: [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)
```

Count the number of tuple matched the provided key.

Параметры

- `space_id` (*uint32_t*) – space identifier
- `index_id` (*uint32_t*) – index identifier
- `type` (*int*) – [iterator_type](#)
- `char* key` (*const*) – encode key in MsgPack Array format ([part1, part2, ...])
- `char* key_end` (*const*) – the end of encoded key

Результат -1 on error (check `:ref:box_error_last()` ‘`c_api-error-box_error_last`>’)

Результат 0 on success

See also: [index_object.count\(\)](#)

6.7 Модуль *latch*

`box_latch_t`

A lock for cooperative multitasking environment

`box_latch_t *box_latch_new(void)`

Allocate and initialize the new latch.

Результат allocated latch object

Тип результата `box_latch_t *`

`void box_latch_delete(box_latch_t *latch)`

Destroy and free the latch.

Параметры

- `latch (box_latch_t*)` – latch to destroy

`void box_latch_lock(box_latch_t *latch)`

Lock a latch. Waits indefinitely until the current fiber can gain access to the latch.

param `box_latch_t* latch` latch to lock

`int box_latch_trylock(box_latch_t *latch)`

Try to lock a latch. Return immediately if the latch is locked.

Параметры

- `latch (box_latch_t*)` – latch to lock

Результат status of operation. 0 - success, 1 - latch is locked

Тип результата `int`

`void box_latch_unlock(box_latch_t *latch)`

Unlock a latch. The fiber calling this function must own the latch.

Параметры

- `latch (box_latch_t*)` – latch to unlock

6.8 Модуль *lua/utils*

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

Push cdata of given `ctypeid` onto the stack.

CTypeID must be used from FFI at least once. Allocated memory returned uninitialized. Only numbers and pointers are supported.

Параметры

- `L (lua_State*)` – Lua State
- `ctypeid (uint32_t)` – FFI's CTypeID of this cdata

Результат memory associated with this cdata

See also: `luaL_checkcdata()`

`void *luaL_checkcdata(struct lua_State *L, int idx, uint32_t *ctypeid)`

Checks whether the function argument `idx` is a cdata

Параметры

- `L (lua_State*)` – Lua State
- `idx (int)` – stack index
- `ctypeid (uint32_t*)` – output argument. FFI's CTypeID of returned cdata

Результат memory associated with this cdata

See also: [luaL_pushcdata\(\)](#)

void `luaL_setcdatagc(struct lua_State *L, int idx)`
Sets finalizer function on a cdata object.

Equivalent to call `ffi.gc(obj, function)`. Finalizer function must be on the top of the stack.

Параметры

- `L (lua_State*)` – Lua State
- `idx (int)` – stack index

uint32_t `luaL_ctypeid(struct lua_State *L, const char *ctypeiname)`
Return CTypeID (FFI) of given CDATA type

Параметры

- `L (lua_State*)` – Lua State
- `char* ctypeiname (const)` – C type name as string (e.g. "struct request" or "uint32_t")

Результат CTypeID

See also: [luaL_pushcdata\(\)](#), [luaL_checkcdata\(\)](#)

int `luaL_cdef(struct lua_State *L, const char *ctypeiname)`
Declare symbols for FFI

Параметры

- `L (lua_State*)` – Lua State
- `char* ctypeiname (const)` – C definitions (e.g. "struct stat")

Результат 0 on success

Результат LUA_ERRRUN, LUA_ERRMEM or ``LUA_ERRERR otherwise.

See also: `ffi.cdef(def)`

void `luaL_pushuint64(struct lua_State *L, uint64_t val)`
Push uint64_t onto the stack

Параметры

- `L (lua_State*)` – Lua State
- `val (uint64_t)` – value to push

void `luaL_pushint64(struct lua_State *L, int64_t val)`
Push int64_t onto the stack

Параметры

- `L (lua_State*)` – Lua State
- `val (int64_t)` – value to push

```
uint64_t luaL_checkuint64(struct lua_State *L, int idx)
```

Checks whether the argument `idx` is a `uint64` or a convertible string and returns this number.

Throws error if the argument can't be converted

```
uint64_t luaL_checkint64(struct lua_State *L, int idx)
```

Checks whether the argument `idx` is a `int64` or a convertible string and returns this number.

Throws error if the argument can't be converted

```
uint64_t luaL_touint64(struct lua_State *L, int idx)
```

Checks whether the argument `idx` is a `uint64` or a convertible string and returns this number.

Результат the converted number or 0 if argument can't be converted

```
int64_t luaL_toint64(struct lua_State *L, int idx)
```

Checks whether the argument `idx` is a `int64` or a convertible string and returns this number.

Результат the converted number or 0 if argument can't be converted

6.9 Модуль `say` (логирование)

```
enum say_level
```

```
enumerator S_FATAL
```

do not use this value directly

```
enumerator S_SYSERROR
```

```
enumerator S_ERROR
```

```
enumerator S_CRIT
```

```
enumerator S_WARN
```

```
enumerator S_INFO
```

```
enumerator S_DEBUG
```

```
say(level, format, ...)
```

Format and print a message to Tarantool log file.

Параметры

- `level` (*int*) – *log level*
- `char* format` (*const*) – `printf()`-like format string
- ... – format arguments

See also *printf(3)*, *say_level*

```
say_error(format, ...)
```

```
say_crit(format, ...)
```

```
say_warn(format, ...)
```

```
say_info(format, ...)
```

```
say_debug(format, ...)
```

```
say_syserror(format, ...)
```

Format and print a message to Tarantool log file.

Параметры

- `char* format` (*const*) – `printf()`-like format string

- ... – format arguments

See also *printf(3)*, *say_level*

Example:

```
say_info("Some useful information: %s", status);
```

6.10 Модуль *schema*

enum SCHEMA

enumerator BOX_SYSTEM_ID_MIN
Start of the reserved range of system spaces.

enumerator BOX_SCHEMA_ID
Space id of `_schema`.

enumerator BOX_SPACE_ID
Space id of `_space`.

enumerator BOX_VSPACE_ID
Space id of `_vspace` view.

enumerator BOX_INDEX_ID
Space id of `_index`.

enumerator BOX_VINDEX_ID
Space id of `_vindex` view.

enumerator BOX_FUNC_ID
Space id of `_func`.

enumerator BOX_VFUNC_ID
Space id of `_vfunc` view.

enumerator BOX_USER_ID
Space id of `_user`.

enumerator BOX_VUSER_ID
Space id of `_vuser` view.

enumerator BOX_PRIV_ID
Space id of `_priv`.

enumerator BOX_VPRIV_ID
Space id of `_vpriv` view.

enumerator BOX_CLUSTER_ID
Space id of `_cluster`.

enumerator BOX_SYSTEM_ID_MAX
End of reserved range of system spaces.

enumerator BOX_ID_NIL
NULL value, returned on error.

6.11 Модуль *trivia/config*

API_EXPORT

Extern modifier for all public functions.

PACKAGE_VERSION_MAJOR

Package major version - 1 for 1.7.0.

PACKAGE_VERSION_MINOR

Package minor version - 7 for 1.7.0.

PACKAGE_VERSION_PATCH

Package patch version - 0 for 1.7.0.

PACKAGE_VERSION

A string with major-minor-patch-commit-id identifier of the release, e.g. 1.7.0-1216-g73f7154.

SYSCONF_DIR

System configuration dir (e.g */etc*)

INSTALL_PREFIX

Install prefix (e.g. */usr*)

BUILD_TYPE

Build type, e.g. Debug or Release

BUILD_INFO

CMake build type signature, e.g. *Linux-x86_64-Debug*

BUILD_OPTIONS

Command line used to run CMake.

COMPILER_INFO

Pathes to C and CXX compilers.

TARANTOOL_C_FLAGS

C compile flags used to build Tarantool.

TARANTOOL_CXX_FLAGS

CXX compile flags used to build Tarantool.

MODULE_LIBDIR

A path to install *.lua module files.

MODULE_LUADIR

A path to install *.so/*.dylib module files.

MODULE_INCLUDEDIR

A path to Lua includes (the same directory where this file is contained)

MODULE_LUAPATH

A constant added to `package.path` in Lua to find *.lua module files.

MODULE_LIBPATH

A constant added to `package.cpath` in Lua to find *.so module files.

6.12 Модуль *tuple*

`box_tuple_format_t`

`box_tuple_format_t *box_tuple_format_default(void)`
Tuple format.

Each Tuple has associated format (class). Default format is used to create tuples which are not attach to any particular space.

`box_tuple_t`
Tuple

`box_tuple_t *box_tuple_new(box_tuple_format_t *format, const char *tuple, const char *tuple_end)`

Allocate and initialize a new tuple from a raw MsgPack Array data.

Параметры

- `format` (`box_tuple_format_t*`) – tuple format. Use `box_tuple_format_default()` to create space-independent tuple.
- `char* tuple` (`const`) – tuple data in MsgPack Array format ([field1, field2, ...])
- `char* tuple_end` (`const`) – the end of data

Результат NULL on out of memory

Результат tuple otherwise

See also: `box_tuple_new()`

`int box_tuple_ref(box_tuple_t *tuple)`

Increase the reference counter of tuple.

Tuples are reference counted. All functions that return tuples guarantee that the last returned tuple is refcounted internally until the next call to API function that yields or returns another tuple.

You should increase the reference counter before taking tuples for long processing in your code. Such tuples will not be garbage collected even if another fiber remove they from space. After processing please decrement the reference counter using `box_tuple_unref()`, otherwise the tuple will leak.

Параметры

- `tuple` (`box_tuple_t*`) – a tuple

Результат -1 on error

Результат 0 otherwise

See also: `box_tuple_unref()`

`void box_tuple_unref(box_tuple_t *tuple)`

Decrease the reference counter of tuple.

Параметры

- `tuple` (`box_tuple_t*`) – a tuple

Результат -1 on error

Результат 0 otherwise

See also: `box_tuple_ref()`

`uint32_t box_tuple_field_count(const box_tuple_t *tuple)`

Return the number of fields in tuple (the size of MsgPack Array).

Параметры

- `tuple` (`box_tuple_t*`) – a tuple

`size_t box_tuple_bsize(const box_tuple_t *tuple)`

Return the number of bytes used to store internal tuple data (MsgPack Array).

Параметры

- `tuple` (*box_tuple_t**) – a tuple

`ssize_t box_tuple_to_buf(const box_tuple_t *tuple, char *buf, size_t size)`

Dump raw MsgPack data to the memory buffer `buf` of size `size`.

Store tuple fields in the memory buffer.

Upon successful return, the function returns the number of bytes written. If buffer size is not enough then the return value is the number of bytes which would have been written if enough space had been available.

Результат -1 on error

Результат number of bytes written on success.

`box_tuple_format_t *box_tuple_format(const box_tuple_t *tuple)`

Return the associated format.

Параметры

- `tuple` (*box_tuple_t**) – a tuple

Результат tuple format

`const char *box_tuple_field(const box_tuple_t *tuple, uint32_t field_id)`

Return the raw tuple field in MsgPack format.

The buffer is valid until next call to `box_tuple_*` functions.

Параметры

- `tuple` (*box_tuple_t**) – a tuple
- `field_id` (*uint32_t*) – zero-based index in MsgPack array.

Результат NULL if `i >= box_tuple_field_count()`

Результат msgpack otherwise

`box_tuple_iterator_t`

Tuple iterator

`box_tuple_iterator_t *box_tuple_iterator(box_tuple_t *tuple)`

Allocate and initialize a new tuple iterator. The tuple iterator allow to iterate over fields at root level of MsgPack array.

Example:

```
box_tuple_iterator_t* it = box_tuple_iterator(tuple);
if (it == NULL) {
    // error handling using box_error_last()
}
const char* field;
while (field = box_tuple_next(it)) {
    // process raw MsgPack data
}

// rewind iterator to first position
box_tuple_rewind(it)
assert(box_tuple_position(it) == 0);
```

```

// rewind three fields
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)`

Destroy and free tuple iterator

uint32_t `box_tuple_position(box_tuple_iterator_t *it)`

Return zero-based next position in iterator. That is, this function return the field id of field that will be returned by the next call to `box_tuple_next()`. Returned value is zero after initialization or rewind and `box_tuple_field_count()` after the end of iteration.

Параметры

- `it (box_tuple_iterator_t*)` – a tuple iterator

Результат position

void `box_tuple_rewind(box_tuple_iterator_t *it)`

Rewind iterator to the initial position.

Параметры

- `it (box_tuple_iterator_t*)` – a tuple iterator

After: `box_tuple_position(it) == 0`

const char *`box_tuple_seek(box_tuple_iterator_t *it, uint32_t field_no)`

Seek the tuple iterator.

The returned buffer is valid until next call to `box_tuple_*` API. Requested `field_no` returned by next call to `box_tuple_next(it)`.

Параметры

- `it (box_tuple_iterator_t*)` – a tuple iterator
- `field_no (uint32_t)` – field number - zero-based position in MsgPack array

After:

- `box_tuple_position(it) == field_no` if returned value is not NULL.
- `box_tuple_position(it) == box_tuple_field_count(tuple)` if returned value is NULL.

const char *`box_tuple_next(box_tuple_iterator_t *it)`

Return the next tuple field from tuple iterator.

The returned buffer is valid until next call to `box_tuple_*` API.

Параметры

- `it (box_tuple_iterator_t*)` –

Результат NULL if there are no more fields

Результат MsgPack otherwise

Before: `box_tuple_position()` is zero-based ID of returned field.

After: `box_tuple_position(it) == box_tuple_field_count(tuple)` if returned value is 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)
```

6.13 Модуль *txn*

```
bool box_txn(void)
```

Return true if there is an active transaction.

```
int box_txn_begin(void)
```

Begin a transaction in the current fiber.

A transaction is attached to caller fiber, therefore one fiber can have only one active transaction.

Результат 0 on success

Результат -1 on error. Perhaps a transaction has already been started

```
int box_txn_commit(void)
```

Commit the current transaction.

Результат 0 on success

Результат -1 on error. Perhaps a disk write failure

```
void box_txn_rollback(void)
```

Rollback the current transaction.

```
void *box_txn_alloc(size_t size)
```

Allocate memory on txn memory pool.

The memory is automatically deallocated when the transaction is committed or rolled back.

Результат NULL on out of memory

7.1 Состав документации

Tarantool documentation consists of:

- a user guide
- developer guide (you're reading it)
- coding style guide for C, Python (for other connectors, we use conventions of the connector programming language community)

7.2 Сборка из исходных файлов

For downloading Tarantool source and building it, the platforms can differ and the preferences can differ. But the steps are always the same. Here in the manual we'll explain what the steps are, and after that you can look at some example scripts on the Internet.

1. Get tools and libraries that will be necessary for building and testing.

The absolutely necessary ones are:

- A program for downloading source repositories. For all platforms, this is `git`. It allows to download the latest complete set of source files from the Tarantool repository at GitHub.
- A C/C++ compiler. Ordinarily, this is `gcc` and `g++` version 4.6 or later. On Mac OS X, this is `Clang` version 3.2 or later.
- A program for managing the build process. For all platforms, this is `CMake`. The `CMake` version should be 2.8 or later.
- Command-line interpreter for Python-based code (namely, for Tarantool test suite). For all platforms, this is `python`. The Python version should be greater than 2.6 – preferably 2.7 – and less than 3.0.

Here are names of tools and libraries which may have to be installed in advance, using `sudo apt-get` (for Ubuntu), `sudo yum install` (for CentOS), or the equivalent on other platforms. Different platforms may use slightly different names. Ignore the ones marked *optional*, *only in Mac OS scripts* unless the platform is Mac OS.

- `gcc` and `g++`, or `clang` # see above
- `git` # see above
- `cmake` # see above
- `python` # see above; for test suite
- `libreadline-dev` or `libreadline6-dev` or `readline-devel` # for interactive mode
- `libssl-dev` # for *digest* module
- `autoconf` # optional, only in Mac OS scripts
- `zlib1g` or `zlib` # optional, only in Mac OS scripts

2. Set up Python modules for running the test suite.

This step is optional. Python modules are not necessary for building Tarantool itself, unless you intend to use the "Run the test suite" option in step 7.

You need the following Python modules:

- `pip`, any version
- `dev`, any version
- `pyYAML` version 3.10
- `argparse` version 1.1
- `msgpack-python` version 0.4.6
- `gevent` version 1.1b5
- `six` version 1.8.0

On Ubuntu, you can get the modules from the repository:

```
sudo apt-get install python-pip python-dev python-yaml <...>
```

On CentOS 6, you can likewise get the modules from the repository:

```
sudo yum install python26 python26-PyYAML <...>
```

If some modules are not available on a repository, it is best to set up the modules by getting a tarball and doing the setup with `python setup.py`, thus:

```
# On some machines, this initial command may be necessary:
# wget https://bootstrap.pypa.io/ez_setup.py -O - | sudo python

# Python module for parsing YAML (pyYAML), for test suite:
# (If wget fails, check at http://pyyaml.org/wiki/PyYAML
# what the current version is.)
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
```


Finally, use Python `pip` to bring in Python packages that may not be up-to-date in the distro repositories. (On CentOS 7, it will be necessary to install `pip` first, with `sudo yum install epel-release` followed by `sudo yum install python-pip`.)

```
pip install tarantool>0.4 --user
```

- Use `git` to download the latest Tarantool source code from the GitHub repository `tarantool/tarantool`, branch 1.7. For example, to a local directory named `~/tarantool`:

```
git clone https://github.com/tarantool/tarantool.git ~/tarantool
```

- Use `git` again so that third-party contributions will be seen as well.

The build depends on the following external libraries:

- Readline development files (`libreadline-dev/readline-devel` package).
- OpenSSL development files (`libssl-dev/openssl-devel` package).
- `libyaml` (`libyaml-dev/libyaml-devel` package).
- `liblz4` (`liblz4-dev/lz4-devel` package).
- GNU `bfd` which is the part of GNU `binutils` (`binutils-dev/binutils-devel` package).

This step is only necessary once, the first time you do a download.

```
cd ~/tarantool
git submodule init
git submodule update --recursive
cd ../
```

On rare occasions, the submodules will need to be updated again with the command:

```
git submodule update --init --recursive
```

Note: There is an alternative – to say `git clone --recursive` earlier in step 3, – but we prefer the method above because it works with older versions of `git`.

- Use CMake to initiate the build.

```
cd ~/tarantool
make clean          # unnecessary, added for good luck
rm CMakeCache.txt  # unnecessary, added for good luck
cmake .             # start initiating with build type=Debug
```

On some platforms, it may be necessary to specify the C and C++ versions, for example:

```
CC=gcc-4.8 CXX=g++-4.8 cmake .
```

The CMake option for specifying build type is `-DCMAKE_BUILD_TYPE=type`, where *type* can be:

- `Debug` – used by project maintainers
- `Release` – used only if the highest performance is required
- `RelWithDebInfo` – used for production, also provides debugging capabilities

The CMake option for hinting that the result will be distributed is `-DENABLE_DIST=ON`. If this option is on, then later `make install` will install `tarantoolctl` files in addition to `tarantool` files.

- Use `make` to complete the build.

```
make
```

This creates the 'tarantool' executable in the directory *src/*

Next, it's highly recommended to say `make install` to install Tarantool to the */usr/local* directory and keep your system clean. However, it is possible to run the Tarantool executable without installation.

7. Run the test suite.

This step is optional. Tarantool's developers always run the test suite before they publish new versions. You should run the test suite too, if you make any changes in the code. Assuming you downloaded to *~/tarantool*, the principal steps are:

```
# make a subdirectory named `bin`
mkdir ~/tarantool/bin
# link python to bin (this may require superuser privilege)
ln /usr/bin/python ~/tarantool/bin/python
# get on the test subdirectory
cd ~/tarantool/test
# run tests using python
PATH=~/tarantool/bin:$PATH ./test-run.py
```

The output should contain reassuring reports, for example:

```
=====
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.
```

To prevent later confusion, clean up what's in the *bin* subdirectory:

```
rm ~/tarantool/bin/python
rmdir ~/tarantool/bin
```

8. Make an rpm package.

This step is optional. It's only for people who want to redistribute Tarantool. Package maintainers who want to build with `rpmbuild` should consult the `rpm-build` instructions for the appropriate platform.

9. Verify your Tarantool installation.

```
tarantool $ ./src/tarantool
```

This will start Tarantool in the interactive mode.

For your added convenience, we provide OS-specific README files with example scripts at GitHub:

- [README.FreeBSD](#) for FreeBSD 10.1
- [README.MacOSX](#) for Mac OS X *El Capitan*
- [README.md](#) for generic GNU/Linux

These example scripts assume that the intent is to download from the 1.7 branch, build the server and run tests after build.

7.3 Сборка документации

This documentation is built using a simplified markup system named `Sphinx` (see <http://sphinx-doc.org>). You can build a local version of this documentation and contribute to it.

You need to install:

- `git` (a program for downloading source repositories)
- `CMake` version 2.8 or later (a program for managing the build process)
- `Python` version greater than 2.6 – preferably 2.7 – and less than 3.0 (`Sphinx` is a Python-based tool)

Also, make sure to install the following Python modules:

- `pip`, any version
- `dev`, any version
- `pyYAML` version 3.10
- `Sphinx` version 1.4.4
- `sphinx-intl` version 0.9.9
- `pelican`, any version
- `BeautifulSoup`, any version
- `gevent` version 1.1b5

See installation details in the [build-from-source](#) section of this documentation. The procedure below implies that all the prerequisites are met.

1. Use `git` to download the latest source code of this documentation from the GitHub repository `tarantool/doc`, branch 1.7. For example, to a local directory named `~/tarantool-doc`:

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

2. Use `CMake` to initiate the build.

```
cd ~/tarantool-doc
make clean           # unnecessary, added for good luck
rm CMakeCache.txt   # unnecessary, added for good luck
cmake .              # start initiating
```

3. Build a local version of the existing documentation package.

Run the `make` command with an appropriate option to specify which documentation version to build.

```
cd ~/tarantool-doc
make all              # all versions
make sphinx-html     # multi-page English version
make sphinx-singlehtml # one-page English version
make sphinx-html-ru  # multi-page Russian version
make sphinx-singlehtml # one-page Russian version
```

Documentation is created and stored at `/www/output`:

- `/www/output/doc` (English versions)
- `/www/output/doc/ru` (Russian versions)

The entry point for each version is `index.html` file in the appropriate directory.

4. Set up a web-server.

Run the following command to set up a web-server (the example below is for Ubuntu, but the procedure is similar for other supported OS's). Make sure to run it from the documentation output folder, as specified below:

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

5. Open your browser and enter 127.0.0.1:8000 into the address box. If your local documentation build is valid, the default version (English multi-page) will be displayed in the browser.

6. To contribute to documentation, use the .rst format for drafting and submit your updates as "Pull Requests" via GitHub.

To comply with the writing and formatting style, use the [guidelines](#) provided in the documentation, common sense and existing documents.

Notes:

- If you suggest creating a new documentation section (i.e., a whole new page), it has to be saved to the relevant section at GitHub.
- If you want to contribute to localizing this documentation (e.g. into Russian), add your translation strings to .po files stored in the corresponding locale directory (e.g. /sphinx/locale/ru/LC_MESSAGES/ for Russian). See more about localizing with Sphinx at <http://www.sphinx-doc.org/en/stable/intl.html>

7.4 Соглашения по разработке

7.4.1 How to work on a bug

Any defect, even minor, if it changes the user-visible server behavior, needs a bug report. Report a bug at <http://github.com/tarantool/tarantool/issues>.

When reporting a bug, try to come up with a test case right away. Set the current maintenance milestone for the bug fix, and specify the series. Assign the bug to yourself. Put the status to 'In progress' Once the patch is ready, put the bug the bug to 'In review' and solicit a review for the fix.

Once there is a positive code review, push the patch and set the status to 'Closed'

Patches for bugs should contain a reference to the respective Launchpad bug page or at least bug id. Each patch should have a test, unless coming up with one is difficult in the current framework, in which case QA should be alerted.

There are two things you need to do when your patch makes it into the master:

- put the bug to 'fix committed',
- delete the remote branch.

7.5 Соглашения по документации

These guidelines are updated on the on-demand basis, covering only those issues that cause pains to the existing writers. At this point, we do not aim to come up with an exhaustive Documentation Style Guide for the Tarantool project.

7.5.1 Markup issues

Wrapping text

The limit is 80 characters per line for plain text, and no limit for any other constructions when wrapping affects ReST readability and/or HTML output. Also, it makes no sense to wrap text into lines shorter than 80 characters unless you have a good reason to do so.

The 80-character limit comes from the ISO/ANSI 80x24 screen resolution, and it's unlikely that readers/writers will use 80-character consoles. Yet it's still a standard for many coding guidelines (including Tarantool). As for writers, the benefit is that an 80-character page guide allows keeping the text window rather narrow most of the time, leaving more space for other applications in a wide-screen environment.

Formatting code snippets

For code snippets, we mainly use the `code-block` directive with an appropriate highlighting language. The most commonly used highlighting languages are:

- .. `code-block:: tarantoolsession`
- .. `code-block:: console`
- .. `code-block:: lua`

For example (a code snippet in Lua):

```
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
```

In rare cases, when we need custom highlight for specific parts of a code snippet and the `code-block` directive is not enough, we use the per-line `codenormal` directive together and explicit output formatting (defined in `doc/sphinx/_static/sphinx_design.css`).

Examples:

- Function syntax (the placeholder *space-name* is displayed in italics):
box.space.space-name:create_index('index-name')
- A tdb session (user input is in bold, command prompt is in blue, computer output is in green):

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

Warning: Every entry of explicit output formatting (`codenormal`, `codebold`, etc) tends to cause troubles when this documentation is translated to other languages. Please avoid using explicit output formatting unless it is REALLY needed.

Using separated links

Avoid separating the link and the target definition (`ref`), like this:

```
This is a paragraph that contains `a link`_.
```

```
.. _a link: http://example.com/
```

Use non-separated links instead:

```
This is a paragraph that contains `a link <http://example.com/>`_.
```

Warning: Every separated link tends to cause troubles when this documentation is translated to other languages. Please avoid using separated links unless it is REALLY needed (e.g. in tables).

Creating labels for local links

We avoid using links that sphinx generates automatically for most objects. Instead, we add our own labels for linking to any place in this documentation.

Our naming convention is as follows:

- Character set: a through z, 0 through 9, dash, underscore.
- Format: `path dash filename dash tag`

Example: `_c_api-box_index-iterator_type` where: `c_api` is the directory name, `box_index` is the file name (without ".rst"), and `iterator_type` is the tag.

The file name is useful for knowing, when you see "ref" where it is pointing to. And if the file name is meaningful, you see that better.

The file name alone, without a path, is enough when the file name is unique within `doc/sphinx`. So, for `fiber.rst` it should be just "fiber" not "reference-fiber". While for "index.rst" (we have a handful of "index.rst" in different directories) please specify the path before the file name, e.g. "reference-index".

Use a dash "-" to delimit the path and the file name. In the documentation source, we use only underscores "_" in paths and file names, reserving dash "-" as the delimiter for local links.

The tag can be anything meaningful. The only guideline is for Tarantool syntax items (such as members), where the preferred tag syntax is `module_or_object_name dash member_name`. For example, `box_space-drop`.

Making comments

Sometimes we may need to leave comments in a ReST file. To make sphinx ignore some text during processing, use the following per-line notation with `.. //` as the comment marker:

```
.. // your comment here
```

The starting symbols `.. //` do not interfere with the other ReST markup, and they are easy to find both visually and using `grep`. There are no symbols to escape in `grep` search, just go ahead with something like this:

```
grep ".. //" doc/sphinx/dev_guide/*.rst
```

These comments don't work properly in nested documentation, though (e.g. if you leave a comment in `module -> object -> method`, sphinx ignores the comment and all nested content that follows in the method description).

7.5.2 Language and style issues

US vs British spelling

We use English US spelling.

7.5.3 Examples and templates

Module and function

Here is an example of documenting a module (`my_fiber`) and a function (`my_fiber.create`).

```
my_fiber.create(function[, function-arguments ])  
Create and start a my_fiber object. The object is created and begins to run immediately.
```

Параметры

- `function` – the function to be associated with the `my_fiber` object
- `function-arguments` – what will be passed to function

Return created `my_fiber` object

Rtype userdata

Example:

```
tarantool> my_fiber = require('my_fiber')  
---  
...  
tarantool> function function_name()  
  > my_fiber.sleep(1000)  
  > end  
---  
...  
tarantool> my_fiber_object = my_fiber.create(function_name)  
---  
...
```

Module, class and method

Here is an example of documenting a module (`my_box.index`), a class (`my_index_object`) and a function (`my_index_object.rename`).

object `my_index_object`

```
my_index_object:rename(index-name)  
Rename an index.
```

Параметры

- `index_object` – an object reference
- `index_name` – a new name for the index (type = string)

Return nil

Possible errors: index_object does not exist.

Example:

```
tarantool> box.space.space55.index.primary:rename('secondary')
---
```

Complexity Factors: Index size, Index type, Number of tuples accessed.

7.6 Протокол IProto

7.6.1 Диаграмма общей схемы данных

```
0    X
+----+
|    | - X байт
+----+
TYPE - тип переменной из библиотеки MsgPack (если это объект из библиотеки MsgPack)

+====+
|    | - объект из библиотеки MsgPack, динамически изменяемого размера
+====+
TYPE - тип переменной из библиотеки MsgPack

+~~~~+
|    | - массив/сопоставление из библиотеки MsgPack, динамически изменяемого размера
+~~~~+
TYPE - тип переменной из библиотеки MsgPack
```

Типы данных из библиотеки MsgPack:

- MP_INT - целое число
- MP_MAP - Map
- MP_ARR - Array
- MP_STRING - String
- MP_FIXSTR - Fixed size string
- MP_OBJECT - Any MsgPack object

7.6.2 Overview

IPROTO is a binary request/response protocol.

7.6.3 Greeting Packet

TARANTOOL'S GREETING:

```
0                                     63
+-----+
|                                     |
```


Tarantool Greeting (server version)		
64 bytes		
+-----+-----+-----+		
BASE64 encoded SALT	NULL	
44 bytes		
+-----+-----+-----+		
64	107	127

The server begins the dialogue by sending a fixed-size (128 bytes) text greeting to the client. The greeting always contains two 64 byte lines of ASCII text, each line ending with newline character ('\n'). The first line contains the server version and protocol type. The second line contains up to 44 bytes of base64-encoded random string, to use in authentication packet, and ends with up to 23 spaces.

7.6.4 Unified packet structure

Once a greeting is read, the protocol becomes pure request/response and features a complete access to Tarantool functionality, including:

- request multiplexing, e.g. ability to asynchronously issue multiple requests via the same connection
- response format that supports zero-copy writes

For data structuring and encoding, the protocol uses msgpack data format, see <http://msgpack.org>

Tarantool protocol mandates use of a few integer constants serving as keys in maps used in the protocol. These constants are defined in src/box/iproto_constants.h

Let's list them here too:

```
-- user keys
<code>      ::= 0x00
<sync>     ::= 0x01
<schema_id> ::= 0x05
<space_id>  ::= 0x10
<index_id>  ::= 0x11
<limit>     ::= 0x12
<offset>    ::= 0x13
<iterator>  ::= 0x14
<key>       ::= 0x20
<tuple>     ::= 0x21
<function_name> ::= 0x22
<username>  ::= 0x23
<expression> ::= 0x27
<ops>       ::= 0x28
<data>      ::= 0x30
<error>     ::= 0x31
```

```
-- -- Value for <code> key in request can be:
-- User command codes
<select>   ::= 0x01
<insert>   ::= 0x02
<replace>  ::= 0x03
<update>   ::= 0x04
<delete>   ::= 0x05
<call>     ::= 0x06
<auth>     ::= 0x07
<eval>     ::= 0x08
```

```

<upsert> ::= 0x09
-- Admin command codes
<ping>   ::= 0x40

-- -- Value for <code> key in response can be:
<OK>     ::= 0x00
<ERROR>  ::= 0x8XXX

```

Both <header> and <body> are msgpack maps:

Request/Response:

```

0      5
+-----+ +-----+ +-----+
| BODY + | |           | |           |
| HEADER | |   HEADER  | |           |
| SIZE   | |           | |           |
+-----+ +-----+ +-----+
MP_INT   MP_MAP           MP_MAP

```

UNIFIED HEADER:

```

+-----+ +-----+ +-----+
| 0x00: CODE | 0x01: SYNC | 0x05: SCHEMA_ID |
| MP_INT: MP_INT | MP_INT: MP_INT | MP_INT: MP_INT |
+-----+ +-----+ +-----+
MP_MAP

```

They only differ in the allowed set of keys and values, the key defines the type of value that follows. If a body has no keys, entire msgpack map for the body may be missing. Such is the case, for example, in <ping> request. `schema_id` may be absent in request's header, that means that there'll be no version checking, but it must be present in the response. If `schema_id` is sent in the header, then it'll be checked.

7.6.5 Authentication

When a client connects to the server, the server responds with a 128-byte text greeting message. Part of the greeting is base-64 encoded session salt - a random string which can be used for authentication. The length of decoded salt (44 bytes) exceeds the amount necessary to sign the authentication message (first 20 bytes). An excess is reserved for future authentication schemas.

PREPARE SCRAMBLE:

```

LEN(ENCODED_SALT) = 44;
LEN(SCRAMBLE)     = 20;

```

prepare 'chap-sha1' scramble:

```

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;

```

```
AUTHORIZATION BODY: CODE = 0x07

+-----+-----+
| (KEY) | (TUPLE)| len == 9 | len == 20 | |
| 0x23:USERNAME | 0x21:| "chap-sha1" | SCRAMBLE | |
| MP_INT:MP_STRING | MP_INT:| MP_STRING | MP_STRING | |
| | | +-----+-----+ |
| | | MP_ARRAY | |
+-----+-----+
MP_MAP
```

<key> holds the user name. <tuple> must be an array of 2 fields: authentication mechanism ("chap-sha1" is the only supported mechanism right now) and password, encrypted according to the specified mechanism. Authentication in Tarantool is optional, if no authentication is performed, session user is 'guest'. The server responds to authentication packet with a standard response with 0 tuples.

7.6.6 Requests

- SELECT: CODE - 0x01 Find tuples matching the search pattern

```
SELECT BODY:

+-----+-----+-----+
| 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 Inserts tuple into the space, if no tuple with same unique keys exists. Otherwise throw *duplicate key* error.
- REPLACE: CODE - 0x03 Insert a tuple into the space or replace an existing one.

```
INSERT/REPLACE BODY:

+-----+-----+
| 0x10: SPACE_ID | 0x21: TUPLE | |
| MP_INT: MP_INT | MP_INT: MP_ARRAY | |
| | | |
+-----+-----+
MP_MAP
```

- UPDATE: CODE - 0x04 Update a tuple

```
UPDATE BODY:

+-----+-----+
| | | |
+-----+-----+
```

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:

Works only for integer fields:

- * Addition OP = '+' . space[key][field_no] += argument
- * Subtraction OP = '-' . space[key][field_no] -= argument
- * Bitwise AND OP = '&' . space[key][field_no] &= argument
- * Bitwise XOR OP = '^' . space[key][field_no] ^= argument
- * Bitwise OR OP = '|' . space[key][field_no] |= argument

Works on any fields:

- * Delete OP = '#'
 - delete <argument> fields starting
 - from <field_no> in the space[<key>]

0	2	
+-----+-----+-----+		
OP	FIELD_NO	ARGUMENT
MP_FIXSTR	MP_INT	MP_INT
+-----+-----+-----+		
MP_ARRAY		

- * Insert OP = '!'
 - insert <argument> before <field_no>
- * Assign OP = '='
 - assign <argument> to field <field_no>.
 - will extend the tuple if <field_no> == <max_field_no> + 1

0	2	
+-----+-----+-----+		
OP	FIELD_NO	ARGUMENT
MP_FIXSTR	MP_INT	MP_OBJECT
+-----+-----+-----+		
MP_ARRAY		

Works on string fields:

- * Splice OP = ':'
 - take the string from space[key][field_no] and
 - substitute <offset> bytes from <position> with <argument>

0	2	
+-----+-----+-----+		

	'	FIELD_NO	POSITION	OFFSET	ARGUMENT	
	MP_FIXSTR	MP_INT	MP_INT	MP_INT	MP_STR	
+-----+-----+-----+-----+-----+-----+						
MP_ARRAY						

It's an error to specify an argument of a type that differs from expected type.

- DELETE: CODE - 0x05 Delete a tuple

DELETE BODY:

+-----+-----+-----+-----+-----+-----+						
	0x10: SPACE_ID	0x11: INDEX_ID	0x20: KEY			
	MP_INT: MP_INT	MP_INT: MP_INT	MP_INT: MP_ARRAY			
+-----+-----+-----+-----+-----+-----+						
MP_MAP						

- CALL: CODE - 0x06 Call a stored function

CALL BODY:

+-----+-----+-----+-----+-----+-----+						
	0x22: FUNCTION_NAME	0x21: TUPLE				
	MP_INT: MP_STRING	MP_INT: MP_ARRAY				
+-----+-----+-----+-----+-----+-----+						
MP_MAP						

- EVAL: CODE - 0x08 Evaluate Lua expression

EVAL BODY:

+-----+-----+-----+-----+-----+-----+						
	0x27: EXPRESSION	0x21: TUPLE				
	MP_INT: MP_STRING	MP_INT: MP_ARRAY				
+-----+-----+-----+-----+-----+-----+						
MP_MAP						

- UPSERT: CODE - 0x09 Update tuple if it would be found elsewhere try to insert tuple. Always use primary index for key.

UPSERT BODY:

+-----+-----+-----+-----+-----+-----+						
				+~~~~~+		
	0x10: SPACE_ID	0x21: TUPLE	(OPS)	OP		
	MP_INT: MP_INT	MP_INT: MP_ARRAY	0x28:			
			MP_INT:	+~~~~~+		
				MP_ARRAY		
+-----+-----+-----+-----+-----+-----+						
MP_MAP						

Operations structure same as for UPDATE operation.

```

0      2
+-----+-----+-----+
|      |      |      |
|  OP   | FIELD_NO | ARGUMENT |
| MP_FIXSTR | MP_INT | MP_INT |
|      |      |      |
+-----+-----+-----+
                MP_ARRAY

```

Supported operations:

'+' - add a value to a numeric field. If the field is not numeric, it's changed to 0 first. If the field does not exist, the operation is skipped. There is no error in case of overflow either, the value simply wraps around in C style. The range of the integer is MsgPack: from -2^{63} to $2^{64}-1$

'-' - same as the previous, but subtract a value

'=' - assign a field to a value. The field must exist, if it does not exist, the operation is skipped.

'' - insert a field. It's only possible to insert a field if this create no nil "gaps" between fields. E.g. it's possible to add a field between existing fields or as the last field of the tuple.

'#' - delete a field. If the field does not exist, the operation is skipped. It's not possible to change with update operations a part of the primary key (this is validated before performing upsert).

7.6.7 Response packet structure

We'll show whole packets here:

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

```

Set of tuples in the response <data> expects a msgpack array of tuples as value EVAL command returns arbitrary *MP_ARRAY* with arbitrary MsgPack values.

ERROR: 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

```

Where 0xXXX is ERRCODE.

Error message is present in the response only if there is an error `<error>` expects as value a msgpack string
Convenience macros which define hexadecimal constants for return codes can be found in <src/box/errcode.h>

7.6.8 Replication packet structure

```
-- replication keys
<server_id>      ::= 0x02
<lsn>           ::= 0x03
<timestamp>     ::= 0x04
<server_uuid>   ::= 0x24
<cluster_uuid> ::= 0x25
<vclock>       ::= 0x26
```

```
-- replication codes
<join>          ::= 0x41
<subscribe>    ::= 0x42
```

JOIN:

In the beginning you must send JOIN

HEADER		BODY	
0x00: 0x41	0x01: SYNC	SERVER_UUID	EMPTY
MP_INT: MP_INT	MP_INT: MP_INT	0x24: UUID	MP_STRING
MP_MAP		MP_MAP	

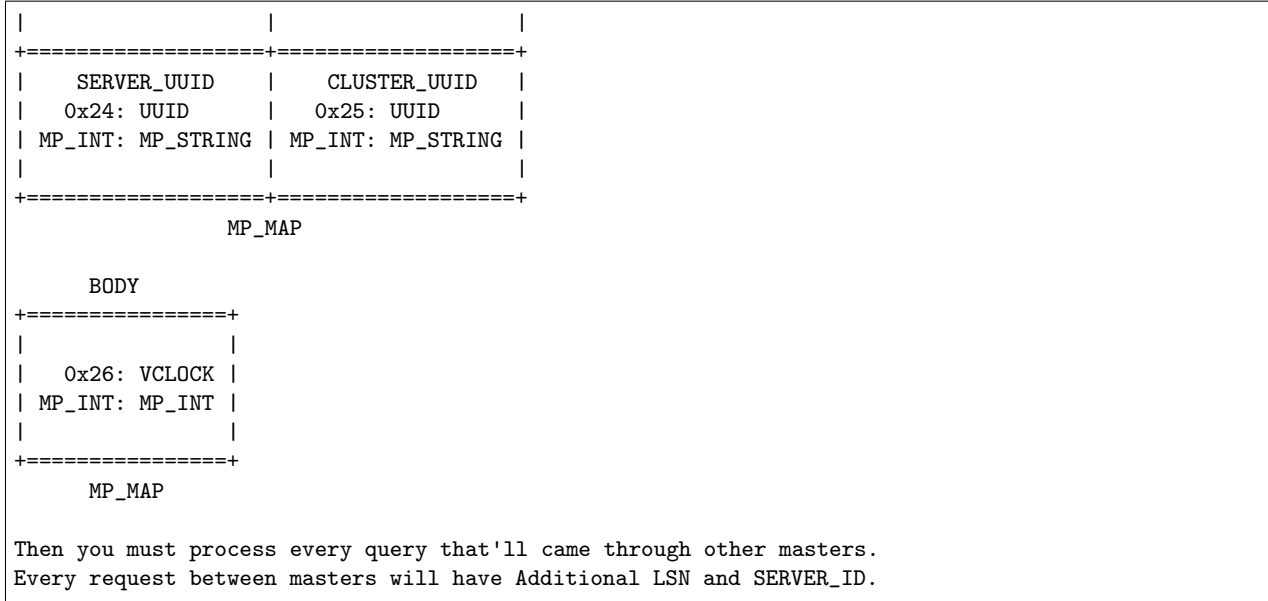
Then server, which we connect to, will send last SNAP file by, simply, creating a number of INSERTs (with additional LSN and ServerID) (don't reply). Then it'll send a vclock's MP_MAP and close a socket.

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	

SUBSCRIBE:

Then you must send SUBSCRIBE:

HEADER	
0x00: 0x41	0x01: SYNC
MP_INT: MP_INT	MP_INT: MP_INT



7.6.9 XLOG / SNAP

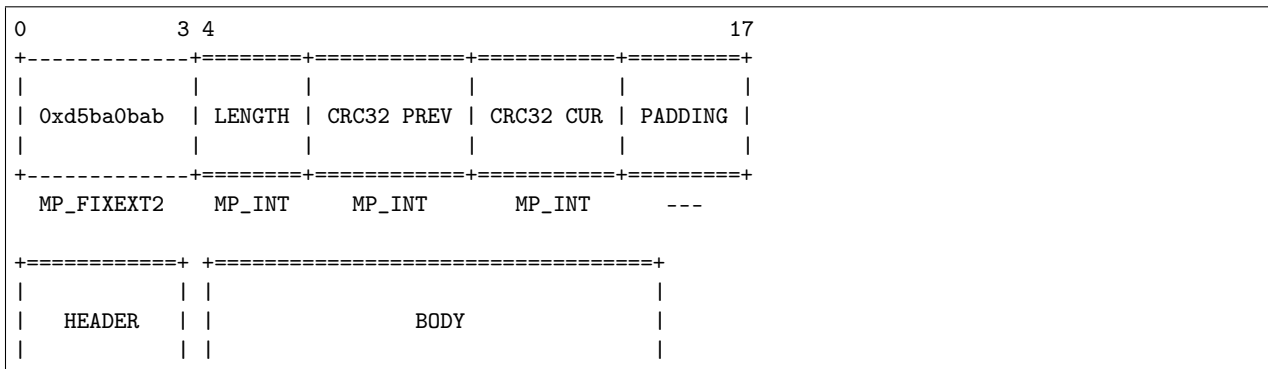
XLOG and SNAP have the same format. They start with:

```
SNAP\n
0.12\n
Server: e6eda543-eda7-4a82-8bf4-7ddd442a9275\n
VClock: {1: 0}\n
\n
...
```

So, **Header** of an SNAP/XLOG consists of:

```
<format>\n
<format_version>\n
Server: <server_uuid>\n
VClock: <vclock_map>\n
\n
```

There are two markers: tuple beginning - **0xd5ba0bab** and EOF marker - **0xd510aded**. So, next, between **Header** and EOF marker there's data with the following schema:




```
+-----+ +-----+
MP_MAP      MP_MAP
```

7.7 Соглашения по разработке на языке C

The project's coding style is based on a version of the Linux kernel coding style.

The latest version of the Linux style can be found at: <http://www.kernel.org/doc/Documentation/CodingStyle>

Since it is open for changes, the version of style that we follow, one from 2007-July-13, will be also copied later in this document.

There are a few additional guidelines, either unique to Tarantool or deviating from the Kernel guidelines.

1. Chapters 10 "Kconfig configuration files", 11 "Data structures", 13 "Printing kernel messages", 14 "Allocating memory" and 17 "Don't re-invent the kernel macros" do not apply, since they are specific to Linux kernel programming environment.
2. The rest of Linux Kernel Coding Style is amended as follows:

7.7.1 General guidelines

We use Git for revision control. The latest development is happening in the 'master' branch. Our git repository is hosted on github, and can be checked out with `git clone git://github.com/tarantool/tarantool.git #` anonymous read-only access

If you have any questions about Tarantool internals, please post them on the developer discussion list, <https://groups.google.com/forum/#!forum/tarantool>. However, please be warned: Launchpad silently deletes posts from non-subscribed members, thus please be sure to have subscribed to the list prior to posting. Additionally, some engineers are always present on `#tarantool` channel on `irc.freenode.net`.

Commenting style

Use Doxygen comment format, Javadoc flavor, i.e. `@tag` rather than `tag`. The main tags in use are `@param`, `@retval`, `@return`, `@see`, `@note` and `@todo`.

Every function, except perhaps a very short and obvious one, should have a comment. A sample function comment may look like below:

```
/** Write all data to a descriptor.
 *
 * This function is equivalent to 'write', except it would ensure
 * that all data is written to the file unless a non-ignorable
 * error occurs.
 *
 * @retval 0 Success
 *
 * @reval 1 An error occurred (not EINTR)
 * /
static int
write_all(int fd, void \*data, size_t len);
```

Public structures and important structure members should be commented as well.

Header files

Use header guards. Put the header guard in the first line in the header, before the copyright or declarations. Use all-uppercase name for the header guard. Derive the header guard name from the file name, and append `_INCLUDED` to get a macro name. For example, `core/log_io.h` -> `CORE_LOG_IO_H_INCLUDED`. In `.c` (implementation) file, include the respective declaration header before all other headers, to ensure that the header is self-sufficient. Header "header.h" is self-sufficient if the following compiles without errors:

```
#include "header.h"
```

Allocating memory

Prefer the supplied slab (`salloc`) and pool (`palloc`) allocators to `malloc()/free()` for any performance-intensive or large memory allocations. Repetitive use of `malloc()/free()` can lead to memory fragmentation and should therefore be avoided.

Always free all allocated memory, even allocated at start-up. We aim at being valgrind leak-check clean, and in most cases it's just as easy to `free()` the allocated memory as it is to write a valgrind suppression. Freeing all allocated memory is also dynamic-load friendly: assuming a plug-in can be dynamically loaded and unloaded multiple times, reload should not lead to a memory leak.

Other

Select GNU C99 extensions are acceptable. It's OK to mix declarations and statements, use `true` and `false`.

The not-so-current list of all GCC C extensions can be found at: <http://gcc.gnu.org/onlinedocs/gcc-4.3.5/gcc/C-Extensions.html>

7.7.2 Linux kernel coding style

This is a short document describing the preferred coding style for the linux kernel. Coding style is very personal, and I won't `_force_` my views on anybody, but this is what goes for anything that I have to be able to maintain, and I'd prefer it for most other things too. Please at least consider the points made here.

First off, I'd suggest printing out a copy of the GNU coding standards, and NOT read it. Burn them, it's a great symbolic gesture.

Anyway, here goes:

Chapter 1: Indentation

Tabs are 8 characters, and thus indentations are also 8 characters. There are heretic movements that try to make indentations 4 (or even 2!) characters deep, and that is akin to trying to define the value of PI to be 3.

Rationale: The whole idea behind indentation is to clearly define where a block of control starts and ends. Especially when you've been looking at your screen for 20 straight hours, you'll find it a lot easier to see how the indentation works if you have large indentations.

Now, some people will claim that having 8-character indentations makes the code move too far to the right, and makes it hard to read on a 80-character terminal screen. The answer to that is that if you need more than 3 levels of indentation, you're screwed anyway, and should fix your program.

In short, 8-char indents make things easier to read, and have the added benefit of warning you when you're nesting your functions too deep. Heed that warning.

The preferred way to ease multiple indentation levels in a switch statement is to align the "switch" and its subordinate "case" labels in the same column instead of "double-indenting" the "case" labels. e.g.:

```
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;
}
```

Don't put multiple statements on a single line unless you have something to hide:

```
if (condition) do_this;
    do_something_everytime;
```

Don't put multiple assignments on a single line either. Kernel coding style is super simple. Avoid tricky expressions.

Outside of comments, documentation and except in Kconfig, spaces are never used for indentation, and the above example is deliberately broken.

Get a decent editor and don't leave whitespace at the end of lines.

Chapter 2: Breaking long lines and strings

Coding style is all about readability and maintainability using commonly available tools.

The limit on the length of lines is 80 columns and this is a strongly preferred limit.

Statements longer than 80 columns will be broken into sensible chunks. Descendants are always substantially shorter than the parent and are placed substantially to the right. The same applies to function headers with a long argument list. Long strings are as well broken into shorter strings. The only exception to this is where exceeding 80 columns significantly increases readability and does not hide information.

```
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;
}
```

Chapter 3: Placing Braces and Spaces

The other issue that always comes up in C styling is the placement of braces. Unlike the indent size, there are few technical reasons to choose one placement strategy over the other, but the preferred way, as shown to us by the prophets Kernighan and Ritchie, is to put the opening brace last on the line, and put the closing brace first, thusly:

```
if (x is true) {
    we do y
}
```

This applies to all non-function statement blocks (if, switch, for, while, do). e.g.:

```
switch (action) {
case KOBJ_ADD:
    return "add";
case KOBJ_REMOVE:
    return "remove";
case KOBJ_CHANGE:
    return "change";
default:
    return NULL;
}
```

However, there is one special case, namely functions: they have the opening brace at the beginning of the next line, thus:

```
int function(int x)
{
    body of function;
}
```

Heretic people all over the world have claimed that this inconsistency is ... well ... inconsistent, but all right-thinking people know that (a) K&R are right and (b) K&R are right. Besides, functions are special anyway (you can't nest them in C).

Note that the closing brace is empty on a line of its own, except in the cases where it is followed by a continuation of the same statement, ie a "while" in a do-statement or an "else" in an if-statement, like this:

```
do {
    body of do-loop;
} while (condition);
```

and

```
if (x == y) {
    ..
} else if (x > y) {
    ...
} else {
    ....
}
```

Rationale: K&R.

Also, note that this brace-placement also minimizes the number of empty (or almost empty) lines, without any loss of readability. Thus, as the supply of new-lines on your screen is not a renewable resource (think 25-line terminal screens here), you have more empty lines to put comments on.

Do not unnecessarily use braces where a single statement will do.

```
if (condition)
    action();
```

This does not apply if one branch of a conditional statement is a single statement. Use braces in both branches.

```
if (condition) {
    do_this();
    do_that();
} else {
    otherwise();
}
```

Chapter 3.1: Spaces

Linux kernel style for use of spaces depends (mostly) on function-versus-keyword usage. Use a space after (most) keywords. The notable exceptions are `sizeof`, `typeof`, `alignof`, and `__attribute__`, which look somewhat like functions (and are usually used with parentheses in Linux, although they are not required in the language, as in: "sizeof info" after "struct fileinfo info;" is declared).

So use a space after these keywords: `if`, `switch`, `case`, `for`, `do`, `while` but not with `sizeof`, `typeof`, `alignof`, or `__attribute__`. E.g.,

```
s = sizeof(struct file);
```

Do not add spaces around (inside) parenthesized expressions. This example is **bad**:

```
s = sizeof( struct file );
```

When declaring pointer data or a function that returns a pointer type, the preferred use of `**` is adjacent to the data name or function name and not adjacent to the type name. Examples:

```
char *linux_banner;
unsigned long long memparse(char *ptr, char **retptr);
char *match_strdup(substring_t *s);
```

Use one space around (on each side of) most binary and ternary operators, such as any of these:

= + - < > * / % | & ^ <= >= == != ? :

but no space after unary operators:

& * + - ~ ! sizeof typeof alignof __attribute__ defined

no space before the postfix increment & decrement unary operators:

++ -

no space after the prefix increment & decrement unary operators:

++ -

and no space around the `'.'` and `>` structure member operators.

Do not leave trailing whitespace at the ends of lines. Some editors with "smart" indentation will insert whitespace at the beginning of new lines as appropriate, so you can start typing the next line of code right away. However, some such editors do not remove the whitespace if you end up not putting a line of code there, such as if you leave a blank line. As a result, you end up with lines containing trailing whitespace.

Git will warn you about patches that introduce trailing whitespace, and can optionally strip the trailing whitespace for you; however, if applying a series of patches, this may make later patches in the series fail by changing their context lines.

Chapter 4: Naming

C is a Spartan language, and so should your naming be. Unlike Modula-2 and Pascal programmers, C programmers do not use cute names like `ThisVariableIsATemporaryCounter`. A C programmer would call that variable `tmp` which is much easier to write, and not the least more difficult to understand.

HOWEVER, while mixed-case names are frowned upon, descriptive names for global variables are a must. To call a global function `foo` is a shooting offense.

GLOBAL variables (to be used only if you *really* need them) need to have descriptive names, as do global functions. If you have a function that counts the number of active users, you should call that `count_active_users()` or similar, you should *not* call it `cntusr()`.

Encoding the type of a function into the name (so-called Hungarian notation) is brain damaged - the compiler knows the types anyway and can check those, and it only confuses the programmer. No wonder MicroSoft makes buggy programs.

LOCAL variable names should be short, and to the point. If you have some random integer loop counter, it should probably be called `i`. Calling it `loop_counter` is non-productive, if there is no chance of it being mis-understood. Similarly, `tmp` can be just about any type of variable that is used to hold a temporary value.

If you are afraid to mix up your local variable names, you have another problem, which is called the function-growth-hormone-imbalance syndrome. See chapter 6 (Functions).

Chapter 5: Typedefs

Please don't use things like `vps_t`.

It's a *mistake* to use typedef for structures and pointers. When you see a

```
vps_t a;
```

in the source, what does it mean?

In contrast, if it says

```
struct virtual_container *a;
```

you can actually tell what `a` is.

Lots of people think that typedefs "help readability". Not so. They are useful only for:

1. totally opaque objects (where the typedef is actively used to *hide* what the object is).

Example: `pte_t` etc. opaque objects that you can only access using the proper accessor functions.

NOTE! Opaqueness and "accessor functions" are not good in themselves. The reason we have them for things like `pte_t` etc. is that there really is absolutely *zero* portably accessible information there.

2. Clear integer types, where the abstraction *helps* avoid confusion whether it is `int` or `long`.

`u8/u16/u32` are perfectly fine typedefs, although they fit into category (d) better than here.

NOTE! Again - there needs to be a *reason* for this. If something is `unsigned long` then there's no reason to do

```
typedef unsigned long myflags_t;
```

but if there is a clear reason for why it under certain circumstances might be an "unsigned int" and under other configurations might be "unsigned long then by all means go ahead and use a typedef.

3. when you use sparse to literally create a `_new_` type for type-checking.
4. New types which are identical to standard C99 types, in certain exceptional circumstances.

Although it would only take a short amount of time for the eyes and brain to become accustomed to the standard types like `'uint32_t'`, some people object to their use anyway.

Therefore, the Linux-specific `'u8/u16/u32/u64'` types and their signed equivalents which are identical to standard types are permitted – although they are not mandatory in new code of your own.

When editing existing code which already uses one or the other set of types, you should conform to the existing choices in that code.

5. Types safe for use in userspace.

In certain structures which are visible to userspace, we cannot require C99 types and cannot use the `'u32'` form above. Thus, we use `__u32` and similar types in all structures which are shared with userspace.

Maybe there are other cases too, but the rule should basically be to NEVER EVER use a typedef unless you can clearly match one of those rules.

In general, a pointer, or a struct that has elements that can reasonably be directly accessed should **never** be a typedef.

Chapter 6: Functions

Functions should be short and sweet, and do just one thing. They should fit on one or two screenfuls of text (the ISO/ANSI screen size is 80x24, as we all know), and do one thing and do that well.

The maximum length of a function is inversely proportional to the complexity and indentation level of that function. So, if you have a conceptually simple function that is just one long (but simple) case-statement, where you have to do lots of small things for a lot of different cases, it's OK to have a longer function.

However, if you have a complex function, and you suspect that a less-than-gifted first-year high-school student might not even understand what the function is all about, you should adhere to the maximum limits all the more closely. Use helper functions with descriptive names (you can ask the compiler to in-line them if you think it's performance-critical, and it will probably do a better job of it than you would have done).

Another measure of the function is the number of local variables. They shouldn't exceed 5-10, or you're doing something wrong. Re-think the function, and split it into smaller pieces. A human brain can generally easily keep track of about 7 different things, anything more and it gets confu/sed. You know you're brilliant, but maybe you'd like to understand what you did 2 weeks from now.

In source files, separate functions with one blank line. If the function is exported, the `EXPORT*` macro for it should follow immediately after the closing function brace line. E.g.:

```
int system_is_up(void)
{
    return system_state == SYSTEM_RUNNING;
}
EXPORT_SYMBOL(system_is_up);
```

In function prototypes, include parameter names with their data types. Although this is not required by the C language, it is preferred in Linux because it is a simple way to add valuable information for the reader.

Chapter 7: Centralized exiting of functions

Albeit deprecated by some people, the equivalent of the goto statement is used frequently by compilers in form of the unconditional jump instruction.

The goto statement comes in handy when a function exits from multiple locations and some common work such as cleanup has to be done.

The rationale is:

- unconditional statements are easier to understand and follow
- nesting is reduced
- errors by not updating individual exit points when making modifications are prevented
- saves the compiler work to optimize redundant code away ;)

```
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;
}
```

Chapter 8: Commenting

Comments are good, but there is also a danger of over-commenting. NEVER try to explain HOW your code works in a comment: it's much better to write the code so that the `_working_` is obvious, and it's a waste of time to explain badly written code. Generally, you want your comments to tell WHAT your code does, not HOW. Also, try to avoid putting comments inside a function body: if the function is so complex that you need to separately comment parts of it, you should probably go back to chapter 6 for a while. You can make small comments to note or warn about something particularly clever (or ugly), but try to avoid excess. Instead, put the comments at the head of the function, telling people what it does, and possibly WHY it does it.

When commenting the kernel API functions, please use the kernel-doc format. See the files `Documentation/kernel-doc-nano-HOWTO.txt` and `scripts/kernel-doc` for details.

Linux style for comments is the C89 `"/* ... */"` style. Don't use C99-style `"// ..."` comments.

The preferred style for long (multi-line) comments is:

```
/*
 * This is the preferred style for multi-line
```



```

* comments in the Linux kernel source code.
* Please use it consistently.
*
* Description: A column of asterisks on the left side,
* with beginning and ending almost-blank lines.
*/

```

It's also important to comment data, whether they are basic types or derived types. To this end, use just one data declaration per line (no commas for multiple data declarations). This leaves you room for a small comment on each item, explaining its use.

Chapter 9: You've made a mess of it

That's OK, we all do. You've probably been told by your long-time Unix user helper that "GNU emacs" automatically formats the C sources for you, and you've noticed that yes, it does do that, but the defaults it uses are less than desirable (in fact, they are worse than random typing - an infinite number of monkeys typing into GNU emacs would never make a good program).

So, you can either get rid of GNU emacs, or change it to use saner values. To do the latter, you can stick the following in your .emacs file:

```

(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")))))

```

This will make emacs go better with the kernel coding style for C files below `~/src/linux-trees`.

But even if you fail in getting emacs to do sane formatting, not everything is lost: use "indent".

Now, again, GNU indent has the same brain-dead settings that GNU emacs has, which is why you need to give it a few command line options. However, that's not too bad, because even the makers of GNU indent recognize the authority of K&R (the GNU people aren't evil, they are just severely misguided in

this matter), so you just give indent the options `kr -i8` (stands for "K&R, 8 character indents"), or use `scripts/Lindent` which indents in the latest style.

"indent" has a lot of options, and especially when it comes to comment re-formatting you may want to take a look at the man page. But remember: "indent" is not a fix for bad programming.

Chapter 10: Kconfig configuration files

For all of the Kconfig* configuration files throughout the source tree, the indentation is somewhat different. Lines under a "config" definition are indented with one tab, while help text is indented an additional two spaces. Example:

```
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.
```

Features that might still be considered unstable should be defined as dependent on "EXPERIMENTAL":

```
config SLUB
    depends on EXPERIMENTAL && !ARCH_USES_SLAB_PAGE_STRUCT
    bool "SLUB (Unqueued Allocator)"
    ...
```

while seriously dangerous features (such as write support for certain filesystems) should advertise this prominently in their prompt string:

```
config ADFS_FS_RW
    bool "ADFS write support (DANGEROUS)"
    depends on ADFS_FS
    ...
```

For full documentation on the configuration files, see the file `Documentation/kbuild/kconfig-language.txt`.

Chapter 11: Data structures

Data structures that have visibility outside the single-threaded environment they are created and destroyed in should always have reference counts. In the kernel, garbage collection doesn't exist (and outside the kernel garbage collection is slow and inefficient), which means that you absolutely have to reference count all your uses.

Reference counting means that you can avoid locking, and allows multiple users to have access to the data structure in parallel - and not having to worry about the structure suddenly going away from under them just because they slept or did something else for a while.

Note that locking is not a replacement for reference counting. Locking is used to keep data structures coherent, while reference counting is a memory management technique. Usually both are needed, and they are not to be confused with each other.

Many data structures can indeed have two levels of reference counting, when there are users of different "classes". The subclass count counts the number of subclass users, and decrements the global count just once when the subclass count goes to zero.

Examples of this kind of "multi-level-reference-counting" can be found in memory management ("struct mm_struct": mm_users and mm_count), and in filesystem code ("struct super_block": s_count and s_active).

Remember: if another thread can find your data structure, and you don't have a reference count on it, you almost certainly have a bug.

Chapter 12: Macros, Enums and RTL

Names of macros defining constants and labels in enums are capitalized.

```
#define CONSTANT 0x12345
```

Enums are preferred when defining several related constants.

CAPITALIZED macro names are appreciated but macros resembling functions may be named in lower case.

Generally, inline functions are preferable to macros resembling functions.

Macros with multiple statements should be enclosed in a do - while block:

```
#define macrofun(a, b, c) |\n    do {                  |\n        if (a == 5)      |\n            do_this(b, c); |\n    } while (0)
```

Things to avoid when using macros:

1. macros that affect control flow:

```
#define FOO(x)           |\n    do {                 |\n        if (blah(x) < 0) |\n            return -EBUGGERED; |\n    } while(0)
```

is a very bad idea. It looks like a function call but exits the "calling" function; don't break the internal parsers of those who will read the code.

2. macros that depend on having a local variable with a magic name:

```
#define FOO(val) bar(index, val)
```

might look like a good thing, but it's confusing as hell when one reads the code and it's prone to breakage from seemingly innocent changes.

3. macros with arguments that are used as l-values: FOO(x) = y; will bite you if somebody e.g. turns FOO into an inline function.
4. forgetting about precedence: macros defining constants using expressions must enclose the expression in parentheses. Beware of similar issues with macros using parameters.

```
#define CONSTANT 0x4000\n#define CONSTEXP (CONSTANT | 3)
```

The cpp manual deals with macros exhaustively. The gcc internals manual also covers RTL which is used frequently with assembly language in the kernel.

Chapter 13: Printing kernel messages

Kernel developers like to be seen as literate. Do mind the spelling of kernel messages to make a good impression. Do not use crippled words like "dont"; use "do not" or "don't" instead. Make the messages concise, clear, and unambiguous.

Kernel messages do not have to be terminated with a period.

Printing numbers in parentheses (%d) adds no value and should be avoided.

There are a number of driver model diagnostic macros in `<linux/device.h>` which you should use to make sure messages are matched to the right device and driver, and are tagged with the right level: `dev_err()`, `dev_warn()`, `dev_info()`, and so forth. For messages that aren't associated with a particular device, `<linux/kernel.h>` defines `pr_debug()` and `pr_info()`.

Coming up with good debugging messages can be quite a challenge; and once you have them, they can be a huge help for remote troubleshooting. Such messages should be compiled out when the `DEBUG` symbol is not defined (that is, by default they are not included). When you use `dev_dbg()` or `pr_debug()`, that's automatic. Many subsystems have `Kconfig` options to turn on `-DDEBUG`. A related convention uses `VERBOSE_DEBUG` to add `dev_vdbg()` messages to the ones already enabled by `DEBUG`.

Chapter 14: Allocating memory

The kernel provides the following general purpose memory allocators: `kmalloc()`, `kzalloc()`, `kcalloc()`, and `vmalloc()`. Please refer to the API documentation for further information about them.

The preferred form for passing a size of a struct is the following:

```
p = kmalloc(sizeof(*p), ...);
```

The alternative form where struct name is spelled out hurts readability and introduces an opportunity for a bug when the pointer variable type is changed but the corresponding `sizeof` that is passed to a memory allocator is not.

Casting the return value which is a void pointer is redundant. The conversion from void pointer to any other pointer type is guaranteed by the C programming language.

Chapter 15: The inline disease

There appears to be a common misperception that `gcc` has a magic "make me faster" speedup option called "inline". While the use of inlines can be appropriate (for example as a means of replacing macros, see Chapter 12), it very often is not. Abundant use of the `inline` keyword leads to a much bigger kernel, which in turn slows the system as a whole down, due to a bigger icache footprint for the CPU and simply because there is less memory available for the pagecache. Just think about it; a pagecache miss causes a disk seek, which easily takes 5 milliseconds. There are a LOT of cpu cycles that can go into these 5 milliseconds.

A reasonable rule of thumb is to not put `inline` at functions that have more than 3 lines of code in them. An exception to this rule are the cases where a parameter is known to be a compiletime constant, and as a result of this constantness you *know* the compiler will be able to optimize most of your function away at compile time. For a good example of this later case, see the `kmalloc()` inline function.

Often people argue that adding `inline` to functions that are static and used only once is always a win since there is no space tradeoff. While this is technically correct, `gcc` is capable of inlining these automatically without help, and the maintenance issue of removing the `inline` when a second user appears outweighs the potential value of the hint that tells `gcc` to do something it would have done anyway.

Chapter 16: Function return values and names

Functions can return values of many different kinds, and one of the most common is a value indicating whether the function succeeded or failed. Such a value can be represented as an error-code integer (-Exxx = failure, 0 = success) or a "succeeded" boolean (0 = failure, non-zero = success).

Mixing up these two sorts of representations is a fertile source of difficult-to-find bugs. If the C language included a strong distinction between integers and booleans then the compiler would find these mistakes for us... but it doesn't. To help prevent such bugs, always follow this convention:

```
If the name of a function is an action or an imperative command,
the function should return an error-code integer. If the name
is a predicate, the function should return a "succeeded" boolean.
```

For example, "add work" is a command, and the `add_work()` function returns 0 for success or `-EBUSY` for failure. In the same way, "PCI device present" is a predicate, and the `pci_dev_present()` function returns 1 if it succeeds in finding a matching device or 0 if it doesn't.

All EXPORTed functions must respect this convention, and so should all public functions. Private (static) functions need not, but it is recommended that they do.

Functions whose return value is the actual result of a computation, rather than an indication of whether the computation succeeded, are not subject to this rule. Generally they indicate failure by returning some out-of-range result. Typical examples would be functions that return pointers; they use `NULL` or the `ERR_PTR` mechanism to report failure.

Chapter 17: Don't re-invent the kernel macros

The header file `include/linux/kernel.h` contains a number of macros that you should use, rather than explicitly coding some variant of them yourself. For example, if you need to calculate the length of an array, take advantage of the macro

```
#define ARRAY_SIZE(x) (sizeof(x) / sizeof((x)[0]))
```

Similarly, if you need to calculate the size of some structure member, use

```
#define FIELD_SIZEOF(t, f) (sizeof(((t*)0)->f))
```

There are also `min()` and `max()` macros that do strict type checking if you need them. Feel free to peruse that header file to see what else is already defined that you shouldn't reproduce in your code.

Chapter 18: Editor modelines and other cruft

Some editors can interpret configuration information embedded in source files, indicated with special markers. For example, emacs interprets lines marked like this:

```
 -*- mode: c -*-
```

Or like this:

```
/*
Local Variables:
compile-command: "gcc -DMAGIC_DEBUG_FLAG foo.c"
End:
*/
```

Vim interprets markers that look like this:

```
/* vim:set sw=8 noet */
```

Do not include any of these in source files. People have their own personal editor configurations, and your source files should not override them. This includes markers for indentation and mode configuration. People may use their own custom mode, or may have some other magic method for making indentation work correctly.

Appendix I: References

- [The C Programming Language, Second Edition](#) by Brian W. Kernighan and Dennis M. Ritchie. Prentice Hall, Inc., 1988. ISBN 0-13-110362-8 (paperback), 0-13-110370-9 (hardback).
- [The Practice of Programming](#) by Brian W. Kernighan and Rob Pike. Addison-Wesley, Inc., 1999. ISBN 0-201-61586-X.
- [GNU manuals](#) - where in compliance with K&R and this text - for `cpp`, `gcc`, `gcc internals` and `indent`
- [WG14 International standardization workgroup for the programming language C](#)
- [Kernel CodingStyle](#), by greg@kroah.com at OLS 2002

7.8 Соглашения по разработке на языке Python

7.8.1 Introduction

This document gives coding conventions for the Python code comprising the standard library in the main Python distribution. Please see the companion informational PEP describing style guidelines for the C code in the C implementation of Python¹.

This document and PEP 257 (Docstring Conventions) were adapted from Guido's original Python Style Guide essay, with some additions from Barry's style guide².

7.8.2 A Foolish Consistency is the Hobgoblin of Little Minds

One of Guido's key insights is that code is read much more often than it is written. The guidelines provided here are intended to improve the readability of code and make it consistent across the wide spectrum of Python code. As PEP 20 says, "Readability counts".

A style guide is about consistency. Consistency with this style guide is important. Consistency within a project is more important. Consistency within one module or function is the most important.

But most importantly: know when to be inconsistent – sometimes the style guide just doesn't apply. When in doubt, use your best judgment. Look at other examples and decide what looks best. And don't hesitate to ask!

Two good reasons to break a particular rule:

1. When applying the rule would make the code less readable, even for someone who is used to reading code that follows the rules.
2. To be consistent with surrounding code that also breaks it (maybe for historic reasons) – although this is also an opportunity to clean up someone else's mess (in true XP style).

¹ [PEP 7, Style Guide for C Code, van Rossum](#)

² [Barry's GNU Mailman style guide](#)

7.8.3 Code lay-out

Indentation

Use 4 spaces per indentation level.

For really old code that you don't want to mess up, you can continue to use 8-space tabs.

Continuation lines should align wrapped elements either vertically using Python's implicit line joining inside parentheses, brackets and braces, or using a hanging indent. When using a hanging indent the following considerations should be applied; there should be no arguments on the first line and further indentation should be used to clearly distinguish itself as a continuation line.

Yes:

```
# Aligned with opening delimiter
foo = long_function_name(var_one, var_two,
                        var_three, var_four)

# More indentation included to distinguish this from the rest.
def long_function_name(
    var_one, var_two, var_three,
    var_four):
    print(var_one)
```

No:

```
# Arguments on first line forbidden when not using vertical alignment
foo = long_function_name(var_one, var_two,
                        var_three, var_four)

# Further indentation required as indentation is not distinguishable
def long_function_name(
    var_one, var_two, var_three,
    var_four):
    print(var_one)
```

Optional:

```
# Extra indentation is not necessary.
foo = long_function_name(
    var_one, var_two,
    var_three, var_four)
```

The closing brace/bracket/parenthesis on multi-line constructs may either line up under the first non-whitespace character of the last line of list, as in:

```
my_list = [
    1, 2, 3,
    4, 5, 6,
]
result = some_function_that_takes_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f',
)
```

or it may be lined up under the first character of the line that starts the multi-line construct, as in:

```

my_list = [
    1, 2, 3,
    4, 5, 6,
]
result = some_function_that_takes_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f',
)

```

Tabs or Spaces?

Never mix tabs and spaces.

The most popular way of indenting Python is with spaces only. The second-most popular way is with tabs only. Code indented with a mixture of tabs and spaces should be converted to using spaces exclusively. When invoking the Python command line interpreter with the `-t` option, it issues warnings about code that illegally mixes tabs and spaces. When using `-tt` these warnings become errors. These options are highly recommended!

For new projects, spaces-only are strongly recommended over tabs. Most editors have features that make this easy to do.

Maximum Line Length

Limit all lines to a maximum of 79 characters.

There are still many devices around that are limited to 80 character lines; plus, limiting windows to 80 characters makes it possible to have several windows side-by-side. The default wrapping on such devices disrupts the visual structure of the code, making it more difficult to understand. Therefore, please limit all lines to a maximum of 79 characters. For flowing long blocks of text (docstrings or comments), limiting the length to 72 characters is recommended.

The preferred way of wrapping long lines is by using Python's implied line continuation inside parentheses, brackets and braces. Long lines can be broken over multiple lines by wrapping expressions in parentheses. These should be used in preference to using a backslash for line continuation.

Backslashes may still be appropriate at times. For example, long, multiple `with`-statements cannot use implicit continuation, so backslashes are acceptable:

```

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())

```

Another such case is with `assert` statements.

Make sure to indent the continued line appropriately. The preferred place to break around a binary operator is *after* the operator, not before it. Some examples:

```

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)
```

Blank Lines

Separate top-level function and class definitions with two blank lines.

Method definitions inside a class are separated by a single blank line.

Extra blank lines may be used (sparingly) to separate groups of related functions. Blank lines may be omitted between a bunch of related one-liners (e.g. a set of dummy implementations).

Use blank lines in functions, sparingly, to indicate logical sections.

Python accepts the control-L (i.e. `^L`) form feed character as whitespace; Many tools treat these characters as page separators, so you may use them to separate pages of related sections of your file. Note, some editors and web-based code viewers may not recognize control-L as a form feed and will show another glyph in its place.

Encodings (PEP 263)

Code in the core Python distribution should always use the ASCII or Latin-1 encoding (a.k.a. ISO-8859-1). For Python 3.0 and beyond, UTF-8 is preferred over Latin-1, see PEP 3120.

Files using ASCII should not have a coding cookie. Latin-1 (or UTF-8) should only be used when a comment or docstring needs to mention an author name that requires Latin-1; otherwise, using `\x`, `\u` or `\U` escapes is the preferred way to include non-ASCII data in string literals.

For Python 3.0 and beyond, the following policy is prescribed for the standard library (see PEP 3131): All identifiers in the Python standard library MUST use ASCII-only identifiers, and SHOULD use English words wherever feasible (in many cases, abbreviations and technical terms are used which aren't English). In addition, string literals and comments must also be in ASCII. The only exceptions are (a) test cases testing the non-ASCII features, and (b) names of authors. Authors whose names are not based on the latin alphabet MUST provide a latin transliteration of their names.

Open source projects with a global audience are encouraged to adopt a similar policy.

Imports

- Imports should usually be on separate lines, e.g.:

```
Yes: import os
     import sys

No:  import sys, os
```

It's okay to say this though:

```
from subprocess import Popen, PIPE
```

- Imports are always put at the top of the file, just after any module comments and docstrings, and before module globals and constants.

Imports should be grouped in the following order:

1. standard library imports
2. related third party imports
3. local application/library specific imports

You should put a blank line between each group of imports.

Put any relevant `__all__` specification after the imports.

- Relative imports for intra-package imports are highly discouraged. Always use the absolute package path for all imports. Even now that PEP 328 is fully implemented in Python 2.5, its style of explicit relative imports is actively discouraged; absolute imports are more portable and usually more readable.
- When importing a class from a class-containing module, it's usually okay to spell this:

```
from myclass import MyClass
from foo.bar.yourclass import YourClass
```

If this spelling causes local name clashes, then spell them

```
import myclass
import foo.bar.yourclass
```

and use `"myclass.MyClass"` and `"foo.bar.yourclass.YourClass"`.

7.8.4 Whitespace in Expressions and Statements

Pet Peeves

Avoid extraneous whitespace in the following situations:

- Immediately inside parentheses, brackets or braces.

```
Yes: spam(ham[1], {eggs: 2})
No:  spam( ham[ 1 ], { eggs: 2 } )
```

- Immediately before a comma, semicolon, or colon:

```
Yes: if x == 4: print x, y; x, y = y, x
No:  if x == 4 : print x , y ; x , y = y , x
```

- Immediately before the open parenthesis that starts the argument list of a function call:

```
Yes: spam(1)
No:  spam (1)
```

- Immediately before the open parenthesis that starts an indexing or slicing:

```
Yes: dict['key'] = list[index]
No:  dict ['key'] = list [index]
```

- More than one space around an assignment (or other) operator to align it with another.

Yes:

```
x = 1
y = 2
long_variable = 3
```

No:

```
x          = 1
y          = 2
long_variable = 3
```

Other Recommendations

- Always surround these binary operators with a single space on either side: assignment (=), augmented assignment (+=, -= etc.), comparisons (==, <, >, !=, <>, <=, >=, in, not in, is, is not), Booleans (and, or, not).
- If operators with different priorities are used, consider adding whitespace around the operators with the lowest priority(ies). Use your own judgement; however, never use more than one space, and always have the same amount of whitespace on both sides of a binary operator.

Yes:

```
i = i + 1
submitted += 1
x = x*2 - 1
hypot2 = x*x + y*y
c = (a+b) * (a-b)
```

No:

```
i=i+1
submitted +=1
x = x * 2 - 1
hypot2 = x * x + y * y
c = (a + b) * (a - b)
```

- Don't use spaces around the = sign when used to indicate a keyword argument or a default parameter value.

Yes:

```
def complex(real, imag=0.0):
    return magic(r=real, i=imag)
```

No:

```
def complex(real, imag = 0.0):
    return magic(r = real, i = imag)
```

- Compound statements (multiple statements on the same line) are generally discouraged.

Yes:

```
if foo == 'blah':
    do_blah_thing()
do_one()
```

```
do_two()
do_three()
```

Rather not:

```
if foo == 'blah': do_blah_thing()
do_one(); do_two(); do_three()
```

- While sometimes it's okay to put an if/for/while with a small body on the same line, never do this for multi-clause statements. Also avoid folding such long lines!

Rather not:

```
if foo == 'blah': do_blah_thing()
for x in lst: total += x
while t < 10: t = delay()
```

Definitely not:

```
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()
```

7.8.5 Comments

Comments that contradict the code are worse than no comments. Always make a priority of keeping the comments up-to-date when the code changes!

Comments should be complete sentences. If a comment is a phrase or sentence, its first word should be capitalized, unless it is an identifier that begins with a lower case letter (never alter the case of identifiers!).

If a comment is short, the period at the end can be omitted. Block comments generally consist of one or more paragraphs built out of complete sentences, and each sentence should end in a period.

You should use two spaces after a sentence-ending period.

When writing English, Strunk and White apply.

Python coders from non-English speaking countries: please write your comments in English, unless you are 120% sure that the code will never be read by people who don't speak your language.

Block Comments

Block comments generally apply to some (or all) code that follows them, and are indented to the same level as that code. Each line of a block comment starts with a # and a single space (unless it is indented text inside the comment).

Paragraphs inside a block comment are separated by a line containing a single #.

Inline Comments

Use inline comments sparingly.

An inline comment is a comment on the same line as a statement. Inline comments should be separated by at least two spaces from the statement. They should start with a `#` and a single space.

Inline comments are unnecessary and in fact distracting if they state the obvious. Don't do this:

```
x = x + 1           # Increment x
```

But sometimes, this is useful:

```
x = x + 1           # Compensate for border
```

Documentation Strings

Conventions for writing good documentation strings (a.k.a. "docstrings") are immortalized in PEP 257.

- Write docstrings for all public modules, functions, classes, and methods. Docstrings are not necessary for non-public methods, but you should have a comment that describes what the method does. This comment should appear after the `def` line.
- PEP 257 describes good docstring conventions. Note that most importantly, the `"` that ends a multiline docstring should be on a line by itself, and preferably preceded by a blank line, e.g.:

```
"""Return a foobang

Optional plotz says to frobnicate the bizbaz first.

"""
```

- For one liner docstrings, it's okay to keep the closing `"` on the same line.

7.8.6 Version Bookkeeping

If you have to have Subversion, CVS, or RCS crud in your source file, do it as follows.

```
__version__ = "$Revision$"
# $Source$
```

These lines should be included after the module's docstring, before any other code, separated by a blank line above and below.

7.8.7 Naming Conventions

The naming conventions of Python's library are a bit of a mess, so we'll never get this completely consistent – nevertheless, here are the currently recommended naming standards. New modules and packages (including third party frameworks) should be written to these standards, but where an existing library has a different style, internal consistency is preferred.

Descriptive: Naming Styles

There are a lot of different naming styles. It helps to be able to recognize what naming style is being used, independently from what they are used for.

The following naming styles are commonly distinguished:

- `b` (single lowercase letter)
- `B` (single uppercase letter)
- `lowercase`
- `lower_case_with_underscores`
- `UPPERCASE`
- `UPPER_CASE_WITH_UNDERScores`
- `CapitalizedWords` (or `CapWords`, or `CamelCase` – so named because of the bumpy look of its letters³). This is also sometimes known as `StudyCaps`.

Note: When using abbreviations in `CapWords`, capitalize all the letters of the abbreviation. Thus `HTTPServerError` is better than `HttpServerError`.

- `mixedCase` (differs from `CapitalizedWords` by initial lowercase character!)
- `Capitalized_Words_With_Underscores` (ugly!)

There's also the style of using a short unique prefix to group related names together. This is not used much in Python, but it is mentioned for completeness. For example, the `os.stat()` function returns a tuple whose items traditionally have names like `st_mode`, `st_size`, `st_mtime` and so on. (This is done to emphasize the correspondence with the fields of the POSIX system call struct, which helps programmers familiar with that.)

The X11 library uses a leading X for all its public functions. In Python, this style is generally deemed unnecessary because attribute and method names are prefixed with an object, and function names are prefixed with a module name.

In addition, the following special forms using leading or trailing underscores are recognized (these can generally be combined with any case convention):

- `_single_leading_underscore`: weak "internal use" indicator. E.g. `from M import *` does not import objects whose name starts with an underscore.
- `single_trailing_underscore_`: used by convention to avoid conflicts with Python keyword, e.g.

```
Tkinter.Toplevel(master, class_='ClassName')
```

- `__double_leading_underscore`: when naming a class attribute, invokes name mangling (inside class `FooBar`, `__boo` becomes `_FooBar__boo`; see below).
- `__double_leading_and_trailing_underscore__`: "magic" objects or attributes that live in user-controlled namespaces. E.g. `__init__`, `__import__` or `__file__`. Never invent such names; only use them as documented.

Prescriptive: Naming Conventions

³ [CamelCase Wikipedia page](#)

Names to Avoid

Never use the characters 'l' (lowercase letter el), 'O' (uppercase letter oh), or 'I' (uppercase letter eye) as single character variable names.

In some fonts, these characters are indistinguishable from the numerals one and zero. When tempted to use 'l', use 'L' instead.

Package and Module Names

Modules should have short, all-lowercase names. Underscores can be used in the module name if it improves readability. Python packages should also have short, all-lowercase names, although the use of underscores is discouraged.

Since module names are mapped to file names, and some file systems are case insensitive and truncate long names, it is important that module names be chosen to be fairly short – this won't be a problem on Unix, but it may be a problem when the code is transported to older Mac or Windows versions, or DOS.

When an extension module written in C or C++ has an accompanying Python module that provides a higher level (e.g. more object oriented) interface, the C/C++ module has a leading underscore (e.g. `_socket`).

Class Names

Almost without exception, class names use the CapWords convention. Classes for internal use have a leading underscore in addition.

Exception Names

Because exceptions should be classes, the class naming convention applies here. However, you should use the suffix "Error" on your exception names (if the exception actually is an error).

Global Variable Names

(Let's hope that these variables are meant for use inside one module only.) The conventions are about the same as those for functions.

Modules that are designed for use via `from M import *` should use the `__all__` mechanism to prevent exporting globals, or use the older convention of prefixing such globals with an underscore (which you might want to do to indicate these globals are "module non-public").

Function Names

Function names should be lowercase, with words separated by underscores as necessary to improve readability. `mixedCase` is allowed only in contexts where that's already the prevailing style (e.g. `threading.py`), to retain backwards compatibility.

Function and method arguments

Always use `self` for the first argument to instance methods.

Always use `cls` for the first argument to class methods.

If a function argument's name clashes with a reserved keyword, it is generally better to append a single trailing underscore rather than use an abbreviation or spelling corruption. Thus `class_` is better than `class`. (Perhaps better is to avoid such clashes by using a synonym.)

Method Names and Instance Variables

Use the function naming rules: lowercase with words separated by underscores as necessary to improve readability.

Use one leading underscore only for non-public methods and instance variables.

To avoid name clashes with subclasses, use two leading underscores to invoke Python's name mangling rules.

Python mangles these names with the class name: if class `Foo` has an attribute named `__a`, it cannot be accessed by `Foo.__a`. (An insistent user could still gain access by calling `Foo._Foo__a`.) Generally, double leading underscores should be used only to avoid name conflicts with attributes in classes designed to be subclassed.

Note: there is some controversy about the use of `__` names (see below).

Constants

Constants are usually defined on a module level and written in all capital letters with underscores separating words. Examples include `MAX_OVERFLOW` and `TOTAL`.

Designing for inheritance

Always decide whether a class's methods and instance variables (collectively: "attributes") should be public or non-public. If in doubt, choose non-public; it's easier to make it public later than to make a public attribute non-public.

Public attributes are those that you expect unrelated clients of your class to use, with your commitment to avoid backward incompatible changes. Non-public attributes are those that are not intended to be used by third parties; you make no guarantees that non-public attributes won't change or even be removed.

We don't use the term "private" here, since no attribute is really private in Python (without a generally unnecessary amount of work).

Another category of attributes are those that are part of the "subclass API" (often called "protected" in other languages). Some classes are designed to be inherited from, either to extend or modify aspects of the class's behavior. When designing such a class, take care to make explicit decisions about which attributes are public, which are part of the subclass API, and which are truly only to be used by your base class.

With this in mind, here are the Pythonic guidelines:

- Public attributes should have no leading underscores.
- If your public attribute name collides with a reserved keyword, append a single trailing underscore to your attribute name. This is preferable to an abbreviation or corrupted spelling. (However, notwithstanding this rule, 'cls' is the preferred spelling for any variable or argument which is known to be a class, especially the first argument to a class method.)

Note 1: See the argument name recommendation above for class methods.

- For simple public data attributes, it is best to expose just the attribute name, without complicated accessor/mutator methods. Keep in mind that Python provides an easy path to future enhancement, should you find that a simple data attribute needs to grow functional behavior. In that case, use properties to hide functional implementation behind simple data attribute access syntax.

Note 1: Properties only work on new-style classes.

Note 2: Try to keep the functional behavior side-effect free, although side-effects such as caching are generally fine.

Note 3: Avoid using properties for computationally expensive operations; the attribute notation makes the caller believe that access is (relatively) cheap.

- If your class is intended to be subclassed, and you have attributes that you do not want subclasses to use, consider naming them with double leading underscores and no trailing underscores. This invokes Python's name mangling algorithm, where the name of the class is mangled into the attribute name. This helps avoid attribute name collisions should subclasses inadvertently contain attributes with the same name.

Note 1: Note that only the simple class name is used in the mangled name, so if a subclass chooses both the same class name and attribute name, you can still get name collisions.

Note 2: Name mangling can make certain uses, such as debugging and `__getattr__()`, less convenient. However the name mangling algorithm is well documented and easy to perform manually.

Note 3: Not everyone likes name mangling. Try to balance the need to avoid accidental name clashes with potential use by advanced callers.

7.8.8 References

7.8.9 Copyright

Author:

- Guido van Rossum <guido@python.org>
- Barry Warsaw <barry@python.org>

7.9 Работа с релизами

7.9.1 How to make a minor release

```
$ git tag -a 1.4.4 -m "Next minor in 1.4 series"
$ vim CMakeLists.txt # edit CPACK_PACKAGE_VERSION_PATCH
$ git push --tags
```

Update the Web site in doc/www

Update all issues, upload the ChangeLog based on `git log` output. The ChangeLog must only include items which are mentioned as issues on github. If anything significant is there, which is not mentioned, something went wrong in release planning and the release should be held up until this is cleared.

Click 'Release milestone'. Create a milestone for the next minor release. Alert the driver to target bugs and blueprints to the new milestone.

- `genindex`

Примечание: Одностраничная документация доступна по [этой ссылке](#).

b

box.cfg, ??
box.error, ??
box.index, ??
box.info, ??
box.schema, ??
box.schema.role, ??
box.schema.user, ??
box.session, ??
box.slab, ??
box.space, ??
box.tuple, ??

C

capi_error, ??
clock, [169](#)
console, [171](#)
crypto.cipher, ??
crypto.digest, ??
csv, [174](#)

d

digest, [177](#)

f

fiber, ??
fio, [191](#)

j

jit, [199](#)
json, [201](#)

l

log, [203](#)

m

msgpack, [204](#)
my_box.index, ??
my_fiber, ??

n

net_box, ??

O

os, [210](#)

p

pickle, [212](#)

S

schema, [270](#)
shard, [246](#)
socket, [214](#)
strict, [223](#)

t

tap, [224](#)
tarantool, [227](#)

u

uuid, [228](#)

y

yaml, [230](#)