Tarantool

Release 2.1.1

Aug 08, 2019

Contents

CHAPTER 1

An application server together with a database manager

Tarantool is a Lua application server integrated with a database management system. It has a "fiber" model which means that many Tarantool applications can run simultaneously on a single thread, while each instance of the Tarantool server itself can run multiple threads for input-output and background maintenance. It incorporates the LuaJIT – "Just In Time" – Lua compiler, Lua libraries for most common applications, and the Tarantool Database Server which is an established NoSQL DBMS. Thus Tarantool serves all the purposes that have made node.js and Twisted popular, plus it supports data persistence.

The code is free. The open-source license is BSD license. The supported platforms are GNU/Linux, Mac OS and FreeBSD.

Tarantool's creator and biggest user is Mail.Ru, the largest internet company in Russia, with 30 million users, 25 million emails per day, and a web site whose Alexa global rank is in the top 40 worldwide. Tarantool services Mail.Ru's hottest data, such as the session data of online users, the properties of online applications, the caches of the underlying data, the distribution and sharding algorithms, and much more. Outside Mail.Ru the software is used by a growing number of projects in online gaming, digital marketing, and social media industries. Although Mail.Ru is the sponsor for product development, the roadmap and the bugs database and the development process are fully open. The software incorporates patches from dozens of community contributors. The Tarantool community writes and maintains most of the drivers for programming languages. The greater Lua community has hundreds of useful packages most of which can become Tarantool extensions.

Users can create, modify and drop Lua functions at runtime. Or they can define Lua programs that are loaded during startup for triggers, background tasks, and interacting with networked peers. Unlike popular application development frameworks based on a "reactor" pattern, networking in server-side Lua is sequential, yet very efficient, as it is built on top of the cooperative multitasking environment that Tarantool itself uses.

One of the built-in Lua packages provides an API for the Database Management System. Thus some developers see Tarantool as a DBMS with a popular stored procedure language, while others see it as a Lua interpreter, while still others see it as a replacement for many components of multi-tier Web applications. Performance can be a few hundred thousand transactions per second on a laptop, scalable upwards or outwards to server farms.

CHAPTER 2

Database features

Tarantool can run without it, but "The Box" – the DBMS server – is a strong distinguishing feature.

The database API allows for permanently storing Lua objects, managing object collections, creating or dropping secondary keys, making changes atomically, configuring and monitoring replication, performing controlled fail-over, and executing Lua code triggered by database events. Remote database instances are accessible transparently via a remote-procedure-invocation API.

Tarantool's DBMS server uses the storage engine concept, where different sets of algorithms and data structures can be used for different situations. Two storage engines are built-in: an in-memory engine which has all the data and indexes in RAM, and a two-level B-tree engine for data sets whose size is 10 to 1000 times the amount of available RAM. All storage engines in Tarantool support transactions and replication by using a common write ahead log (WAL). This ensures consistency and crash safety of the persistent state. Changes are not considered complete until the WAL is written. The logging subsystem supports group commit.

Tarantool's in-memory storage engine (memtx) keeps all the data in random-access memory, and therefore has very low read latency. It also keeps persistent copies of the data in non-volatile storage, such as disk, when users request "snapshots". If an instance of the server stops and the random-access memory is lost, then restarts, it reads the latest snapshot and then replays the transactions that are in the log – therefore no data is lost.

Tarantool's in-memory engine is lock-free in typical situations. Instead of the operating system's concurrency primitives, such as mutexes, Tarantool uses cooperative multitasking to handle thousands of connections simultaneously. There is a fixed number of independent execution threads. The threads do not share state. Instead they exchange data using low-overhead message queues. While this approach limits the number of cores that the instance will use, it removes competition for the memory bus and ensures peak scalability of memory access and network throughput. CPU utilization of a typical highly-loaded Tarantool instance is under 10%. Searches are possible via secondary index keys as well as primary keys.

Tarantool's disk-based storage engine is a fusion of ideas from modern filesystems, log-structured merge trees and classical B-trees. All data is organized into ranges. Each range is represented by a file on disk. Range size is a configuration option and normally is around 64MB. Each range is a collection of pages, serving different purposes. Pages in a fully merged range contain non-overlapping ranges of keys. A range can be partially merged if there were a lot of changes in its key range recently. In that case some pages represent new keys and values in the range. The disk-based storage engine is append only: new data never overwrites old data. The disk-based storage engine is named vinyl.

Tarantool supports multi-part index keys. The possible index types are HASH, TREE, BITSET, and RTREE.

Tarantool supports asynchronous replication, locally or to remote hosts. The replication architecture can be master-master, that is, many nodes may both handle the loads and receive what others have handled, for the same data sets.

Tarantool supports basic SQL structures and persistence for SQL operations (with acceptable limitations). All tables and triggers created in SQL are available after server restart.

CHAPTER 3

User's Guide

3.1 Preface

Welcome to Tarantool! This is the User's Guide. We recommend reading it first, and consulting Reference materials for more detail afterwards, if needed.

3.1.1 How to read the documentation

To get started, you can install and launch Tarantool using a Docker container, a binary package, or the online Tarantool server at http://try.tarantool.org. Either way, as the first tryout, you can follow the introductory exercises from Chapter 2 "Getting started". If you want more hands-on experience, proceed to Tutorials after you are through with Chapter 2.

Chapter 3 "Database" is about using Tarantool as a NoSQL DBMS, whereas Chapter 4 "Application server" is about using Tarantool as an application server.

Chapter 5 "Server administration" and Chapter 6 "Replication" are primarily for administrators.

Chapter 7 "Connectors" is strictly for users who are connecting from a different language such as C or Perl or Python — other users will find no immediate need for this chapter.

Chapter 8 "FAQ" gives answers to some frequently asked questions about Tarantool.

For experienced users, there are also Reference materials, a Contributor's Guide and an extensive set of comments in the source code.

3.1.2 Getting in touch with the Tarantool community

Please report bugs or make feature requests at http://github.com/tarantool/tarantool/issues.

You can contact developers directly in telegram or in a Tarantool discussion group (English or Russian).

3.1.3 Conventions used in this manual

Square brackets [and] enclose optional syntax.

Two dots in a row .. mean the preceding tokens may be repeated.

A vertical bar | means the preceding and following tokens are mutually exclusive alternatives.

3.2 Getting started

In this chapter, we explain how to install Tarantool, how to start it, and how to create a simple database.

This chapter contains the following sections:

3.2.1 Using a Docker image

For trial and test purposes, we recommend using official Tarantool images for Docker. An official image contains a particular Tarantool version (1.6, 1.10 or 2.1) and all popular external modules for Tarantool. Everything is already installed and configured in Linux. These images are the easiest way to install and use Tarantool.

Note: If you're new to Docker, we recommend going over this tutorial before proceeding with this chapter.

Launching a container

If you don't have Docker installed, please follow the official installation guide for your OS.

To start a fully functional Tarantool instance, run a container with minimal options:

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

This command runs a new container named 'mytarantool'. Docker starts it from an official image named 'tarantool/tarantool:2', with Tarantool version 2.1 and all external modules already installed.

Tarantool will be accepting incoming connections on localhost:3301. You may start using it as a key-value storage right away.

Tarantool persists data inside the container. To make your test data available after you stop the container, this command also mounts the host's directory /data/dir/on/host (you need to specify here an absolute path to an existing local directory) in the container's directory /var/lib/tarantool (by convention, Tarantool in a container uses this directory to persist data). So, all changes made in the mounted directory on the container's side are applied to the host's disk.

Tarantool's database module in the container is already configured and started. You needn't do it manually, unless you use Tarantool as an application server and run it with an application.

Attaching to Tarantool

To attach to Tarantool that runs inside the container, say:

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

This command:

- Instructs Tarantool to open an interactive console port for incoming connections.
- Attaches to the Tarantool server inside the container under 'admin' user via a standard Unix socket.

Tarantool displays a prompt:

```
tarantool.sock>
```

Now you can enter requests on the command line.

Note: On production machines, Tarantool's interactive mode is for system administration only. But we use it for most examples in this manual, because the interactive mode is convenient for learning.

Creating a database

While you're attached to the console, let's create a simple test database.

First, create the first space (named 'tester'):

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

Format the created space by specifying field names and types:

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

Create the first index (named 'primary'):

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

This is a primary index based on the 'id' field of each tuple.

Insert three tuples (our name for "records") into the space:

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

To select a tuple using the 'primary' index, say:

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

The terminal screen now looks like this:

```
tarantool.sock> s = box.schema.space.create('tester')
---
...
tarantool.sock> s:format({
              > {name = 'id', type = 'unsigned'},
              > {name = 'band_name', type = 'string'},
              > {name = 'year', type = 'unsigned'}
              > })
---
...
tarantool.sock> s:create_index('primary', {
              > type = 'hash',
              > parts = {'id'}
              > })
---
- unique: true
  parts:
  - type: unsigned
    is_nullable: false
    fieldno: 1
  id: 0
  space_id: 512
  name: primary
  type: HASH
...
tarantool.sock> s:insert{1, 'Roxette', 1986}
---
- [1, 'Roxette', 1986]
...
tarantool.sock> s:insert{2, 'Scorpions', 2015}
---
- [2, 'Scorpions', 2015]
...
tarantool.sock> s:insert{3, 'Ace of Base', 1993}
---
- [3, 'Ace of Base', 1993]
...
tarantool.sock> s:select{3}
---
- - [3, 'Ace of Base', 1993]
...
```

To add a secondary index based on the 'band_name' field, say:

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

To select tuples using the 'secondary' index, say:

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

Stopping a container

When the testing is over, stop the container politely:

```
$ docker stop mytarantool
```

This was a temporary container, and its disk/memory data were flushed when you stopped it. But since you mounted a data directory from the host in the container, Tarantool's data files were persisted to the host's disk. Now if you start a new container and mount that data directory in it, Tarantool will recover all data from disk and continue working with the persisted data.

3.2.2 Using a binary package

For production purposes, we recommend official binary packages. You can choose from two Tarantool versions: 1.10 (stable) or 2.1 (beta). An automatic build system creates, tests and publishes packages for every push into a corresponding branch (1.10 or 2.1) at Tarantool's GitHub repository.

To download and install the package that's appropriate for your OS, start a shell (terminal) and enter the command-line instructions provided for your OS at Tarantool's download page.

Starting Tarantool

To start a Tarantool instance, say this:

```
$ # if you downloaded a binary with apt-get or yum, say this:
$ /usr/bin/tarantool
$ # if you downloaded and untarred a binary tarball to ~/tarantool, say this:
$ ~/tarantool/bin/tarantool
```

Tarantool starts in the interactive mode and displays a prompt:

```
tarantool>
```

Now you can enter requests on the command line.

Note: On production machines, Tarantool's interactive mode is for system administration only. But we use it for most examples in this manual, because the interactive mode is convenient for learning.

Creating a database

Here is how to create a simple test database after installation.

1. To let Tarantool store data in a separate place, create a new directory dedicated for tests:

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

   You can delete the directory when the tests are over.

2. Check if the default port the database instance will listen to is vacant.

   Depending on the release, during installation Tarantool may start a demonstrative global example.lua instance that listens to the 3301 port by default. The example.lua file showcases basic configuration and can be found in the /etc/tarantool/instances.enabled or /etc/tarantool/instances.available directories.

However, we encourage you to perform the instance startup manually, so you can learn.

Make sure the default port is vacant:

1. To check if the demonstrative instance is running, say:

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

2. If it does, kill the corresponding process. In this example:

```
$ kill 6851
```

3. To start Tarantool's database module and make the instance accept TCP requests on port 3301, say:

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

4. Create the first space (named `'tester'`):

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

5. Format the created space by specifying field names and types:

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

6. Create the first index (named `'primary'`):

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

This is a primary index based on the id field of each tuple.

7. Insert three tuples (our name for "records") into the space:

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

8. To select a tuple using the `'primary'` index, say:

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

The terminal screen now looks like this:

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

(continues on next page)

(continued from previous page)

```
---
...
tarantool> s:create_index('primary', {
         > type = 'hash',
         > parts = {'id'}
         > })
---
- unique: true
  parts:
  - type: unsigned
    is_nullable: false
    fieldno: 1
  id: 0
  space_id: 512
  name: primary
  type: HASH
...
tarantool> s:insert{1, 'Roxette', 1986}
---
- [1, 'Roxette', 1986]
...
tarantool> s:insert{2, 'Scorpions', 2015}
---
- [2, 'Scorpions', 2015]
...
tarantool> s:insert{3, 'Ace of Base', 1993}
---
- [3, 'Ace of Base', 1993]
...
tarantool> s:select{3}
---
- - [3, 'Ace of Base', 1993]
...
```

9. To add a secondary index based on the 'band_name' field, say:

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

10. To select tuples using the 'secondary' index, say:

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

11. Now, to prepare for the example in the next section, try this:

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

Connecting remotely

In the request box.cfg{listen = 3301} that we made earlier, the listen value can be any form of a URI (uniform resource identifier). In this case, it's just a local port: port 3301. You can send requests to the

listen URI via:

(1) telnet,

(2) a connector,

(3) another instance of Tarantool (using the console module), or

(4) tarantoolctl utility.

Let's try (4).

Switch to another terminal. On Linux, for example, this means starting another instance of a Bash shell. You can switch to any working directory in the new terminal, not necessarily to ~/tarantool_sandbox.

Start the tarantoolctl utility:

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

This means "use tarantoolctl connect to connect to the Tarantool instance that's listening on localhost:3301".

Try this request:

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

This means "send a request to that Tarantool instance, and display the result". The result in this case is one of the tuples that was inserted earlier. Your terminal screen should now look like this:

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

You can repeat box.space...:insert{} and box.space...:select{} indefinitely, on either Tarantool instance.

When the testing is over:

- To drop the space: s:drop()
- To stop tarantoolctl: Ctrl+C or Ctrl+D
- To stop Tarantool (an alternative): the standard Lua function os.exit()
- To stop Tarantool (from another terminal): sudo pkill -f tarantool
- To destroy the test: rm -r ~/tarantool_sandbox

3.3 Database

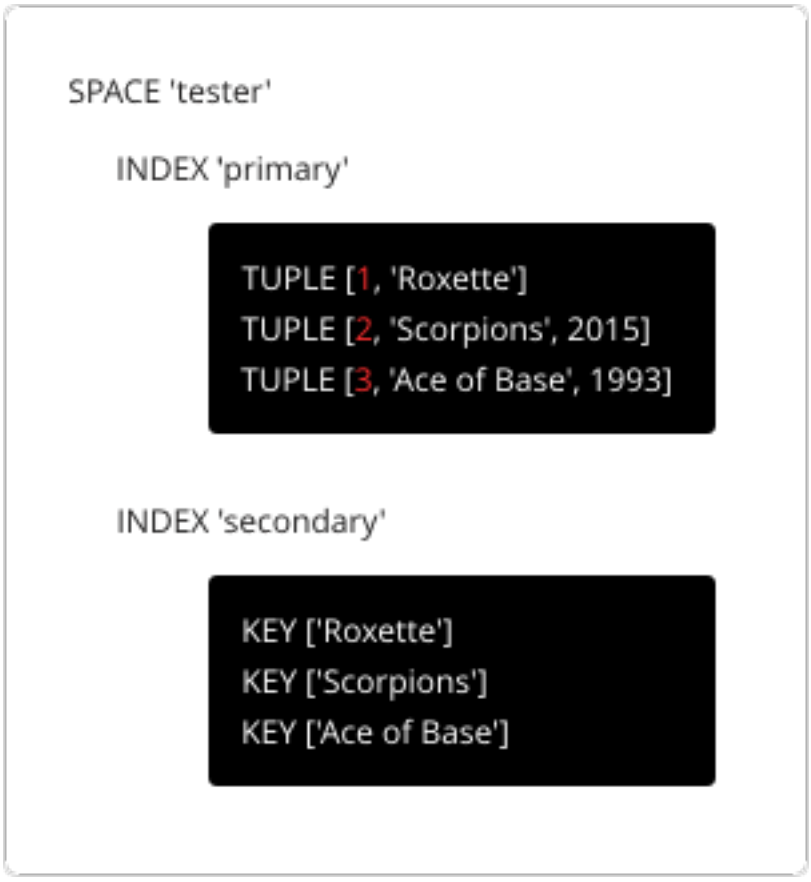
In this chapter, we introduce the basic concepts of working with Tarantool as a database manager.

This chapter contains the following sections:

3.3.1 Data model

This section describes how Tarantool stores values and what operations with data it supports.

If you tried to create a database as suggested in our "Getting started" exercises, then your test database now looks like this:



Space

A space – 'tester' in our example – is a container.

When Tarantool is being used to store data, there is always at least one space. Each space has a unique name specified by the user. Besides, each space has a unique numeric identifier which can be specified by the user, but usually is assigned automatically by Tarantool. Finally, a space always has an engine: memtx (default) – in-memory engine, fast but limited in size, or vinyl – on-disk engine for huge data sets.

A space is a container for tuples. To be functional, it needs to have a primary index. It can also have secondary indexes.

Tuple

A tuple plays the same role as a "row" or a "record", and the components of a tuple (which we call "fields") play the same role as a "row column" or "record field", except that:

- fields can be composite structures, such as arrays or maps, and
- fields don't need to have names.

Any given tuple may have any number of fields, and the fields may be of different types. The identifier of a field is the field's number, base 1 (in Lua and other 1-based languages) or base 0 (in PHP or C/C++). For example, "1" or "0" can be used in some contexts to refer to the first field of a tuple.

Tuples in Tarantool are stored as MsgPack arrays.

When Tarantool returns a tuple value in console, it uses the YAML format, for example: [3, 'Ace of Base', 1993].

Index

An index is a group of key values and pointers.

As with spaces, you should specify the index name, and let Tarantool come up with a unique numeric identifier ("index id").

An index always has a type. The default index type is 'TREE'. TREE indexes are provided by all Tarantool engines, can index unique and non-unique values, support partial key searches, comparisons and ordered results. Additionally, memtx engine supports HASH, RTREE and BITSET indexes.

An index may be multi-part, that is, you can declare that an index key value is composed of two or more fields in the tuple, in any order. For example, for an ordinary TREE index, the maximum number of parts is 255.

An index may be unique, that is, you can declare that it would be illegal to have the same key value twice.

The first index defined on a space is called the primary key index, and it must be unique. All other indexes are called secondary indexes, and they may be non-unique.

An index definition may include identifiers of tuple fields and their expected types (see allowed indexed field types below).

In our example, we first defined the primary index (named 'primary') based on field #1 of each tuple:

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

The effect is that, for all tuples in space 'tester', field #1 must exist and must contain an unsigned integer. The index type is 'hash', so values in field #1 must be unique, because keys in HASH indexes are unique.

After that, we defined a secondary index (named 'secondary') based on field #2 of each tuple:

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

The effect is that, for all tuples in space 'tester', field #2 must exist and must contain a string. The index type is 'tree', so values in field #2 must not be unique, because keys in TREE indexes may be non-unique.

Note: Space definitions and index definitions are stored permanently in Tarantool's system spaces _space and _index (for details, see reference on box.space submodule).

You can add, drop, or alter the definitions at runtime, with some restrictions. See syntax details in reference on box module.

Data types

Tarantool is both a database and an application server. Hence a developer often deals with two type sets: the programming language types (e.g. Lua) and the types of the Tarantool storage format (MsgPack).

Lua vs MsgPack

Scalar / compound	MsgPack type	Lua type	Example value
scalar	nil	"nil"	msgpack.NULL
scalar	boolean	"boolean"	true
scalar	string	"string"	'A B C'
scalar	integer	"number"	12345
scalar	double	"number"	1.2345
scalar	bin	"cdata"	[!!binary 3t7e]
compound	map	"table" (with string keys)	{'a': 5, 'b': 6}
compound	array	"table" (with integer keys)	[1, 2, 3, 4, 5]
compound	array	tuple ("cdata")	[12345, 'A B C']

In Lua, a nil type has only one possible value, also called nil (displayed as null on Tarantool's command line, since the output is in the YAML format). Nils may be compared to values of any types with == (is-equal) or ~= (is-not-equal), but other operations will not work. Nils may not be used in Lua tables; the workaround is to use msgpack.NULL

A boolean is either true or false.

A string is a variable-length sequence of bytes, usually represented with alphanumeric characters inside single quotes. In both Lua and MsgPack, strings are treated as binary data, with no attempts to determine a string's character set or to perform any string conversion – unless there is an optional collation. So, usually, string sorting and comparison are done byte-by-byte, without any special collation rules applied. (Example: numbers are ordered by their point on the number line, so 2345 is greater than 500; meanwhile, strings are ordered by the encoding of the first byte, then the encoding of the second byte, and so on, so '2345' is less than '500'.)

In Lua, a number is double-precision floating-point, but Tarantool allows both integer and floating-point values. Tarantool will try to store a Lua number as floating-point if the value contains a decimal point or is very large (greater than 100 trillion = 1e14), otherwise Tarantool will store it as an integer. To ensure that even very large numbers are stored as integers, use the tonumber64 function, or the LL (Long Long) suffix, or the ULL (Unsigned Long Long) suffix. Here are examples of numbers using regular notation, exponential notation, the ULL suffix and the tonumber64 function: -55, -2.7e+20, 100000000000000ULL, tonumber64('18446744073709551615').

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

Lua tables with string keys are stored as MsgPack maps; Lua tables with integer keys starting with 1 – as MsgPack arrays. Nils may not be used in Lua tables; the workaround is to use msgpack.NULL

A tuple is a light reference to a MsgPack array stored in the database. It is a special type (cdata) to avoid conversion to a Lua table on retrieval. A few functions may return tables with multiple tuples. For more tuple examples, see box.tuple.

Note: Tarantool uses the MsgPack format for database storage, which is variable-length. So, for example, the smallest number requires only one byte, but the largest number requires nine bytes.

Examples of insert requests with different data types:

```
tarantool> box.space.K:insert{1,nil,true,'A B C',12345,1.2345}
---
```

(continues on next page)

(continued from previous page)

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

Indexed field types

Indexes restrict values which Tarantool's MsgPack may contain. This is why, for example, 'unsigned' is a separate indexed field type, compared to 'integer' data type in MsgPack: they both store 'integer' values, but an 'unsigned' index contains only non-negative integer values and an 'integer' index contains all integer values.

Here's how Tarantool indexed field types correspond to MsgPack data types.

Indexed field type	MsgPack data type (and possible values)	Index type	Examples
unsigned (may also be called 'uint' or 'num', but 'num' is deprecated)	integer (integer between 0 and 18446744073709551615, i.e. about 18 quintillion)	TREE, BITSET or HASH	123456
integer (may also be called 'int')	integer (integer between -9223372036854775808 and 18446744073709551615)	TREE or HASH	-2^63
number	integer (integer between -9223372036854775808 and 18446744073709551615) double (single-precision floating point number or double-precision floating point number)	TREE or HASH	1.234 -44 1.447e+44
string (may also be called 'str')	string (any set of octets, up to the maximum length)	TREE, BITSET or HASH	'A B C' '\65 \66 \67'
varbinary	bin (any set of octets, up to the maximum length)	TREE or HASH	'\65 \66 \67'
boolean	bool (true or false)	TREE or HASH	true
array	array (list of numbers representing points in a geometric figure)	RTREE	{10, 11} {3, 5, 9, 10}
scalar	null bool (true or false) integer (integer between -9223372036854775808 and 18446744073709551615) double (single-precision floating point number or double-precision floating point number) string (any set of octets) varbinary (any set of octets) Note: When there is a mix of types, the key order is: null, then booleans, then numbers, then strings, then varbinary.	TREE or HASH	msgpack.NULL true -1 1.234 '' 'py'

Collations

By default, when Tarantool compares strings, it uses what we call a "binary" collation. The only consideration here is the numeric value of each byte in the string. Therefore, if the string is encoded with ASCII or UTF-8, then `'A' < 'B' < 'a'`, because the encoding of 'A' (what used to be called the "ASCII value") is 65, the encoding of 'B' is 66, and the encoding of 'a' is 98. Binary collation is best if you prefer fast deterministic simple maintenance and searching with Tarantool indexes.

But if you want the ordering that you see in phone books and dictionaries, then you need Tarantool's optional collations, such as unicode and unicode_ci, which allow for `'a' < 'A' < 'B'` and `'a' = 'A' < 'B'`

respectively.

The unicode and unicode_ci optional collations use the ordering according to the Default Unicode Collation Element Table (DUCET) and the rules described in Unicode® Technical Standard #10 Unicode Collation Algorithm (UTS #10 UCA). The only difference between the two collations is about weights:

- unicode collation observes L1 and L2 and L3 weights (strength = 'tertiary'),
- unicode_ci collation observes only L1 weights (strength = 'primary'), so for example 'a' = 'A' = 'á' = 'Á'.

As an example, take some Russian words:

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

. . . and show the difference in ordering and selecting by index:

- with unicode collation:

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

- with unicode_ci collation:

```
tarantool> box.space.T:create_index('I', {parts = {{1,'str', collation='unicode_ci'}}})
...
tarantool> box.space.S.index.I:select()
---
- - ['ЕЛЕ']
  - ['елейный']
  - ['ёлка']
  - ['еловый']
  - ['елозить']
  - ['Ёлочка']
  - ['ёлочный']
```

(continues on next page)

(continued from previous page)

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

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

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

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

Sequences

A sequence is a generator of ordered integer values.

As with spaces and indexes, you should specify the sequence name, and let Tarantool come up with a unique numeric identifier ("sequence id").

As well, you can specify several options when creating a new sequence. The options determine what value will be generated whenever the sequence is used.

Options for box.schema.sequence.create()

Option name	Type and meaning	Default	Examples
start	Integer. The value to generate the first time a sequence is used	1	start=0
min	Integer. Values smaller than this cannot be generated	1	min=-1000
max	Integer. Values larger than this cannot be generated	9223372036854775807	max=0
cycle	Boolean. Whether to start again when values cannot be generated	false	cycle=true
cache	Integer. The number of values to store in a cache	0	cache=0
step	Integer. What to add to the previous generated value, when generating a new value	1	step=-1
if_not_exists	Boolean. If this is true and a sequence with this name exists already, ignore other options and use the existing values	false	if_not_exists=true

Once a sequence exists, it can be altered, dropped, reset, forced to generate the next value, or associated with an index.

For an initial example, we generate a sequence named 'S'.

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

The result shows that the new sequence has all default values, except for the two that were specified, min and start.

Then we get the next value, with the next() function.

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

The result is the same as the start value. If we called next() again, we would get 6 (because the previous value plus the step value is 6), and so on.

Then we create a new table, and say that its primary key may be generated from the sequence.

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

Then we insert a tuple, without specifying a value for the primary key.

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

The result is a new tuple where the first field has a value of 6. This arrangement, where the system automatically generates the values for a primary key, is sometimes called "auto-incrementing" or "identity".

For syntax and implementation details, see the reference for box.schema.sequence.

Persistence

In Tarantool, updates to the database are recorded in the so-called write ahead log (WAL) files. This ensures data persistence. When a power outage occurs or the Tarantool instance is killed incidentally, the in-memory database is lost. In this situation, WAL files are used to restore the data. Namely, Tarantool reads the WAL files and redoes the requests (this is called the "recovery process"). You can change the timing of the WAL writer, or turn it off, by setting wal_mode.

Tarantool also maintains a set of snapshot files. These files contain an on-disk copy of the entire data set for a given moment. Instead of reading every WAL file since the databases were created, the recovery process can load the latest snapshot file and then read only those WAL files that were produced after the snapshot file was made. After checkpointing, old WAL files can be removed to free up space.

To force immediate creation of a snapshot file, you can use Tarantool's box.snapshot() request. To enable automatic creation of snapshot files, you can use Tarantool's checkpoint daemon. The checkpoint daemon sets intervals for forced checkpoints. It makes sure that the states of both memtx and vinyl storage engines are synchronized and saved to disk, and automatically removes old WAL files.

Snapshot files can be created even if there is no WAL file.

Note: The memtx engine makes only regular checkpoints with the interval set in checkpoint daemon configuration.

The vinyl engine runs checkpointing in the background at all times.

See the Internals section for more details about the WAL writer and the recovery process.

Operations

Data operations

The basic data operations supported in Tarantool are:

- five data-manipulation operations (INSERT, UPDATE, UPSERT, DELETE, REPLACE), and
- one data-retrieval operation (SELECT).

All of them are implemented as functions in box.space submodule.

Examples:

- INSERT: Add a new tuple to space 'tester'.

  The first field, field[1], will be 999 (MsgPack type is integer).

  The second field, field[2], will be 'Taranto' (MsgPack type is string).

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

- UPDATE: Update the tuple, changing field field[2].

  The clause "{999}", which has the value to look up in the index of the tuple's primary-key field, is mandatory, because update() requests must always have a clause that specifies a unique key, which in this case is field[1].

  The clause "{{'=', 2, 'Tarantino'}}" specifies that assignment will happen to field[2] with the new value.

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

- UPSERT: Upsert the tuple, changing field field[2] again.

  The syntax of upsert() is similar to the syntax of update(). However, the execution logic of these two requests is different. UPSERT is either UPDATE or INSERT, depending on the database's state. Also, UPSERT execution is postponed until after transaction commit, so, unlike update(), upsert() doesn't return data back.

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

- REPLACE: Replace the tuple, adding a new field.

  This is also possible with the update() request, but the update() request is usually more complicated.

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

- SELECT: Retrieve the tuple.

  The clause "{999}" is still mandatory, although it does not have to mention the primary key.

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

- DELETE: Delete the tuple.

  In this example, we identify the primary-key field.

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

Summarizing the examples:

- Functions insert and replace accept a tuple (where a primary key comes as part of the tuple).
- Function upsert accepts a tuple (where a primary key comes as part of the tuple), and also the update operations to execute.
- Function delete accepts a full key of any unique index (primary or secondary).
- Function update accepts a full key of any unique index (primary or secondary), and also the operations to execute.
- Function select accepts any key: primary/secondary, unique/non-unique, full/partial.

See reference on box.space for more details on using data operations.

Note: Besides Lua, you can use Perl, PHP, Python or other programming language connectors. The client server protocol is open and documented. See this annotated BNF.

Index operations

Index operations are automatic: if a data-manipulation request changes a tuple, then it also changes the index keys defined for the tuple.

The simple index-creation operation that we've illustrated before is:

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

This creates a unique TREE index on the first field of all tuples (often called "Field#1"), which is assumed to be numeric.

The simple SELECT request that we've illustrated before is:

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

This looks for a single tuple via the first index. Since the first index is always unique, the maximum number of returned tuples will be: one.

The following SELECT variations exist:

1. The search can use comparisons other than equality.

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

The comparison operators are LT, LE, EQ, REQ, GE, GT (for "less than", "less than or equal", "equal", "reversed equal", "greater than or equal", "greater than" respectively). Comparisons make sense if and only if the index type is 'TREE'.

This type of search may return more than one tuple; if so, the tuples will be in descending order by key when the comparison operator is LT or LE or REQ, otherwise in ascending order.

2. The search can use a secondary index.

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

   For a primary-key search, it is optional to specify an index name. For a secondary-key search, it is mandatory.

3. The search may be for some or all key parts.

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

4. The search may be for all fields, using a table for the value:

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

   or the search can be for one field, using a table or a scalar:

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

   In the second case, the result will be two tuples: {1, 'A'} and {1, 'B'}.

   You can specify even zero fields, causing all three tuples to be returned. (Notice that partial key searches are available only in TREE indexes.)

Examples

- BITSET example:

  ```
  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'})
  ```

  The result will be:

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

because (7 AND 2) is not equal to 0, and (3 AND 2) is not equal to 0.

- RTREE example:

```
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'})
```

The result will be:

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

because a rectangle whose corners are at coordinates 4,7,5,9 is entirely within a rectangle whose corners are at coordinates 3,5,9,10.

Additionally, there exist index iterator operations. They can only be used with code in Lua and C/C++. Index iterators are for traversing indexes one key at a time, taking advantage of features that are specific to an index type, for example evaluating Boolean expressions when traversing BITSET indexes, or going in descending order when traversing TREE indexes.

See also other index operations like alter() and drop() in reference for box.index submodule.

Complexity factors

In reference for box.space and box.index submodules, there are notes about which complexity factors might affect the resource usage of each function.

Complexity factor	Effect
Index size	The number of index keys is the same as the number of tuples in the data set. For a TREE index, if there are more keys, then the lookup time will be greater, although of course the effect is not linear. For a HASH index, if there are more keys, then there is more RAM used, but the number of low-level steps tends to remain constant.
Index type	Typically, a HASH index is faster than a TREE index if the number of tuples in the space is greater than one.
Number of indexes accessed	Ordinarily, only one index is accessed to retrieve one tuple. But to update the tuple, there must be N accesses if the space has N different indexes. Note re storage engine: Vinyl optimizes away such accesses if secondary index fields are unchanged by the update. So, this complexity factor applies only to memtx, since it always makes a full-tuple copy on every update.
Number of tuples accessed	A few requests, for example SELECT, can retrieve multiple tuples. This factor is usually less important than the others.
WAL settings	The important setting for the write-ahead log is wal_mode. If the setting causes no writing or delayed writing, this factor is unimportant. If the setting causes every data-change request to wait for writing to finish on a slow device, this factor is more important than all the others.

3.3.2 Transaction control

Transactions in Tarantool occur in fibers on a single thread. That is why Tarantool has a guarantee of execution atomicity. That requires emphasis.

Threads, fibers and yields

How does Tarantool process a basic operation? As an example, let's take this query:

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

This is equivalent to the following SQL statement for a table that stores primary keys in field[1]:

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

This query will be processed with three operating system threads:

1. If we issue the query on a remote client, then the network thread on the server side receives the query, parses the statement and changes it to a server executable message which has already been checked, and which the server instance can understand without parsing everything again.
2. The network thread ships this message to the instance's transaction processor thread using a lock-free message bus. Lua programs execute directly in the transaction processor thread, and do not require parsing and preparation.

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

3. The transaction processor thread sends a message to the write-ahead logging (WAL) thread to commit the transaction. When done, the WAL thread replies with a COMMIT or ROLLBACK result, which is returned to the client.

Notice that there is only one transaction processor thread in Tarantool. Some people are used to the idea that there can be multiple threads operating on the database, with (say) thread #1 reading row #x, while thread #2 writes row #y. With Tarantool, no such thing ever happens. Only the transaction processor thread can access the database, and there is only one transaction processor thread for each Tarantool instance.

Like any other Tarantool thread, the transaction processor thread can handle many fibers. A fiber is a set of computer instructions that may contain "yield" signals. The transaction processor thread will execute all computer instructions until a yield, then switch to execute the instructions of a different fiber. Thus (say) the thread reads row #x for the sake of fiber #1, then writes row #y for the sake of fiber #2.

Yields must happen, otherwise the transaction processor thread would stick permanently on the same fiber. There are two types of yields:

- implicit yields: every data-change operation or network-access causes an implicit yield, and every statement that goes through the Tarantool client causes an implicit yield.
- explicit yields: in a Lua function, you can (and should) add "yield" statements to prevent hogging. This is called cooperative multitasking.

Cooperative multitasking

Cooperative multitasking means: 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": a transaction commit, an operating system call, or an explicit "yield" request. 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.

Transactions

In the absence of transactions, any function that contains yield points may see changes in the database state caused by fibers that preempt. Multi-statement transactions exist to provide isolation: each transaction sees a consistent database state and commits all its changes atomically. At commit time, a yield happens and all transaction changes are written to the write ahead log in a single batch. Or, if needed, transaction changes can be rolled back – completely or to a specific savepoint.

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

The cooperative scheduler ensures that, in absence of yields, a multi-statement transaction is not preempted and hence is never aborted. Therefore, understanding yields is essential to writing abort-free code.

Note: You can't mix storage engines in a transaction today.

Implicit yields

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

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

Many functions in modules fio, net_box, console and socket (the "os" and "network" requests) yield.

Example #1

- Engine = memtx select() insert() has one yield, at the end of insertion, caused by implicit commit; select() has nothing to write to the WAL and so does not yield.
- Engine = vinyl select() insert() has between one and three yields, since select() may yield if the data is not in cache, insert() may yield waiting for available memory, and there is an implicit yield at commit.
- The sequence begin() insert() insert() commit() yields only at commit if the engine is memtx, and can yield up to 3 times if the engine is vinyl.

Example #2

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

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

If wal_mode = 'none', then implicit yielding at commit time does not take place, 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 (in memtx), if it is inside a function or Lua program being executed on the server instance, but causes yielding (in both memtx and vinyl) 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.

3.3.3 Access control

Understanding security details is primarily an issue for administrators. However, ordinary users should at least skim this section to get 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 system space, where usernames and password-hashes are stored.
- There are functions for saying that certain users are allowed to do certain things ("privileges").
- There is a _priv system 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").

Details follow.

Users

There is a current user for any program working with Tarantool, local or remote. If a remote connection is using a binary port, the current user, by default, is 'guest'. If the connection is using an admin-console port, the current user is 'admin'. When executing a Lua initialization script, the current user is also 'admin'.

The current user name can be found with box.session.user().

The current user can be changed:

- For a binary port connection – with the AUTH protocol command, supported by most clients;
- For an admin-console connection and in a Lua initialization script – with box.session.su;
- For a binary-port connection invoking a stored function with the CALL command – if the SETUID property is enabled for the function, Tarantool temporarily replaces the current user with the function's creator, with all the creator's privileges, during function execution.

Passwords

Each user (except 'guest') may have a password. The password is any alphanumeric string.

Tarantool passwords are stored in the _user system space with a cryptographic hash function so that, if the password is 'x', the stored hash-password is a long string like 'lL3OvhkIPOKh+Vn9Avlkx69M/Ck='. When a client connects to a Tarantool instance, the instance sends a random salt value which the client must mix with the hashed-password before sending to the instance. 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.

Note: For more details of the password hashing algorithm (e.g. for the purpose of writing a new client application), read the scramble.h header file.

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. Administrators should also advise users to choose long unobvious passwords, but it is ultimately up to the users to choose or change their own passwords.

There are two functions for managing passwords in Tarantool: box.schema.user.password() for changing a user's password and box.schema.user.passwd() for getting a hash-password.

Owners and privileges

Tarantool has one database. It may be called "box.schema" or "universe". The database contains database objects, including spaces, indexes, users, roles, sequences, and functions.

The owner of a database object is the user who created it. The owner of the database itself, and the owner of objects that are created initially – the system spaces and the default users – is 'admin'.

Owners automatically have privileges for what they create. They can share these privileges with other users or with roles, using box.schema.user.grant requests. The following privileges can be granted:

- 'read', e.g. allow select from a space
- 'write', e.g. allow update on a space
- 'execute', e.g. allow call of a function, or (less commonly) allow use of a role
- 'create', e.g. allow box.schema.space.create (access to certain system spaces is also necessary)
- 'alter', e.g. allow box.space.x.index.y:alter (access to certain system spaces is also necessary)
- 'drop', e.g. allow box.sequence.x:drop (currently this can be granted but has no effect)
- 'usage', e.g. whether any action is allowable regardless of other privileges (sometimes revoking 'usage' is a convenient way to block a user temporarily without dropping the user)
- 'session', e.g. whether the user can 'connect'.

To create objects, users need the 'create' privilege and at least 'read' and 'write' privileges on the system space with a similar name (for example, on the _space if the user needs to create spaces).

To access objects, users need an appropriate privilege on the object (for example, the 'execute' privilege on function F if the users need to execute function F). See below some examples for granting specific privileges that a grantor – that is, 'admin' or the object creator – can make.

To drop an object, users must be the object's creator or be 'admin'. As the owner of the entire database, 'admin' can drop any object including other users.

To grant privileges to a user, the object owner says grant(). To revoke privileges from a user, the object owner says revoke(). In either case, there are up to five parameters:

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

- user-name is the user (or role) that will receive or lose the privilege;

- privilege is any of 'read', 'write', 'execute', 'create', 'alter', 'drop', 'usage', or 'session' (or a comma-separated list);
- object-type is any of 'space', 'index', 'sequence', 'function', role-name, or 'universe';
- object-name is what the privilege is for (omitted if object-type is 'universe');
- options is a list inside braces for example {if_not_exists=true|false} (usually omitted because the default is acceptable).

Example for granting many privileges at once

In this example user 'admin' grants many privileges on many objects to user 'U', with a single request.

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

Examples for granting privileges for specific operations

In these examples the object's creator grants precisely the minimal privileges necessary for particular operations, to user 'U'.

```
-- So that 'U' can create spaces:
  box.schema.user.grant('U','create','universe')
  box.schema.user.grant('U','write', 'space', '_schema')
  box.schema.user.grant('U','write', 'space', '_space')
-- So that 'U' can  create indexes (assuming 'U' created the space)
  box.schema.user.grant('U','read', 'space', '_space')
  box.schema.user.grant('U','read,write', 'space', '_index')
-- So that 'U' can  create indexes on space T (assuming 'U' did not create space T)
  box.schema.user.grant('U','create','space','T')
  box.schema.user.grant('U','read', 'space', '_space')
  box.schema.user.grant('U','write', 'space', '_index')
-- So that 'U' can  alter indexes on space T (assuming 'U' did not create the index)
  box.schema.user.grant('U','alter','space','T')
  box.schema.user.grant('U','read','space','_space')
  box.schema.user.grant('U','read','space','_index')
  box.schema.user.grant('U','read','space','_space_sequence')
  box.schema.user.grant('U','write','space','_index')
-- So that 'U' can create users or roles:
  box.schema.user.grant('U','create','universe')
  box.schema.user.grant('U','read,write', 'space', '_user')
  box.schema.user.grant('U','write','space', '_priv')
-- So that 'U' can create sequences:
  box.schema.user.grant('U','create','universe')
  box.schema.user.grant('U','read,write','space','_sequence')
-- So that 'U' can create functions:
  box.schema.user.grant('U','create','universe')
  box.schema.user.grant('U','read,write','space','_func')
-- So that 'U' can grant access on objects that 'U' created
  box.schema.user.grant('U','read','space','_user')
-- So that 'U' can select or get from a space named 'T'
  box.schema.user.grant('U','read','space','T')
-- So that 'U' can update or insert or delete or truncate a space named 'T'
  box.schema.user.grant('U','write','space','T')
-- So that 'U' can execute a function named 'F'
  box.schema.user.grant('U','execute','function','F')
-- So that 'U' can use the "S:next()" function with a sequence named S
  box.schema.user.grant('U','read,write','sequence','S')
-- So that 'U' can use the "S:set()" or "S:reset() function with a sequence named S
  box.schema.user.grant('U','write','sequence','S')
```

Example for creating users and objects then granting privileges

Here we create a Lua function that will be executed under the user id of its creator, even if called by another user.

First, we create two spaces ('u' and 'i') and grant a no-password user ('internal') full access to them. Then we define a function ('read_and_modify') and the no-password user becomes this function's creator. Finally, we grant another user ('public_user') access to execute Lua functions created by the no-password user.

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

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

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

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

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

Roles

A role is a container for privileges which can be granted to regular users. Instead of granting or revoking individual privileges, you can put all the privileges in a role and then grant or revoke the role.

Role information is stored in the _user space, but the third field in the tuple – the type field – is 'role' rather than 'user'.

An important feature in role management is that roles can be nested. For example, role R1 can be granted a privilege "role R2", so users with the role R1 will subsequently get all privileges from both roles R1 and R2. In other words, a user gets all the privileges that are granted to a user's roles, directly or indirectly.

There are actually two ways to grant or revoke a role: box.schema.user.grant-or-revoke(user-name-or-role-name,'execute', 'role',role-name...) or box.schema.user.grant-or-revoke(user-name-or-role-name,role-name...). The second way is preferable.

The 'usage' and 'session' privileges cannot be granted to roles.

Example

```
-- This example will work for a user with many privileges, such as 'admin'
-- or a user with the pre-defined 'super' role
-- Create space T with a primary index
box.schema.space.create('T')
box.space.T:create_index('primary', {})
-- Create user U1 so that later we can change the current user to 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 user U1 (order doesn't matter)
-- There are two ways to grant a role; here we use the shorter way
box.schema.role.grant('R1', 'R2')
box.schema.user.grant('U1', 'R1')
-- Grant read/write privileges for space T to role R2
-- (but not to role R1 and not to user U1)
box.schema.role.grant('R2', 'read,write', 'space', 'T')
-- Change the current user to user U1
box.session.su('U1')
-- An insertion to space T will now succeed because, due to nested roles,
-- user U1 has write privilege on space T
box.space.T:insert{1}
```

For more detail see box.schema.user.grant() and box.schema.role.grant() in the built-in modules reference.

Sessions and security

A session is the state of a connection to Tarantool. It contains:

- an integer id identifying the connection,
- the current user associated with the connection,
- text description of the connected peer, and
- session local state, such as Lua variables and functions.

In Tarantool, a single session can execute multiple concurrent transactions. Each transaction is identified by a unique integer id, which can be queried at start of the transaction using box.session.sync().

Note: To track all connects and disconnects, you can use connection and authentication triggers.

3.3.4 Triggers

Triggers, also known as callbacks, are functions which the server executes when certain events happen.

There are six types of triggers in Tarantool:

- connection triggers, which are executed when a session begins or ends,
- authentication triggers, which are executed during authentication,
- replace triggers, which are for database events,
- transaction triggers, which are executed during commit or rollback,
- server triggers, which are executed when the server starts or stops.

- member triggers, which are executed when a SWIM member is updated.

All triggers have the following characteristics:

- Triggers associate a function with an event. The request to "define a trigger" implies passing the trigger's function to one of the "on_event()" functions:
  - box.session.on_connect() and box.session.on_disconnect(), or
  - box.session.on_auth(), or
  - space_object:on_replace() and space_object:before_replace(), or
  - box.on_commit() and box.on_rollback(), or
  - box.ctl.on_schema_init() and box.ctl.on_shutdown(), or
  - swim_object:on_member_event().
- Triggers are defined only by the 'admin' user.
- Triggers are stored in the Tarantool instance's memory, not in the database. Therefore triggers disappear when the instance is shut down. To make them permanent, put function definitions and trigger settings into Tarantool's initialization script.
- Triggers 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 function.
- There can be multiple triggers for one event. In this case, triggers are executed in the reverse order that they were defined in. (Exception: member triggers are executed in the order that they appear in the member list.)
- Triggers must work within the event context. However, effects are undefined if a function contains requests which normally could not occur immediately after the event, but only before the return from the event. For example, putting os.exit() or box.rollback() in a trigger function would be bringing in requests outside the event context.
- Triggers are replaceable. The request to "redefine a trigger" implies passing a new trigger function and an old trigger function to one of the "on_event()" functions.
- The "on_event()" functions all have parameters which are function pointers, and they all return function pointers. Remember that a Lua function definition such as "function f() x = x + 1 end" is the same as "f = function () x = x + 1 end" – in both cases f gets a function pointer. And "trigger = box.session.on_connect(f)" is the same as "trigger = box.session.on_connect(function () x = x + 1 end)" – in both cases trigger gets the function pointer which was passed.

To get a list of triggers, you can use:

- box.session.on_connect() – with no arguments – to return a table of all connect-trigger functions;
- box.session.on_auth() to return all authentication-trigger functions;
- box.session.on_disconnect() to return all disconnect-trigger functions;
- space_object:on_replace() to return all replace-trigger functions made for on_replace().
- space_object:before_replace() to return all replace-trigger functions made for before_replace().
- box.ctl.on_shutdown() to return all shutdown-trigger functions made for on_shutdown().
- box.ctl.on_schema_init() to return all initialization-trigger functions made for on_schema_init().
- swim_object:on_member_event() to return all member triggers made for on_member_event().

Example

Here we log connect and disconnect events into Tarantool server log.

```
log = require('log')

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

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

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

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

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

3.3.5 Limitations

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 of up to 20 dimensions. 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 2,147,483,647 (box.schema.FIELD_MAX). The practical maximum is whatever is specified by the space's field_count member, or the maximal tuple length.

Number of bytes in a tuple

> The maximal number of bytes in a tuple is roughly equal to memtx_max_tuple_size or vinyl_max_tuple_size (with a metadata overhead of about 20 bytes per tuple, which is added on top of useful bytes). By default, the value of either memtx_max_tuple_size or vinyl_max_tuple_size is 1,048,576. To increase it, specify a larger value when starting the Tarantool instance. For example, box.cfg{memtx_max_tuple_size=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) but the practical maximum is around 65,000.

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 memtx_memory, 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

65000 (box.schema.NAME_MAX).

Number of replicas in a replica set

32 (vclock.VCLOCK_MAX).

3.3.6 Storage engines

A storage engine is a set of very-low-level routines which actually store and retrieve tuple values. Tarantool offers a choice of two storage engines:

- memtx (the in-memory storage engine) is the default and was the first to arrive.
- vinyl (the on-disk storage engine) is a working key-value engine and will especially appeal to users who like to see data go directly to disk, so that recovery time might be shorter and database size might be larger.

  On the other hand, vinyl lacks some functions and options that are available with memtx. Where that is the case, the relevant description in this manual contains a note beginning with the words "Note re storage engine".

Further in this section we discuss the details of storing data using the vinyl storage engine.

To specify that the engine should be vinyl, add the clause engine = 'vinyl' when creating a space, for example:

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

Differences between memtx and vinyl storage engines

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 (each query is usually run under 1 ms), and the memtx engine is justifiably the default for Tarantool, but on-disk engine such as vinyl is preferable when the database is larger than the available memory and adding more memory is not a realistic option.

Option	memtx	vinyl
Supported index type	TREE, HASH, RTREE or BITSET	TREE
Temporary spaces	Supported	Not supported
random() function	Supported	Not supported
alter() function	Supported	Supported starting from the 1.10.2 release (the primary index cannot be modified)
len() function	Returns the number of tuples in the space	Returns the maximum approximate number of tuples in the space
count() function	Takes a constant amount of time	Takes a variable amount of time depending on a state of a DB
delete() function	Returns the deleted tuple, if any	Always returns nil
yield	Does not yield on the select requests unless the transaction is commited to WAL	Yields on the select requests or on its equivalents: get() or pairs()

Storing data with vinyl

Tarantool is a transactional and persistent DBMS that maintains 100% of its data in RAM. The greatest advantages of in-memory databases are their speed and ease of use: they demonstrate consistently high performance, but you never need to tune them.

A few years ago we decided to extend the product by implementing a classical storage engine similar to those used by regular DBMSes: it uses RAM for caching, while the bulk of its data is stored on disk. We decided to make it possible to set a storage engine independently for each table in the database, which is the same way that MySQL approaches it, but we also wanted to support transactions from the very beginning.

The first question we needed to answer was whether to create our own storage engine or use an existing library. The open-source community offered a few viable solutions. The RocksDB library was the fastest growing open-source library and is currently one of the most prominent out there. There were also several lesser-known libraries to consider, such as WiredTiger, ForestDB, NestDB, and LMDB.

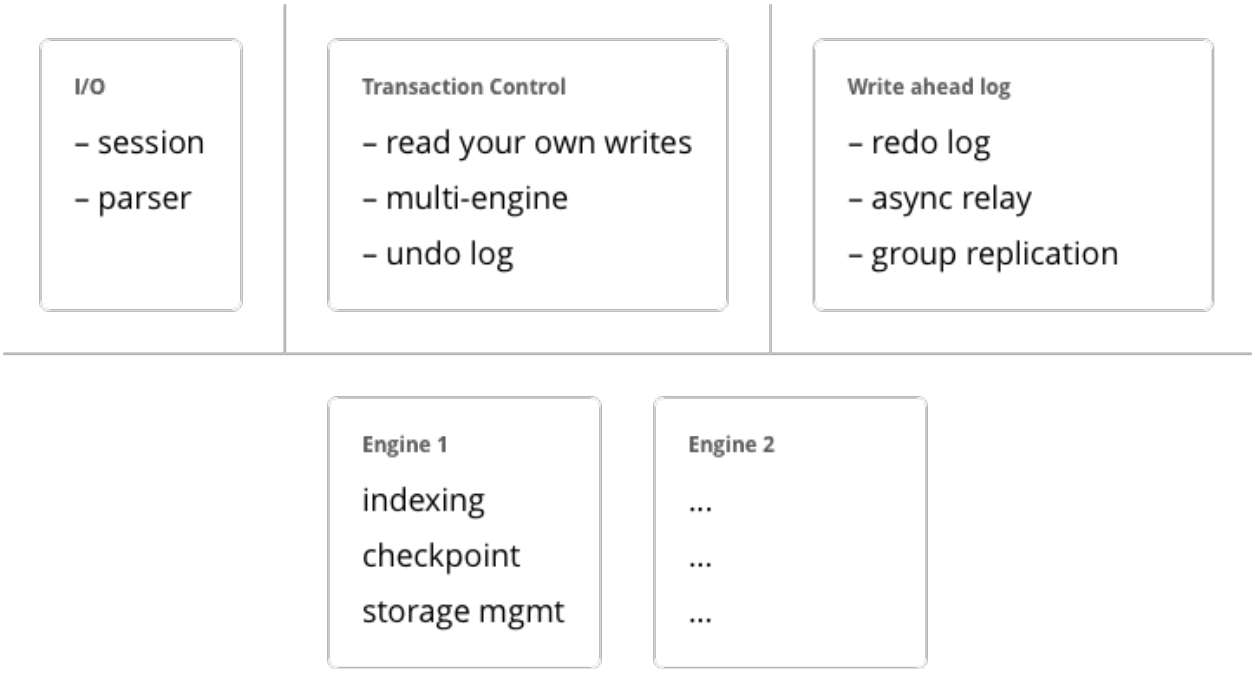
Nevertheless, after studying the source code of existing libraries and considering the pros and cons, we opted for our own storage engine. One reason is that the existing third-party libraries expected requests to come from multiple operating system threads and thus contained complex synchronization primitives for controlling parallel data access. If we had decided to embed one of these in Tarantool, we would have made our users bear the overhead of a multithreaded application without getting anything in return. The thing is, Tarantool has an actor-based architecture. The way it processes transactions in a dedicated thread allows it to do away with the unnecessary locks, interprocess communication, and other overhead that accounts for up to 80% of processor time in multithreaded DBMSes.

The Tarantool process consists of a fixed number of “actor” threads

If you design a database engine with cooperative multitasking in mind right from the start, it not only significantly speeds up the development process, but also allows the implementation of certain optimization tricks that would be too complex for multithreaded engines. In short, using a third-party solution wouldn’t have yielded the best result.

Algorithm

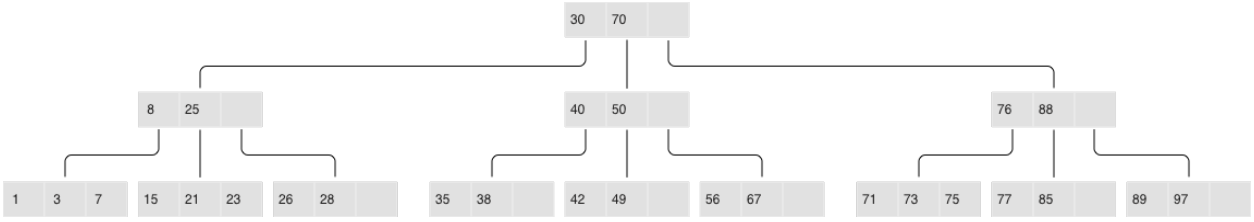
Once the idea of using an existing library was off the table, we needed to pick an architecture to build upon. There are two competing approaches to on-disk data storage: the older one relies on B-trees and



their variations; the newer one advocates the use of log-structured merge-trees, or "LSM" trees. MySQL, PostgreSQL, and Oracle use B-trees, while Cassandra, MongoDB, and CockroachDB have adopted LSM trees.

B-trees are considered better suited for reads and LSM trees—for writes. However, with SSDs becoming more widespread and the fact that SSDs have read throughput that's several times greater than write throughput, the advantages of LSM trees in most scenarios was more obvious to us.

Before dissecting LSM trees in Tarantool, let's take a look at how they work. To do that, we'll begin by analyzing a regular B-tree and the issues it faces. A B-tree is a balanced tree made up of blocks, which contain sorted lists of key- value pairs. (Topics such as filling and balancing a B-tree or splitting and merging blocks are outside of the scope of this article and can easily be found on Wikipedia). As a result, we get a container sorted by key, where the smallest element is stored in the leftmost node and the largest one in the rightmost node. Let's have a look at how insertions and searches in a B-tree happen.



Classical B-tree

If you need to find an element or check its membership, the search starts at the root, as usual. If the key is found in the root block, the search stops; otherwise, the search visits the rightmost block holding the largest element that's not larger than the key being searched (recall that elements at each level are sorted). If the first level yields no results, the search proceeds to the next level. Finally, the search ends up in one of the leaves and probably locates the needed key. Blocks are stored and read into RAM one by one, meaning the algorithm reads logB(N) blocks in a single search, where N is the number of elements in the B-tree. In the simplest case, writes are done similarly: the algorithm finds the block that holds the necessary element and updates (inserts) its value.

To better understand the data structure, let's consider a practical example: say we have a B-tree with 100,000,000 nodes, a block size of 4096 bytes, and an element size of 100 bytes. Thus each block will hold up to 40 elements (all overhead considered), and the B-tree will consist of around 2,570,000 blocks and 5 levels: the first four will have a size of 256 Mb, while the last one will grow up to 10 Gb. Obviously, any modern computer will be able to store all of the levels except the last one in filesystem cache, so read requests will require just a single I/O operation.

But if we change our perspective —B-trees don't look so good anymore. Suppose we need to update a single element. Since working with B-trees involves reading and writing whole blocks, we would have to read in one whole block, change our 100 bytes out of 4096, and then write the whole updated block to disk. In other words,we were forced to write 40 times more data than we actually modified!

If you take into account the fact that an SSD block has a size of 64 Kb+ and not every modification changes a whole element, the extra disk workload can be greater still.

Authors of specialized literature and blogs dedicated to on-disk data storage have coined two terms for these phenomena: extra reads are referred to as "read amplification" and writes as "write amplification".

The amplification factor (multiplication coefficient) is calculated as the ratio of the size of actual read (or written) data to the size of data needed (or actually changed). In our B-tree example, the amplification factor would be around 40 for both reads and writes.

The huge number of extra I/O operations associated with updating data is one of the main issues addressed by LSM trees. Let's see how they work.

The key difference between LSM trees and regular B-trees is that LSM trees don't just store data (keys and values), but also data operations: insertions and deletions.

LSM tree:

- Stores statements, not values:
  - REPLACE
  - DELETE
  - UPSERT
- Every statement is marked by LSN Append-only files, garbage is collected after a checkpoint
- Transactional log of all filesystem changes: vylog

For example, an element corresponding to an insertion operation has, apart from a key and a value, an extra byte with an operation code ("REPLACE" in the image above). An element representing the deletion operation contains a key (since storing a value is unnecessary) and the corresponding operation code—"DELETE". Also, each LSM tree element has a log sequence number (LSN), which is the value of a monotonically increasing sequence that uniquely identifies each operation. The whole tree is first ordered by key in ascending order, and then, within a single key scope, by LSN in descending order.

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

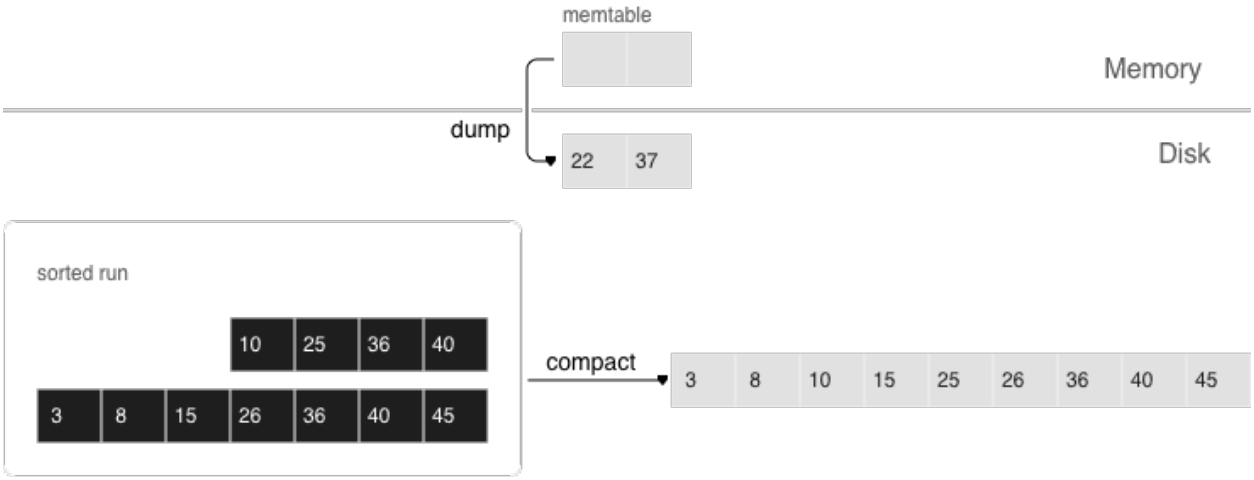
A single level of an LSM tree

Filling an LSM tree

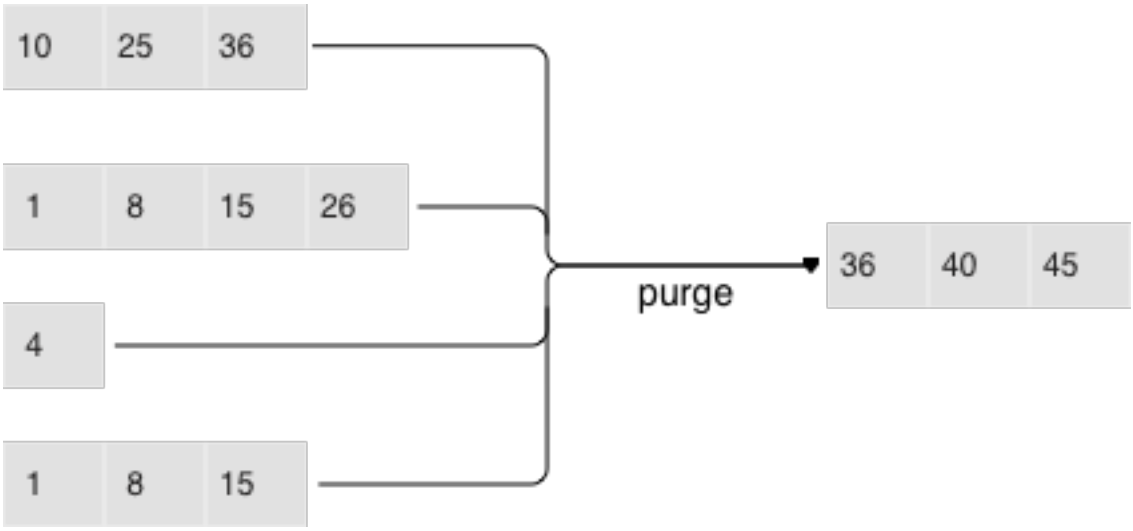
Unlike a B-tree, which is stored completely on disk and can be partly cached in RAM, when using an LSM tree, memory is explicitly separated from disk right from the start. The issue of volatile memory and data persistence is beyond the scope of the storage algorithm and can be solved in various ways—for example, by logging changes.

The part of an LSM tree that's stored in RAM is called L0 (level zero). The size of RAM is limited, so L0 is allocated a fixed amount of memory. For example, in Tarantool, the L0 size is controlled by the vinyl_memory parameter. Initially, when an LSM tree is empty, operations are written to L0. Recall that all elements are ordered by key in ascending order, and then within a single key scope, by LSN in descending order, so when a new value associated with a given key gets inserted, it's easy to locate the older value and delete it. L0 can be structured as any container capable of storing a sorted sequence of elements. For example, in Tarantool, L0 is implemented as a B+*-tree. Lookups and insertions are standard operations for the data structure underlying L0, so I won't dwell on those.

Sooner or later the number of elements in an LSM tree exceeds the L0 size and that's when L0 gets written to a file on disk (called a "run") and then cleared for storing new elements. This operation is called a "dump".

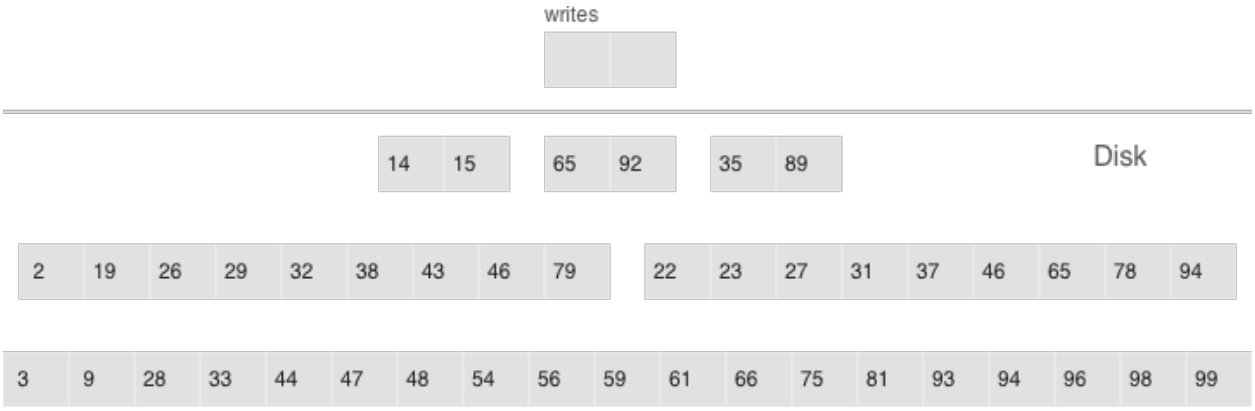


Dumps on disk form a sequence ordered by LSN: LSN ranges in different runs don't overlap, and the leftmost runs (at the head of the sequence) hold newer operations. Think of these runs as a pyramid, with the newest ones closer to the top. As runs keep getting dumped, the pyramid grows higher. Note that newer runs may contain deletions or replacements for existing keys. To remove older data, it's necessary to perform garbage collection (this process is sometimes called "merge" or "compaction") by combining several older runs into a new one. If two versions of the same key are encountered during a compaction, only the newer one is retained; however, if a key insertion is followed by a deletion, then both operations can be discarded.



The key choices determining an LSM tree's efficiency are which runs to compact and when to compact them. Suppose an LSM tree stores a monotonically increasing sequence of keys (1, 2, 3, . . . ,) with no deletions. In this case, compacting runs would be useless: all of the elements are sorted, the tree doesn't have any garbage, and the location of any key can unequivocally be determined. On the other hand, if an LSM tree contains many deletions, doing a compaction would free up some disk space. However, even if there are no deletions, but key ranges in different runs overlap a lot, compacting such runs could speed up lookups as there would be fewer runs to scan. In this case, it might make sense to compact runs after each dump. But keep in mind that a compaction causes all data stored on disk to be overwritten, so with few reads it's recommended to perform it less often.

To ensure it's optimally configurable for any of the scenarios above, an LSM tree organizes all runs into a pyramid: the newer the data operations, the higher up the pyramid they are located. During a compaction, the algorithm picks two or more neighboring runs of approximately equal size, if possible.



- Multi-level compaction can span any number of levels
- A level can contain multiple runs

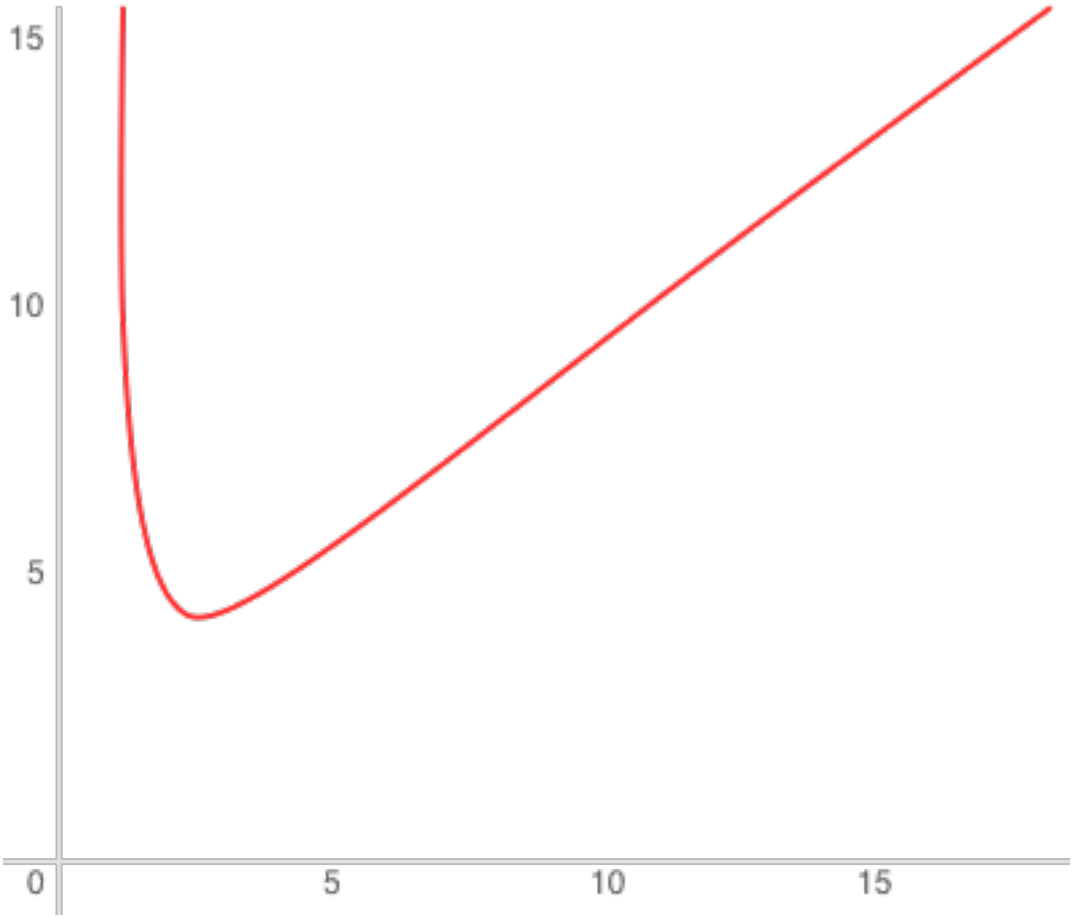
All of the neighboring runs of approximately equal size constitute an LSM tree level on disk. The ratio of run sizes at different levels determines the pyramid's proportions, which allows optimizing the tree for write-intensive or read-intensive scenarios.

Suppose the L0 size is 100 Mb, the ratio of run sizes at each level (the vinyl_run_size_ratio parameter) is 5, and there can be no more than 2 runs per level (the vinyl_run_count_per_level parameter). After the first 3 dumps, the disk will contain 3 runs of 100 Mb each—which constitute L1 (level one). Since 3 > 2, the runs will be compacted into a single 300 Mb run, with the older ones being deleted. After 2 more dumps, there will be another compaction, this time of 2 runs of 100 Mb each and the 300 Mb run, which will produce one 500 Mb run. It will be moved to L2 (recall that the run size ratio is 5), leaving L1 empty. The next 10 dumps will result in L2 having 3 runs of 500 Mb each, which will be compacted into a single 1500 Mb run. Over the course of 10 more dumps, the following will happen: 3 runs of 100 Mb each will be compacted twice, as will two 100 Mb runs and one 300 Mb run, which will yield 2 new 500 Mb runs in L2. Since L2 now has 3 runs, they will also be compacted: two 500 Mb runs and one 1500 Mb run will produce a 2500 Mb run that will be moved to L3, given its size.

This can go on infinitely, but if an LSM tree contains lots of deletions, the resulting compacted run can be moved not only down, but also up the pyramid due to its size being smaller than the sizes of the original runs that were compacted. In other words, it's enough to logically track which level a certain run belongs to, based on the run size and the smallest and greatest LSN among all of its operations.

Controlling the form of an LSM tree

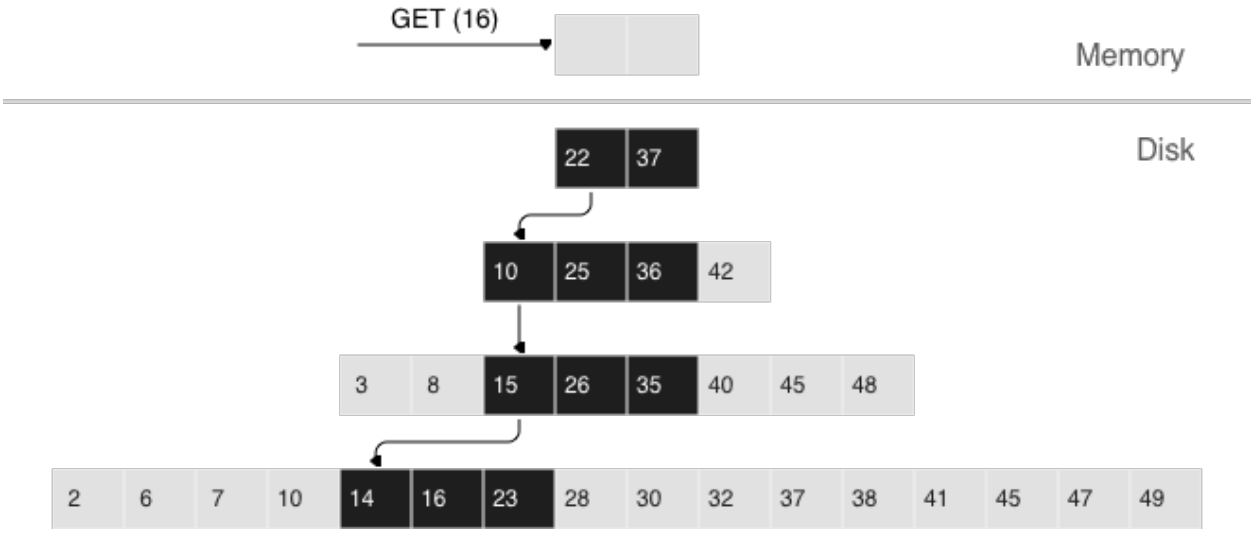
If it's necessary to reduce the number of runs for lookups, then the run size ratio can be increased, thus bringing the number of levels down. If, on the other hand, you need to minimize the compaction-related overhead, then the run size ratio can be decreased: the pyramid will grow higher, and even though runs will be compacted more often, they will be smaller, which will reduce the total amount of work done. In general, write amplification in an LSM tree is described by this formula: log_{x}(\frac {N} {L0}) × x or, alternatively, x × \frac {ln (\frac {N} {C0})} {ln(x)}, where N is the total size of all tree elements, L0 is the level zero size, and x is the level size ratio (the level_size_ratio parameter). At \frac {N} {C0} = 40 (the disk-to- memory ratio), the plot would look something like this:



As for read amplification, it's proportional to the number of levels. The lookup cost at each level is no greater than that for a B-tree. Getting back to the example of a tree with 100,000,000 elements: given 256 Mb of RAM and the default values of vinyl_level_size_ratio and run_count_per_level, write amplification would come out to about 13, while read amplification could be as high as 150. Let's try to figure out why this happens.

Search

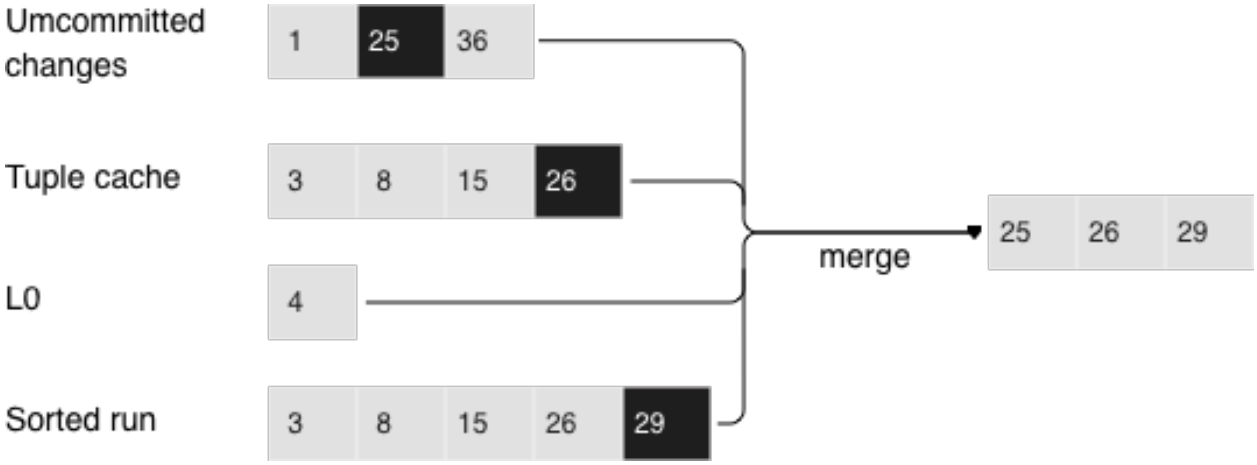
When doing a lookup in an LSM tree, what we need to find is not the element itself, but the most recent operation associated with it. If it's a deletion, then the tree doesn't contain this element. If it's an insertion, we need to grab the topmost value in the pyramid, and the search can be stopped after finding the first matching key. In the worst-case scenario, that is if the tree doesn't hold the needed element, the algorithm will have to sequentially visit all of the levels, starting from L0.



Unfortunately, this scenario is quite common in real life. For example, when inserting a value into a tree, it's necessary to make sure there are no duplicates among primary/unique keys. So to speed up membership checks, LSM trees use a probabilistic data structure called a "Bloom filter", which will be covered a bit later, in a section on how vinyl works under the hood.

Range searching

In the case of a single-key search, the algorithm stops after encountering the first match. However, when searching within a certain key range (for example, looking for all the users with the last name "Ivanov"), it's necessary to scan all tree levels.



Searching within a range of [24,30)

The required range is formed the same way as when compacting several runs: the algorithm picks the key with the largest LSN out of all the sources, ignoring the other associated operations, then moves on to the next key and repeats the procedure.

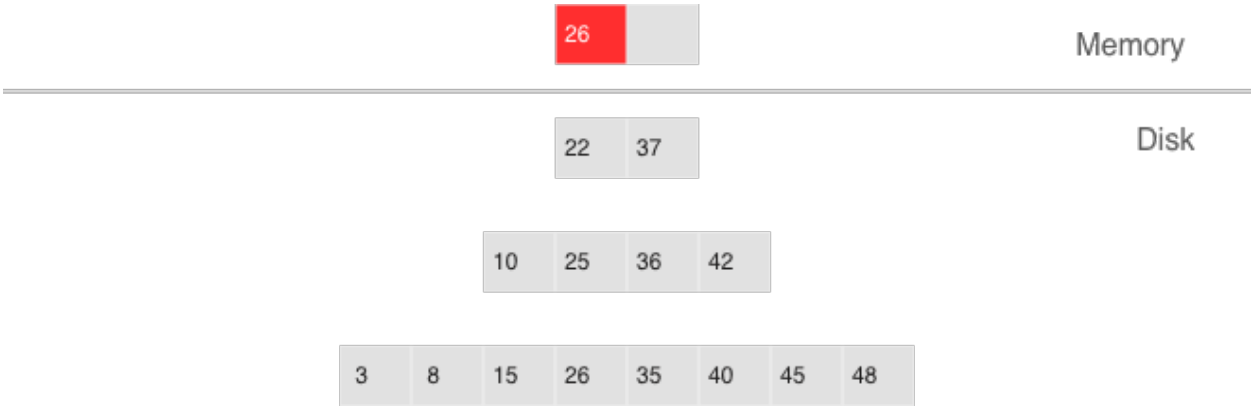
Deletion

Why would one store deletions? And why doesn't it lead to a tree overflow in the case of for i=1,10000000 put(i) delete(i) end?

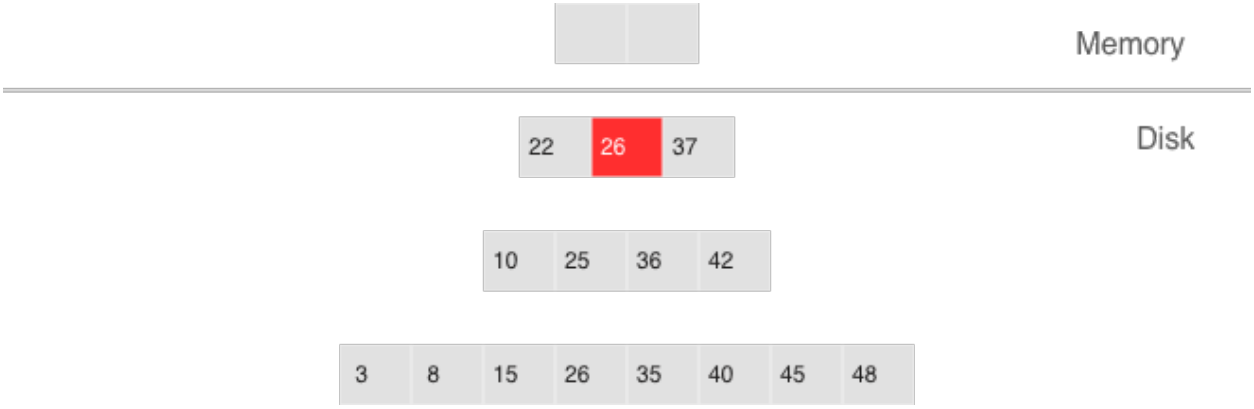
With regards to lookups, deletions signal the absence of a value being searched; with compactions, they clear the tree of "garbage" records with older LSNs.

While the data is in RAM only, there's no need to store deletions. Similarly, you don't need to keep them following a compaction if they affect, among other things, the lowest tree level, which contains the oldest dump. Indeed, if a value can't be found at the lowest level, then it doesn't exist in the tree.

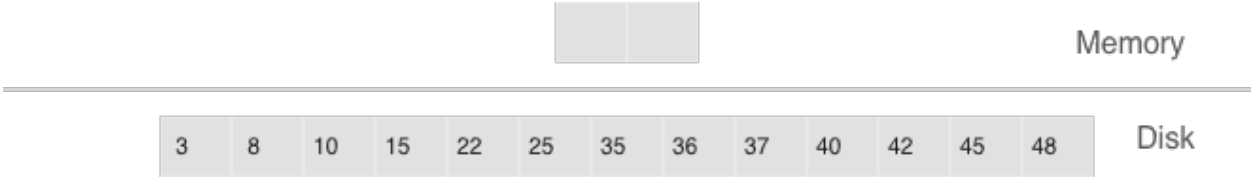
- We can't delete from append-only files
- Tombstones (delete markers) are inserted into L0 instead



Deletion, step 1: a tombstone is inserted into L0



Deletion, step 2: the tombstone passes through intermediate levels



Deletion, step 3: in the case of a major compaction, the tombstone is removed from the tree

If a deletion is known to come right after the insertion of a unique value, which is often the case when modifying a value in a secondary index, then the deletion can safely be filtered out while compacting intermediate tree levels. This optimization is implemented in vinyl.

Advantages of an LSM tree

Apart from decreasing write amplification, the approach that involves periodically dumping level L0 and compacting levels L1-Lk has a few advantages over the approach to writes adopted by B-trees:

- Dumps and compactions write relatively large files: typically, the L0 size is 50-100 Mb, which is thousands of times larger than the size of a B-tree block.
- This large size allows efficiently compressing data before writing it. Tarantool compresses data automatically, which further decreases write amplification.
- There is no fragmentation overhead, since there's no padding/empty space between the elements inside a run.
- All operations create new runs instead of modifying older data in place. This allows avoiding those nasty locks that everyone hates so much. Several operations can run in parallel without causing any conflicts. This also simplifies making backups and moving data to replicas.
- Storing older versions of data allows for the efficient implementation of transaction support by using multiversion concurrency control.

Disadvantages of an LSM tree and how to deal with them

One of the key advantages of the B-tree as a search data structure is its predictability: all operations take no longer than log_{B}(N) to run. Conversely, in a classical LSM tree, both read and write speeds can differ by a factor of hundreds (best case scenario) or even thousands (worst case scenario). For example, adding just one element to L0 can cause it to overflow, which can trigger a chain reaction in levels L1, L2, and so on. Lookups may find the needed element in L0 or may need to scan all of the tree levels. It's also necessary to optimize reads within a single level to achieve speeds comparable to those of a B-tree. Fortunately, most disadvantages can be mitigated or even eliminated with additional algorithms and data structures. Let's take a closer look at these disadvantages and how they're dealt with in Tarantool.

Unpredictable write speed

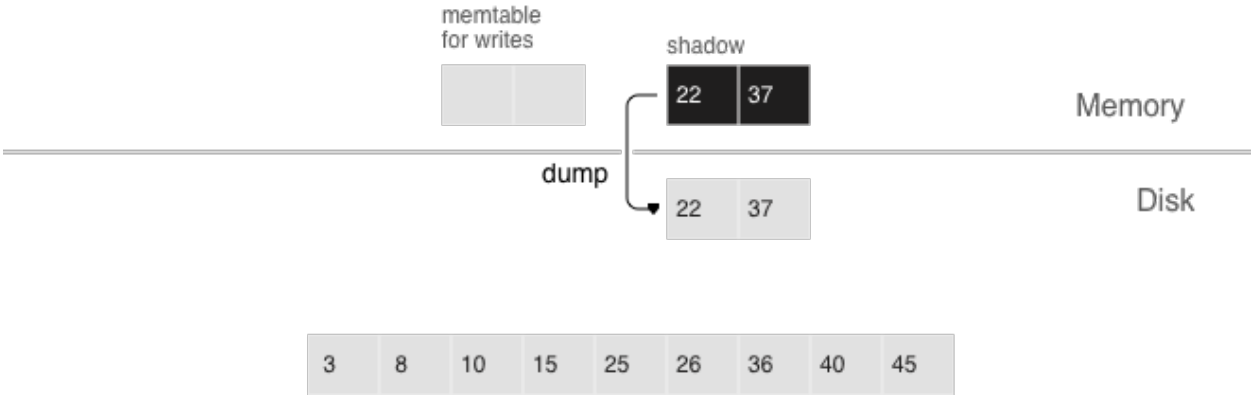
In an LSM tree, insertions almost always affect L0 only. How do you avoid idle time when the memory area allocated for L0 is full?

Clearing L0 involves two lengthy operations: writing to disk and memory deallocation. To avoid idle time while L0 is being dumped, Tarantool uses writeaheads. Suppose the L0 size is 256 Mb. The disk write speed is 10 Mbps. Then it would take 26 seconds to dump L0. The insertion speed is 10,000 RPS, with each key having a size of 100 bytes. While L0 is being dumped, it's necessary to reserve 26 Mb of RAM, effectively slicing the L0 size down to 230 Mb.

Tarantool does all of these calculations automatically, constantly updating the rolling average of the DBMS workload and the histogram of the disk speed. This allows using L0 as efficiently as possible and it prevents write requests from timing out. But in the case of workload surges, some wait time is still possible. That's why we also introduced an insertion timeout (the vinyl_timeout parameter), which is set to 60 seconds by default. The write operation itself is executed in dedicated threads. The number of these threads (2 by default) is controlled by the vinyl_write_threads parameter. The default value of 2 allows doing dumps and compactions in parallel, which is also necessary for ensuring system predictability.

In Tarantool, compactions are always performed independently of dumps, in a separate execution thread. This is made possible by the append-only nature of an LSM tree: after dumps runs are never changed, and compactions simply create new runs.

Delays can also be caused by L0 rotation and the deallocation of memory dumped to disk: during a dump, L0 memory is owned by two operating system threads, a transaction processing thread and a write thread. Even though no elements are being added to the rotated L0, it can still be used for lookups. To avoid read locks when doing lookups, the write thread doesn't deallocate the dumped memory, instead delegating this task to the transaction processor thread. Following a dump, memory deallocation itself happens instantaneously: to achieve this, L0 uses a special allocator that deallocates all of the memory with a single operation.

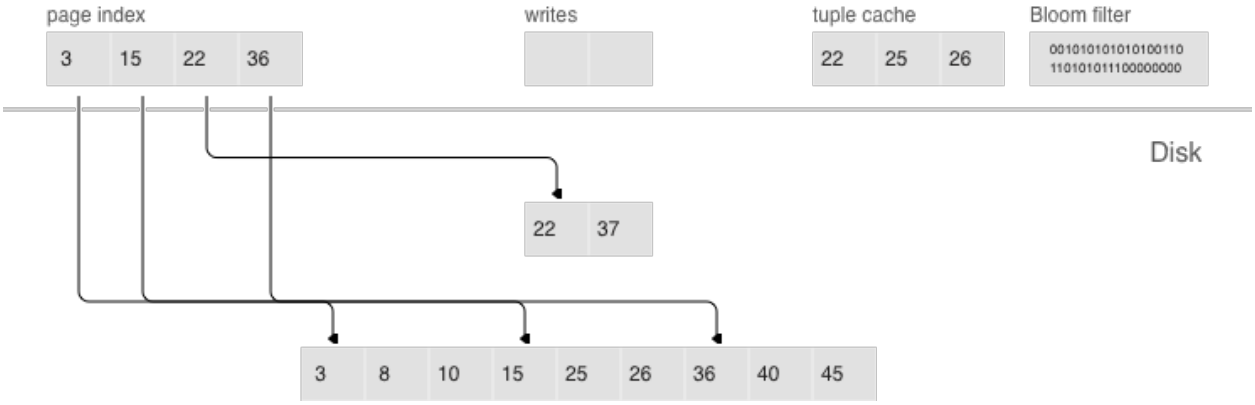


- anticipatory dump
- throttling

The dump is performed from the so-called "shadow" L0 without blocking new insertions and lookups

Unpredictable read speed

Optimizing reads is the most difficult optimization task with regards to LSM trees. The main complexity factor here is the number of levels: any optimization causes not only much slower lookups, but also tends to require significantly larger RAM resources. Fortunately, the append-only nature of LSM trees allows us to address these problems in ways that would be nontrivial for traditional data structures.



- page index
- bloom filters
- tuple range cache
- multi-level compaction

Compression and page index

In B-trees, data compression is either the hardest problem to crack or a great marketing tool—rather than something really useful. In LSM trees, compression works as follows:

During a dump or compaction all of the data within a single run is split into pages. The page size (in bytes) is controlled by the vinyl_page_size parameter and can be set separately for each index. A page doesn't have to be exactly of vinyl_page_size size—depending on the data it holds, it can be a little bit smaller or larger. Because of this, pages never have any empty space inside.

Data is compressed by Facebook's streaming algorithm called "zstd". The first key of each page, along with the page offset, is added to a "page index", which is a separate file that allows the quick retrieval of any page. After a dump or compaction, the page index of the created run is also written to disk.

All .index files are cached in RAM, which allows finding the necessary page with a single lookup in a .run file (in vinyl, this is the extension of files resulting from a dump or compaction). Since data within a page is sorted, after it's read and decompressed, the needed key can be found using a regular binary search. Decompression and reads are handled by separate threads, and are controlled by the vinyl_read_threads parameter.

Tarantool uses a universal file format: for example, the format of a .run file is no different from that of an .xlog file (log file). This simplifies backup and recovery as well as the usage of external tools.

Bloom filters

Even though using a page index enables scanning fewer pages per run when doing a lookup, it's still necessary to traverse all of the tree levels. There's a special case, which involves checking if particular data is absent when scanning all of the tree levels and it's unavoidable: I'm talking about insertions into a unique index. If the data being inserted already exists, then inserting the same data into a unique index should lead to an error. The only way to throw an error in an LSM tree before a transaction is committed is to do a search before inserting the data. Such reads form a class of their own in the DBMS world and are called "hidden" or "parasitic" reads.

Another operation leading to hidden reads is updating a value in a field on which a secondary index is defined. Secondary keys are regular LSM trees that store differently ordered data. In most cases, in order not to have to store all of the data in all of the indexes, a value associated with a given key is kept in whole only in the primary index (any index that stores both a key and a value is called "covering" or "clustered"), whereas the secondary index only stores the fields on which a secondary index is defined, and the values of the fields that are part of the primary index. Thus, each time a change is made to a value in a field on which a secondary index is defined, it's necessary to first remove the old key from the secondary index—and only then can the new key be inserted. At update time, the old value is unknown, and it is this value that needs to be read in from the primary key "under the hood".

For example:

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

To minimize the number of disk reads, especially for nonexistent data, nearly all LSM trees use probabilistic data structures, and Tarantool is no exception. A classical Bloom filter is made up of several (usually 3-to-5) bit arrays. When data is written, several hash functions are calculated for each key in order to get corresponding array positions. The bits at these positions are then set to 1. Due to possible hash collisions, some bits might be set to 1 twice. We're most interested in the bits that remain 0 after all keys have been added. When looking for an element within a run, the same hash functions are applied to produce bit positions in the arrays. If any of the bits at these positions is 0, then the element is definitely not in the run. The probability of a false positive in a Bloom filter is calculated using Bayes' theorem: each hash function

is an independent random variable, so the probability of a collision simultaneously occurring in all of the bit arrays is infinitesimal.

The key advantage of Bloom filters in Tarantool is that they're easily configurable. The only parameter that can be specified separately for each index is called bloom_fpr (FPR stands for "false positive ratio") and it has the default value of 0.05, which translates to a 5% FPR. Based on this parameter, Tarantool automatically creates Bloom filters of the optimal size for partial- key and full-key searches. The Bloom filters are stored in the .index file, along with the page index, and are cached in RAM.

Caching

A lot of people think that caching is a silver bullet that can help with any performance issue. "When in doubt, add more cache". In vinyl, caching is viewed rather as a means of reducing the overall workload and consequently, of getting a more stable response time for those requests that don't hit the cache. vinyl boasts a unique type of cache among transactional systems called a "range tuple cache". Unlike, say, RocksDB or MySQL, this cache doesn't store pages, but rather ranges of index values obtained from disk, after having performed a compaction spanning all tree levels. This allows the use of caching for both single-key and key-range searches. Since this method of caching stores only hot data and not, say, pages (you may need only some data from a page), RAM is used in the most efficient way possible. The cache size is controlled by the vinyl_cache parameter.

Garbage collection control

Chances are that by now you've started losing focus and need a well-deserved dopamine reward. Feel free to take a break, since working through the rest of the article is going to take some serious mental effort.

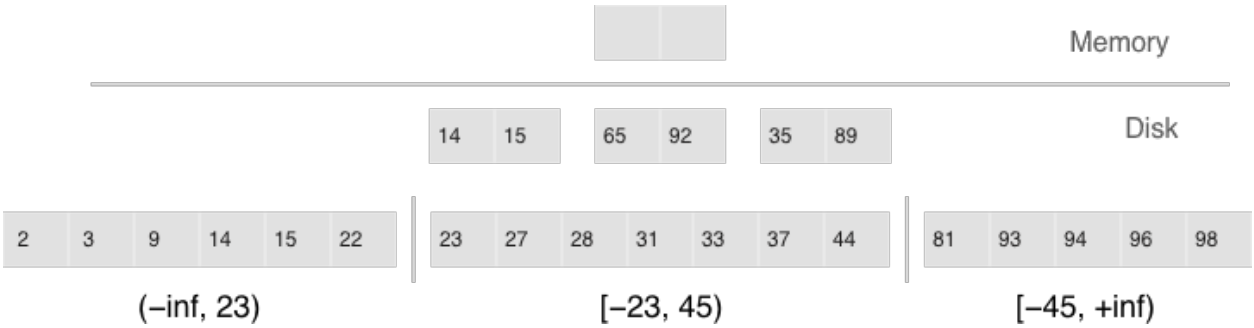
An LSM tree in vinyl is just a small piece of the puzzle. Even with a single table (or so-called "space"), vinyl creates and maintains several LSM trees, one for each index. But even a single index can be comprised of dozens of LSM trees. Let's try to understand why this might be necessary.

Recall our example with a tree containing 100,000,000 records, 100 bytes each. As time passes, the lowest LSM level may end up holding a 10 Gb run. During compaction, a temporary run of approximately the same size will be created. Data at intermediate levels takes up some space as well, since the tree may store several operations associated with a single key. In total, storing 10 Gb of actual data may require up to 30 Gb of free space: 10 Gb for the last tree level, 10 Gb for a temporary run, and 10 Gb for the remaining data. But what if the data size is not 10 Gb, but 1 Tb? Requiring that the available disk space always be several times greater than the actual data size is financially unpractical, not to mention that it may take dozens of hours to create a 1 Tb run. And in the case of an emergency shutdown or system restart, the process would have to be started from scratch.

Here's another scenario. Suppose the primary key is a monotonically increasing sequence—for example, a time series. In this case, most insertions will fall into the right part of the key range, so it wouldn't make much sense to do a compaction just to append a few million more records to an already huge run.

But what if writes predominantly occur in a particular region of the key range, whereas most reads take place in a different region? How do you optimize the form of the LSM tree in this case? If it's too high, read performance is impacted; if it's too low—write speed is reduced.

Tarantool "factorizes" this problem by creating multiple LSM trees for each index. The approximate size of each subtree may be controlled by the vinyl_range_size configuration parameter. We call such subtrees "ranges".



Factorizing large LSM trees via ranging

- Ranges reflect a static layout of sorted runs
- Slices connect a sorted run into a range

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

Split and coalesce don't entail a compaction, the creation of new runs, or other resource-intensive operations. An LSM tree is just a collection of runs. vinyl has a special metadata log that helps keep track of which run belongs to which subtree(s). This has the .vylog extension and its format is compatible with an .xlog file. Similarly to an .xlog file, the metadata log gets rotated at each checkpoint. To avoid the creation of extra runs with split and coalesce, we have also introduced an auxiliary entity called "slice". It's a reference to a run containing a key range and it's stored only in the metadata log. Once the reference counter drops to zero, the corresponding file gets removed. When it's necessary to perform a split or to coalesce, Tarantool creates slice objects for each new tree, removes older slices, and writes these operations to the metadata log, which literally stores records that look like this: <tree id, slice id> or <slice id, run id, min, max>.

This way all of the heavy lifting associated with splitting a tree into two subtrees is postponed until a compaction and then is performed automatically. A huge advantage of dividing all of the keys into ranges is the ability to independently control the L0 size as well as the dump and compaction processes for each subtree, which makes these processes manageable and predictable. Having a separate metadata log also simplifies the implementation of both "truncate" and "drop". In vinyl, they're processed instantly, since they only work with the metadata log, while garbage collection is done in the background.

Advanced features of vinyl

Upsert

In the previous sections, we mentioned only two operations stored by an LSM tree: deletion and replacement. Let's take a look at how all of the other operations can be represented. An insertion can be represented via a replacement—you just need to make sure there are no other elements with the specified key. To perform an update, it's necessary to read the older value from the tree, so it's easier to represent this operation as a replacement as well—this speeds up future read requests by the key. Besides, an update must return the new value, so there's no avoiding hidden reads.

In B-trees, the cost of hidden reads is negligible: to update a block, it first needs to be read from disk anyway. Creating a special update operation for an LSM tree that doesn't cause any hidden reads is really

tempting.

Such an operation must contain not only a default value to be inserted if a key has no value yet, but also a list of update operations to perform if a value does exist.

At transaction execution time, Tarantool just saves the operation in an LSM tree, then "executes" it later, during a compaction.

The upsert operation:

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

- Non-reading update or insert
- Delayed execution
- Background upsert squashing prevents upserts from piling up

Unfortunately, postponing the operation execution until a compaction doesn't leave much leeway in terms of error handling. That's why Tarantool tries to validate upserts as fully as possible before writing them to an LSM tree. However, some checks are only possible with older data on hand, for example when the update operation is trying to add a number to a string or to remove a field that doesn't exist.

A semantically similar operation exists in many products including PostgreSQL and MongoDB. But anywhere you look, it's just syntactic sugar that combines the update and replace operations without avoiding hidden reads. Most probably, the reason is that LSM trees as data storage structures are relatively new.

Even though an upsert is a very important optimization and implementing it cost us a lot of blood, sweat, and tears, we must admit that it has limited applicability. If a table contains secondary keys or triggers, hidden reads can't be avoided. But if you have a scenario where secondary keys are not required and the update following the transaction completion will certainly not cause any errors, then the operation is for you.

I'd like to tell you a short story about an upsert. It takes place back when vinyl was only beginning to "mature" and we were using an upsert in production for the first time. We had what seemed like an ideal environment for it: we had tons of keys, the current time was being used as values; update operations were inserting keys or modifying the current time; and we had few reads. Load tests yielded great results.

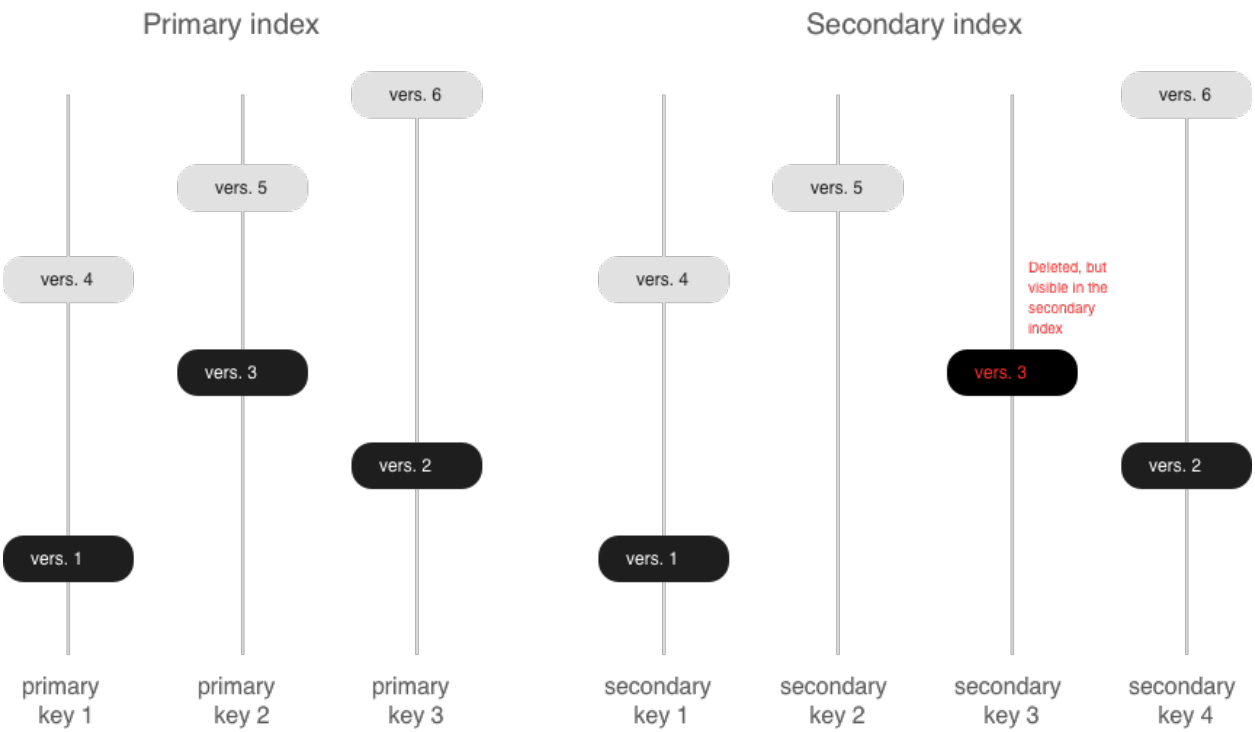
Nevertheless, after a couple of days, the Tarantool process started eating up 100% of our CPU, and the system performance dropped close to zero.

We started digging into the issue and found out that the distribution of requests across keys was significantly different from what we had seen in the test environment. It was... well, quite nonuniform. Most keys were updated once or twice a day, so the database was idle for the most part, but there were much hotter keys with tens of thousands of updates per day. Tarantool handled those just fine. But in the case of lookups by key with tens of thousands of upserts, things quickly went downhill. To return the most recent value, Tarantool had to read and "replay" the whole history consisting of all of the upserts. When designing upserts, we had hoped this would happen automatically during a compaction, but the process never even got to that stage: the L0 size was more than enough, so there were no dumps.

We solved the problem by adding a background process that performed readaheads on any keys that had more than a few dozen upserts piled up, so all those upserts were squashed and substituted with the read value.

Secondary keys

Update is not the only operation where optimizing hidden reads is critical. Even the replace operation, given secondary keys, has to read the older value: it needs to be independently deleted from the secondary indexes, and inserting a new element might not do this, leaving some garbage behind.



If secondary indexes are not unique, then collecting "garbage" from them can be put off until a compaction, which is what we do in Tarantool. The append-only nature of LSM trees allowed us to implement full-blown serializable transactions in vinyl. Read-only requests use older versions of data without blocking any writes. The transaction manager itself is fairly simple for now: in classical terms, it implements the MVTO (multiversion timestamp ordering) class, whereby the winning transaction is the one that finished earlier. There are no locks and associated deadlocks. Strange as it may seem, this is a drawback rather than an advantage: with parallel execution, you can increase the number of successful transactions by simply holding some of them on lock when necessary. We're planning to improve the transaction manager soon. In the current release, we focused on making the algorithm behave 100% correctly and predictably. For example, our transaction manager is one of the few on the NoSQL market that supports so-called "gap locks".

3.4 Application server

In this chapter, we introduce the basics of working with Tarantool as a Lua application server.

This chapter contains the following sections:

3.4.1 Launching an application

Using Tarantool as an application server, you can write your own applications. Tarantool's native language for writing applications is Lua, so a typical application would be a file that contains your Lua script. But you can also write applications in C or C++.

Note: If you're new to Lua, we recommend going over the interactive Tarantool tutorial before proceeding with this chapter. To launch the tutorial, say tutorial() in Tarantool console:

```
tarantool> tutorial()
---
```

(continues on next page)

(continued from previous page)

```
- |
 Tutorial -- Screen #1 -- Hello, Moon
 ====================================

 Welcome to the Tarantool tutorial.
 It will introduce you to Tarantool's Lua application server
 and database server, which is what's running what you're seeing.
 This is INTERACTIVE -- you're expected to enter requests
 based on the suggestions or examples in the screen's text.
 <...>
```

Let's create and launch our first Lua application for Tarantool. Here's a simplest Lua application, the good old "Hello, world!":

```
#!/usr/bin/env tarantool
print('Hello, world!')
```

We save it in a file. Let it be myapp.lua in the current directory.

Now let's discuss how we can launch our application with Tarantool.

Launching in Docker

If we run Tarantool in a Docker container, the following command will start Tarantool 1.9 without any application:

```
$ # create a temporary container and run it in interactive mode
$ docker run --rm -t -i tarantool/tarantool:1
```

To run Tarantool with our application, we can say:

```
$ # create a temporary container and
$ # launch Tarantool with our application
$ docker run --rm -t -i \
        -v `pwd`/myapp.lua:/opt/tarantool/myapp.lua \
        -v /data/dir/on/host:/var/lib/tarantool \
        tarantool/tarantool:1 tarantool /opt/tarantool/myapp.lua
```

Here two resources on the host get mounted in the container:

- our application file (myapp.lua) and
- Tarantool data directory (/data/dir/on/host).

By convention, the directory for Tarantool application code inside a container is /opt/tarantool, and the directory for data is /var/lib/tarantool.

Launching a binary program

If we run Tarantool from a binary package or from a source build, we can launch our application:

- in the script mode,
- as a server application, or
- as a daemon service.

The simplest way is to pass the filename to Tarantool at start:

```
$ tarantool myapp.lua
Hello, world!
$
```

Tarantool starts, executes our script in the script mode and exits.

Now let's turn this script into a server application. We use box.cfg from Tarantool's built-in Lua module to:

- launch the database (a database has a persistent on-disk state, which needs to be restored after we start an application) and
- configure Tarantool as a server that accepts requests over a TCP port.

We also add some simple database logic, using space.create() and create_index() to create a space with a primary index. We use the function box.once() to make sure that our logic will be executed only once when the database is initialized for the first time, so we don't try to create an existing space or index on each invocation of the script:

```
#!/usr/bin/env tarantool
-- Configure database
box.cfg {
   listen = 3301
}
box.once("bootstrap", function()
   box.schema.space.create('tweedledum')
   box.space.tweedledum:create_index('primary',
      { type = 'TREE', parts = {1, 'unsigned'}})
end)
```

Now we launch our application in the same manner as before:

```
$ tarantool myapp.lua
Hello, world!
2017-08-11 16:07:14.250 [41436] main/101/myapp.lua C> version 2.1.0-429-g4e5231702
2017-08-11 16:07:14.250 [41436] main/101/myapp.lua C> log level 5
2017-08-11 16:07:14.251 [41436] main/101/myapp.lua I> mapping 1073741824 bytes for tuple arena...
2017-08-11 16:07:14.255 [41436] main/101/myapp.lua I> recovery start
2017-08-11 16:07:14.255 [41436] main/101/myapp.lua I> recovering from `./00000000000000000000.snap'
2017-08-11 16:07:14.271 [41436] main/101/myapp.lua I> recover from `./00000000000000000000.xlog'
2017-08-11 16:07:14.271 [41436] main/101/myapp.lua I> done `./00000000000000000000.xlog'
2017-08-11 16:07:14.272 [41436] main/102/hot_standby I> recover from `./00000000000000000000.xlog'
2017-08-11 16:07:14.274 [41436] iproto/102/iproto I> binary: started
2017-08-11 16:07:14.275 [41436] iproto/102/iproto I> binary: bound to [::]:3301
2017-08-11 16:07:14.275 [41436] main/101/myapp.lua I> done `./00000000000000000000.xlog'
2017-08-11 16:07:14.278 [41436] main/101/myapp.lua I> ready to accept requests
```

This time, Tarantool executes our script and keeps working as a server, accepting TCP requests on port 3301. We can see Tarantool in the current session's process list:

```
$ ps | grep "tarantool"
  PID TTY          TIME CMD
41608 ttys001    0:00.47 tarantool myapp.lua <running>
```

But the Tarantool instance will stop if we close the current terminal window. To detach Tarantool and our application from the terminal window, we can launch it in the daemon mode. To do so, we add some parameters to box.cfg{}:

- background = true that actually tells Tarantool to work as a daemon service,

- log = 'dir-name' that tells the Tarantool daemon where to store its log file (other log settings are available in Tarantool log module), and
- pid_file = 'file-name' that tells the Tarantool daemon where to store its pid file.

For example:

```
box.cfg {
   listen = 3301
   background = true,
   log = '1.log',
   pid_file = '1.pid'
}
```

We launch our application in the same manner as before:

```
$ tarantool myapp.lua
Hello, world!
$
```

Tarantool executes our script, gets detached from the current shell session (you won't see it with ps | grep "tarantool") and continues working in the background as a daemon attached to the global session (with SID = 0):

```
$ ps -ef | grep "tarantool"
  PID SID     TIME  CMD
42178   0  0:00.72 tarantool myapp.lua <running>
```

Now that we have discussed how to create and launch a Lua application for Tarantool, let's dive deeper into programming practices.

3.4.2 Creating an application

Further we walk you through key programming practices that will give you a good start in writing Lua applications for Tarantool. For an adventure, this is a story of implementing... a real microservice based on Tarantool! We implement a backend for a simplified version of Pokémon Go, a location-based augmented reality game released in mid-2016. In this game, players use a mobile device's GPS capability to locate, capture, battle and train virtual monsters called "pokémon", who appear on the screen as if they were in the same real-world location as the player.

To stay within the walk-through format, let's narrow the original gameplay as follows. We have a map with pokémon spawn locations. Next, we have multiple players who can send catch-a-pokémon requests to the server (which runs our Tarantool microservice). The server replies whether the pokémon is caught or not, increases the player's pokémon counter if yes, and triggers the respawn-a-pokémon method that spawns a new pokémon at the same location in a while.

We leave client-side applications outside the scope of this story. Yet we promise a mini-demo in the end to simulate real users and give us some fun. :-)

* * *

First, what would be the best way to deliver our microservice?

Modules, rocks and applications

To make our game logic available to other developers and Lua applications, let's put it into a Lua module.

A module (called "rock" in Lua) is an optional library which enhances Tarantool functionality. So, we can install our logic as a module in Tarantool and use it from any Tarantool application or module. Like applications, modules in Tarantool can be written in Lua (rocks), C or C++.

Modules are good for two things:

- easier code management (reuse, packaging, versioning), and
- hot code reload without restarting the Tarantool instance.

Technically, a module is a file with source code that exports its functions in an API. For example, here is a Lua module named mymodule.lua that exports one function named myfun:

```
local exports = {}
exports.myfun = function(input_string)
  print('Hello', input_string)
end
return exports
```

To launch the function myfun() – from another module, from a Lua application, or from Tarantool itself, – we need to save this module as a file, then load this module with the require() directive and call the exported function.

For example, here's a Lua application that uses myfun() function from mymodule.lua module:

```
-- loading the module
local mymodule = require('mymodule')

-- calling myfun() from within test() function
local test = function()
  mymodule.myfun()
end
```

A thing to remember here is that the require() directive takes load paths to Lua modules from the package.path variable. This is a semicolon-separated string, where a question mark is used to interpolate the module name. By default, this variable contains system-wide Lua paths and the working directory. But if we put our modules inside a specific folder (e.g. scripts/), we need to add this folder to package.path before any calls to require():

```
package.path = 'scripts/?.lua;' .. package.path
```

For our microservice, a simple and convenient solution would be to put all methods in a Lua module (say pokemon.lua) and to write a Lua application (say game.lua) that initializes the gaming environment and starts the game loop.

* * *

Now let's get down to implementation details. In our game, we need three entities:

- map, which is an array of pokémons with coordinates of respawn locations; in this version of the game, let a location be a rectangle identified with two points, upper-left and lower-right;
- player, which has an ID, a name, and coordinates of the player's location point;
- pokémon, which has the same fields as the player, plus a status (active/inactive, that is present on the map or not) and a catch probability (well, let's give our pokémons a chance to escape :-))

We'll store these entities as tuples in Tarantool spaces. But to deliver our backend application as a microservice, the good practice would be to send/receive our data in the universal JSON format, thus using Tarantool as a document storage.

Avro schemas

To store JSON data as tuples, we will apply a savvy practice which reduces data footprint and ensures all stored documents are valid. We will use Tarantool module avro-schema which checks the schema of a JSON document and converts it to a Tarantool tuple. The tuple will contain only field values, and thus take a lot less space than the original document. In avro-schema terms, converting JSON documents to tuples is "flattening", and restoring the original documents is "unflattening". The usage is quite straightforward:

(1) For each entity, we need to define a schema in Apache Avro schema syntax, where we list the entity's fields with their names and Avro data types.

(2) At initialization, we call avro-schema.create() that creates objects in memory for all schema entities, and compile() that generates flatten/unflatten methods for each entity.

(3) Further on, we just call flatten/unflatten methods for a respective entity on receiving/sending the entity's data.

Here's what our schema definitions for the player and pokémon entities look like:

```
local schema = {
    player = {
        type="record",
        name="player_schema",
        fields={
            {name="id", type="long"},
            {name="name", type="string"},
            {
                name="location",
                type= {
                    type="record",
                    name="player_location",
                    fields={
                        {name="x", type="double"},
                        {name="y", type="double"}
                    }
                }
            }
        }
    },
    pokemon = {
        type="record",
        name="pokemon_schema",
        fields={
            {name="id", type="long"},
            {name="status", type="string"},
            {name="name", type="string"},
            {name="chance", type="double"},
            {
                name="location",
                type= {
                    type="record",
                    name="pokemon_location",
                    fields={
                        {name="x", type="double"},
                        {name="y", type="double"}
                    }
                }
            }
        }
```

(continues on next page)

(continued from previous page)

```
    }
}
```

And here's how we create and compile our entities at initialization:

```
-- load avro-schema module with require()
local avro = require('avro_schema')

-- create models
local ok_m, pokemon = avro.create(schema.pokemon)
local ok_p, player = avro.create(schema.player)
if ok_m and ok_p then
    -- compile models
    local ok_cm, compiled_pokemon = avro.compile(pokemon)
    local ok_cp, compiled_player = avro.compile(player)
    if ok_cm and ok_cp then
        -- start the game
        <...>
    else
        log.error('Schema compilation failed')
    end
else
    log.info('Schema creation failed')
end
return false
```

As for the map entity, it would be an overkill to introduce a schema for it, because we have only one map in the game, it has very few fields, and – which is most important – we use the map only inside our logic, never exposing it to external users.

* * *

Next, we need methods to implement the game logic. To simulate object-oriented programming in our Lua code, let's store all Lua functions and shared variables in a single local variable (let's name it as game). This will allow us to address functions or variables from within our module as self.func_name or self.var_name. Like this:

```
local game = {
    -- a local variable
    num_players = 0,

    -- a method that prints a local variable
    hello = function(self)
      print('Hello! Your player number is ' .. self.num_players .. '.')
    end,

    -- a method that calls another method and returns a local variable
    sign_in = function(self)
      self.num_players = self.num_players + 1
      self:hello()
      return self.num_players
    end
}
```

In OOP terms, we can now regard local variables inside game as object fields, and local functions as object methods.

Note: In this manual, Lua examples use local variables. Use global variables with caution, since the module's users may be unaware of them.

To enable/disable the use of undeclared global variables in your Lua code, use Tarantool's strict module.

So, our game module will have the following methods:

- catch() to calculate whether the pokémon was caught (besides the coordinates of both the player and pokémon, this method will apply a probability factor, so not every pokémon within the player's reach will be caught);
- respawn() to add missing pokémons to the map, say, every 60 seconds (we assume that a frightened pokémon runs away, so we remove a pokémon from the map on any catch attempt and add it back to the map in a while);
- notify() to log information about caught pokémons (like "Player 1 caught pokémon A");
- start() to initialize the game (it will create database spaces, create and compile avro schemas, and launch respawn()).

Besides, it would be convenient to have methods for working with Tarantool storage. For example:

- add_pokemon() to add a pokémon to the database, and
- map() to populate the map with all pokémons stored in Tarantool.

We'll need these two methods primarily when initializing our game, but we can also call them later, for example to test our code.

Bootstrapping a database

Let's discuss game initialization. In start() method, we need to populate Tarantool spaces with pokémon data. Why not keep all game data in memory? Why use a database? The answer is: persistence. Without a database, we risk losing data on power outage, for example. But if we store our data in an in-memory database, Tarantool takes care to persist it on disk whenever it's changed. This gives us one more benefit: quick startup in case of failure. Tarantool has a smart algorithm that quickly loads all data from disk into memory on startup, so the warm-up takes little time.

We'll be using functions from Tarantool built-in box module:

- box.schema.create_space('pokemons') to create a space named pokemon for storing information about pokémons (we don't create a similar space for players, because we intend to only send/receive player information via API calls, so we needn't store it);
- box.space.pokemons:create_index('primary', {type = 'hash', parts = {1, 'unsigned'}}) to create a primary HASH index by pokémon ID;
- box.space.pokemons:create_index('status', {type = 'tree', parts = {2, 'str'}}) to create a secondary TREE index by pokémon status.

Notice the parts = argument in the index specification. The pokémon ID is the first field in a Tarantool tuple since it's the first member of the respective Avro type. So does the pokémon status. The actual JSON document may have ID or status fields at any position of the JSON map.

The implementation of start() method looks like this:

```
-- create game object
start = function(self)
    -- create spaces and indexes
```

(continues on next page)

(continued from previous page)

```
    box.once('init', function()
        box.schema.create_space('pokemons')
        box.space.pokemons:create_index(
            "primary", {type = 'hash', parts = {1, 'unsigned'}}
        )
        box.space.pokemons:create_index(
            "status", {type = "tree", parts = {2, 'str'}}
        )
    end)

    -- create models
    local ok_m, pokemon = avro.create(schema.pokemon)
    local ok_p, player = avro.create(schema.player)
    if ok_m and ok_p then
        -- compile models
        local ok_cm, compiled_pokemon = avro.compile(pokemon)
        local ok_cp, compiled_player = avro.compile(player)
        if ok_cm and ok_cp then
            -- start the game
            <...>
        else
            log.error('Schema compilation failed')
        end
    else
        log.info('Schema creation failed')
    end
    return false
end
```

GIS

Now let's discuss catch(), which is the main method in our gaming logic.

Here we receive the player's coordinates and the target pokémon's ID number, and we need to answer whether the player has actually caught the pokémon or not (remember that each pokémon has a chance to escape).

First thing, we validate the received player data against its Avro schema. And we check whether such a pokémon exists in our database and is displayed on the map (the pokémon must have the active status):

```
catch = function(self, pokemon_id, player)
    -- check player data
    local ok, tuple = self.player_model.flatten(player)
    if not ok then
        return false
    end
    -- get pokemon data
    local p_tuple = box.space.pokemons:get(pokemon_id)
    if p_tuple == nil then
        return false
    end
    local ok, pokemon = self.pokemon_model.unflatten(p_tuple)
    if not ok then
        return false
    end
    if pokemon.status ~= self.state.ACTIVE then
```

(continues on next page)

(continued from previous page)

```
        return false
    end
    -- more catch logic to follow
    <...>
end
```

Next, we calculate the answer: caught or not.

To work with geographical coordinates, we use Tarantool gis module.

To keep things simple, we don't load any specific map, assuming that we deal with a world map. And we do not validate incoming coordinates, assuming again that all received locations are within the planet Earth.

We use two geo-specific variables:

- wgs84, which stands for the latest revision of the World Geodetic System standard, WGS84. Basically, it comprises a standard coordinate system for the Earth and represents the Earth as an ellipsoid.
- nationalmap, which stands for the US National Atlas Equal Area. This is a projected coordinates system based on WGS84. It gives us a zero base for location projection and allows positioning our players and pokémons in meters.

Both these systems are listed in the EPSG Geodetic Parameter Registry, where each system has a unique number. In our code, we assign these listing numbers to respective variables:

```
wgs84 = 4326,
nationalmap = 2163,
```

For our game logic, we need one more variable, catch_distance, which defines how close a player must get to a pokémon before trying to catch it. Let's set the distance to 100 meters.

```
catch_distance = 100,
```

Now we're ready to calculate the answer. We need to project the current location of both player (p_pos) and pokémon (m_pos) on the map, check whether the player is close enough to the pokémon (using catch_distance), and calculate whether the player has caught the pokémon (here we generate some random value and let the pokémon escape if the random value happens to be less than 100 minus pokémon's chance value):

```
-- project locations
local m_pos = gis.Point(
    {pokemon.location.x, pokemon.location.y}, self.wgs84
):transform(self.nationalmap)
local p_pos = gis.Point(
    {player.location.x, player.location.y}, self.wgs84
):transform(self.nationalmap)

-- check catch distance condition
if p_pos:distance(m_pos) > self.catch_distance then
    return false
end
-- try to catch pokemon
local caught = math.random(100) >= 100 - pokemon.chance
if caught then
    -- update and notify on success
    box.space.pokemons:update(
        pokemon_id, {{'=', self.STATUS, self.state.CAUGHT}}
    )
```

(continues on next page)

(continued from previous page)

```
    self:notify(player, pokemon)
end
return caught
```

Index iterators

By our gameplay, all caught pokémons are returned back to the map. We do this for all pokémons on the map every 60 seconds using respawn() method. We iterate through pokémons by status using Tarantool index iterator function index:pairs and reset the statuses of all "caught" pokémons back to "active" using box.space.pokemons:update().

```
respawn = function(self)
    fiber.name('Respawn fiber')
    for _, tuple in box.space.pokemons.index.status:pairs(
            self.state.CAUGHT) do
        box.space.pokemons:update(
            tuple[self.ID],
            {{'=', self.STATUS, self.state.ACTIVE}}
        )
    end
 end
```

For readability, we introduce named fields:

```
ID = 1, STATUS = 2,
```

The complete implementation of start() now looks like this:

```
-- create game object
start = function(self)
    -- create spaces and indexes
    box.once('init', function()
        box.schema.create_space('pokemons')
        box.space.pokemons:create_index(
            "primary", {type = 'hash', parts = {1, 'unsigned'}}
        )
        box.space.pokemons:create_index(
            "status", {type = "tree", parts = {2, 'str'}}
        )
    end)

    -- create models
    local ok_m, pokemon = avro.create(schema.pokemon)
    local ok_p, player = avro.create(schema.player)
    if ok_m and ok_p then
        -- compile models
        local ok_cm, compiled_pokemon = avro.compile(pokemon)
        local ok_cp, compiled_player = avro.compile(player)
        if ok_cm and ok_cp then
            -- start the game
            self.pokemon_model = compiled_pokemon
            self.player_model = compiled_player
            self.respawn()
            log.info('Started')
            return true
```

(continues on next page)

(continued from previous page)

```
        else
            log.error('Schema compilation failed')
        end
    else
        log.info('Schema creation failed')
    end
    return false
end
```

Fibers

But wait! If we launch it as shown above – self.respawn() – the function will be executed only once, just like all the other methods. But we need to execute respawn() every 60 seconds. Creating a fiber is the Tarantool way of making application logic work in the background at all times.

A fiber exists for executing instruction sequences but it is not a thread. The key difference is that threads use preemptive multitasking, while fibers use cooperative multitasking. This gives fibers the following two advantages over threads:

- Better controllability. Threads often depend on the kernel's thread scheduler to preempt a busy thread and resume another thread, so preemption may occur unpredictably. Fibers yield themselves to run another fiber while executing, so yields are controlled by application logic.
- Higher performance. Threads require more resources to preempt as they need to address the system kernel. Fibers are lighter and faster as they don't need to address the kernel to yield.

Yet fibers have some limitations as compared with threads, the main limitation being no multi-core mode. All fibers in an application belong to a single thread, so they all use the same CPU core as the parent thread. Meanwhile, this limitation is not really serious for Tarantool applications, because a typical bottleneck for Tarantool is the HDD, not the CPU.

A fiber has all the features of a Lua coroutine and all 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.

Well, performance or controllability are of little importance in our case. We'll launch respawn() in a fiber to make it work in the background all the time. To do so, we'll need to amend respawn():

```
respawn = function(self)
    -- let 's give our fiber a name;
    -- this will produce neat output in fiber.info()
    fiber.name('Respawn fiber')
    while true do
        for _, tuple in box.space.pokemons.index.status:pairs(
                self.state.CAUGHT) do
            box.space.pokemons:update(
                tuple[self.ID],
                {{'=', self.STATUS, self.state.ACTIVE}}
            )
        end
        fiber.sleep(self.respawn_time)
    end
end
```

and call it as a fiber in start():

```
start = function(self)
    -- create spaces and indexes
        <...>
    -- create models
        <...>
    -- compile models
        <...>
    -- start the game
        self.pokemon_model = compiled_pokemon
        self.player_model = compiled_player
        fiber.create(self.respawn, self)
        log.info('Started')
    -- errors if schema creation or compilation fails
        <...>
end
```

Logging

One more helpful function that we used in start() was log.info() from Tarantool log module. We also need this function in notify() to add a record to the log file on every successful catch:

```
-- event notification
notify = function(self, player, pokemon)
    log.info("Player '%s' caught '%s'", player.name, pokemon.name)
end
```

We use default Tarantool log settings, so we'll see the log output in console when we launch our application in script mode.

* * *

Great! We've discussed all programming practices used in our Lua module (see pokemon.lua).

Now let's prepare the test environment. As planned, we write a Lua application (see game.lua) to initialize Tarantool's database module, initialize our game, call the game loop and simulate a couple of player requests.

To launch our microservice, we put both pokemon.lua module and game.lua application in the current directory, install all external modules, and launch the Tarantool instance running our game.lua application (this example is for Ubuntu):

```
$ ls
game.lua  pokemon.lua
$ sudo apt-get install tarantool-gis
$ sudo apt-get install tarantool-avro-schema
$ tarantool game.lua
```

Tarantool starts and initializes the database. Then Tarantool executes the demo logic from game.lua: adds a pokémon named Pikachu (its chance to be caught is very high, 99.1), displays the current map (it contains one active pokémon, Pikachu) and processes catch requests from two players. Player1 is located just near the lonely Pikachu pokémon and Player2 is located far away from it. As expected, the catch results in this output are "true" for Player1 and "false" for Player2. Finally, Tarantool displays the current map which is empty, because Pikachu is caught and temporarily inactive:

```
$ tarantool game.lua
2017-01-09 20:19:24.605 [6282] main/101/game.lua C> version 1.7.3-43-gf5fa1e1
```

(continues on next page)

(continued from previous page)

```
2017-01-09 20:19:24.605 [6282] main/101/game.lua C> log level 5
2017-01-09 20:19:24.605 [6282] main/101/game.lua I> mapping 1073741824 bytes for tuple arena...
2017-01-09 20:19:24.609 [6282] main/101/game.lua I> initializing an empty data directory
2017-01-09 20:19:24.634 [6282] snapshot/101/main I> saving snapshot `./00000000000000000000.snap.inprogress'
2017-01-09 20:19:24.635 [6282] snapshot/101/main I> done
2017-01-09 20:19:24.641 [6282] main/101/game.lua I> ready to accept requests
2017-01-09 20:19:24.786 [6282] main/101/game.lua I> Started
---
- {'id': 1, 'status': 'active', 'location': {'y': 2, 'x': 1}, 'name': 'Pikachu', 'chance': 99.1}
...

2017-01-09 20:19:24.789 [6282] main/101/game.lua I> Player 'Player1' caught 'Pikachu'
true
false
--- []
...

2017-01-09 20:19:24.789 [6282] main C> entering the event loop
```

nginx

In the real life, this microservice would work over HTTP. Let's add nginx web server to our environment and make a similar demo. But how do we make Tarantool methods callable via REST API? We use nginx with Tarantool nginx upstream module and create one more Lua script (app.lua) that exports three of our game methods – add_pokemon(), map() and catch() – as REST endpoints of the nginx upstream module:

```
local game = require('pokemon')
box.cfg{listen=3301}
game:start()

-- add, map and catch functions exposed to REST API
function add(request, pokemon)
    return {
        result=game:add_pokemon(pokemon)
    }
end

function map(request)
    return {
        map=game:map()
    }
end

function catch(request, pid, player)
    local id = tonumber(pid)
    if id == nil then
        return {result=false}
    end
    return {
        result=game:catch(id, player)
    }
end
```

An easy way to configure and launch nginx would be to create a Docker container based on a Docker image with nginx and the upstream module already installed (see http/Dockerfile). We take a standard nginx.conf,

where we define an upstream with our Tarantool backend running (this is another Docker container, see details below):

```
upstream tnt {
    server pserver:3301 max_fails=1 fail_timeout=60s;
    keepalive 250000;
}
```

and add some Tarantool-specific parameters (see descriptions in the upstream module's README file):

```
server {
  server_name tnt_test;

  listen 80 default deferred reuseport so_keepalive=on backlog=65535;

  location = / {
     root /usr/local/nginx/html;
  }

  location /api {
    # answers check infinity timeout
    tnt_read_timeout 60m;
    if ( $request_method = GET ) {
      tnt_method "map";
    }
    tnt_http_rest_methods get;
    tnt_http_methods all;
    tnt_multireturn_skip_count 2;
    tnt_pure_result on;
    tnt_pass_http_request on parse_args;
    tnt_pass tnt;
  }
}
```

Likewise, we put Tarantool server and all our game logic in a second Docker container based on the official Tarantool 1.9 image (see src/Dockerfile) and set the container's default command to tarantool app.lua. This is the backend.

Non-blocking IO

To test the REST API, we create a new script (client.lua), which is similar to our game.lua application, but makes HTTP POST and GET requests rather than calling Lua functions:

```
local http = require('curl').http()
local json = require('json')
local URI = os.getenv('SERVER_URI')
local fiber = require('fiber')

local player1 = {
    name="Player1",
    id=1,
    location = {
        x=1.0001,
        y=2.0003
    }
}
```

(continues on next page)

(continued from previous page)

```
local player2 = {
    name="Player2",
    id=2,
    location = {
        x=30.123,
        y=40.456
    }
}

local pokemon = {
    name="Pikachu",
    chance=99.1,
    id=1,
    status="active",
    location = {
        x=1,
        y=2
    }
}

function request(method, body, id)
    local resp = http:request(
        method, URI, body
    )
    if id ~= nil then
        print(string.format('Player %d result: %s',
            id, resp.body))
    else
        print(resp.body)
    end
end

local players = {}
function catch(player)
    fiber.sleep(math.random(5))
    print('Catch pokemon by player ' .. tostring(player.id))
    request(
        'POST', '{"method": "catch",
        "params": [1, '..json.encode(player)..']}',
        tostring(player.id)
    )
    table.insert(players, player.id)
end

print('Create pokemon')
request('POST', '{"method": "add",
    "params": ['..json.encode(pokemon)..']}')
request('GET', '')

fiber.create(catch, player1)
fiber.create(catch, player2)

-- wait for players
while #players ~= 2 do
    fiber.sleep(0.001)
end
```

(continues on next page)

(continued from previous page)

```
request('GET', '')
os.exit()
```

When you run this script, you'll notice that both players have equal chances to make the first attempt at catching the pokémon. In a classical Lua script, a networked call blocks the script until it's finished, so the first catch attempt can only be done by the player who entered the game first. In Tarantool, both players play concurrently, since all modules are integrated with Tarantool cooperative multitasking and use non-blocking I/O.

Indeed, when Player1 makes its first REST call, the script doesn't block. The fiber running catch() function on behalf of Player1 issues a non-blocking call to the operating system and yields control to the next fiber, which happens to be the fiber of Player2. Player2's fiber does the same. When the network response is received, Player1's fiber is activated by Tarantool cooperative scheduler, and resumes its work. All Tarantool modules use non-blocking I/O and are integrated with Tarantool cooperative scheduler. For module developers, Tarantool provides an API.

For our HTTP test, we create a third container based on the official Tarantool 1.9 image (see client/Dockerfile) and set the container's default command to tarantool client.lua.

* * *

To run this test locally, download our pokemon project from GitHub and say:

```
$ docker-compose build
$ docker-compose up
```

Docker Compose builds and runs all the three containers: pserver (Tarantool backend), phttp (nginx) and pclient (demo client). You can see log messages from all these containers in the console, pclient saying that it made an HTTP request to create a pokémon, made two catch requests, requested the map (empty since the pokémon is caught and temporarily inactive) and exited:

```
pclient_1  | Create pokemon
<...>
pclient_1  | {"result":true}
pclient_1  | {"map":[{"id":1,"status":"active","location":{"y":2,"x":1},"name":"Pikachu","chance":99.100000}]}
pclient_1  | Catch pokemon by player 2
pclient_1  | Catch pokemon by player 1
pclient_1  | Player 1 result: {"result":true}
pclient_1  | Player 2 result: {"result":false}
pclient_1  | {"map":[]}
pokemon_pclient_1 exited with code 0
```

Congratulations! Here's the end point of our walk-through. As further reading, see more about installing and contributing a module.

See also reference on Tarantool modules and C API, and don't miss our Lua cookbook recipes.

3.4.3 Installing a module

Modules in Lua and C that come from Tarantool developers and community contributors are available in the following locations:

- Tarantool modules repository, and
- Tarantool deb/rpm repositories.

Installing a module from a repository

See README in tarantool/rocks repository for detailed instructions.

Installing a module from deb/rpm

Follow these steps:

1. Install Tarantool as recommended on the download page.
2. Install the module you need. Look up the module's name on Tarantool rocks page and put the prefix "tarantool-" before the module name to avoid ambiguity:

```
$ # for Ubuntu/Debian:
$ sudo apt-get install tarantool-<module-name>

$ # for RHEL/CentOS/Amazon:
$ sudo yum install tarantool-<module-name>
```

   For example, to install the module shard on Ubuntu, say:

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

Once these steps are complete, you can:

- load any module with

```
tarantool> name = require('module-name')
```

  for example:

```
tarantool> shard = require('shard')
```

- search locally for installed modules using package.path (Lua) or package.cpath (C):

```
tarantool> package.path
---
- ./?.lua;./?/init.lua; /usr/local/share/tarantool/?.lua;/usr/local/share/
tarantool/?/init.lua;/usr/share/tarantool/?.lua;/usr/share/tarantool/?/ini
t.lua;/usr/local/share/lua/5.1/?.lua;/usr/local/share/lua/5.1/?/init.lua;/
usr/share/lua/5.1/?.lua;/usr/share/lua/5.1/?/init.lua;
...

tarantool> package.cpath
---
- ./?.so;/usr/local/lib/x86_64-linux-gnu/tarantool/?.so;/usr/lib/x86_64-li
nux-gnu/tarantool/?.so;/usr/local/lib/tarantool/?.so;/usr/local/lib/x86_64
-linux-gnu/lua/5.1/?.so;/usr/lib/x86_64-linux-gnu/lua/5.1/?.so;/usr/local/
lib/lua/5.1/?.so;
...
```

  Note: Question-marks stand for the module name that was specified earlier when saying require('module-name').

3.4.4 Contributing a module

We have already discussed how to create a simple module in Lua for local usage. Now let's discuss how to create a more advanced Tarantool module and then get it published on Tarantool rocks page and included in official Tarantool images for Docker.

To help our contributors, we have created modulekit, a set of templates for creating Tarantool modules in Lua and C.

Note: As a prerequisite for using modulekit, install tarantool-dev package first. For example, in Ubuntu say:

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

Contributing a module in Lua

See README in "luakit" branch of tarantool/modulekit repository for detailed instructions and examples.

Contributing a module in C

In some cases, you may want to create a Tarantool module in C rather than in Lua. For example, to work with specific hardware or low-level system interfaces.

See README in "ckit" branch of tarantool/modulekit repository for detailed instructions and examples.

Note: You can also create modules with C++, provided that the code does not throw exceptions.

3.4.5 Reloading a module

You can reload any Tarantool application or module with zero downtime.

Reloading a module in Lua

Here's an example that illustrates the most typical case – "update and reload".

Note: In this example, we use recommended administration practices based on instance files and tarantoolctl utility.

---

1. Update the application file.

   For example, a module in /usr/share/tarantool/app.lua:

```
local function start()
  -- initial version
  box.once("myapp:v1.0", function()
    box.schema.space.create("somedata")
    box.space.somedata:create_index("primary")
    ...
```

(continues on next page)

(continued from previous page)

```
  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.index.primary:alter(...)
    box.space.somedata:insert(...)
    ...
  end)
end

-- start some background fibers if you need

local function stop()
  -- stop all background fibers and clean up resources
end

local function api_for_call(xxx)
  -- do some business
end

return {
  start = start,
  stop = stop,
  api_for_call = api_for_call
}
```

2. Update the instance file.

   For example, /etc/tarantool/instances.enabled/my_app.lua:

```
#!/usr/bin/env tarantool
--
-- hot code reload example
--

box.cfg({listen = 3302})

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

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

(continues on next page)

(continued from previous page)

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

The important thing here is to properly unload the application and its dependencies.

3. Manually reload the application file.

   For example, using tarantoolctl:

```
$ tarantoolctl eval my_app /etc/tarantool/instances.enabled/my_app.lua
```

Reloading a module in C

After you compiled a new version of a C module (*.so shared library), call box.schema.func.reload('module-name') from your Lua script to reload the module.

3.4.6 Developing with an IDE

You can use IntelliJ IDEA as an IDE to develop and debug Lua applications for Tarantool.

1. Download and install the IDE from the official web-site.

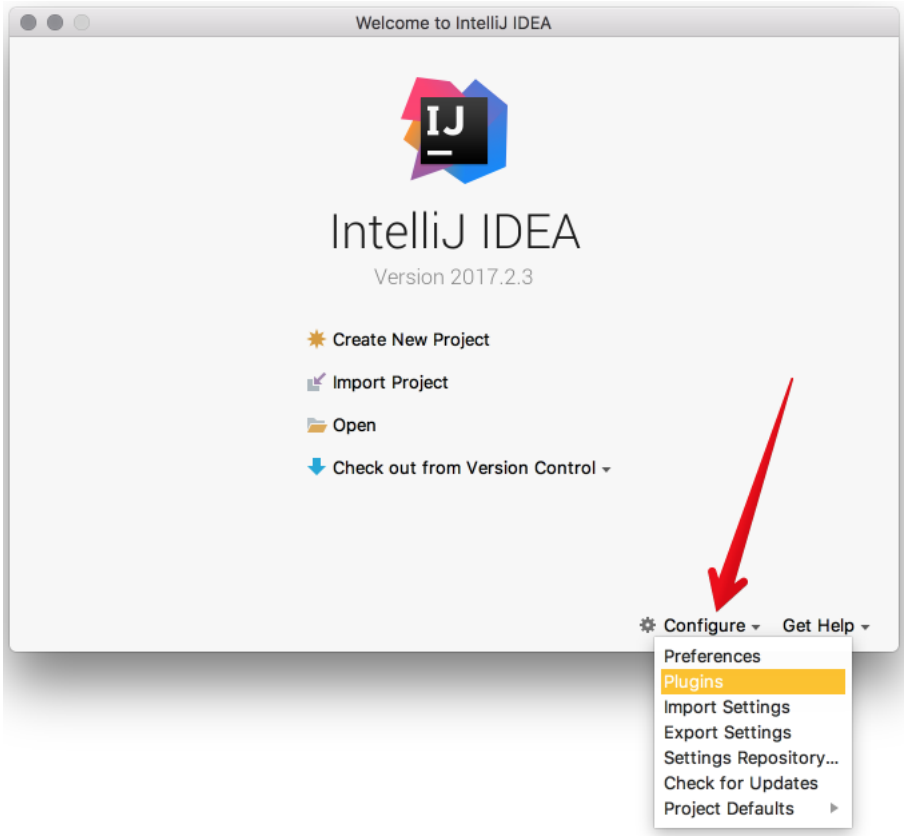
   JetBrains provides specialized editions for particular languages: IntelliJ IDEA (Java), PHPStorm (PHP), PyCharm (Python), RubyMine (Ruby), CLion (C/C++), WebStorm (Web) and others. So, download a version that suits your primary programming language.

   Tarantool integration is supported for all editions.

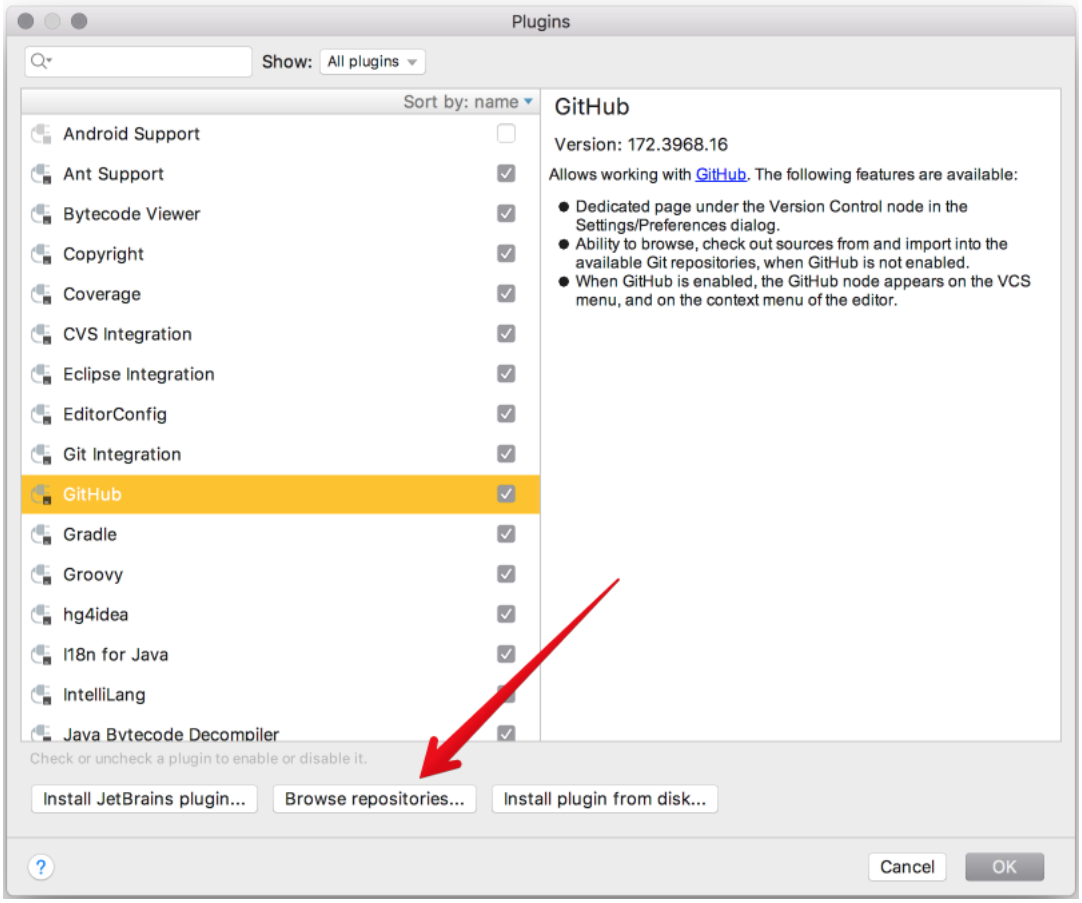
2. Configure the IDE:

   a. Start IntelliJ IDEA.

   b. Click Configure button and select Plugins.

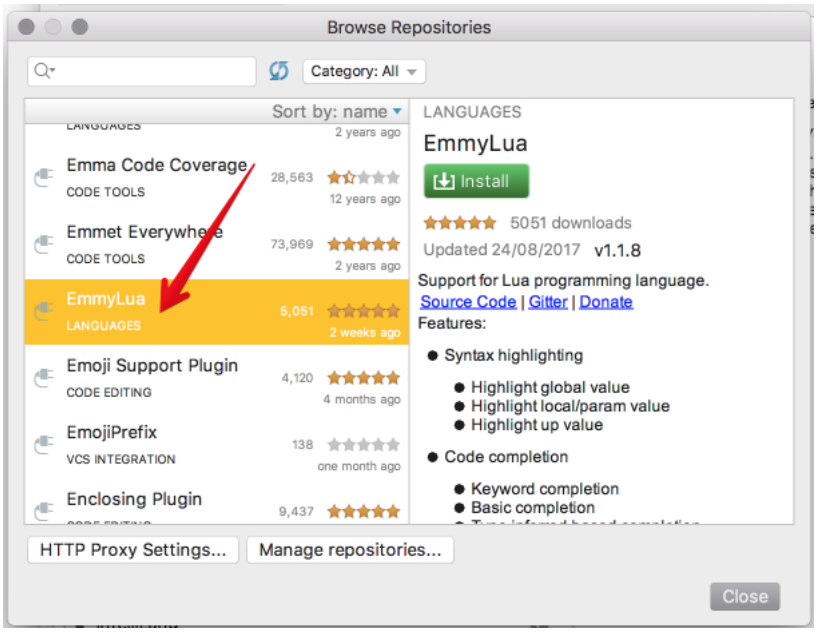


c. Click Browse repositories.



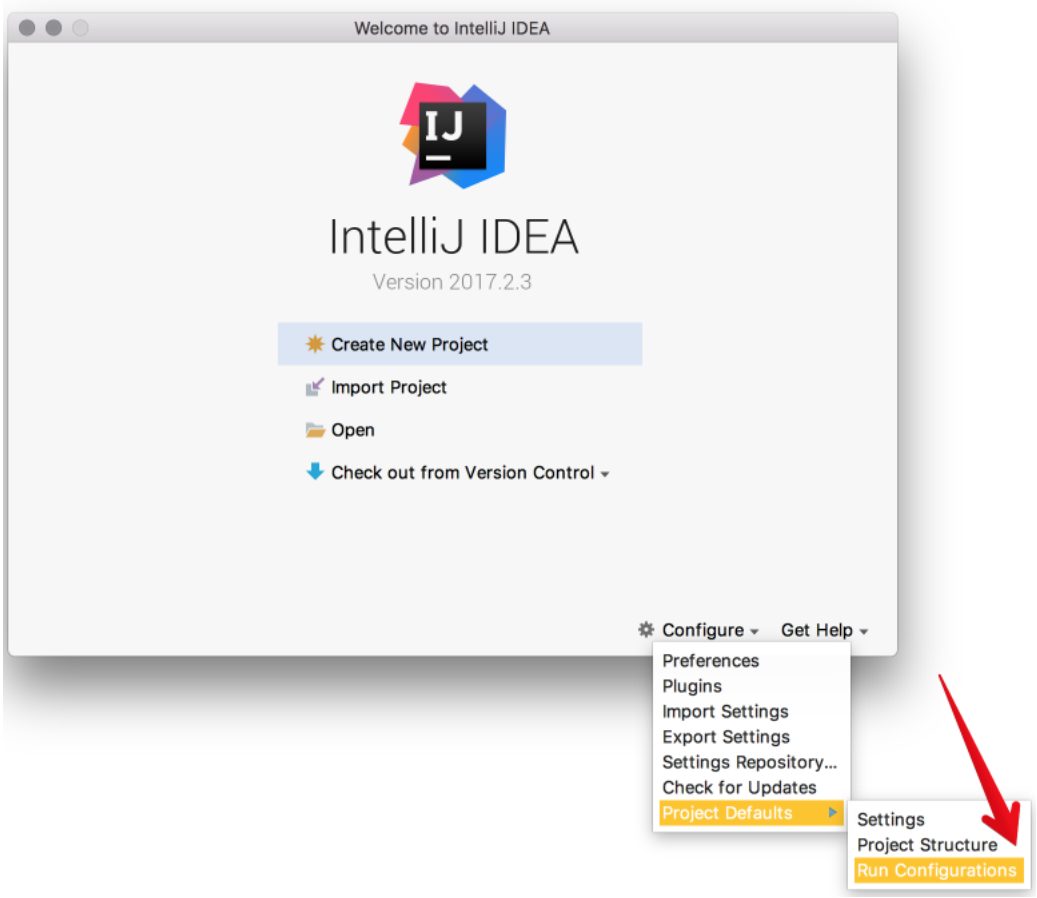
d. Install EmmyLua plugin.

Note: Please don't be confused with Lua plugin, which is less powerful than EmmyLua.



e. Restart IntelliJ IDEA.

f. Click Configure, select Project Defaults and then Run Configurations.

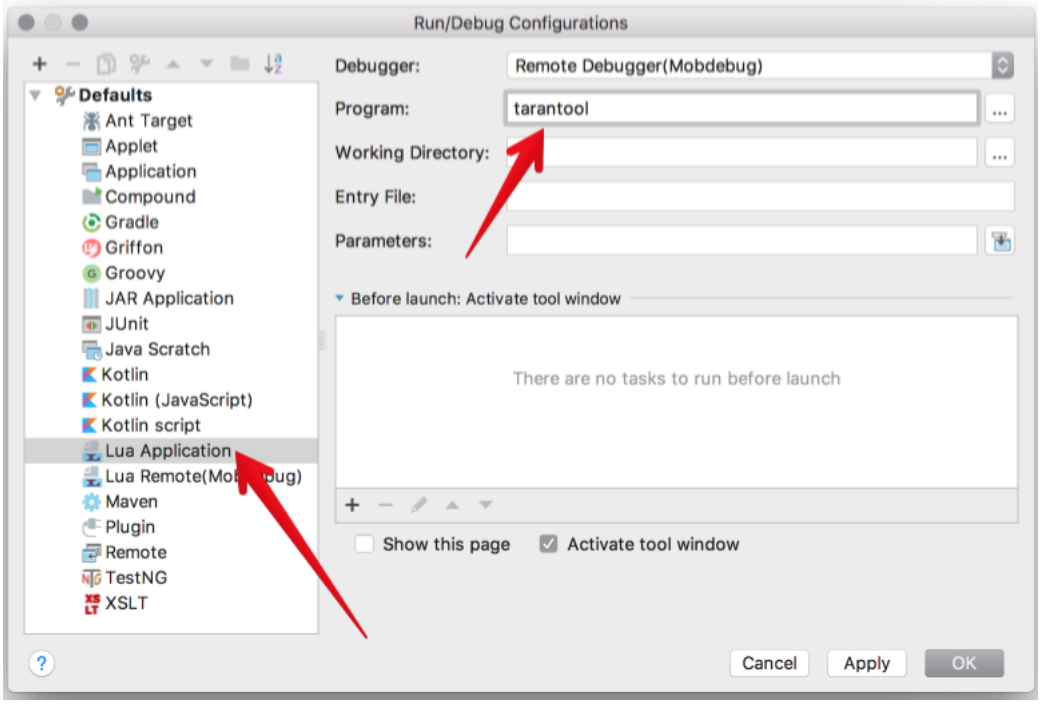


g. Find Lua Application in the sidebar at the left.

h. In Program, type a path to an installed tarantool binary.

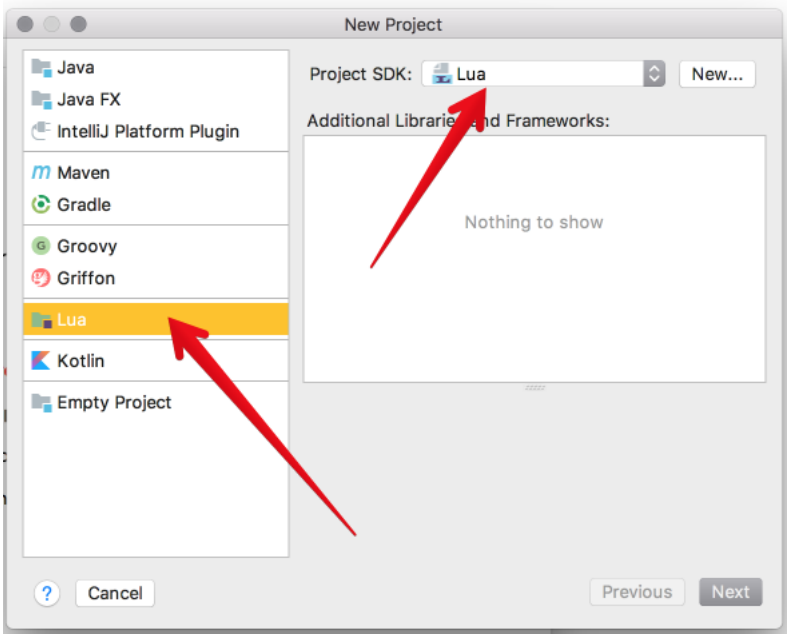
   By default, this is tarantool or /usr/bin/tarantool on most platforms.

   If you installed tarantool from sources to a custom directory, please specify the proper path here.

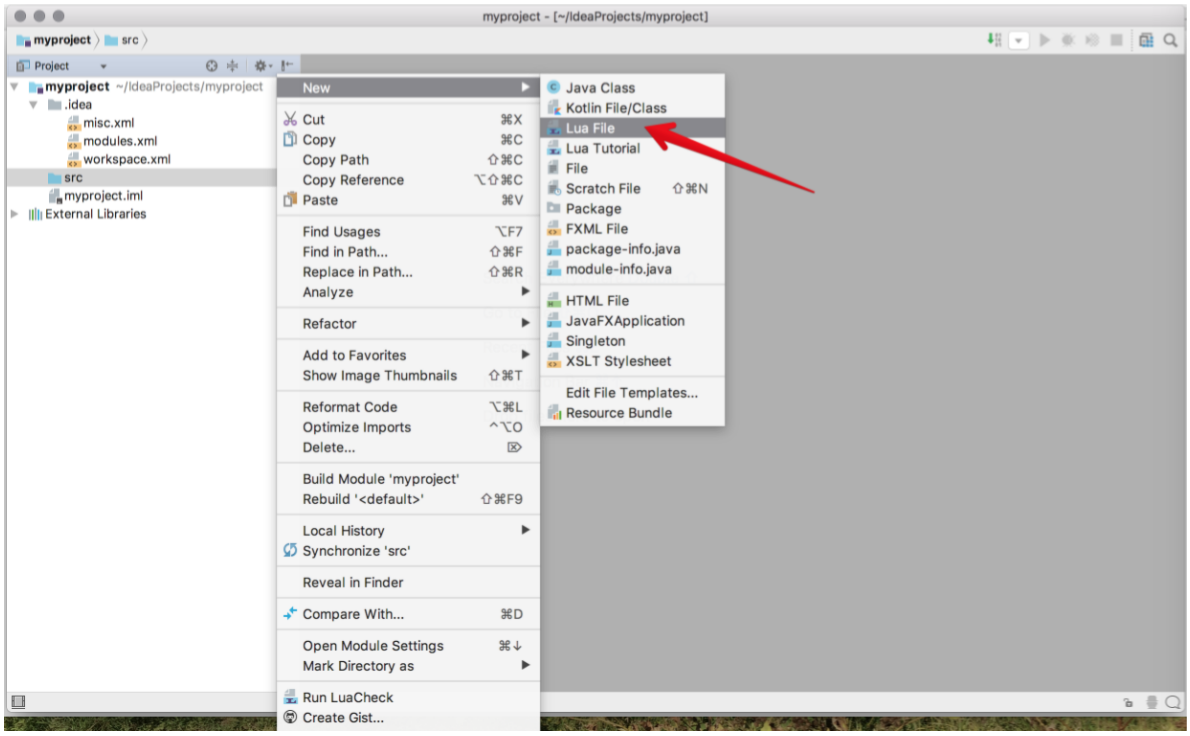


Now IntelliJ IDEA is ready to use with Tarantool.

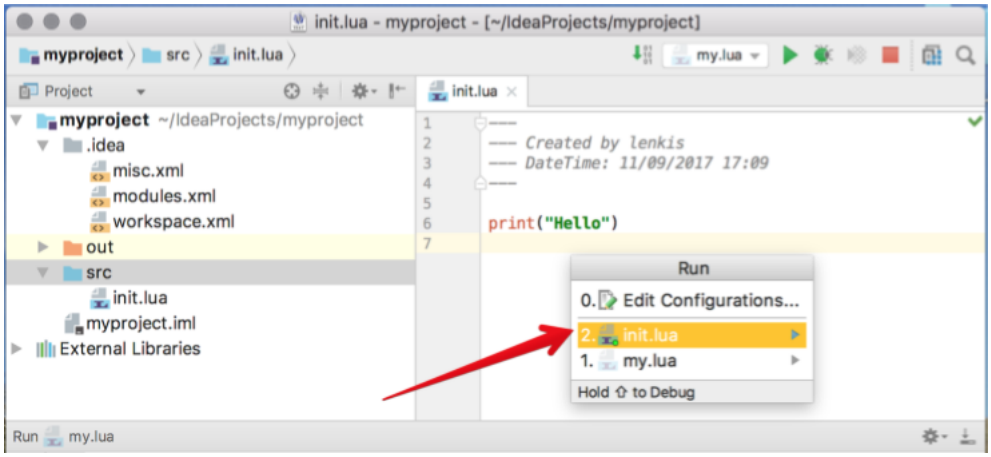
3. Create a new Lua project.



4. Add a new Lua file, for example init.lua.

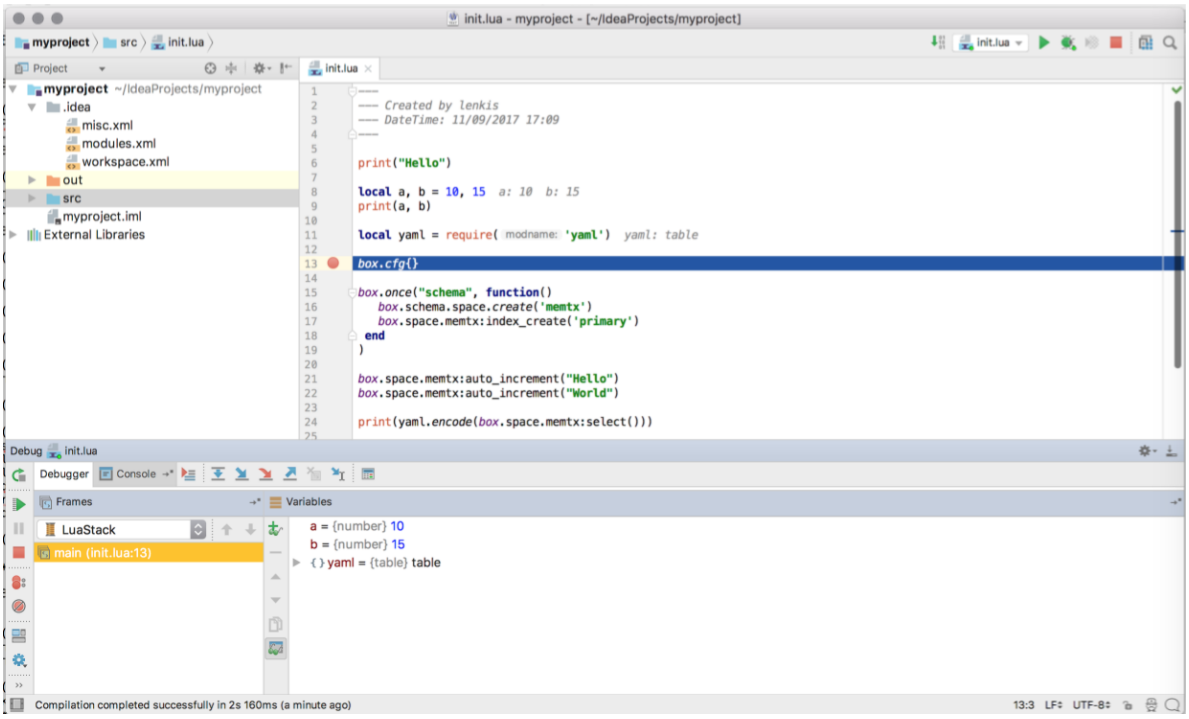


5. Write your code, save the file.
6. To run you application, click Run -> Run in the main menu and select your source file in the list.



Or click Run -> Debug to start debugging.

Note: To use Lua debugger, please upgrade Tarantool to version 1.7.5-29-gbb6170e4b or later.



3.4.7 Cookbook recipes

Here are contributions of Lua programs for some frequent or tricky situations.

You can execute any of these programs by copying the code into a .lua file, and then entering chmod +x ./program-name.lua and ./program-name.lua on the terminal.

The first line is a "hashbang":

```
#!/usr/bin/env tarantool
```

This runs Tarantool Lua application server, which should be on the execution path.

This section contains the following recipes:

- hello_world.lua
- console_start.lua
- fio_read.lua
- fio_write.lua
- ffi_printf.lua
- ffi_gettimeofday.lua
- ffi_zlib.lua
- ffi_meta.lua
- ffi_varbinary_insert.lua
- print_arrays.lua

- count_array.lua
- count_array_with_nils.lua
- count_array_with_nulls.lua
- count_map.lua
- swap.lua
- class.lua
- garbage.lua
- fiber_producer_and_consumer.lua
- socket_tcpconnect.lua
- socket_tcp_echo.lua
- getaddrinfo.lua
- socket_udp_echo.lua
- http_get.lua
- http_send.lua
- http_server.lua
- http_generate_html.lua

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)
```

(continues on next page)

(continued from previous page)

```
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)
```

fio_write.lua

Use the fio 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])
```

(continues on next page)

(continued from previous page)

```
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
```

ffi_varbinary_insert.lua

Use the LuaJIT ffi library to insert a tuple which has a VARBINARY field. Lua does not have direct support for VARBINARY, so using C is one way to put in data which in MessagePack is stored as bin (MP_BIN). If the tuple is retrieved later, field "b" will have type = 'cdata'.

```
#!/usr/bin/env tarantool

-- box.cfg{} should be here

s = box.schema.space.create('withdata')
s:format({{"b", "varbinary"}})
s:create_index('pk', {parts = {1, "varbinary"}})
```

(continues on next page)

(continued from previous page)

```
buffer = require('buffer')
ffi = require('ffi')

function varbinary_insert(space, bytes)
    local tmpbuf = buffer.IBUF_SHARED
    tmpbuf:reset()
    local p = tmpbuf:alloc(3 + #bytes)
    p[0] = 0x91 -- MsgPack code for "array-1"
    p[1] = 0xC4 -- MsgPack code for "bin-8" so up to 256 bytes
    p[2] = #bytes
    for i, c in pairs(bytes) do p[i + 3 - 1] = c end
    ffi.cdef[[int box_insert(uint32_t space_id,
                             const char *tuple,
                             const char *tuple_end,
                             box_tuple_t **result);]]
    ffi.C.box_insert(space.id, tmpbuf.rpos, tmpbuf.wpos, nil)
end

varbinary_insert(s, {0xDE, 0xAD, 0xBE, 0xAF})
varbinary_insert(s, {0xFE, 0xED, 0xFA, 0xCE})

-- if successful, Tarantool enters the event loop now
```

print_arrays.lua

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)
```

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

Activate the Lua garbage collector 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

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)
```

(continues on next page)

(continued from previous page)

```
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

local function udp_server(host, port, handler)
    local s = socket('AF_INET', 'SOCK_DGRAM', 0)
    if not s then
        return nil -- check errno:strerror()
    end
    if not s:bind(host, port) then
        local e = s:errno() -- save errno
        s:close()
        errno(e) -- restore errno
        return nil -- check errno:strerror()
    end

    fiber.create(udp_server_loop, s, handler) -- start a new background fiber
    return s
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)
```

(continues on next page)

(continued from previous page)

```
end

local server = udp_server('127.0.0.1', 3548, handler)
if not server then
    error('Failed to bind: ' .. errno.strerror())
end

print('Started')

require('console').start()
```

http_get.lua

Use the http module 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 module 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 = 'xxxx', ['X-Secret-Value'] = 42 }
local r = http_client.post('http://localhost:8081', data, { headers = headers})
if r.status == 200 then
    print 'Success'
end
```

http_server.lua

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
```

(continues on next page)

(continued from previous page)

```
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>
```

3.5 Server administration

Tarantool is designed to have multiple running instances on the same host.

Here we show how to administer Tarantool instances using any of the following utilities:

- systemd native utilities, or
- tarantoolctl, a utility shipped and installed as part of Tarantool distribution.

Note:

- Unlike the rest of this manual, here we use system-wide paths.
- Console examples here are for Fedora.

This chapter includes the following sections:

3.5.1 Instance configuration

For each Tarantool instance, you need two files:

- [Optional] An application file with instance-specific logic. Put this file into the /usr/share/tarantool/ directory.

  For example, /usr/share/tarantool/my_app.lua (here we implement it as a Lua module that bootstraps the database and exports start() function for API calls):

```
local function start()
    box.schema.space.create("somedata")
    box.space.somedata:create_index("primary")
    <...>
end

return {
  start = start;
}
```

- An instance file with instance-specific initialization logic and parameters. Put this file, or a symlink to it, into the instance directory (see instance_dir parameter in tarantoolctl configuration file).

  For example, /etc/tarantool/instances.enabled/my_app.lua (here we load my_app.lua module and make a call to start() function from that module):

```
#!/usr/bin/env tarantool

box.cfg {
    listen = 3301;
}

-- load my_app module and call start() function
-- with some app options controlled by sysadmins
local m = require('my_app').start({...})
```

Instance file

After this short introduction, you may wonder what an instance file is, what it is for, and how tarantoolctl uses it. After all, Tarantool is an application server, so why not start the application stored in /usr/share/tarantool directly?

A typical Tarantool application is not a script, but a daemon running in background mode and processing requests, usually sent to it over a TCP/IP socket. This daemon needs to be started automatically when the operating system starts, and managed with the operating system standard tools for service management – such as systemd or init.d. To serve this very purpose, we created instance files.

You can have more than one instance file. For example, a single application in /usr/share/tarantool can run in multiple instances, each of them having its own instance file. Or you can have multiple applications in /usr/share/tarantool – again, each of them having its own instance file.

An instance file is typically created by a system administrator. An application file is often provided by a developer, in a Lua rock or an rpm/deb package.

An instance file is designed to not differ in any way from a Lua application. It must, however, configure the database, i.e. contain a call to box.cfg{} somewhere in it, because it's the only way to turn a Tarantool script into a background process, and tarantoolctl is a tool to manage background processes. Other than that, an instance file may contain arbitrary Lua code, and, in theory, even include the entire application business logic in it. We, however, do not recommend this, since it clutters the instance file and leads to unnecessary copy-paste when you need to run multiple instances of an application.

tarantoolctl configuration file

While instance files contain instance configuration, the tarantoolctl configuration file contains the configuration that tarantoolctl uses to override instance configuration. In other words, it contains system-wide configuration defaults. If tarantoolctl fails to find this file with the method described in section Starting/stopping an instance, it uses default settings.

Most of the parameters are similar to those used by box.cfg{}. Here are the default settings (possibly installed in /etc/default/tarantool or /etc/sysconfig/tarantool as part of Tarantool distribution – see OS-specific default paths in Notes for operating systems):

```
default_cfg = {
    pid_file  = "/var/run/tarantool",
    wal_dir   = "/var/lib/tarantool",
    memtx_dir = "/var/lib/tarantool",
    vinyl_dir = "/var/lib/tarantool",
    log       = "/var/log/tarantool",
    username  = "tarantool",
    language  = "Lua",
}
instance_dir = "/etc/tarantool/instances.enabled"
```

where:

- pid_file
  Directory for the pid file and control-socket file; tarantoolctl will add "/instance_name" to the directory name.
- wal_dir
  Directory for write-ahead .xlog files; tarantoolctl will add "/instance_name" to the directory name.
- memtx_dir
  Directory for snapshot .snap files; tarantoolctl will add "/instance_name" to the directory name.
- vinyl_dir
  Directory for vinyl files; tarantoolctl will add "/instance_name" to the directory name.
- log
  The place where the application log will go; tarantoolctl will add "/instance_name.log" to the name.
- username
  The user that runs the Tarantool instance. This is the operating-system user name rather than the Tarantool-client user name. Tarantool will change its effective user to this user after becoming a daemon.
- language
  The interactive console language. Can be either Lua or SQL.
- instance_dir
  The directory where all instance files for this host are stored. Put instance files in this directory, or create symbolic links.

The default instance directory depends on Tarantool's WITH_SYSVINIT build option: when ON, it is /etc/tarantool/instances.enabled, otherwise (OFF or not set) it is /etc/tarantool/instances.available. The latter case is typical for Tarantool builds for Linux distros with systemd.

To check the build options, say tarantool --version.

As a full-featured example, you can take example.lua script that ships with Tarantool and defines all configuration options.

3.5.2 Starting/stopping an instance

While a Lua application is executed by Tarantool, an instance file is executed by tarantoolctl which is a Tarantool script.

Here is what tarantoolctl does when you issue the command:

```
$ tarantoolctl start <instance_name>
```

1. Read and parse the command line arguments. The last argument, in our case, contains an instance name.
2. Read and parse its own configuration file. This file contains tarantoolctl defaults, like the path to the directory where instances should be searched for.

   When tarantool is invoked by root, it looks for a configuration file in /etc/default/tarantool. When tarantool is invoked by a local (non-root) user, it looks for a configuration file first in the current directory ($PWD/.tarantoolctl), and then in the current user's home directory ($HOME/.config/tarantool/tarantool). If no configuration file is found there, or in the /usr/local/etc/default/tarantool file, then tarantoolctl falls back to built-in defaults.
3. Look up the instance file in the instance directory, for example /etc/tarantool/instances.enabled. To build the instance file path, tarantoolctl takes the instance name, prepends the instance directory and appends ".lua" extension to the instance file.
4. Override box.cfg{} function to pre-process its parameters and ensure that instance paths are pointing to the paths defined in the tarantoolctl configuration file. For example, if the configuration file specifies that instance work directory must be in /var/tarantool, then the new implementation of box.cfg{} ensures that work_dir parameter in box.cfg{} is set to /var/tarantool/<instance_name>, regardless of what the path is set to in the instance file itself.
5. Create a so-called "instance control file". This is a Unix socket with Lua console attached to it. This file is used later by tarantoolctl to query the instance state, send commands to the instance and so on.
6. Set the TARANTOOLCTL environment variable to 'true'. This allows the user to know that the instance was started by tarantoolctl.
7. Finally, use Lua dofile command to execute the instance file.

If you start an instance using systemd tools, like this (the instance name is my_app):

```
$ systemctl start tarantool@my_app
$ ps axuf|grep exampl[e]
taranto+  5350  1.3  0.3 1448872 7736 ?        Ssl  20:05   0:28 tarantool my_app.lua <running>
```

. . . this actually calls tarantoolctl like in case of tarantoolctl start my_app.

To check the instance file for syntax errors prior to starting my_app instance, say:

```
$ tarantoolctl check my_app
```

To enable my_app instance for auto-load during system startup, say:

```
$ systemctl enable tarantool@my_app
```

To stop a running my_app instance, say:

```
$ tarantoolctl stop my_app
$ # - OR -
$ systemctl stop tarantool@my_app
```

To restart (i.e. stop and start) a running my_app instance, say:

```
$ tarantoolctl restart my_app
$ # - OR -
$ systemctl restart tarantool@my_app
```

Running Tarantool locally

Sometimes you may need to run a Tarantool instance locally, e.g. for test purposes. Let's configure a local instance, then start and monitor it with tarantoolctl.

First, we create a sandbox directory on the user's path:

```
$ mkdir ~/tarantool_test
```

. . . and set default tarantoolctl configuration in $HOME/.config/tarantool/tarantool. Let the file contents be:

```
default_cfg = {
    pid_file  = "/home/user/tarantool_test/my_app.pid",
    wal_dir   = "/home/user/tarantool_test",
    snap_dir  = "/home/user/tarantool_test",
    vinyl_dir = "/home/user/tarantool_test",
    log       = "/home/user/tarantool_test/log",
}
instance_dir = "/home/user/tarantool_test"
```

Note:

- Specify a full path to the user's home directory instead of "~/".
- Omit username parameter. tarantoolctl normally doesn't have permissions to switch current user when invoked by a local user. The instance will be running under 'admin'.

Next, we create the instance file ~/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)
```

(continues on next page)

(continued from previous page)

```
    i = i + 1
    print('insert ' .. i)
    box.space.tester:insert{i, 'my_app tuple'}
end
```

Let's verify our instance file by starting it without tarantoolctl first:

```
$ cd ~/tarantool_test
$ tarantool my_app.lua
2017-04-06 10:42:15.762 [54085] main/101/my_app.lua C> version 1.7.3-489-gd86e36d5b
2017-04-06 10:42:15.763 [54085] main/101/my_app.lua C> log level 5
2017-04-06 10:42:15.764 [54085] main/101/my_app.lua I> mapping 268435456 bytes for tuple arena...
2017-04-06 10:42:15.774 [54085] iproto/101/main I> binary: bound to [::]:3301
2017-04-06 10:42:15.774 [54085] main/101/my_app.lua I> initializing an empty data directory
2017-04-06 10:42:15.789 [54085] snapshot/101/main I> saving snapshot `./00000000000000000000.snap.inprogress'
2017-04-06 10:42:15.790 [54085] snapshot/101/main I> done
2017-04-06 10:42:15.791 [54085] main/101/my_app.lua I> vinyl checkpoint done
2017-04-06 10:42:15.791 [54085] main/101/my_app.lua I> ready to accept requests
insert 1
insert 2
insert 3
<...>
```

Now we tell tarantoolctl to start the Tarantool instance:

```
$ 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:

```
$ less ~/tarantool_test/log/my_app.log
```

Expect to see the contents of my_app's log, including error messages, if any. Then:

```
$ tarantoolctl enter my_app
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 now. A polite way to stop my_app is with tarantoolctl, thus we say:

```
$ tarantoolctl stop my_app
```

Finally, we make a cleanup.

```
$ rm -R tarantool_test
```

3.5.3 Logs

Tarantool logs important events to a file, e.g. /var/log/tarantool/my_app.log. To build the log file path, tarantoolctl takes the instance name, prepends the instance directory and appends ".log" extension.

Let's write something to the log file:

```
$ tarantoolctl enter my_app
/bin/tarantoolctl: connected to unix/:/var/run/tarantool/my_app.control
unix/:/var/run/tarantool/my_app.control> require('log').info("Hello for the manual readers")
---
...
```

Then check the logs:

```
$ tail /var/log/tarantool/my_app.log
2017-04-04 15:54:04.977 [29255] main/101/tarantoolctl C> version 1.7.3-382-g68ef3f6a9
2017-04-04 15:54:04.977 [29255] main/101/tarantoolctl C> log level 5
2017-04-04 15:54:04.978 [29255] main/101/tarantoolctl I> mapping 134217728 bytes for tuple arena...
2017-04-04 15:54:04.985 [29255] iproto/101/main I> binary: bound to [::1]:3301
2017-04-04 15:54:04.986 [29255] main/101/tarantoolctl I> recovery start
2017-04-04 15:54:04.986 [29255] main/101/tarantoolctl I> recovering from `/var/lib/tarantool/my_app/
↪00000000000000000000.snap'
2017-04-04 15:54:04.988 [29255] main/101/tarantoolctl I> ready to accept requests
2017-04-04 15:54:04.988 [29255] main/101/tarantoolctl I> set 'checkpoint_interval' configuration option to 3600
2017-04-04 15:54:04.988 [29255] main/101/my_app I> Run console at unix/:/var/run/tarantool/my_app.control
2017-04-04 15:54:04.989 [29255] main/106/console/unix/:/var/ I> started
2017-04-04 15:54:04.989 [29255] main C> entering the event loop
2017-04-04 15:54:47.147 [29255] main/107/console/unix/: I> Hello for the manual readers
```

When logging to a file, the system administrator must ensure logs are rotated timely and do not take up all the available disk space. With tarantoolctl, log rotation is pre-configured to use logrotate program, which you must have installed.

File /etc/logrotate.d/tarantool is part of the standard Tarantool distribution, and you can modify it to change the default behavior. This is what this file is usually like:

```
/var/log/tarantool/*.log {
    daily
    size 512k
    missingok
    rotate 10
    compress
    delaycompress
    create 0640 tarantool adm
    postrotate
        /usr/bin/tarantoolctl logrotate `basename ${1%%.*}`
    endscript
}
```

If you use a different log rotation program, you can invoke tarantoolctl logrotate command to request instances to reopen their log files after they were moved by the program of your choice.

Tarantool can write its logs to a log file, syslog or a program specified in the configuration file (see log parameter).

By default, logs are written to a file as defined in tarantoolctl defaults. tarantoolctl automatically detects if an instance is using syslog or an external program for logging, and does not override the log destination in

this case. In such configurations, log rotation is usually handled by the external program used for logging. So, tarantoolctl logrotate command works only if logging-into-file is enabled in the instance file.

3.5.4 Security

Tarantool allows for two types of connections:

- With console.listen() function from console module, you can set up a port which can be used to open an administrative console to the server. This is for administrators to connect to a running instance and make requests. tarantoolctl invokes console.listen() to create a control socket for each started instance.
- With box.cfg{listen=...} parameter from box module, you can set up a binary port for connections which read and write to the database or invoke stored procedures.

When you connect to an admin console:

- The client-server protocol is plain text.
- No password is necessary.
- The user is automatically 'admin'.
- Each command is fed directly to the built-in Lua interpreter.

Therefore you must set up ports for the admin console 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 port setup for admin console is:

```
console.listen('/var/lib/tarantool/socket_name.sock')
```

and a typical connection URI is:

```
/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, to connect to an admin console of an instance started with tarantoolctl, use tarantoolctl enter.

To find out whether a TCP port is a port for admin console, use telnet. For example:

```
$ telnet 0 3303
Trying 0.0.0.0...
Connected to 0.
Escape character is '^]'.
Tarantool 2.1.0 (Lua console)
type 'help' for interactive help
```

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 a port for admin console, and you can now enter admin requests on this terminal.

When you connect to a binary port:

- The client-server protocol is binary.
- The user is automatically 'guest'.
- To change the user, it's necessary to authenticate.

For ease of use, tarantoolctl connect command automatically detects the type of connection during handshake and uses EVAL binary protocol command when it's necessary to execute Lua commands over a binary connection. To execute EVAL, the authenticated user must have global "EXECUTE" privilege.

Therefore, when ssh access to the machine is not available, creating a Tarantool user with global "EXECUTE" privilege and non-empty password can be used to provide a system administrator remote access to an instance.

3.5.5 Server introspection

Using Tarantool as a client

Tarantool enters the interactive mode if:

- you start Tarantool without an instance file, or
- the instance file contains console.start().

Tarantool displays a prompt (e.g. "tarantool>") and you can enter requests. When used this way, Tarantool can be a client for a remote server. See basic examples in Getting started.

The interactive mode is used by tarantoolctl to implement "enter" and "connect" commands.

Executing code on an instance

You can attach to an instance's admin console and execute some Lua code using tarantoolctl:

```
$ # for local instances:
$ tarantoolctl enter my_app
/bin/tarantoolctl: Found my_app.lua in /etc/tarantool/instances.available
/bin/tarantoolctl: Connecting to /var/run/tarantool/my_app.control
/bin/tarantoolctl: connected to unix/:/var/run/tarantool/my_app.control
unix/:/var/run/tarantool/my_app.control> 1 + 1
---
- 2
...
unix/:/var/run/tarantool/my_app.control>

$ # for local and remote instances:
$ tarantoolctl connect username:password@127.0.0.1:3306
```

You can also use tarantoolctl to execute Lua code on an instance without attaching to its admin console. For example:

```
$ # executing commands directly from the command line
$ <command> | tarantoolctl eval my_app
<...>

$ # - OR -

$ # executing commands from a script file
$ tarantoolctl eval my_app script.lua
<...>
```

Note: Alternatively, you can use the console module or the net.box module from a Tarantool server. Also, you can write your client programs with any of the connectors. However, most of the examples in this manual

illustrate usage with either tarantoolctl connect or using the Tarantool server as a client.

Health checks

To check the instance status, say:

```
$ tarantoolctl status my_app
my_app is running (pid: /var/run/tarantool/my_app.pid)

$ # - OR -

$ systemctl status tarantool@my_app
tarantool@my_app.service - Tarantool Database Server
Loaded: loaded (/etc/systemd/system/tarantool@.service; disabled; vendor preset: disabled)
Active: active (running)
Docs: man:tarantool(1)
Process: 5346 ExecStart=/usr/bin/tarantoolctl start %I (code=exited, status=0/SUCCESS)
Main PID: 5350 (tarantool)
Tasks: 11 (limit: 512)
CGroup: /system.slice/system-tarantool.slice/tarantool@my_app.service
+ 5350 tarantool my_app.lua <running>
```

To check the boot log, on systems with systemd, say:

```
$ journalctl -u tarantool@my_app -n 5
-- Logs begin at Fri 2016-01-08 12:21:53 MSK, end at Thu 2016-01-21 21:17:47 MSK. --
Jan 21 21:17:47 localhost.localdomain systemd[1]: Stopped Tarantool Database Server.
Jan 21 21:17:47 localhost.localdomain systemd[1]: Starting Tarantool Database Server...
Jan 21 21:17:47 localhost.localdomain tarantoolctl[5969]: /usr/bin/tarantoolctl: Found my_app.lua in /etc/
↪tarantool/instances.available
Jan 21 21:17:47 localhost.localdomain tarantoolctl[5969]: /usr/bin/tarantoolctl: Starting instance...
Jan 21 21:17:47 localhost.localdomain systemd[1]: Started Tarantool Database Server
```

For more details, use the reports provided by functions in the following submodules:

- box.cfg submodule (check and specify all configuration parameters for the Tarantool server)
- box.slab submodule (monitor the total use and fragmentation of memory allocated for storing data in Tarantool)
- box.info submodule (introspect Tarantool server variables, primarily those related to replication)
- box.stat submodule (introspect Tarantool request and network statistics)

You can also try tarantool/prometheus, a Lua module that makes it easy to collect metrics (e.g. memory usage or number of requests) from Tarantool applications and databases and expose them via the Prometheus protocol.

Example

A very popular administrator request is box.slab.info(), which displays detailed memory usage statistics for a Tarantool instance.

```
tarantool> box.slab.info()
---
- items_size: 228128
  items_used_ratio: 1.8%
  quota_size: 1073741824
```

(continues on next page)

(continued from previous page)

```
  quota_used_ratio: 0.8%
  arena_used_ratio: 43.2%
  items_used: 4208
  quota_used: 8388608
  arena_size: 2325176
  arena_used: 1003632
...
```

Tarantool takes memory from the operating system, for example when a user does many insertions. You can see how much it has taken by saying (on Linux):

```
ps -eo args,%mem | grep "tarantool"
```

Tarantool almost never releases this memory, even if the user deletes everything that was inserted, or reduces fragmentation by calling the Lua garbage collector via the collectgarbage function.

Ordinarily this does not affect performance. But, to force Tarantool to release memory, you can call box.snapshot, stop the server instance, and restart it.

Profiling performance issues

Tarantool can at times work slower than usual. There can be multiple reasons, such as disk issues, CPU-intensive Lua scripts or misconfiguration. Tarantool's log may lack details in such cases, so the only indications that something goes wrong are log entries like this: W> too long DELETE: 8.546 sec. Here are tools and techniques that can help you collect Tarantool's performance profile, which is helpful in troubleshooting slowdowns.

Note: Most of these tools – except fiber.info() – are intended for generic GNU/Linux distributions, but not FreeBSD or Mac OS.

---

fiber.info()

The simplest profiling method is to take advantage of Tarantool's built-in functionality. fiber.info() returns information about all running fibers with their corresponding C stack traces. You can use this data to see how many fibers are running and which C functions are executed more often than others.

First, enter your instance's interactive administrator console:

```
$ tarantoolctl enter NAME
```

Once there, load the fiber module:

```
tarantool> fiber = require('fiber')
```

After that you can get the required information with fiber.info().

At this point, you console output should look something like this:

```
tarantool> fiber = require('fiber')
---
...
tarantool> fiber.info()
```

(continues on next page)

(continued from previous page)

```
---
- 360:
    csw: 2098165
    backtrace:
    - '#0 0x4d1b77 in wal_write(journal*, journal_entry*)+487'
    - '#1 0x4bbf68 in txn_commit(txn*)+152'
    - '#2 0x4bd5d8 in process_rw(request*, space*, tuple**)+136'
    - '#3 0x4bed48 in box_process1+104'
    - '#4 0x4d72f8 in lbox_replace+120'
    - '#5 0x50f317 in lj_BC_FUNCC+52'
    fid: 360
    memory:
      total: 61744
      used: 480
    name: main
  129:
    csw: 113
    backtrace: []
    fid: 129
    memory:
      total: 57648
      used: 0
    name: 'console/unix/:'
...
```

We highly recommend to assign meaningful names to fibers you create so that you can find them in the fiber.info() list. In the example below, we create a fiber named myworker:

```
tarantool> fiber = require('fiber')
---
...
tarantool> f = fiber.create(function() while true do fiber.sleep(0.5) end end)
---
...
tarantool> f:name('myworker') <!-- assigning the name to a fiber
---
...
tarantool> fiber.info()
---
- 102:
    csw: 14
    backtrace:
    - '#0 0x501a1a in fiber_yield_timeout+90'
    - '#1 0x4f2008 in lbox_fiber_sleep+72'
    - '#2 0x5112a7 in lj_BC_FUNCC+52'
    fid: 102
    memory:
      total: 57656
      used: 0
    name: myworker <!-- newly created background fiber
  101:
    csw: 284
    backtrace: []
    fid: 101
    memory:
      total: 57656
```

(continues on next page)

(continued from previous page)

```
    used: 0
  name: interactive
...
```

You can kill any fiber with fiber.kill(fid):

```
tarantool> fiber.kill(102)
---
...
tarantool> fiber.info()
---
- 101:
    csw: 324
    backtrace: []
    fid: 101
    memory:
      total: 57656
      used: 0
    name: interactive
...
```

If you want to dynamically obtain information with fiber.info(), the shell script below may come in handy. It connects to a Tarantool instance specified by NAME every 0.5 seconds, grabs the fiber.info() output and writes it to the fiber-info.txt file:

```
$ rm -f fiber.info.txt
$ watch -n 0.5 "echo 'require(\"fiber\").info()' | tarantoolctl enter NAME | tee -a fiber-info.txt"
```

If you can't understand which fiber causes performance issues, collect the metrics of the fiber.info() output for 10-15 seconds using the script above and contact the Tarantool team at support@tarantool.org.

Poor man's profilers

pstack <pid>

To use this tool, first install it with a package manager that comes with your Linux distribution. This command prints an execution stack trace of a running process specified by the PID. You might want to run this command several times in a row to pinpoint the bottleneck that causes the slowdown.

Once installed, say:

```
$ pstack $(pidof tarantool INSTANCENAME.lua)
```

Next, say:

```
$ echo $(pidof tarantool INSTANCENAME.lua)
```

to show the PID of the Tarantool instance that runs the INSTANCENAME.lua file.

You should get similar output:

```
Thread 19 (Thread 0x7f09d1bff700 (LWP 24173)):
#0 0x00007f0a1a5423f2 in ?? () from /lib64/libgomp.so.1
#1 0x00007f0a1a53fdc0 in ?? () from /lib64/libgomp.so.1
#2 0x00007f0a1ad5adc5 in start_thread () from /lib64/libpthread.so.0
```

(continues on next page)

(continued from previous page)

```
#3 0x00007f0a1a050ced in clone () from /lib64/libc.so.6
Thread 18 (Thread 0x7f09d13fe700 (LWP 24174)):
#0 0x00007f0a1a5423f2 in ?? () from /lib64/libgomp.so.1
#1 0x00007f0a1a53fdc0 in ?? () from /lib64/libgomp.so.1
#2 0x00007f0a1ad5adc5 in start_thread () from /lib64/libpthread.so.0
#3 0x00007f0a1a050ced in clone () from /lib64/libc.so.6
<...>
Thread 2 (Thread 0x7f09c8bfe700 (LWP 24191)):
#0 0x00007f0a1ad5e6d5 in pthread_cond_wait@@GLIBC_2.3.2 () from /lib64/libpthread.so.0
#1 0x000000000045d901 in wal_writer_pop(wal_writer*) ()
#2 0x000000000045db01 in wal_writer_f(__va_list_tag*) ()
#3 0x0000000000429abc in fiber_cxx_invoke(int (*)(__va_list_tag*), __va_list_tag*) ()
#4 0x00000000004b52a0 in fiber_loop ()
#5 0x00000000006099cf in coro_init ()
Thread 1 (Thread 0x7f0a1c47fd80 (LWP 24172)):
#0 0x00007f0a1a0512c3 in epoll_wait () from /lib64/libc.so.6
#1 0x00000000006051c8 in epoll_poll ()
#2 0x0000000000607533 in ev_run ()
#3 0x0000000000428e13 in main ()
```

gdb -ex “bt” -p <pid>

As with pstack, the GNU debugger (also known as gdb) needs to be installed before you can start using it. Your Linux package manager can help you with that.

Once the debugger is installed, say:

```
$ gdb -ex "set pagination 0" -ex "thread apply all bt" --batch -p $(pidof tarantool INSTANCENAME.lua)
```

Next, say:

```
$ echo $(pidof tarantool INSTANCENAME.lua)
```

to show the PID of the Tarantool instance that runs the INSTANCENAME.lua file.

After using the debugger, your console output should look like this:

```
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

[CUT]

Thread 1 (Thread 0x7f72289ba940 (LWP 20535)):
#0 _int_malloc (av=av@entry=0x7f7226e0eb20 <main_arena>, bytes=bytes@entry=504) at malloc.c:3697
#1 0x00007f7226acf21a in __libc_calloc (n=<optimized out>, elem_size=<optimized out>) at malloc.c:3234
#2 0x00000000004631f8 in vy_merge_iterator_reserve (capacity=3, itr=0x7f72264af9e0) at /usr/src/tarantool/
↪src/box/vinyl.c:7629
#3 vy_merge_iterator_add (itr=itr@entry=0x7f72264af9e0, is_mutable=is_mutable@entry=true, belong_
↪range=belong_range@entry=false) at /usr/src/tarantool/src/box/vinyl.c:7660
#4 0x00000000004703df in vy_read_iterator_add_mem (itr=0x7f72264af990) at /usr/src/tarantool/src/box/
↪vinyl.c:8387
#5 vy_read_iterator_use_range (itr=0x7f72264af990) at /usr/src/tarantool/src/box/vinyl.c:8453
#6 0x000000000047657d in vy_read_iterator_start (itr=<optimized out>) at /usr/src/tarantool/src/box/vinyl.
↪c:8501
#7 0x00000000004766b5 in vy_read_iterator_next (itr=itr@entry=0x7f72264af990,
↪result=result@entry=0x7f72264afad8) at /usr/src/tarantool/src/box/vinyl.c:8592
#8 0x000000000047689d in vy_index_get (tx=tx@entry=0x7f7226468158, index=index@entry=0x2563860, key=
↪<optimized out>, part_count=<optimized out>, result=result@entry=0x7f72264afad8) at /usr/src/tarantool/
↪src/box/vinyl.c:5705
```

(continues on next page)

(continued from previous page)

```
#9 0x0000000000477601 in vy_replace_impl (request=<optimized out>, request=<optimized out>,
↪stmt=0x7f72265a7150, space=0x2567ea0, tx=0x7f7226468158) at /usr/src/tarantool/src/box/vinyl.c:5920
#10 vy_replace (tx=0x7f7226468158, stmt=stmt@entry=0x7f72265a7150, space=0x2567ea0, request=<optimized
↪out>) at /usr/src/tarantool/src/box/vinyl.c:6608
#11 0x00000000004615a9 in VinylSpace::executeReplace (this=<optimized out>, txn=<optimized out>, space=
↪<optimized out>, request=<optimized out>) at /usr/src/tarantool/src/box/vinyl_space.cc:108
#12 0x00000000004bd723 in process_rw (request=request@entry=0x7f72265a70f8,
↪space=space@entry=0x2567ea0, result=result@entry=0x7f72264afbc8) at /usr/src/tarantool/src/box/box.cc:182
#13 0x00000000004bed48 in box_process1 (request=0x7f72265a70f8, result=result@entry=0x7f72264afbc8) at /
↪usr/src/tarantool/src/box/box.cc:700
#14 0x00000000004bf389 in box_replace (space_id=space_id@entry=513, tuple=<optimized out>, tuple_end=
↪<optimized out>, result=result@entry=0x7f72264afbc8) at /usr/src/tarantool/src/box/box.cc:754
#15 0x00000000004d72f8 in lbox_replace (L=0x413c5780) at /usr/src/tarantool/src/box/lua/index.c:72
#16 0x000000000050f317 in lj_BC_FUNCC ()
#17 0x00000000004d37c7 in execute_lua_call (L=0x413c5780) at /usr/src/tarantool/src/box/lua/call.c:282
#18 0x000000000050f317 in lj_BC_FUNCC ()
#19 0x0000000000529c7b in lua_cpcall ()
#20 0x00000000004f6aa3 in luaT_cpcall (L=L@entry=0x413c5780, func=func@entry=0x4d36d0 <execute_lua_
↪call>, ud=ud@entry=0x7f72264afde0) at /usr/src/tarantool/src/lua/utils.c:962
#21 0x00000000004d3fe7 in box_process_lua (handler=0x4d36d0 <execute_lua_call>,
↪out=out@entry=0x7f7213020600, request=request@entry=0x413c5780) at /usr/src/tarantool/src/box/lua/call.
↪c:382
#22 box_lua_call (request=request@entry=0x7f72130401d8, out=out@entry=0x7f7213020600) at /usr/src/
↪tarantool/src/box/lua/call.c:405
#23 0x00000000004c0f27 in box_process_call (request=request@entry=0x7f72130401d8,
↪out=out@entry=0x7f7213020600) at /usr/src/tarantool/src/box/box.cc:1074
#24 0x000000000041326c in tx_process_misc (m=0x7f7213040170) at /usr/src/tarantool/src/box/iproto.cc:942
#25 0x0000000000504554 in cmsg_deliver (msg=0x7f7213040170) at /usr/src/tarantool/src/cbus.c:302
#26 0x0000000000504c2e in fiber_pool_f (ap=<error reading variable: value has been optimized out>) at /usr/
↪src/tarantool/src/fiber_pool.c:64
#27 0x000000000041122c in fiber_cxx_invoke(fiber_func, typedef __va_list_tag __va_list_tag *) (f=
↪<optimized out>, ap=<optimized out>) at /usr/src/tarantool/src/fiber.h:645
#28 0x00000000005011a0 in fiber_loop (data=<optimized out>) at /usr/src/tarantool/src/fiber.c:641
#29 0x0000000000688fbf in coro_init () at /usr/src/tarantool/third_party/coro/coro.c:110
```

Run the debugger in a loop a few times to collect enough samples for making conclusions about why Tarantool demonstrates suboptimal performance. Use the following script:

```
$ rm -f stack-trace.txt
$ watch -n 0.5 "gdb -ex 'set pagination 0' -ex 'thread apply all bt' --batch -p $(pidof tarantool
↪INSTANCENAME.lua) | tee -a stack-trace.txt"
```

Structurally and functionally, this script is very similar to the one used with fiber.info() above.

If you have any difficulties troubleshooting, let the script run for 10-15 seconds and then send the resulting stack-trace.txt file to the Tarantool team at support@tarantool.org.

> Warning: Use the poor man's profilers with caution: each time they attach to a running process, this stops the process execution for about a second, which may leave a serious footprint in high-load services.

gperftools

To use the CPU profiler from the Google Performance Tools suite with Tarantool, first take care of the prerequisites:

- For Debian/Ubuntu, run:

```
$ apt-get install libgoogle-perftools4
```

- For RHEL/CentOS/Fedora, run:

```
$ yum install gperftools-libs
```

Once you do this, install Lua bindings:

```
$ tarantoolctl rocks install gperftools
```

Now you're ready to go. Enter your instance's interactive administrator console:

```
$ tarantoolctl enter NAME
```

To start profiling, say:

```
tarantool> cpuprof = require('gperftools.cpu')
tarantool> cpuprof.start('/home/<username>/tarantool-on-production.prof')
```

It takes at least a couple of minutes for the profiler to gather performance metrics. After that, save the results to disk (you can do that as many times as you need):

```
tarantool> cpuprof.flush()
```

To stop profiling, say:

```
tarantool> cpuprof.stop()
```

You can now analyze the output with the pprof utility that comes with the gperftools package:

```
$ pprof --text /usr/bin/tarantool /home/<username>/tarantool-on-production.prof
```

Note: On Debian/Ubuntu, the pprof utility is called google-pprof.

Your output should look similar to this:

```
Total: 598 samples
      83 13.9% 13.9% 83 13.9% epoll_wait
      54 9.0% 22.9% 102 17.1%
vy_mem_tree_insert.constprop.35
      32 5.4% 28.3% 34 5.7% __write_nocancel
      28 4.7% 32.9% 42 7.0% vy_mem_iterator_start_from
      26 4.3% 37.3% 26 4.3% _IO_str_seekoff
      21 3.5% 40.8% 21 3.5% tuple_compare_field
      19 3.2% 44.0% 19 3.2%
::TupleCompareWithKey::compare
      19 3.2% 47.2% 38 6.4% tuple_compare_slowpath
      12 2.0% 49.2% 23 3.8% __libc_calloc
       9 1.5% 50.7% 9 1.5%
::TupleCompare::compare@42efc0
       9 1.5% 52.2% 9 1.5% vy_cache_on_write
       9 1.5% 53.7% 57 9.5% vy_merge_iterator_next_key
       8 1.3% 55.0% 8 1.3% __nss_passwd_lookup
       6 1.0% 56.0% 25 4.2% gc_onestep
```

(continues on next page)

(continued from previous page)

```
    6 1.0% 57.0% 6 1.0% lj_tab_next
    5 0.8% 57.9% 5 0.8% lj_alloc_malloc
    5 0.8% 58.7% 131 21.9% vy_prepare
```

perf

This tool for performance monitoring and analysis is installed separately via your package manager. Try running the perf command in the terminal and follow the prompts to install the necessary package(s).

Note: By default, some perf commands are restricted to root, so, to be on the safe side, either run all commands as root or prepend them with sudo.

To start gathering performance statistics, say:

```
$ perf record -g -p $(pidof tarantool INSTANCENAME.lua)
```

This command saves the gathered data to a file named perf.data inside the current working directory. To stop this process (usually, after 10-15 seconds), press ctrl+C. In your console, you'll see:

```
^C[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.225 MB perf.data (1573 samples) ]
```

Now run the following command:

```
$ perf report -n -g --stdio | tee perf-report.txt
```

It formats the statistical data in the perf.data file into a performance report and writes it to the perf-report.txt file.

The resulting output should look similar to this:

```
# Samples: 14K of event 'cycles'
# Event count (approx.): 9927346847
#
# Children Self Samples Command Shared Object Symbol
# ........ ........ ............ ......... ................... ....................................
#
    35.50% 0.55% 79 tarantool tarantool [.] lj_gc_step
            |
            --34.95%--lj_gc_step
                      |
                      |--29.26%--gc_onestep
                      | |
                      | |--13.85%--gc_sweep
                      | | |
                      | | |--5.59%--lj_alloc_free
                      | | |
                      | | |--1.33%--lj_tab_free
                      | | | |
                      | | | --1.01%--lj_alloc_free
                      | | |
                      | | --1.17%--lj_cdata_free
                      | |
```

(continues on next page)

(continued from previous page)

```
                  | |--5.41%--gc_finalize
                  | | |
                  | | |--1.06%--lj_obj_equal
                  | | |
                  | | --0.95%--lj_tab_set
                  | |
                  | |--4.97%--rehashtab
                  | | |
                  | | --3.65%--lj_tab_resize
                  | | |
                  | | |--0.74%--lj_tab_set
                  | | |
                  | | --0.72%--lj_tab_newkey
                  | |
                  | |--0.91%--propagatemark
                  | |
                  | --0.67%--lj_cdata_free
                  |
                   --5.43%--propagatemark
                             |
                              --0.73%--gc_mark
```

Unlike the poor man's profilers, gperftools and perf have low overhead (almost negligible as compared with pstack and gdb): they don't result in long delays when attaching to a process and therefore can be used without serious consequences.

jit.p

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

Example

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

```
box.space.t:drop()
box.schema.space.create('t')
box.space.t:create_index('i')
function f1() for i = 1,500000 do
  box.space.t:insert{i}
  box.space.t:delete{i}
  end
return 1
end
function f3() f1() end
jit_p = require("jit.profile")
sampletable = {}
jit_p.start("f", function(thread, samples, vmstate)
  local dump=jit_p.dumpstack(thread, "f", 1)
  sampletable[dump] = (sampletable[dump] or 0) + samples
end)
f3()
jit_p.stop()
for d,v in pairs(sampletable) do print(v, d) end
```

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

3.5.6 Daemon supervision

Server signals

Tarantool processes these signals during the event loop in the transaction processor thread:

Signal	Effect
SIGHUP	May cause log file rotation. See the example in reference on Tarantool logging parameters.
SIGUSR1	May cause a database checkpoint. See box.snapshot.
SIGTERM	May cause graceful shutdown (information will be saved first).
SIGINT (also known as keyboard interrupt)	May cause graceful shutdown.
SIGKILL	Causes an immediate shutdown.

Other signals will result in behavior defined by the operating system. Signals other than SIGKILL may be ignored, especially if Tarantool is executing a long-running procedure which prevents return to the event loop in the transaction processor thread.

Automatic instance restart

On systemd-enabled platforms, systemd automatically restarts all Tarantool instances in case of failure. To demonstrate it, let's try to destroy an instance:

```
$ systemctl status tarantool@my_app|grep PID
Main PID: 5885 (tarantool)
$ tarantoolctl enter my_app
/bin/tarantoolctl: Found my_app.lua in /etc/tarantool/instances.available
/bin/tarantoolctl: Connecting to /var/run/tarantool/my_app.control
/bin/tarantoolctl: connected to unix/:/var/run/tarantool/my_app.control
unix/:/var/run/tarantool/my_app.control> os.exit(-1)
/bin/tarantoolctl: unix/:/var/run/tarantool/my_app.control: Remote host closed connection
```

Now let's make sure that systemd has restarted the instance:

```
$ systemctl status tarantool@my_app|grep PID
Main PID: 5914 (tarantool)
```

Finally, let's check the boot logs:

```
$ journalctl -u tarantool@my_app -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@my_app.service: Unit entered failed state.
Jan 21 21:09:45 localhost.localdomain systemd[1]: tarantool@my_app.service: Failed with result 'exit-code'.
Jan 21 21:09:45 localhost.localdomain systemd[1]: tarantool@my_app.service: Service hold-off time over,␣
↪scheduling restart.
Jan 21 21:09:45 localhost.localdomain systemd[1]: Stopped Tarantool Database Server.
Jan 21 21:09:45 localhost.localdomain systemd[1]: Starting Tarantool Database Server...
Jan 21 21:09:45 localhost.localdomain tarantoolctl[5910]: /usr/bin/tarantoolctl: Found my_app.lua in /etc/
↪tarantool/instances.available
```

(continues on next page)

(continued from previous page)

```
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.
```

Core dumps

Tarantool makes a core dump if it receives any of the following signals: SIGSEGV, SIGFPE, SIGABRT or SIGQUIT. This is automatic if Tarantool crashes.

On systemd-enabled platforms, coredumpctl automatically saves core dumps and stack traces in case of a crash. Here is a general "how to" for how to enable core dumps on a Unix system:

1. Ensure session limits are configured to enable core dumps, i.e. say ulimit -c unlimited. Check "man 5 core" for other reasons why a core dump may not be produced.
2. Set a directory for writing core dumps to, and make sure that the directory is writable. On Linux, the directory path is set in a kernel parameter configurable via /proc/sys/kernel/core_pattern.
3. Make sure that core dumps include stack trace information. If you use a binary Tarantool distribution, this is automatic. If you build Tarantool from source, you will not get detailed information if you pass -DCMAKE_BUILD_TYPE=Release to CMake.

To simulate a crash, you can execute an illegal command against a Tarantool instance:

```
$ # !!! please never do this on a production system !!!
$ tarantoolctl enter my_app
unix/:/var/run/tarantool/my_app.control> require('ffi').cast('char *', 0)[0] = 48
/bin/tarantoolctl: unix/:/var/run/tarantool/my_app.control: Remote host closed connection
```

Alternatively, if you know the process ID of the instance (here we refer to it as $PID), you can abort a Tarantool instance by running gdb debugger:

```
$ gdb -batch -ex "generate-core-file" -p $PID
```

or manually sending a SIGABRT signal:

```
$ kill -SIGABRT $PID
```

Note: To find out the process id of the instance ($PID), you can:

- look it up in the instance's box.info.pid,
- find it with ps -A | grep tarantool, or
- say systemctl status tarantool@my_app|grep PID.

On a systemd-enabled system, to see the latest crashes of the Tarantool daemon, say:

```
$ 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
```

To save a core dump into a file, say:

```
$ coredumpctl -o filename.core info <pid>
```

Stack traces

Since Tarantool stores tuples in memory, core files may be large. For investigation, you normally don't need the whole file, but only a "stack trace" or "backtrace".

To save a stack trace into a file, say:

```
$ gdb -se "tarantool" -ex "bt full" -ex "thread apply all bt" --batch -c core> /tmp/tarantool_trace.txt
```

where:

- "tarantool" is the path to the Tarantool executable,
- "core" is the path to the core file, and
- "/tmp/tarantool_trace.txt" is a sample path to a file for saving the stack trace.

Note: Occasionally, you may find that the trace file contains output without debug symbols – the lines will contain "??" instead of names. If this happens, check the instructions on these Tarantool wiki pages: How to debug core dump of stripped tarantool and How to debug core from different OS.

To see the stack trace and other useful information in console, say:

```
$ coredumpctl info 21035
          PID: 21035 (tarantool)
          UID: 995 (tarantool)
          GID: 992 (tarantool)
       Signal: 6 (ABRT)
    Timestamp: Sat 2016-01-23 15:51:42 MSK (4h 36min ago)
 Command Line: tarantool my_app.lua <running>
   Executable: /usr/bin/tarantool
Control Group: /system.slice/system-tarantool.slice/tarantool@my_app.service
         Unit: tarantool@my_app.service
        Slice: system-tarantool.slice
      Boot ID: 7c686e2ef4dc4e3ea59122757e3067e2
   Machine ID: a4a878729c654c7093dc6693f6a8e5ee
     Hostname: localhost.localdomain
      Message: Process 21035 (tarantool) of user 995 dumped core.

               Stack trace of thread 21035:
               #0  0x00007f84993aa618 raise (libc.so.6)
               #1  0x00007f84993ac21a abort (libc.so.6)
               #2  0x0000560d0a9e9233 _ZL12sig_fatal_cbi (tarantool)
               #3  0x00007f849a211220 __restore_rt (libpthread.so.0)
               #4  0x0000560d0aaa5d9d lj_cconv_ct_ct (tarantool)
               #5  0x0000560d0aaa687f lj_cconv_ct_tv (tarantool)
               #6  0x0000560d0aaabe33 lj_cf_ffi_meta___newindex (tarantool)
               #7  0x0000560d0aaae2f7 lj_BC_FUNCC (tarantool)
               #8  0x0000560d0aa9aabd lua_pcall (tarantool)
               #9  0x0000560d0aa71400 lbox_call (tarantool)
               #10 0x0000560d0aa6ce36 lua_fiber_run_f (tarantool)
               #11 0x0000560d0a9e8d0c _ZL16fiber_cxx_invokePFiP13__va_list_tagES0_ (tarantool)
               #12 0x0000560d0aa7b255 fiber_loop (tarantool)
               #13 0x0000560d0ab38ed1 coro_init (tarantool)
               ...
```

Debugger

To start gdb debugger on the core dump, say:

```
$ coredumpctl gdb <pid>
```

It is highly recommended to install tarantool-debuginfo package to improve gdb experience, for 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
```

Symbolic names are present in stack traces even if you don't have tarantool-debuginfo package installed.

3.5.7 Disaster recovery

The minimal fault-tolerant Tarantool configuration would be a replication cluster that includes a master and a replica, or two masters.

The basic recommendation is to configure all Tarantool instances in a cluster to create snapshot files at a regular basis.

Here follow action plans for typical crash scenarios.

Master-replica

Configuration: One master and one replica.

Problem: The master has crashed.

Your actions:

1. Ensure the master is stopped for good. For example, log in to the master machine and use systemctl stop tarantool@<instance_name>.
2. Switch the replica to master mode by setting box.cfg.read_only parameter to false and let the load be handled by the replica (effective master).
3. Set up a replacement for the crashed master on a spare host, with replication parameter set to replica (effective master), so it begins to catch up with the new master's state. The new instance should have box.cfg.read_only parameter set to true.

You lose the few transactions in the master write ahead log file, which it may have not transferred to the replica before crash. If you were able to salvage the master .xlog file, you may be able to recover these. In order to do it:

1. Find out the position of the crashed master, as reflected on the new master.
    a. Find out instance UUID from the crashed master xlog:

```
$ head -5 *.xlog | grep Instance
Instance: ed607cad-8b6d-48d8-ba0b-dae371b79155
```

b. On the new master, use the UUID to find the position:

```
tarantool> box.info.vclock[box.space._cluster.index.uuid:select{'ed607cad-8b6d-48d8-ba0b-
↪dae371b79155'}[1][1]]
---
- 23425
<...>
```

2. Play the records from the crashed .xlog to the new master, starting from the new master position:

a. Issue this request locally at the new master's machine to find out instance ID of the new master:

```
tarantool> box.space._cluster:select{}
---
- - [1, '88580b5c-4474-43ab-bd2b-2409a9af80d2']
...
```

b. Play the records to the new master:

```
$ tarantoolctl <new_master_uri> <xlog_file> play --from 23425 --replica 1
```

Master-master

Configuration: Two masters.

Problem: Master#1 has crashed.

Your actions:

1. Let the load be handled by master#2 (effective master) alone.

2. Follow the same steps as in the master-replica recovery scenario to create a new master and salvage lost data.

Data loss

Configuration: Master-master or master-replica.

Problem: Data was deleted at one master and this data loss was propagated to the other node (master or replica).

The following steps are applicable only to data in memtx storage engine. Your actions:

1. Put all nodes in read-only mode and disable checkpointing with box.backup.start(). Disabling the checkpointing is necessary to prevent the Tarantool garbage collector from removing files made with older checkpoints.
2. Get the latest valid .snap file and use tarantoolctl cat command to calculate at which lsn the data loss occurred.
3. Start a new instance (instance#1) and use tarantoolctl play command to play to it the contents of .snap/.xlog files up to the calculated lsn.
4. Bootstrap a new replica from the recovered master (instance#1).

3.5.8 Backups

Tarantool has an append-only storage architecture: it appends data to files but it never overwrites earlier data. The Tarantool garbage collector removes old files after a checkpoint. You can prevent or delay the garbage collector's action by configuring the checkpoint daemon. Backups can be taken at any time, with minimal overhead on database performance.

backup.start() and backup.stop()

Two functions are helpful for backups in certain situations.

box.backup.start() informs the server that some activities that might interfer with backup should be suspended – suspend checkpointing, suspend Tarantool garbage collection, and effectively enter read-only mode.

Later box.backup.stop() informs the server that normal operations may resume. Starting with Tarantool 1.10.1 there is a new optional argument, box.backup.start(n), where n indicates the checkpoint to use relative to the latest checkpoint – for example n = 0 means "backup will be based on the latest checkpoint", n = 1 means "backup will be based on the first checkpoint before the latest checkpoint (counting backwards)", and so on, and the default value for n is zero.

box.backup.start() returns a table with the names of snapshot and vinyl files that should be copied. Example:

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

Hot backup (memtx)

This is a special case when there are only in-memory tables.

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. Use tar to make a (possibly compressed) copy of the latest .snap and .xlog files on the memtx_dir and wal_dir directories.
2. If there is a security policy, encrypt the .tar file.
3. 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 memtx_dir and wal_dir directories.

Hot backup (vinyl/memtx)

Vinyl stores its files in vinyl_dir, and creates a folder for each database space. Dump and compaction processes are append-only and create new files. The Tarantool garbage collector may remove old files after each checkpoint.

To take a mixed backup:

1. Issue box.backup.start() on the administrative console. This will suspend garbage collection till the next box.backup.stop() and will return a list of files to back up.
2. Copy the files from the list to a safe location. This will include memtx snapshot files, vinyl run and index files, at a state consistent with the last checkpoint.
3. Issue box.backup.stop() so the garbage collector can continue.

Continuous remote backup (memtx)

The replication feature 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.

Continuous backup (memtx)

The logged changes done since the last cold backup must be secured, while the system is running.

For this purpose, you need a file copy utility that will do the copying remotely and continuously, copying only the parts of a write ahead log file that are changing. One such utility is rsync.

Alternatively, you need 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.

3.5.9 Upgrades

Upgrading a Tarantool database

If you created a database with an older Tarantool version and have now installed a newer version, make the request box.schema.upgrade(). This updates Tarantool system spaces to match the currently installed version of Tarantool.

For example, here is what happens when you run box.schema.upgrade() with a database created with Tarantool version 1.6.4 to version 1.7.2 (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
---
...
```

Upgrading a Tarantool instance

Tarantool is backward compatible between two adjacent versions. For example, you should have no or little trouble when upgrading from Tarantool 1.6 to 1.7, or from Tarantool 1.7 to 2.x. Meanwhile Tarantool 2.x may have incompatible changes when migrating from Tarantool 1.6. to 2.x directly.

How to upgrade from Tarantool 1.7 to 2.x

1. Stop the Tarantool server.
2. Make a copy of all data (see an appropriate hot backup procedure in Backups) and the package from which the current (old) version was installed (for rollback purposes).
3. Update the Tarantool server. See installation instructions at Tarantool download page.
4. Launch the updated Tarantool server using tarantoolctl or systemctl.

How to upgrade from Tarantool 1.6 to 2.x

The procedure is fully analogous to upgrading from 1.7 to 2.x.

How to upgrade from Tarantool 1.6 to 1.7

This procedure is for upgrading a standalone Tarantool instance in production from 1.6.x to 1.7.x. Notice that this will always imply a downtime. To upgrade without downtime, you need several Tarantool servers running in a replication cluster (see below).

Tarantool 1.7 has an incompatible .snap and .xlog file format: 1.6 files are supported during upgrade, but you won't be able to return to 1.6 after running under 1.7 for a while. It also renames a few configuration parameters, but old parameters are supported. The full list of breaking changes is available in release notes for Tarantool 1.7.

1. Check with application developers whether application files need to be updated due to incompatible changes (see 1.7 release notes). If yes, back up the old application files.
2. Stop the Tarantool server.
3. Make a copy of all data (see an appropriate hot backup procedure in Backups) and the package from which the current (old) version was installed (for rollback purposes).
4. Update the Tarantool server. See installation instructions at Tarantool download page.
5. Update the Tarantool database. Put the request box.schema.upgrade() inside a box.once() function in your Tarantool initialization file. On startup, this will create new system spaces, update data type names (e.g. num -> unsigned, str -> string) and options in Tarantool system spaces.
6. Update application files, if needed.
7. Launch the updated Tarantool server using tarantoolctl or systemctl.

Upgrading Tarantool in a replication cluster

Tarantool 1.7 can work as a replica for Tarantool 1.6 and vice versa. Replicas perform capability negotiation on handshake, and new 1.7 replication features are not used with 1.6 replicas. This allows upgrading clustered configurations.

This procedure allows for a rolling upgrade without downtime and works for any cluster configuration: master-master or master-replica.

1. Upgrade Tarantool at all replicas (or at any master in a master-master cluster). See details in Upgrading a Tarantool instance.
2. Verify installation on the replicas:

   a. Start Tarantool.

   b. Attach to the master and start working as before.

   The master runs the old Tarantool version, which is always compatible with the next major version.

3. Upgrade the master. The procedure is similar to upgrading a replica.
4. Verify master installation:
   a. Start Tarantool with replica configuration to catch up.
   b. Switch to master mode.
5. Upgrade the database on any master node in the cluster. Make the request box.schema.upgrade(). This updates Tarantool system spaces to match the currently installed version of Tarantool. Changes are propagated to other nodes via the regular replication mechanism.

3.5.10 Notes for operating systems

Mac OS

On Mac OS, you can administer Tarantool instances only with tarantoolctl. No native system tools are supported.

FreeBSD

To make tarantoolctl work along with init.d utilities on FreeBSD, use paths other than those suggested in Instance configuration. Instead of /usr/share/tarantool/ directory, use /usr/local/etc/tarantool/ and create the following subdirectories:

- default for tarantoolctl defaults (see example below),
- instances.available for all available instance files, and
- instances.enabled for instance files to be auto-started by sysvinit.

Here is an example of tarantoolctl defaults on FreeBSD:

```
default_cfg = {
    pid_file  = "/var/run/tarantool", -- /var/run/tarantool/${INSTANCE}.pid
    wal_dir   = "/var/db/tarantool", -- /var/db/tarantool/${INSTANCE}/
    snap_dir  = "/var/db/tarantool", -- /var/db/tarantool/${INSTANCE}
    vinyl_dir = "/var/db/tarantool", -- /var/db/tarantool/${INSTANCE}
    logger    = "/var/log/tarantool", -- /var/log/tarantool/${INSTANCE}.log
    username  = "tarantool",
}

-- instances.available - all available instances
-- instances.enabled - instances to autostart by sysvinit
instance_dir = "/usr/local/etc/tarantool/instances.available"
```

Gentoo Linux

The section below is about a dev-db/tarantool package installed from the official layman overlay (named tarantool).

The default instance directory is /etc/tarantool/instances.available, can be redefined in /etc/default/tarantool.

Tarantool instances can be managed (start/stop/reload/status/...) using OpenRC. Consider the example how to create an OpenRC-managed instance:

```
$ cd /etc/init.d
$ ln -s tarantool your_service_name
$ ln -s /usr/share/tarantool/your_service_name.lua /etc/tarantool/instances.available/your_service_name.lua
```

Checking that it works:

```
$ /etc/init.d/your_service_name start
$ tail -f -n 100 /var/log/tarantool/your_service_name.log
```

3.5.11 Bug reports

If you found a bug in Tarantool, you're doing us a favor by taking the time to tell us about it.

Please create an issue at Tarantool repository at GitHub. We encourage you to include the following information:

- Steps needed to reproduce the bug, and an explanation why this differs from the expected behavior according to our manual. Please provide specific unique information. For example, instead of "I can't get certain information", say "box.space.x:delete() didn't report what was deleted".
- Your operating system name and version, the Tarantool name and version, and any unusual details about your machine and its configuration.
- Related files like a stack trace or a Tarantool log file.

If this is a feature request or if it affects a special category of users, be sure to mention that.

Usually within one or two workdays a Tarantool team member will write an acknowledgment, or some questions, or suggestions for a workaround.

3.5.12 Troubleshooting guide

For this guide, you need to install Tarantool stat module:

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

Problem: INSERT/UPDATE-requests result in ER_MEMORY_ISSUE error

Possible reasons

- Lack of RAM (parameters arena_used_ratio and quota_used_ratio in box.slab.info() report are getting close to 100%).

  To check these parameters, say:

  ```
  $ # attaching to a Tarantool instance
  $ tarantoolctl enter <instance_name>
  $ # -- OR --
  $ tarantoolctl connect <URI>
  ```

```
-- requesting arena_used_ratio value
tarantool> require('stat').stat()['slab.arena_used_ratio']

-- requesting quota_used_ratio value
tarantool> require('stat').stat()['slab.quota_used_ratio']
```

Solution

Try either of the following measures:

- In Tarantool's instance file, increase the value of box.cfg{memtx_memory} (if memory resources are available).

  In versions of Tarantool before 1.10, the server needs to be restarted to change this parameter. The Tarantool server will be unavailable while restarting from .xlog files, unless you restart it using hot standby mode. In the latter case, nearly 100% server availability is guaranteed.

- Clean up the database.

- Check the indicators of memory fragmentation:

  ```
  -- requesting quota_used_ratio value
  tarantool> require('stat').stat()['slab.quota_used_ratio']

  -- requesting items_used_ratio value
  tarantool> require('stat').stat()['slab.items_used_ratio']
  ```

  In case of heavy memory fragmentation (quota_used_ratio is getting close to 100%, items_used_ratio is about 50%), we recommend restarting Tarantool in the hot standby mode.

Problem: Tarantool generates too heavy CPU load

Possible reasons

The transaction processor thread consumes over 60% CPU.

Solution

Attach to the Tarantool instance with tarantoolctl utility, analyze the query statistics with box.stat() and spot the CPU consumption leader. The following commands can help:

```
$ # attaching to a Tarantool instance
$ tarantoolctl enter <instance_name>
$ # -- OR --
$ tarantoolctl connect <URI>
```

```
-- checking the RPS of calling stored procedures
tarantool> require('stat').stat()['stat.op.call.rps']
```

The critical RPS value is 75 000, boiling down to 10 000 - 20 000 for a rich Lua application (a Lua module of 200+ lines).

```
-- checking RPS per query type
tarantool> require('stat').stat()['stat.op.<query_type>.rps']
```

The critical RPS value for SELECT/INSERT/UPDATE/DELETE requests is 100 000.

If the load is mostly generated by SELECT requests, we recommend adding a slave server and let it process part of the queries.

If the load is mostly generated by INSERT/UPDATE/DELETE requests, we recommend sharding the database.

Problem: Query processing times out

Possible reasons

Note: All reasons that we discuss here can be identified by messages in Tarantool's log file, all starting with the words 'Too long...'.

---

1. Both fast and slow queries are processed within a single connection, so the readahead buffer is cluttered with slow queries.

   Solution

   Try either of the following measures:

   - Increase the readahead buffer size (box.cfg{readahead} parameter).

     This parameter can be changed on the fly, so you don't need to restart Tarantool. Attach to the Tarantool instance with tarantoolctl utility and call box.cfg{} with a new readahead value:

     ```
     $ # attaching to a Tarantool instance
     $ tarantoolctl enter <instance_name>
     $ # -- OR --
     $ tarantoolctl connect <URI>
     ```

     ```
     -- changing the readahead value
     tarantool> box.cfg{readahead = 10 * 1024 * 1024}
     ```

     Example: Given 1000 RPS, 1 Kbyte of query size, and 10 seconds of maximal query processing time, the minimal readahead buffer size must be 10 Mbytes.

   - On the business logic level, split fast and slow queries processing by different connections.

2. Slow disks.

   Solution

   Check disk performance (use iostat, iotop or strace utility to check iowait parameter) and try to put .xlog files and snapshot files on different physical disks (i.e. use different locations for wal_dir and memtx_dir).

Problem: Replication "lag" and "idle" contain negative values

This is about box.info.replication.(upstream.)lag and box.info.replication.(upstream.)idle values in box.info.replication section.

Possible reasons

Operating system clock on the hosts is not synchronized, or the NTP server is faulty.

Solution

Check NTP server settings.

If you found no problems with the NTP server, just do nothing then. Lag calculation uses operating system clock from two different machines. If they get out of sync, the remote master clock can get consistently behind the local instance's clock.

Problem: Replication "idle" keeps growing, but no related log messages appear

This is about box.info.replication.(upstream.)idle value in box.info.replication section.

Possible reasons

Some server was assigned different IP addresses, or some server was specified twice in box.cfg{}, so duplicate connections were established.

Solution

Upgrade Tarantool 1.6 to 1.7, where this error is fixed: in case of duplicate connections, replication is stopped and the following message is added to the log: 'Incorrect value for option ''replication_source'': duplicate connection with the same replica UUID'.

Problem: Replication statistics differ on replicas within a replica set

This is about a replica set that consists of one master and several replicas. In a replica set of this type, values in box.info.replication section, like box.info.replication.lsn, come from the master and must be the same on all replicas within the replica set. The problem is that they get different.

Possible reasons

Replication is broken.

Solution

Restart replication.

Problem: Master-master replication is stopped

This is about box.info.replication(.upstream).status = stopped.

Possible reasons

In a master-master replica set of two Tarantool instances, one of the masters has tried to perform an action already performed by the other server, for example re-insert a tuple with the same unique key. This would cause an error message like 'Duplicate key exists in unique index 'primary' in space <space_name>'.

Solution

Restart replication with the following commands (at each master instance):

```
$ # attaching to a Tarantool instance
$ tarantoolctl enter <instance_name>
$ # -- OR --
$ tarantoolctl connect <URI>
```

```
-- restarting replication
tarantool> original_value = box.cfg.replication
tarantool> box.cfg{replication={}}
tarantool> box.cfg{replication=original_value}
```

We also recommend using text primary keys or setting up master-slave replication.

Problem: Tarantool works much slower than before

Possible reasons

Inefficient memory usage (RAM is cluttered with a huge amount of unused objects).

Solution

Call the Lua garbage collector with the collectgarbage('count') function and measure its execution time with the Tarantool functions clock.bench() or clock.proc().

Example of calculating memory usage statistics:

```
$ # attaching to a Tarantool instance
$ tarantoolctl enter <instance_name>
$ # -- OR --
$ tarantoolctl connect <URI>
```

```
-- loading Tarantool's "clock" module with time-related routines
tarantool> local clock = require 'clock'
-- starting the timer
tarantool> local b = clock.proc()
-- launching garbage collection
tarantool> local c = collectgarbage('count')
-- stopping the timer after garbage collection is completed
tarantool> return c, clock.proc() - b
```

If the returned clock.proc() value is greater than 0.001, this may be an indicator of inefficient memory usage (no active measures are required, but we recommend to optimize your Tarantool application code).

If the value is greater than 0.01, your application definitely needs thorough code analysis aimed at optimizing memory usage.

3.6 Replication

Replication allows multiple Tarantool instances to work on copies of the same databases. The databases are kept in sync because each instance can communicate its changes to all the other instances.

This chapter includes the following sections:

3.6.1 Replication architecture

Replication mechanism

A pack of instances which operate on copies of the same databases make up a replica set. Each instance in a replica set has a role, master or replica.

A replica gets all updates from the master by continuously fetching and applying its write ahead log (WAL). Each record in the WAL represents a single Tarantool data-change request such as INSERT, UPDATE or DELETE, and is assigned a monotonically growing log sequence number (LSN). In essence, Tarantool replication is row-based: each data-change request is fully deterministic and operates on a single tuple. However, unlike a classical row-based log, which contains entire copies of the changed rows, Tarantool's WAL contains copies of the requests. For example, for UPDATE requests, Tarantool only stores the primary key of the row and the update operations, to save space.

Invocations of stored programs are not written to the WAL. Instead, records of the actual data-change requests, performed by the Lua code, are written to the WAL. This ensures that possible non-determinism of Lua does not cause replication to go out of sync.

Data definition operations on temporary spaces, such as creating/dropping, adding indexes, truncating, etc., are written to the WAL, since information about temporary spaces is stored in non-temporary system spaces, such as box.space._space. Data change operations on temporary spaces are not written to the WAL and are not replicated.

Data change operations on replication-local spaces (spaces created with is_local = true) are written to the WAL but are not replicated.

To create a valid initial state, to which WAL changes can be applied, every instance of a replica set requires a start set of checkpoint files, such as .snap files for memtx and .run files for vinyl. A replica joining an existing replica set, chooses an existing master and automatically downloads the initial state from it. This is called an initial join.

When an entire replica set is bootstrapped for the first time, there is no master which could provide the initial checkpoint. In such a case, replicas connect to each other and elect a master, which then creates the starting set of checkpoint files, and distributes it to all the other replicas. This is called an automatic bootstrap of a replica set.

When a replica contacts a master (there can be many masters) for the first time, it becomes part of a replica set. On subsequent occasions, it should always contact a master in the same replica set. Once connected to the master, the replica requests all changes that happened after the latest local LSN (there can be many LSNs – each master has its own LSN).

Each replica set is identified by a globally unique identifier, called the replica set UUID. The identifier is created by the master which creates the very first checkpoint, and is part of the checkpoint file. It is stored in system space box.space._schema. For example:

```
tarantool> box.space._schema:select{'cluster'}
---
- - ['cluster', '6308acb9-9788-42fa-8101-2e0cb9d3c9a0']
...
```

Additionally, each instance in a replica set is assigned its own UUID, when it joins the replica set. It is called an instance UUID and is a globally unique identifier. The instance UUID is checked to ensure that instances do not join a different replica set, e.g. because of a configuration error. A unique instance identifier is also necessary to apply rows originating from different masters only once, that is, to implement multi-master replication. This is why each row in the write ahead log, in addition to its log sequence number, stores the instance identifier of the instance on which it was created. But using a UUID as such an identifier would take too much space in the write ahead log, thus a shorter integer number is assigned to the instance when it joins a replica set. This number is then used to refer to the instance in the write ahead log. It is called instance id. All identifiers are stored in system space box.space._cluster. For example:

```
tarantool> box.space._cluster:select{}
---
- - [1, '88580b5c-4474-43ab-bd2b-2409a9af80d2']
...
```

Here the instance ID is 1 (unique within the replica set), and the instance UUID is 88580b5c-4474-43ab-bd2b-2409a9af80d2 (globally unique).

Using instance IDs is also handy for tracking the state of the entire replica set. For example, box.info.vclock describes the state of replication in regard to each connected peer.

```
tarantool> box.info.vclock
---
- {1: 827, 2: 584}
...
```

Here vclock contains log sequence numbers (827 and 584) for instances with instance IDs 1 and 2.

Starting in Tarantool 1.7.7, it is possible for administrators to assign the instance UUID and the replica set UUID values, rather than let the system generate them – see the description of the replicaset_uuid configuration parameter.

Replication setup

To enable replication, you need to specify two parameters in a box.cfg{} request:

- replication which defines the replication source(s), and
- read_only which is true for a replica and false for a master.

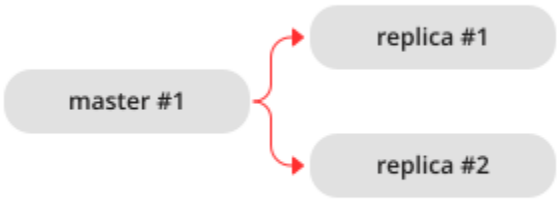
Both these parameters are "dynamic". This allows a replica to become a master and vice versa on the fly with the help of a box.cfg{} request.

Later we will give a detailed example of bootstrapping a replica set.

Replication roles: master and replica

The replication role (master or replica) is set by the read_only configuration parameter. The recommended role is "read_only" (replica) for all but one instance in the replica set.

In a master-replica configuration, every change that happens on the master will be visible on the replicas, but not vice versa.



A simple two-instance replica set with the master on one machine and the replica on a different machine provides two benefits:

- failover, because if the master goes down then the replica can take over, and
- load balancing, because clients can connect to either the master or the replica for read requests.

In a master-master configuration (also called "multi-master"), every change that happens on either instance will be visible on the other one.

The failover benefit in this case is still present, and the load-balancing benefit is enhanced, because any instance can handle both read and write requests. Meanwhile, for multi-master configurations, it is necessary to understand the replication guarantees provided by the asynchronous protocol that Tarantool implements.

Tarantool multi-master replication guarantees that each change on each master is propagated to all instances and is applied only once. Changes from the same instance are applied in the same order as on the originating instance. Changes from different instances, however, can be mixed and applied in a different order on different instances. This may lead to replication going out of sync in certain cases.

For example, assuming the database is only appended to (i.e. it contains only insertions), a multi-master configuration is safe. If there are also deletions, but it is not mission critical that deletion happens in the same order on all replicas (e.g. the DELETE is used to prune expired data), a master-master configuration is also safe.

UPDATE operations, however, can easily go out of sync. For example, assignment and increment are not commutative, and may yield different results if applied in different order on different instances.

More generally, it is only safe to use Tarantool master-master replication if all database changes are commutative: the end result does not depend on the order in which the changes are applied. You can start learning more about conflict-free replicated data types here.

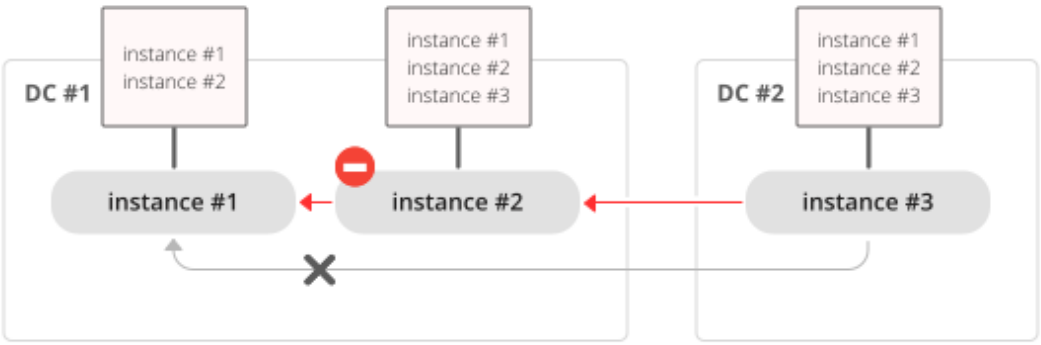
Replication topologies: cascade, ring and full mesh

Replication topology is set by the replication configuration parameter. The recommended topology is a full mesh, because it makes potential failover easy.

Some database products offer cascading replication topologies: creating a replica on a replica. Tarantool does not recommend such setup.

The problem with a cascading replica set is that some instances have no connection to other instances and may not receive changes from them. One essential change that must be propagated across all instances in a replica set is an entry in box.space._cluster system space with the replica set UUID. Without knowing the replica set UUID, a master refuses to accept connections from such instances when replication topology changes. Here is how this can happen:

We have a chain of three instances. Instance #1 contains entries for instances #1 and #2 in its _cluster space. Instances #2 and #3 contain entries for instances #1, #2 and #3 in their _cluster spaces.



Now instance #2 is faulty. Instance #3 tries connecting to instance #1 as its new master, but the master refuses the connection since it has no entry for instance #3.

Ring replication topology is, however, supported:

So, if you need a cascading topology, you may first create a ring to ensure all instances know each other's UUID, and then disconnect the chain in the place you desire.

A stock recommendation for a master-master replication topology, however, is a full mesh:

You then can decide where to locate instances of the mesh – within the same data center, or spread across a few data centers. Tarantool will automatically ensure that each row is applied only once on each instance. To remove a degraded instance from a mesh, simply change the replication configuration parameter.

This ensures full cluster availability in case of a local failure, e.g. one of the instances failing in one of the data centers, as well as in case of an entire data center failure.

The maximal number of replicas in a mesh is 32.

3.6.2 Bootstrapping a replica set

Master-replica bootstrap

Let us first bootstrap a simple master-replica set containing two instances, each located on its own machine. For easier administration, we make the instance files almost identical.

Here is an example of the master's instance file:

```
-- instance file for the master
box.cfg{
  listen = 3301,
  replication = {'replicator:password@192.168.0.101:3301',  -- master URI
                 'replicator:password@192.168.0.102:3301'}, -- replica URI
  read_only = false
}
box.once("schema", function()
   box.schema.user.create('replicator', {password = 'password'})
   box.schema.user.grant('replicator', 'replication') -- grant replication role
   box.schema.space.create("test")
   box.space.test:create_index("primary")
   print('box.once executed on master')
end)
```

where:

- the box.cfg() listen parameter defines a URI (port 3301 in our example), on which the master can accept connections from replicas.
- the box.cfg() replication parameter defines the URIs at which all instances in the replica set can accept connections. It includes the replica's URI as well, although the replica is not a replication source right now.

  Note: For security reasons, we recommend that administrators prevent unauthorized replication sources by associating a password with every user that has a replication role. That way, the URI for replication parameter must have the long form username:password@host:port.

- the read_only = false parameter setting enables data-change operations on the instance and makes the instance act as a master, not as a replica. That is the only parameter setting in our instance files that will differ.
- the box.once() function contains database initialization logic that should be executed only once during the replica set lifetime.

In this example, we create a space with a primary index, and a user for replication purposes. We also say print('box.once executed on master') so that it will later be visible on a console whether box.once() was executed.

Note: Replication requires privileges. We can grant privileges for accessing spaces directly to the user who will start the instance. However, it is more usual to grant privileges for accessing spaces to a role, and then grant the role to the user who will start the replica.

Here we use Tarantool's predefined role named "replication" which by default grants "read" privileges for all database objects ("universe"), and we can change privileges for this role as required.

In the replica's instance file, we set the read_only parameter to "true", and say print('box.once executed on replica') so that later it will be visible that box.once() was not executed more than once. Otherwise the replica's instance file is identical to the master's instance file.

```
-- instance file for the replica
box.cfg{
  listen = 3301,
  replication = {'replicator:password@192.168.0.101:3301',  -- master URI
                 'replicator:password@192.168.0.102:3301'}, -- replica URI
```

(continues on next page)

(continued from previous page)

```
  read_only = true
}
box.once("schema", function()
   box.schema.user.create('replicator', {password = 'password'})
   box.schema.user.grant('replicator', 'replication') -- grant replication role
   box.schema.space.create("test")
   box.space.test:create_index("primary")
   print('box.once executed on replica')
end)
```

Note: The replica does not inherit the master's configuration parameters, such as those making the checkpoint daemon run on the master. To get the same behavior, set the relevant parameters explicitly so that they are the same on both master and replica.

Now we can launch the two instances. The master...

```
$ # launching the master
$ tarantool master.lua
2017-06-14 14:12:03.847 [18933] main/101/master.lua C> version 1.7.4-52-g980d30092
2017-06-14 14:12:03.848 [18933] main/101/master.lua C> log level 5
2017-06-14 14:12:03.849 [18933] main/101/master.lua I> mapping 268435456 bytes for tuple arena...
2017-06-14 14:12:03.859 [18933] iproto/101/main I> binary: bound to [::]:3301
2017-06-14 14:12:03.861 [18933] main/105/applier/replicator@192.168.0. I> can't connect to master
2017-06-14 14:12:03.861 [18933] main/105/applier/replicator@192.168.0. coio.cc:105 !> SystemError connect,␣
↪called on fd 14, aka 192.168.0.102:56736: Connection refused
2017-06-14 14:12:03.861 [18933] main/105/applier/replicator@192.168.0. I> will retry every 1 second
2017-06-14 14:12:03.861 [18933] main/104/applier/replicator@192.168.0. I> remote master is 1.7.4 at 192.168.0.
↪101:3301
2017-06-14 14:12:19.878 [18933] main/105/applier/replicator@192.168.0. I> remote master is 1.7.4 at 192.168.0.
↪102:3301
2017-06-14 14:12:19.879 [18933] main/101/master.lua I> initializing an empty data directory
2017-06-14 14:12:19.908 [18933] snapshot/101/main I> saving snapshot `/var/lib/tarantool/master/
↪00000000000000000000.snap.inprogress'
2017-06-14 14:12:19.914 [18933] snapshot/101/main I> done
2017-06-14 14:12:19.914 [18933] main/101/master.lua I> vinyl checkpoint done
2017-06-14 14:12:19.917 [18933] main/101/master.lua I> ready to accept requests
2017-06-14 14:12:19.918 [18933] main/105/applier/replicator@192.168.0. I> failed to authenticate
2017-06-14 14:12:19.918 [18933] main/105/applier/replicator@192.168.0. xrow.cc:431 E> ER_LOADING: Instance␣
↪bootstrap hasn't finished yet
box.once executed on master
2017-06-14 14:12:19.920 [18933] main C> entering the event loop
```

... (the display confirms that box.once() was executed on the master) – and the replica:

```
$ # launching the replica
$ tarantool replica.lua
2017-06-14 14:12:19.486 [18934] main/101/replica.lua C> version 1.7.4-52-g980d30092
2017-06-14 14:12:19.486 [18934] main/101/replica.lua C> log level 5
2017-06-14 14:12:19.487 [18934] main/101/replica.lua I> mapping 268435456 bytes for tuple arena...
2017-06-14 14:12:19.494 [18934] iproto/101/main I> binary: bound to [::]:3311
2017-06-14 14:12:19.495 [18934] main/104/applier/replicator@192.168.0. I> remote master is 1.7.4 at 192.168.0.
↪101:3301
2017-06-14 14:12:19.495 [18934] main/105/applier/replicator@192.168.0. I> remote master is 1.7.4 at 192.168.0.
↪102:3302
```

(continues on next page)

(continued from previous page)

```
2017-06-14 14:12:19.496 [18934] main/104/applier/replicator@192.168.0. I> failed to authenticate
2017-06-14 14:12:19.496 [18934] main/104/applier/replicator@192.168.0. xrow.cc:431 E> ER_LOADING: Instance␣
↪bootstrap hasn't finished yet
```

In both logs, there are messages saying that the replica was bootstrapped from the master:

```
$ # bootstrapping the replica (from the master 's log)
<...>
2017-06-14 14:12:20.503 [18933] main/106/main I> initial data sent.
2017-06-14 14:12:20.505 [18933] relay/[::ffff:192.168.0.101]:/101/main I> recover from `/var/lib/tarantool/master/
↪00000000000000000000.xlog'
2017-06-14 14:12:20.505 [18933] main/106/main I> final data sent.
2017-06-14 14:12:20.522 [18933] relay/[::ffff:192.168.0.101]:/101/main I> recover from `/Users/e.shebunyaeva/
↪work/tarantool-test-repl/master_dir/00000000000000000000.xlog'
2017-06-14 14:12:20.922 [18933] main/105/applier/replicator@192.168.0. I> authenticated
```

```
$ # bootstrapping the replica (from the replica 's log)
<...>
2017-06-14 14:12:20.498 [18934] main/104/applier/replicator@192.168.0. I> authenticated
2017-06-14 14:12:20.498 [18934] main/101/replica.lua I> bootstrapping replica from 192.168.0.101:3301
2017-06-14 14:12:20.512 [18934] main/104/applier/replicator@192.168.0. I> initial data received
2017-06-14 14:12:20.512 [18934] main/104/applier/replicator@192.168.0. I> final data received
2017-06-14 14:12:20.517 [18934] snapshot/101/main I> saving snapshot `/var/lib/tarantool/replica/
↪00000000000000000005.snap.inprogress'
2017-06-14 14:12:20.518 [18934] snapshot/101/main I> done
2017-06-14 14:12:20.519 [18934] main/101/replica.lua I> vinyl checkpoint done
2017-06-14 14:12:20.520 [18934] main/101/replica.lua I> ready to accept requests
2017-06-14 14:12:20.520 [18934] main/101/replica.lua I> set 'read_only' configuration option to true
2017-06-14 14:12:20.520 [18934] main C> entering the event loop
```

Notice that box.once() was executed only at the master, although we added box.once() to both instance files.

We could as well launch the replica first:

```
$ # launching the replica
$ tarantool replica.lua
2017-06-14 14:35:36.763 [18952] main/101/replica.lua C> version 1.7.4-52-g980d30092
2017-06-14 14:35:36.765 [18952] main/101/replica.lua C> log level 5
2017-06-14 14:35:36.765 [18952] main/101/replica.lua I> mapping 268435456 bytes for tuple arena...
2017-06-14 14:35:36.772 [18952] iproto/101/main I> binary: bound to [::]:3301
2017-06-14 14:35:36.772 [18952] main/104/applier/replicator@192.168.0. I> can't connect to master
2017-06-14 14:35:36.772 [18952] main/104/applier/replicator@192.168.0. coio.cc:105 !> SystemError connect,␣
↪called on fd 13, aka 192.168.0.101:56820: Connection refused
2017-06-14 14:35:36.772 [18952] main/104/applier/replicator@192.168.0. I> will retry every 1 second
2017-06-14 14:35:36.772 [18952] main/105/applier/replicator@192.168.0. I> remote master is 1.7.4 at 192.168.0.
↪102:3301
```

. . . and the master later:

```
$ # launching the master
$ tarantool master.lua
2017-06-14 14:35:43.701 [18953] main/101/master.lua C> version 1.7.4-52-g980d30092
2017-06-14 14:35:43.702 [18953] main/101/master.lua C> log level 5
2017-06-14 14:35:43.702 [18953] main/101/master.lua I> mapping 268435456 bytes for tuple arena...
2017-06-14 14:35:43.709 [18953] iproto/101/main I> binary: bound to [::]:3301
2017-06-14 14:35:43.709 [18953] main/105/applier/replicator@192.168.0. I> remote master is 1.7.4 at 192.168.0.
↪102:3301
```

(continues on next page)

(continued from previous page)

```
2017-06-14 14:35:43.709 [18953] main/104/applier/replicator@192.168.0. I> remote master is 1.7.4 at 192.168.0.
↪101:3301
2017-06-14 14:35:43.709 [18953] main/101/master.lua I> initializing an empty data directory
2017-06-14 14:35:43.721 [18953] snapshot/101/main I> saving snapshot `/var/lib/tarantool/master/
↪00000000000000000000.snap.inprogress'
2017-06-14 14:35:43.722 [18953] snapshot/101/main I> done
2017-06-14 14:35:43.723 [18953] main/101/master.lua I> vinyl checkpoint done
2017-06-14 14:35:43.723 [18953] main/101/master.lua I> ready to accept requests
2017-06-14 14:35:43.724 [18953] main/105/applier/replicator@192.168.0. I> failed to authenticate
2017-06-14 14:35:43.724 [18953] main/105/applier/replicator@192.168.0. xrow.cc:431 E> ER_LOADING: Instance␣
↪bootstrap hasn't finished yet
box.once executed on master
2017-06-14 14:35:43.726 [18953] main C> entering the event loop
2017-06-14 14:35:43.779 [18953] main/103/main I> initial data sent.
2017-06-14 14:35:43.780 [18953] relay/[::ffff:192.168.0.101]:/101/main I> recover from `/var/lib/tarantool/master/
↪00000000000000000000.xlog'
2017-06-14 14:35:43.780 [18953] main/103/main I> final data sent.
2017-06-14 14:35:43.796 [18953] relay/[::ffff:192.168.0.102]:/101/main I> recover from `/var/lib/tarantool/master/
↪00000000000000000000.xlog'
2017-06-14 14:35:44.726 [18953] main/105/applier/replicator@192.168.0. I> authenticated
```

In this case, the replica would wait for the master to become available, so the launch order doesn't matter. Our box.once() logic would also be executed only once, at the master.

```
$ # the replica has eventually connected to the master
$ # and got bootstrapped (from the replica's log)
2017-06-14 14:35:43.777 [18952] main/104/applier/replicator@192.168.0. I> remote master is 1.7.4 at 192.168.0.
↪101:3301
2017-06-14 14:35:43.777 [18952] main/104/applier/replicator@192.168.0. I> authenticated
2017-06-14 14:35:43.777 [18952] main/101/replica.lua I> bootstrapping replica from 192.168.0.199:3310
2017-06-14 14:35:43.788 [18952] main/104/applier/replicator@192.168.0. I> initial data received
2017-06-14 14:35:43.789 [18952] main/104/applier/replicator@192.168.0. I> final data received
2017-06-14 14:35:43.793 [18952] snapshot/101/main I> saving snapshot `/var/lib/tarantool/replica/
↪00000000000000000005.snap.inprogress'
2017-06-14 14:35:43.793 [18952] snapshot/101/main I> done
2017-06-14 14:35:43.795 [18952] main/101/replica.lua I> vinyl checkpoint done
2017-06-14 14:35:43.795 [18952] main/101/replica.lua I> ready to accept requests
2017-06-14 14:35:43.795 [18952] main/101/replica.lua I> set 'read_only' configuration option to true
2017-06-14 14:35:43.795 [18952] main C> entering the event loop
```

Controlled failover

To perform a controlled failover, that is, swap the roles of the master and replica, all we need to do is to set read_only=true at the master, and read_only=false at the replica. The order of actions is important here. If a system is running in production, we do not want concurrent writes happening both at the replica and the master. Nor do we want the new replica to accept any writes until it has finished fetching all replication data from the old master. To compare replica and master state, we can use box.info.signature.

1. Set read_only=true at the master.

   ```
   # at the master
   tarantool> box.cfg{read_only=true}
   ```

2. Record the master's current position with box.info.signature, containing the sum of all LSNs in the master's vector clock.

```
# at the master
tarantool> box.info.signature
```

3. Wait until the replica's signature is the same as the master's.

```
# at the replica
tarantool> box.info.signature
```

4. Set read_only=false at the replica to enable write operations.

```
# at the replica
tarantool> box.cfg{read_only=false}
```

These four steps ensure that the replica doesn't accept new writes until it's done fetching writes from the master.

Master-master bootstrap

Now let us bootstrap a two-instance master-master set. For easier administration, we make master#1 and master#2 instance files fully identical.

We re-use the master's instance file from the master-replica example above.

```
-- instance file for any of the two masters
box.cfg{
  listen      = 3301,
  replication = {'replicator:password@192.168.0.101:3301',  -- master1 URI
                 'replicator:password@192.168.0.102:3301'}, -- master2 URI
  read_only   = false
}
box.once("schema", function()
   box.schema.user.create('replicator', {password = 'password'})
   box.schema.user.grant('replicator', 'replication') -- grant replication role
   box.schema.space.create("test")
   box.space.test:create_index("primary")
   print('box.once executed on master #1')
end)
```

In the replication parameter, we define the URIs of both masters in the replica set and say print('box.once executed on master #1') so it will be clear when and where the box.once() logic is executed.

Now we can launch the two masters. Again, the launch order doesn't matter. The box.once() logic will also be executed only once, at the master which is elected as the replica set leader at bootstrap.

```
$ # launching master #1
$ tarantool master1.lua
2017-06-14 15:39:03.062 [47021] main/101/master1.lua C> version 1.7.4-52-g980d30092
2017-06-14 15:39:03.062 [47021] main/101/master1.lua C> log level 5
2017-06-14 15:39:03.063 [47021] main/101/master1.lua I> mapping 268435456 bytes for tuple arena...
2017-06-14 15:39:03.065 [47021] iproto/101/main I> binary: bound to [::]:3301
2017-06-14 15:39:03.065 [47021] main/105/applier/replicator@192.168.0.10 I> can't connect to master
2017-06-14 15:39:03.065 [47021] main/105/applier/replicator@192.168.0.10 coio.cc:107 !> SystemError connect,
↪called on fd 14, aka 192.168.0.102:57110: Connection refused
```

(continues on next page)

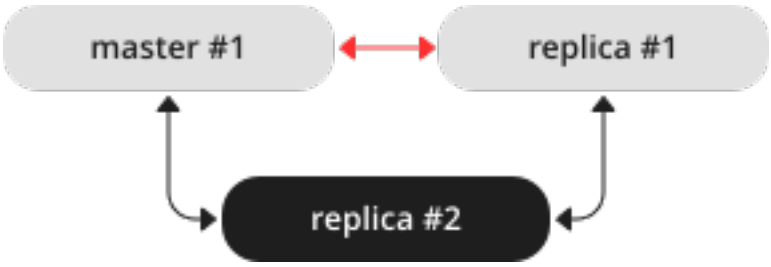
(continued from previous page)

```
2017-06-14 15:39:03.065 [47021] main/105/applier/replicator@192.168.0.10 I> will retry every 1 second
2017-06-14 15:39:03.065 [47021] main/104/applier/replicator@192.168.0.10 I> remote master is 1.7.4 at 192.168.0.
↪101:3301
2017-06-14 15:39:08.070 [47021] main/105/applier/replicator@192.168.0.10 I> remote master is 1.7.4 at 192.168.0.
↪102:3301
2017-06-14 15:39:08.071 [47021] main/105/applier/replicator@192.168.0.10 I> authenticated
2017-06-14 15:39:08.071 [47021] main/101/master1.lua I> bootstrapping replica from 192.168.0.102:3301
2017-06-14 15:39:08.073 [47021] main/105/applier/replicator@192.168.0.10 I> initial data received
2017-06-14 15:39:08.074 [47021] main/105/applier/replicator@192.168.0.10 I> final data received
2017-06-14 15:39:08.074 [47021] snapshot/101/main I> saving snapshot `/Users/e.shebunyaeva/work/tarantool-
↪test-repl/master1_dir/00000000000000000008.snap.inprogress'
2017-06-14 15:39:08.074 [47021] snapshot/101/main I> done
2017-06-14 15:39:08.076 [47021] main/101/master1.lua I> vinyl checkpoint done
2017-06-14 15:39:08.076 [47021] main/101/master1.lua I> ready to accept requests
box.once executed on master #1
2017-06-14 15:39:08.077 [47021] main C> entering the event loop
```

```
$ # launching master #2
$ tarantool master2.lua
2017-06-14 15:39:07.452 [47022] main/101/master2.lua C> version 1.7.4-52-g980d30092
2017-06-14 15:39:07.453 [47022] main/101/master2.lua C> log level 5
2017-06-14 15:39:07.453 [47022] main/101/master2.lua I> mapping 268435456 bytes for tuple arena...
2017-06-14 15:39:07.455 [47022] iproto/101/main I> binary: bound to [::]:3301
2017-06-14 15:39:07.455 [47022] main/104/applier/replicator@192.168.0.19 I> remote master is 1.7.4 at 192.168.0.
↪101:3301
2017-06-14 15:39:07.455 [47022] main/105/applier/replicator@192.168.0.10 I> remote master is 1.7.4 at 192.168.0.
↪102:3301
2017-06-14 15:39:07.455 [47022] main/101/master2.lua I> initializing an empty data directory
2017-06-14 15:39:07.457 [47022] snapshot/101/main I> saving snapshot `/Users/e.shebunyaeva/work/tarantool-
↪test-repl/master2_dir/00000000000000000000.snap.inprogress'
2017-06-14 15:39:07.457 [47022] snapshot/101/main I> done
2017-06-14 15:39:07.458 [47022] main/101/master2.lua I> vinyl checkpoint done
2017-06-14 15:39:07.459 [47022] main/101/master2.lua I> ready to accept requests
2017-06-14 15:39:07.460 [47022] main C> entering the event loop
2017-06-14 15:39:08.072 [47022] main/103/main I> initial data sent.
2017-06-14 15:39:08.073 [47022] relay/[::ffff:192.168.0.102]:/101/main I> recover from `/Users/e.shebunyaeva/
↪work/tarantool-test-repl/master2_dir/00000000000000000000.xlog'
2017-06-14 15:39:08.073 [47022] main/103/main I> final data sent.
2017-06-14 15:39:08.077 [47022] relay/[::ffff:192.168.0.102]:/101/main I> recover from `/Users/e.shebunyaeva/
↪work/tarantool-test-repl/master2_dir/00000000000000000000.xlog'
2017-06-14 15:39:08.461 [47022] main/104/applier/replicator@192.168.0.10 I> authenticated
```

3.6.3 Adding instances

Adding a replica



To add a second replica instance to the master-replica set from our bootstrapping example, we need an analog of the instance file that we created for the first replica in that set:

```
-- instance file for replica #2
box.cfg{
  listen = 3301,
  replication = ('replicator:password@192.168.0.101:3301',  -- master URI
                 'replicator:password@192.168.0.102:3301',  -- replica #1 URI
                 'replicator:password@192.168.0.103:3301'), -- replica #2 URI
  read_only = true
}
box.once("schema", function()
   box.schema.user.create('replicator', {password = 'password'})
   box.schema.user.grant('replicator', 'replication') -- grant replication role
   box.schema.space.create("test")
   box.space.test:create_index("primary")
   print('box.once executed on replica #2')
end)
```

Here we add the URI of replica #2 to the replication parameter, so now it contains three URIs.

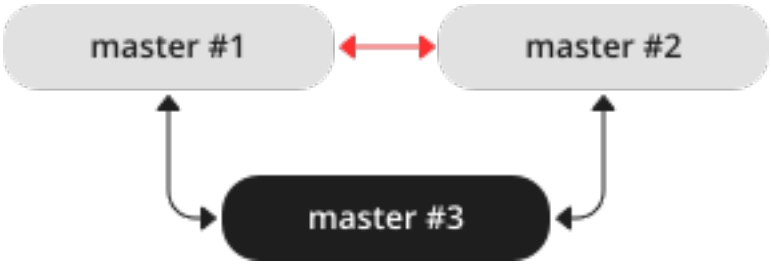
After we launch the new replica instance, it gets connected to the master instance and retrieves the master's write-ahead-log and snapshot files:

```
$ # launching replica #2
$ tarantool replica2.lua
2017-06-14 14:54:33.927 [46945] main/101/replica2.lua C> version 1.7.4-52-g980d30092
2017-06-14 14:54:33.927 [46945] main/101/replica2.lua C> log level 5
2017-06-14 14:54:33.928 [46945] main/101/replica2.lua I> mapping 268435456 bytes for tuple arena...
2017-06-14 14:54:33.930 [46945] main/104/applier/replicator@192.168.0.10 I> remote master is 1.7.4 at 192.168.0.
↪101:3301
2017-06-14 14:54:33.930 [46945] main/104/applier/replicator@192.168.0.10 I> authenticated
2017-06-14 14:54:33.930 [46945] main/101/replica2.lua I> bootstrapping replica from 192.168.0.101:3301
2017-06-14 14:54:33.933 [46945] main/104/applier/replicator@192.168.0.10 I> initial data received
2017-06-14 14:54:33.933 [46945] main/104/applier/replicator@192.168.0.10 I> final data received
2017-06-14 14:54:33.934 [46945] snapshot/101/main I> saving snapshot `/var/lib/tarantool/replica2/
↪00000000000000000010.snap.inprogress'
2017-06-14 14:54:33.934 [46945] snapshot/101/main I> done
2017-06-14 14:54:33.935 [46945] main/101/replica2.lua I> vinyl checkpoint done
2017-06-14 14:54:33.935 [46945] main/101/replica2.lua I> ready to accept requests
2017-06-14 14:54:33.935 [46945] main/101/replica2.lua I> set 'read_only' configuration option to true
2017-06-14 14:54:33.936 [46945] main C> entering the event loop
```

Since we are adding a read-only instance, there is no need to dynamically update the replication parameter on the other running instances. This update would be required if we added a master instance.

However, we recommend specifying the URI of replica #3 in all instance files of the replica set. This will keep all the files consistent with each other and with the current replication topology, and so will help to avoid configuration errors in case of further configuration updates and replica set restart.

Adding a master



To add a third master instance to the master-master set from our bootstrapping example, we need an analog of the instance files that we created to bootstrap the other master instances in that set:

```
-- instance file for master #3
box.cfg{
  listen      = 3301,
  replication = {'replicator:password@192.168.0.101:3301',  -- master#1 URI
                 'replicator:password@192.168.0.102:3301',  -- master#2 URI
                 'replicator:password@192.168.0.103:3301'}, -- master#3 URI
  read_only   = true, -- temporarily read-only
}
box.once("schema", function()
   box.schema.user.create('replicator', {password = 'password'})
   box.schema.user.grant('replicator', 'replication') -- grant "replication" role
   box.schema.space.create("test")
   box.space.test:create_index("primary")
end)
```

Here we make the following changes:

- Add the URI of master #3 to the replication parameter.
- Temporarily specify read_only=true to disable data-change operations on the instance. After launch, master #3 will act as a replica until it retrieves all data from the other masters in the replica set.

After we launch master #3, it gets connected to the other master instances and retrieves their write-ahead-log and snapshot files:

```
$ # launching master #3
$ tarantool master3.lua
2017-06-14 17:10:00.556 [47121] main/101/master3.lua C> version 1.7.4-52-g980d30092
2017-06-14 17:10:00.557 [47121] main/101/master3.lua C> log level 5
2017-06-14 17:10:00.557 [47121] main/101/master3.lua I> mapping 268435456 bytes for tuple arena...
2017-06-14 17:10:00.559 [47121] iproto/101/main I> binary: bound to [::]:3301
2017-06-14 17:10:00.559 [47121] main/104/applier/replicator@192.168.0.10 I> remote master is 1.7.4 at 192.168.0.
↪101:3301
2017-06-14 17:10:00.559 [47121] main/105/applier/replicator@192.168.0.10 I> remote master is 1.7.4 at 192.168.0.
↪102:3301
2017-06-14 17:10:00.559 [47121] main/106/applier/replicator@192.168.0.10 I> remote master is 1.7.4 at 192.168.0.
↪103:3301
2017-06-14 17:10:00.559 [47121] main/105/applier/replicator@192.168.0.10 I> authenticated
2017-06-14 17:10:00.559 [47121] main/101/master3.lua I> bootstrapping replica from 192.168.0.102:3301
2017-06-14 17:10:00.562 [47121] main/105/applier/replicator@192.168.0.10 I> initial data received
2017-06-14 17:10:00.562 [47121] main/105/applier/replicator@192.168.0.10 I> final data received
2017-06-14 17:10:00.562 [47121] snapshot/101/main I> saving snapshot `/Users/e.shebunyaeva/work/tarantool-
↪test-repl/master3_dir/00000000000000000009.snap.inprogress'
2017-06-14 17:10:00.562 [47121] snapshot/101/main I> done
```

(continues on next page)

(continued from previous page)

```
2017-06-14 17:10:00.564 [47121] main/101/master3.lua I> vinyl checkpoint done
2017-06-14 17:10:00.564 [47121] main/101/master3.lua I> ready to accept requests
2017-06-14 17:10:00.565 [47121] main/101/master3.lua I> set 'read_only' configuration option to true
2017-06-14 17:10:00.565 [47121] main C> entering the event loop
2017-06-14 17:10:00.565 [47121] main/104/applier/replicator@192.168.0.10 I> authenticated
```

Next, we add the URI of master #3 to the replication parameter on the existing two masters. Replication-related parameters are dynamic, so we only need to make a box.cfg{} request on each of the running instances:

```
# adding master #3 URI to replication sources
tarantool> box.cfg{replication =
         > {'replicator:password@192.168.0.101:3301',
         > 'replicator:password@192.168.0.102:3301',
         > 'replicator:password@192.168.0.103:3301'}}
---
...
```

When master #3 catches up with the other masters' state, we can disable read-only mode for this instance:

```
# making master #3 a real master
tarantool> box.cfg{read_only=false}
---
...
```

We also recommend to specify master #3 URI in all instance files in order to keep all the files consistent with each other and with the current replication topology.

Orphan status

Starting with Tarantool version 1.9, there is a change to the procedure when an instance joins a replica set. During box.cfg() the instance will try to join all masters listed in box.cfg.replication. If the instance does not succeed with at least the number of masters specified in replication_connect_quorum, then it will switch to orphan status. While an instance is in orphan status, it is read-only.

To "join" a master, a replica instance must "connect" to the master node and then "sync".

"Connect" means contact the master over the physical network and receive acknowledgment. If there is no acknowledgment after box.replication_connect_timeout seconds (usually 4 seconds), and retries fail, then the connect step fails.

"Sync" means receive updates from the master in order to make a local database copy. Syncing is complete when the replica has received all the updates, or at least has received enough updates that the replica's lag (see replication.upstream.lag in box.info()) is less than or equal to the number of seconds specified in box.cfg.replication_sync_lag. If replication_sync_lag is unset (nil) or set to TIMEOUT_INFINITY, then the replica skips the "sync" state and switches to "follow" immediately.

The following situations are possible.

Situation 1: bootstrap

Here box.cfg{} is being called for the first time. A replica is joining but no replica set exists yet.

1. Set status to 'orphan'.
2. Try to connect to all nodes from box.cfg.replication, or to the number of nodes required by replication_connect_quorum. Retrying up to 3 times in 30 seconds is possible because this is bootstrap, replication_connect_timeout is overridden.

3. Abort if not connected to all nodes in box.cfg.replication or replication_connect_quorum.
4. This instance might be elected as the replica set 'leader'. Criteria for electing a leader include vclock value (largest is best), and whether it is read-only or read-write (read-write is best unless there is no other choice). The leader is the master that other instances must join. The leader is the master that executes box_once() functions.
5. If this instance is elected as the replica set leader, then perform an "automatic bootstrap":
   a. Set status to 'running'.
   b. Return from box.cfg{}.

   Otherwise this instance will be a replica joining an existing replica set, so:
   a. Bootstrap from the leader. See examples in section Bootstrapping a replica set.
   b. In background, sync with all the other nodes in the replication set.

Situation 2: recovery

Here box.cfg{} is not being called for the first time. It is being called again in order to perform recovery.

1. Perform recovery from the last local snapshot and the WAL files.
2. Connect to at least replication_connect_quorum nodes.
3. Sync with all connected nodes, until the difference is not more than replication_sync_lag seconds.

Situation 3: configuration update

Here box.cfg{} is not being called for the first time. It is being called again because some replication parameter or something in the replica set has changed.

1. Try to connect to all nodes from box.cfg.replication, or to the number of nodes required by replication_connect_quorum, within the time period specified in replication_connect_timeout.
2. Try to sync with the connected nodes, within the time period specified in replication_sync_timeout.
3. If earlier steps fail, change status to 'orphan'. (Attempts to sync will continue in the background and when/if they succeed then 'orphan' status will end.)
4. If earlier steps succeed, set status to 'running' (master) or 'follow' (replica).

Situation 4: rebootstrap

Here box.cfg{} is not being called. The replica connected successfully at some point in the past, and is now ready for an update from the master. But the master cannot provide an update. This can happen by accident, or more likely can happen because the replica is slow (its lag is large), and the WAL (.xlog) files containing the updates have been deleted. This is not crippling. The replica can discard what it received earlier, and then ask for the master's latest snapshot (.snap) file contents. Since it is effectively going through the bootstrap process a second time, this is called "rebootstrapping". However, there has to be one difference from an ordinary bootstrap – the replica's replica id will remain the same. If it changed, then the master would think that the replica is a new addition to the cluster, and would maintain a record of an instance ID of a replica that has ceased to exist. Rebootstrapping was introduced in Tarantool version 1.10.2 and is completely automatic.

Server startup with replication

In addition to the recovery process described in the section Recovery process, 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 which specifies replication source(-s). We will refer to this replica, which is starting up due to

box.cfg, as the "local" replica to distinguish it from the other replicas in a replica set, which we will refer to as "distant" replicas.

If there is no snapshot .snap file and the 'replication' parameter is empty: then the local replica assumes it is an unreplicated "standalone" instance, or is the first replica of a new replica set. It will generate new UUIDs for itself and for the replica set. The replica UUID is stored in the _cluster space; the replica set UUID is stored in the _schema space. Since a snapshot contains all the data in all the spaces, that means the local replica's snapshot will contain the replica UUID and the replica set UUID. Therefore, when the local replica restarts on later occasions, it will be able to recover these UUIDs when it reads the .snap file.

If there is no snapshot .snap file and the 'replication' parameter is not empty and the '_cluster' space contains no other replica UUIDs: then the local replica assumes it is not a standalone instance, but is not yet part of a replica set. It must now join the replica set. It will send its replica UUID to the first distant replica which is listed in replication and which will act as a master. This is called the "join request". When a distant replica receives a join request, it will send back:

(1) the distant replica's replica set UUID,

(2) the contents of the distant replica's .snap file. When the local replica receives this information, it puts the replica set UUID in its _schema space, puts the distant replica's UUID and connection information in its _cluster space, and makes a snapshot containing all the data sent by the distant replica. Then, if the local replica has data in its WAL .xlog files, it sends that data to the distant replica. The distant replica will receive this and update its own copy of the data, and add the local replica's UUID to its _cluster space.

If there is no snapshot .snap file and the 'replication' parameter is not empty and the "_cluster" space contains other replica UUIDs: then the local replica assumes it is not a standalone instance, and is already part of a replica set. It will send its replica UUID and replica set UUID to all the distant replicas which are listed in replication. This is called the "on-connect handshake". When a distant replica receives an on-connect handshake:

(1) the distant replica compares its own copy of the replica set UUID to the one in the on-connect handshake. If there is no match, then the handshake fails and the local replica will display an error.

(2) the distant replica looks for a record of the connecting instance in its _cluster space. If there is none, then the handshake fails. Otherwise the handshake is successful. The distant replica will read any new information from its own .snap and .xlog files, and send the new requests to the local replica.

In the end, the local replica knows what replica set it belongs to, the distant replica knows that the local replica is a member of the replica set, and both replicas have the same database contents.

If there is a snapshot file and replication source is not empty: first the local replica goes through the recovery process described in the previous section, using its own .snap and .xlog files. Then it sends a "subscribe" request to all the other replicas of the replica set. The subscribe request contains the server vector clock. The vector clock has a collection of pairs 'server id, lsn' for every replica in the _cluster system space. Each distant replica, upon receiving a subscribe request, will read its .xlog files' requests and send them to the local replica if (lsn of .xlog file request) is greater than (lsn of the vector clock in the subscribe request). After all the other replicas of the replica set have responded to the local replica's subscribe request, the replica startup is complete.

The following temporary limitations applied for Tarantool versions earlier than 1.7.7:

- The URIs in the replication parameter should all be in the same order on all replicas. This is not mandatory but is an aid to consistency.
- The replicas of a replica set should be started up at slightly different times. This is not mandatory but prevents a situation where each replica is waiting for the other replica to be ready.

The following limitation still applies for the current Tarantool version:

- 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.

3.6.4 Removing instances

To remove an instance from a replica set politely, follow these steps:

1. On the instance, run box.cfg{} with a blank replication source:

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

   The other instances in the replica set will carry on. If later the removed instance rejoins, it will receive all the updates that the other instances made while it was away.

2. If the instance is decommissioned forever, delete the instance's record from the following locations:

   a. the replication parameter at all running instances in the replica set:

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

   b. the box.space._cluster tuple on any master instance in the replica set. For example, for a record with instance id = 3:

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

3.6.5 Monitoring a replica set

To learn what instances belong in the replica set, and obtain statistics for all these instances, issue a box.info.replication request:

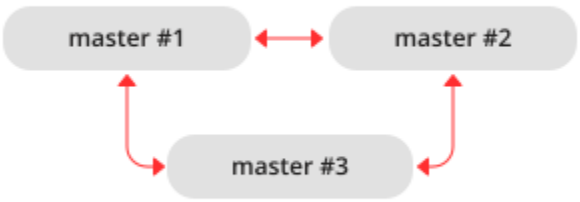
```
tarantool> box.info.replication
---
  replication:
    1:
      id: 1
      uuid: b8a7db60-745f-41b3-bf68-5fcce7a1e019
      lsn: 88
    2:
      id: 2
      uuid: cd3c7da2-a638-4c5d-ae63-e7767c3a6896
      lsn: 31
      upstream:
        status: follow
```

(continues on next page)

(continued from previous page)

```
      idle: 43.187747001648
      peer: replicator@192.168.0.102:3301
      lag: 0
    downstream:
   vclock: {1: 31}
  3:
    id: 3
    uuid: e38ef895-5804-43b9-81ac-9f2cd872b9c4
    lsn: 54
    upstream:
      status: follow
      idle: 43.187621831894
      peer: replicator@192.168.0.103:3301
      lag: 2
    downstream:
      vclock: {1: 54}
...
```

This report is for a master-master replica set of three instances, each having its own instance id, UUID and log sequence number.



The request was issued at master #1, and the reply includes statistics for the other two masters, given in regard to master #1.

The primary indicators of replication health are:

- idle, the time (in seconds) since the instance received the last event from a master.

  A replica sends heartbeat messages to the master every second, and the master is programmed to reconnect automatically if it does not see heartbeat messages within replication_timeout seconds.

  Therefore, in a healthy replication setup, idle should never exceed replication_timeout: if it does, either the replication is lagging seriously behind, because the master is running ahead of the replica, or the network link between the instances is down.

- lag, the time difference between the local time at the instance, recorded when the event was received, and the local time at another master recorded when the event was written to the write ahead log on that master.

  Since the lag calculation uses the operating system clocks from two different machines, do not be surprised if it's negative: a time drift may lead to the remote master clock being consistently behind the local instance's clock.

  For multi-master configurations, lag is the maximal lag.

3.6.6 Recovering from a degraded state

"Degraded state" is a situation when the master becomes unavailable – due to hardware or network failure, or due to a programming bug.

In a master-replica set, if a master disappears, error messages appear on the replicas stating that the connection is lost:

```
$ # messages from a replica 's log
2017-06-14 16:23:10.993 [19153] main/105/applier/replicator@192.168.0. I> can't read row
2017-06-14 16:23:10.993 [19153] main/105/applier/replicator@192.168.0. coio.cc:349 !> SystemError
unexpected EOF when reading from socket, called on fd 17, aka 192.168.0.101:57815,
peer of 192.168.0.101:3301: Broken pipe
2017-06-14 16:23:10.993 [19153] main/105/applier/replicator@192.168.0. I> will retry every 1 second
2017-06-14 16:23:10.993 [19153] relay/[::ffff:192.168.0.101]:/101/main I> the replica has closed its socket, exiting
2017-06-14 16:23:10.993 [19153] relay/[::ffff:192.168.0.101]:/101/main C> exiting the relay loop
```

. . . and the master's status is reported as "disconnected":

```
# report from replica #1
tarantool> box.info.replication
---
- 1:
    id: 1
    uuid: 70e8e9dc-e38d-4046-99e5-d25419267229
    lsn: 542
    upstream:
      peer: replicator@192.168.0.101:3301
      lag: 0.00026607513427734
      status: disconnected
      idle: 182.36929893494
      message: connect, called on fd 13, aka 192.168.0.101:58244
  2:
    id: 2
    uuid: fb252ac7-5c34-4459-84d0-54d248b8c87e
    lsn: 0
  3:
    id: 3
    uuid: fd7681d8-255f-4237-b8bb-c4fb9d99024d
    lsn: 0
    downstream:
      vclock: {1: 542}
...
```

```
# report from replica #2
tarantool> box.info.replication
---
- 1:
    id: 1
    uuid: 70e8e9dc-e38d-4046-99e5-d25419267229
    lsn: 542
    upstream:
      peer: replicator@192.168.0.101:3301
      lag: 0.00027203559875488
      status: disconnected
      idle: 186.76988101006
      message: connect, called on fd 13, aka 192.168.0.101:58253
  2:
    id: 2
    uuid: fb252ac7-5c34-4459-84d0-54d248b8c87e
    lsn: 0
    upstream:
      status: follow
      idle: 186.76960110664
      peer: replicator@192.168.0.102:3301
      lag: 0.00020599365234375
  3:
    id: 3
    uuid: fd7681d8-255f-4237-b8bb-c4fb9d99024d
    lsn: 0
...
```

To declare that one of the replicas must now take over as a new master:

1. Make sure that the old master is gone for good:
   - change network routing rules to avoid any more packets being delivered to the master, or
   - shut down the master instance, if you have access to the machine, or
   - power off the container or the machine.
2. Say box.cfg{read_only=false, listen=URI} on the replica, and box.cfg{replication=URI} on the other replicas in the set.

Note: If there are updates on the old master that were not propagated before the old master went down, re-apply them manually to the new master using tarantoolctl cat and tarantoolctl play commands.

There is no automatic way for a replica to detect that the master is gone forever, since sources of failure and replication environments vary significantly. So the detection of degraded state requires an external observer.

3.6.7 Reseeding a replica

If any of a replica's .xlog/.snap/.run files are corrupted or deleted, you can "re-seed" the replica:

1. Stop the replica and destroy all local database files (the ones with extensions .xlog/.snap/.run/.inprogress).
2. Delete the replica's record from the following locations:
   a. the replication parameter at all running instances in the replica set.

b. the box.space._cluster tuple on the master instance.

See section Removing instances for details.

3. Restart the replica with the same instance file to contact the master again. The replica will then catch up with the master by retrieving all the master's tuples.

Note: Remember that this procedure works only if the master's WAL files are present.

3.6.8 Preventing duplicate actions

Tarantool guarantees that every update is applied only once on every replica. However, due to the asynchronous nature of replication, the order of updates is not guaranteed. We now analyze this problem with more details, provide examples of replication going out of sync, and suggest solutions.

Replication stops

In a replica set of two masters, suppose master #1 tries to do something that master #2 has already done. For example, try to insert a tuple with the same unique key:

```
tarantool> box.space.tester:insert{1, 'data'}
```

This would cause an error saying Duplicate key exists in unique index 'primary' in space 'tester' and the replication would be stopped. (This is the behavior when the replication_skip_conflict configuration parameter has its default recommended value, false.)

```
$ # error messages from master #1
2017-06-26 21:17:03.233 [30444] main/104/applier/rep_user@100.96.166.1 I> can't read row
2017-06-26 21:17:03.233 [30444] main/104/applier/rep_user@100.96.166.1 memtx_hash.cc:226 E> ER_TUPLE_
↪FOUND:
Duplicate key exists in unique index 'primary' in space 'tester'
2017-06-26 21:17:03.233 [30444] relay/[::ffff:100.96.166.178]/101/main I> the replica has closed its socket, exiting
2017-06-26 21:17:03.233 [30444] relay/[::ffff:100.96.166.178]/101/main C> exiting the relay loop

$ # error messages from master #2
2017-06-26 21:17:03.233 [30445] main/104/applier/rep_user@100.96.166.1 I> can't read row
2017-06-26 21:17:03.233 [30445] main/104/applier/rep_user@100.96.166.1 memtx_hash.cc:226 E> ER_TUPLE_
↪FOUND:
Duplicate key exists in unique index 'primary' in space 'tester'
2017-06-26 21:17:03.234 [30445] relay/[::ffff:100.96.166.178]/101/main I> the replica has closed its socket, exiting
2017-06-26 21:17:03.234 [30445] relay/[::ffff:100.96.166.178]/101/main C> exiting the relay loop
```

If we check replication statuses with box.info, we will see that replication at master #1 is stopped (1. upstream.status = stopped). Additionally, no data is replicated from that master (section 1.downstream is missing in the report), because the downstream has encountered the same error:

```
# replication statuses (report from master #3)
tarantool> box.info
---
- version: 1.7.4-52-g980d30092
  id: 3
  ro: false
  vclock: {1: 9, 2: 1000000, 3: 3}
  uptime: 557
```

(continues on next page)

(continued from previous page)

```
  lsn: 3
  vinyl: []
  cluster:
    uuid: 34d13b1a-f851-45bb-8f57-57489d3b3c8b
  pid: 30445
  status: running
  signature: 1000012
  replication:
    1:
      id: 1
      uuid: 7ab6dee7-dc0f-4477-af2b-0e63452573cf
      lsn: 9
      upstream:
        peer: replicator@192.168.0.101:3301
        lag: 0.00050592422485352
        status: stopped
        idle: 445.8626639843
        message: Duplicate key exists in unique index 'primary' in space 'tester'
    2:
      id: 2
      uuid: 9afbe2d9-db84-4d05-9a7b-e0cbbf861e28
      lsn: 1000000
      upstream:
        status: follow
        idle: 201.99915885925
        peer: replicator@192.168.0.102:3301
        lag: 0.0015020370483398
      downstream:
        vclock: {1: 8, 2: 1000000, 3: 3}
    3:
      id: 3
      uuid: e826a667-eed7-48d5-a290-64299b159571
      lsn: 3
  uuid: e826a667-eed7-48d5-a290-64299b159571
...
```

When replication is later manually resumed:

```
# resuming stopped replication (at all masters)
tarantool> original_value = box.cfg.replication
tarantool> box.cfg{replication={}}
tarantool> box.cfg{replication=original_value}
```

. . . the faulty row in the write-ahead-log files is skipped.

Replication runs out of sync

In a master-master cluster of two instances, suppose we make the following operation:

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

When this operation is applied on both instances in the replica set:

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

(continues on next page)

(continued from previous page)

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

. . . we can have the following results, depending on the order of execution:

- each master's row contains the UUID from master #1,
- each master's row contains the UUID from master #2,
- master #1 has the UUID of master #2, and vice versa.

Commutative changes

The cases described in the previous paragraphs represent examples of non-commutative operations, i.e. operations whose result depends on the execution order. On the contrary, for commutative operations, the execution order does not matter.

Consider for example the following command:

```
tarantool> box.space.tester:upsert{{1, 0}, {{'+', 2, 1)}
```

This operation is commutative: we get the same result no matter in which order the update is applied on the other masters.

3.7 Connectors

This chapter documents APIs for various programming languages.

3.7.1 Protocol

Tarantool's binary 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, instance 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, you needn't concern yourself 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 the page about Tarantool's binary protocol.

3.7.2 Packet example

The Tarantool API exists so that a client program can send a request packet to a server instance, 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 on the page about Tarantool's binary 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, you could send that packet to the Tarantool instance, and interpret the response (the page about Tarantool's binary protocol has a description of the packet format for responses as well as requests). But it would be easier, and less error-prone, if you 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 Setting up the server for connector examples

This chapter has examples that show how to connect to a Tarantool instance via the Perl, PHP, Python, node.js, and C connectors. The examples contain hard code that will work if and only if the following conditions are met:

- the Tarantool instance (tarantool) is running on localhost (127.0.0.1) and is listening on port 3301 (box.cfg.listen = '3301'),
- space examples has id = 999 (box.space.examples.id = 999) and has a primary-key index for a numeric field (box.space[999].index[0].parts[1].type = "unsigned"),
- user 'guest' has privileges for reading and writing.

It is easy to meet all the conditions by starting the instance 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.

3.7.6 R

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

3.7.7 Erlang

See Erlang tarantool driver.

3.7.8 Perl

The most commonly used Perl driver is tarantool-perl. It is not supplied as part of the Tarantool repository; it must be installed separately. The most common way to install it is by cloning from GitHub.

To avoid minor warnings that may appear the first time tarantool-perl is installed, start with installing some other modules that tarantool-perl uses, with CPAN, the Comprehensive Perl Archive Network:

```
$ sudo cpan install AnyEvent
$ sudo cpan install Devel::GlobalDestruction
```

Then, to install tarantool-perl itself, say:

```
$ git clone https://github.com/tarantool/tarantool-perl.git tarantool-perl
$ cd tarantool-perl
$ git submodule init
$ git submodule update --recursive
$ perl Makefile.PL
$ make
$ sudo make install
```

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 instance is listening at localhost:3301 and that the space 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 instance at localhost:3301, then send an space_object:INSERT request, then — if all is well — end without displaying any messages. If Tarantool is not running on localhost with listen port = 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. for 'guest' we do not 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 request and does not show all that's necessary for good practice. For that, please see the tarantool-perl repository.

3.7.9 PHP

tarantool-php is the official PHP connector for Tarantool. It is not supplied as part of the Tarantool repository and must be installed separately (see installation instructions in the connector's README file).

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 instance is listening at localhost:3301 and that the space 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 instance 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', [99999, 'BB']);
    echo "Insert succeeded\n";
} catch (Exception $e) {
    echo $e->getMessage(), "\n";
}
```

The example program only shows one request and does not show all that's necessary for good practice. For that, please see tarantool/tarantool-php project at GitHub.

Besides, there is another community-driven GitHub project which includes an alternative connector written in pure PHP, an object mapper, a queue and other packages.

3.7.10 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 the tarantool-python connector 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 instance is listening at localhost:3301 and that the space examples exists, as described earlier. To run the program, say python example.py. The program will connect to the Tarantool server, will send the INSERT request, and will not

throw any 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, please see tarantool-python project at GitHub. For an example of using Python API with queue managers for Tarantool, see queue-python project at GitHub.

3.7.11 Node.js

The most commonly used node.js driver is the Node Tarantool driver. It is not supplied as part of the Tarantool repository; it must be installed separately. The most common way to install it is with npm. For example, on Ubuntu, the installation could look like this after npm has been installed:

```
$ npm install tarantool-driver --global
```

Here is a complete node.js program that inserts [99999,'BB'] into space[999] via the node.js API. Before trying to run, check that the server instance is listening at localhost:3301 and that the space examples exists, as described earlier. To run, paste the code into a file named example.rs and say node example.rs. The program will connect using an application-specific definition of the space. The program will open a socket connection with the Tarantool instance at localhost:3301, then send an INSERT request, then — if all is well — end after saying "Insert succeeded". If Tarantool is not running on localhost with listen port = 3301, the program will print "Connect failed". If the 'guest' user does not have authorization to connect, the program will print "Auth failed". If the insert request fails for any reason, for example because the tuple already exists, the program will print "Insert failed".

```
var TarantoolConnection = require('tarantool-driver');
var conn = new TarantoolConnection({port: 3301});
var insertTuple = [99999, "BB"];
conn.connect().then(function() {
    conn.auth("guest", "").then(function() {
        conn.insert(999, insertTuple).then(function() {
            console.log("Insert succeeded");
            process.exit(0);
    }, function(e) { console.log("Insert failed");  process.exit(1); });
    }, function(e) { console.log("Auth failed");    process.exit(1); });
    }, function(e) { console.log("Connect failed"); process.exit(1); });
```

The example program only shows one request and does not show all that's necessary for good practice. For that, please see The node.js driver repository.

3.7.12 C#

The most commonly used C# driver is progaudi.tarantool, previously named tarantool-csharp. It is not supplied as part of the Tarantool repository; it must be installed separately. The makers recommend cross-platform installation using Nuget.

To be consistent with the other instructions in this chapter, here is a way to install the driver directly on Ubuntu 16.04.

1. Install .net core from Microsoft. Follow .net core installation instructions.

Note:

- Mono will not work, nor will .Net from xbuild. Only .net core supported on Linux and Mac.

- Read the Microsoft End User License Agreement first, because it is not an ordinary open-source agreement and there will be a message during installation saying "This software may collect information about you and your use of the software, and send that to Microsoft." Still you can set environment variables to opt out from telemetry.

---

2. Create a new console project.

```
$ cd ~
$ mkdir progaudi.tarantool.test
$ cd progaudi.tarantool.test
$ dotnet new console
```

3. Add progaudi.tarantool reference.

```
$ dotnet add package progaudi.tarantool
```

4. Change code in Program.cs.

```
$ cat <<EOT > Program.cs
using System;
using System.Threading.Tasks;
using ProGaudi.Tarantool.Client;

public class HelloWorld
{
  static public void Main ()
  {
    Test().GetAwaiter().GetResult();
  }
  static async Task Test()
  {
    var box = await Box.Connect("127.0.0.1:3301");
    var schema = box.GetSchema();
    var space = await schema.GetSpace("examples");
    await space.Insert((99999, "BB"));
  }
}
EOT
```

5. Build and run your application.

   Before trying to run, check that the server is listening at localhost:3301 and that the space examples exists, as described earlier.

```
$ dotnet restore
$ dotnet run
```

   The program will:

   - connect using an application-specific definition of the space,
   - open a socket connection with the Tarantool server at localhost:3301,
   - send an INSERT request, and — if all is well — end without saying anything.

   If Tarantool is not running on localhost with listen port = 3301, or if user 'guest' does not have authorization to connect, or if the INSERT request fails for any reason, the program will print an error message, among other things (stacktrace, etc).

The example program only shows one request and does not show all that's necessary for good practice. For that, please see the progaudi.tarantool driver repository.

3.7.13 C

Here follow two examples of using Tarantool's high-level C API.

Example 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 a server instance is listening at localhost:3301 and that the space examples exists, as described earlier. To run the program, say ./example. The program will connect to the Tarantool instance, and will send the request. If Tarantool is not running on localhost with listen address = 3301, the program will print "Connection refused". If the insert fails, the program will print "Insert failed" and an error number (see all error codes in the source file /src/box/errcode.h).

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 instance 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 a server instance.

```
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 connection 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 tnt_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)
```

Example 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");
```

(continues on next page)

(continued from previous page)

```
        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 GitHub.

3.7.14 Interpreting function return values

For all connectors, calling a function via Tarantool causes a return in the 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 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 *arg; arg = tnt_object(NULL);     /* MAKE REQUEST */
   tnt_object_add_array(arg, 0);
   struct tnt_request *req1 = tnt_request_call(NULL);  /* CALL function f() */
   tnt_request_set_funcz(req1, "f");
   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);
```

(continues on next page)

(continued from previous page)

```
  }
  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(arg);
  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 MsgPack specification). The rest are as described in the table Common Types and MsgPack Encodings.

3.8 SQL

In this section we will go through SQL:2016's "Feature taxonomy and definition for mandatory features".

For each feature in that list, we will come up with a simple example SQL statement. If Tarantool appears to handle the example, we will mark it "Okay", else we will mark it "Fail". Since this is rough and arbitrary, we believe that tests which are unfairly marked "Okay" will probably be balanced by tests which are unfairly marked "Fail".

Feature ID	Feature	Example	Test
E011	Numeric data types		
E011-01	INTEGER and SMALLINT	create table t (s1 int primary key);	Okay.

Continued on next page

Table 1 – continued from previous page

Feature ID	Feature	Example	Test
E011-02	REAL, DOUBLE PRECISION, and FLOAT data types	create table tr (s1 float primary key);	Okay. Note: Floating point SQL types are not planned to be compatible between 2.1 and 2.2 releases. The reason is that in 2.1 we set 'number' format for columns of these types, but will restrict it to 'float32' and 'float64' in 2.2. The format change requires data migration and cannot be done automatically, because in 2.1 we have no information to distinguish 'number' columns (created from Lua) from FLOAT/DOUBLE/REAL ones (created from SQL).
E011-03	DECIMAL and NUMERIC data types	create table td (s1 numeric primary key);	Fail, DECIMAL and NUMERIC data types are not supported and a number containing post-decimal digits will be treated as approximate numeric.
E011-04	Arithmetic operators	select 10+1,9-2,8*3,7/2 from t;	Okay.
E011-05	Numeric comparisons	select * from t where 1 < 2;	Okay.
E011-06	Implicit casting among the numeric data types	select * from t where s1 = 1.00;	Okay, but only because Tarantool doesn't distinguish between numeric data types.
E021	Character string types		
E021-01	Character data type (including all its spellings)	create table t44 (s1 char primary key);	Fail, CHAR is not supported. This type of Fail will only be counted once.
E021-02	CHARACTER VARYING data type (including all its spellings)	create table t45 (s1 varchar primary key);	Fail, only the spelling VARCHAR is allowed. Note: VARCHAR(N) does not check the string length.
E021-03	Character literals	insert into t45 values ('');	Okay, and the bad practice of accepting ""'s for character literals is avoided.

Continued on next page

Table 1 – continued from previous page

Feature ID	Feature	Example	Test
E021-04	CHARACTER_LENGTH function	select character_length(s1) from t;	Fail. There is no such function. There is a function LENGTH(), which is okay.
E021-05	OCTET_LENGTH	select octet_length(s1) from t;	Fail. There is no such function.
E021-06	SUBSTRING function.	select substring(s1 from 1 for 1) from t;	Fail. There is no such function. There is a function SUBSTR(x,n,n) which is okay.
E021-07	Character concatenation	select 'a' || 'b' from t;	Okay.
E021-08	UPPER and LOWER functions	select upper('a'), lower('B') from t;	Okay.
E021-09	TRIM function	select trim('a ') from t;	Okay.
E021-10	Implicit casting among the fixed-length and variable-length character string types	select * from tm where char_column > varchar_column;	Fail, there is no fixed-length character string type.
E021-11	POSITION function	select position(x in y) from z;	Fail. Tarantool's function uses ',' rather than 'in'
E021-12	Character comparison	select * from t where s1 > 'a';	Okay. We should note here that comparisons use a binary collation by default, but it is easy to specify unicode or unicode_ci collations, or create new collations.
E031	Identifiers	create table rank (ceil int primary key);	Fail. Tarantool's list of reserved words differs from the standard's list of reserved words.
E031-01	Delimited Identifiers	create table "t47" (s1 int primary key);	Okay. And enclosing identifiers inside double quotes means they won't be converted to upper case or lower case, this is behavior that some other DBMSs sadly lack.
E031-02	Lower case identifiers	create table t48 (s1 int primary key);	Okay.
E031-03	Trailing underscore	create table t49_ (s1 int primary key);	Okay.
E051	Basic query specification		
E051-01	SELECT DISTINCT	select distinct s1 from t;	Okay.
E051-02	GROUP BY clause	select distinct s1 from t group by s1;	Okay.

Continued on next page

Table 1 – continued from previous page

Feature ID	Feature	Example	Test
E051-04	GROUP BY can contain columns not in select list	select s1 from t group by lower(s1);	Okay.
E051-05	Select list items can be renamed	select s1 as K from t order by K;	Okay.
E051-06	HAVING clause	select count(*) from t having count(*) > 0;	Okay. GROUP BY is not mandatory before HAVING.
E051-07	Qualified * in select list	select t.* from t;	Okay.
E051-08	Correlation names in the FROM clause	select * from t as K;	Okay.
E051-09	Rename columns in the FROM clause	select * from t as x(q,c);	Fail.
E061	Basic predicates and search conditions		
E061-01	Comparison predicate	select * from t where 0 = 0;	Okay.
E061-02	BETWEEN predicate	select * from t where '' between '' and ' ';	Okay.
E061-03	IN predicate with list of values	select * from t where s1 in ('a',upper('a'));	Okay.
E061-04	LIKE predicate	select * from t where s1 like '_';	Okay.
E061-05	LIKE predicate: ESCAPE clause	VALUES ('abc_' LIKE 'abcX_' ESCAPE 'X');	Okay.
E061-06	NULL predicate	select * from t where s1 is not null;	Okay.
E061-07	Quantified comparison predicate	select * from t where s1 = any (select s1 from t);	Fail. Syntax error.
E061-08	EXISTS predicate	select * from t where not exists (select * from t);	Okay.
E061-09	Subqueries in comparison predicate	select * from t where s1 > (select s1 from t);	Okay.
E061-11	Subqueries in IN predicate	select * from t where s1 in (select s1 from t);	Okay.
E061-12	Subqueries in quantified comparison predicate	select * from t where s1 >= all (select s1 from t);	Fail. Syntax error.
E061-13	Correlated subqueries	select * from t where s1 = (select s1 from t2 where t2.s2 = t.s1);	Okay.
E061-14	Search condition	select * from t where 0 <> 0 or 'a' < 'b' and s1 is null;	Okay.
E071	Basic query expressions		
E071-01	UNION DISTINCT table operator	select * from t union distinct select * from t;	Fail. However, "select * from t union select * from t;" is okay.
E071-02	UNION ALL table operator	select * from t union all select * from t;	Okay.

Continued on next page

Table 1 – continued from previous page

Feature ID	Feature	Example	Test
E071-03	EXCEPT DISTINCT table operator	select * from t except distinct select * from t;	Fail. However, select * from t except select * from t; is okay.
E071-05	Columns combined via table operators need not have exactly the same data type.	select s1 from t union select 5 from t;	Okay, but only because Tarantool doesn't distinguish data types very well.
E071-06	Table operators in subqueries	select * from t where 'a' in (select * from t union select * from t);	Okay.
E081	Basic privileges		
E081-01	Select privilege at the table level		Fail. Syntax error. (Tarantool doesn't support privileges.)
E081-02	DELETE privilege		Fail. (Tarantool doesn't support privileges.)
E081-03	INSERT privilege at the table level		Fail. (Tarantool doesn't support privileges.)
E081-04	UPDATE privilege at the table level		Fail. (Tarantool doesn't support privileges.)
E081-05	UPDATE privilege at column level		Fail. (Tarantool doesn't support privileges.)
E081-06	REFERENCES privilege at the table level		Fail. (Tarantool doesn't support privileges.)
E081-07	REFERENCES privilege at column level		Fail. (Tarantool doesn't support privileges.)
E081-08	WITH GRANT OPTION		Fail. (Tarantool doesn't support privileges.)
E081-09	USAGE privilege		Fail. (Tarantool doesn't support privileges.)
E081-10	EXECUTE privilege		Fail. (Tarantool doesn't support privileges.)
E091	Set functions		
E091-01	AVG	select avg(s1) from t7;	Fail. No warning that nulls were eliminated.
E091-02	COUNT	select count(*) from t7 where s1 > 0;	Okay.
E091-03	MAX	select max(s1) from t7 where s1 > 0;	Okay.
E091-04	MIN	select min(s1) from t7 where s1 > 0;	Okay.
E091-05	SUM	select sum(1) from t7 where s1 > 0;	Okay.
E091-06	ALL quantifier	select sum(all s1) from t7 where s1 > 0;	Okay.
E091-07	DISTINCT quantifier	select sum(distinct s1) from t7 where s1 > 0;	Okay.
E101	Basic data manipulation		

Continued on next page

Table 1 – continued from previous page

Feature ID	Feature	Example	Test
E101-01	INSERT statement	insert into t (s1,s2) values (1,''),(2,null), (3,55);	Okay.
E101-03	Searched UPDATE statement	update t set s1 = null where s1 in (select s1 from t2);	Okay.
E101-04	Searched DELETE statement	delete from t where s1 in (select s1 from t);	Okay.
E111	Single row SELECT statement	select count(*) from t;	Okay.
E121	Basic cursor support		
E121-01	DECLARE CURSOR		Fail. Tarantool doesn't support cursors.
E121-02	ORDER BY columns need not be in select list	select s1 from t order by s2;	Okay.
E121-03	Value expressions in select list	select s1 from t7 order by -s1;	Okay.
E121-04	OPEN statement		Fail. Tarantool doesn't support cursors.
E121-06	Positioned UPDATE statement		Fail. Tarantool doesn't support cursors.
E121-07	Positioned DELETE statement		Fail. Tarantool doesn't support cursors.
E121-08	CLOSE statement		Fail. Tarantool doesn't support cursors.
E121-10	FETCH statement implicit next		Fail. Tarantool doesn't support cursors.
E121-17	WITH HOLD cursors		Fail. Tarantool doesn't support cursors.
E131	Null value support (nulls in lieu of values)	select s1 from t7 where s1 is null;	Okay.
E141	Basic integrity constraints		
E141-01	NOT NULL constraints	create table t8 (s1 int primary key, s2 int not null);	Okay.
E141-02	UNIQUE constraints of NOT NULL columns	create table t9 (s1 int primary key , s2 int not null unique);	Okay.
E141-03	PRIMARY KEY constraints	create table t10 (s1 int primary key);	Okay, although Tarantool shouldn't always insist on having a primary key.
E141-04	Basic FOREIGN KEY constraint with the NO ACTION default for both referential delete action and referential update action.	create table t11 (s0 int primary key, s1 int references t10);	Okay.

Continued on next page

Table 1 – continued from previous page

Feature ID	Feature	Example	Test
E141-06	CHECK constraints	create table t12 (s1 int primary key, s2 int, check (s1 = s2));	Okay.
E141-07	Column defaults	create table t13 (s1 int primary key, s2 int default -1);	Okay.
E141-08	NOT NULL inferred on primary key	create table t14 (s1 int primary key);	Okay. We are unable to insert NULL although we don't explicitly say the column is NOT NULL.
E141-10	Names in a foreign key can be specified in any order	create table t15 (s1 int, s2 int, primary key (s1, s2)); create table t16 (s1 int primary key, s2 int, foreign key (s2,s1) references t15 (s1,s2));	Okay.
E151	Transaction support		
E151-01	COMMIT statement	commit;	Fail. We have to say START TRANSACTION first.
E151-02	ROLLBACK statement	rollback;	Okay.
E152	Basic SET TRANSACTION statement		
E152-01	SET TRANSACTION statement ISOLATION SERIALIZABLE clause	set transaction isolation level serializable;	Fail. Syntax error.
E152-02	SET TRANSACTION statement READ ONLY and READ WRITE clauses	set transaction read only;	Fail. Syntax error.
E153	Updatable queries with subqueries		
E161	SQL comments using leading double minus	--comment;	Okay.
E171	SQLSTATE support	drop table no_such_table;	Fail. At least, the error message doesn't hint that SQLSTATE exists.
E182	Host language binding		Okay. Any of the Tarantool connectors should be able to call box.execute().
F031	Basic schema manipulation		
F031-01	CREATE TABLE statement to create persistent base tables	create table t20 (t20_1 int not null);	Fail. We always have to say PRIMARY KEY (we only count this flaw once).
F031-02	CREATE VIEW statement	create view t21 as select * from t20;	Okay.
F031-03	GRANT statement		Fail. Tarantool doesn't support privileges except via NoSQL.

Continued on next page

Table 1 – continued from previous page

Feature ID	Feature	Example	Test
F031-04	ALTER TABLE statement: add column	alter table t7 add column t7_2 varchar default 'q';	Fail. Table alterations work but not this clause.
F031-13	DROP TABLE statement: RESTRICT clause	drop table t20 restrict;	Fail. Syntax error, and RESTRICT is not assumed.
F031-16	DROP VIEW statement: RESTRICT clause	drop view v2 restrict;	Fail. Syntax error, and RESTRICT is not assumed.
F031-19	REVOKE statement: RESTRICT clause		Fail. Tarantool does not support privileges except via NoSQL.
F041	Basic joined table		
F041-01	Inner join but not necessarily the INNER keyword	select a.s1 from t7 a join t7 b;	Okay.
F041-02	INNER keyword	select a.s1 from t7 a inner join t7 b;	Okay.
F041-03	LEFT OUTER JOIN	select t7.*,t22.* from t22 left outer join t7 on (t22_1=s1);	Okay.
F041-04	RIGHT OUTER JOIN	select t7.*,t22.* from t22 right outer join t7 on (t22_1=s1);	Fail. Syntax error.
F041-05	Outer joins can be nested	select t7.*,t22.* from t22 left outer join t7 on (t22_1=s1) left outer join t23;.	Okay.
F041-07	The inner table in a left or right outer join can also be used in an inner join	select t7.* from t22 left outer join t7 on (t22_1=s1) inner join t22 on (t22_4=t22_5);	Okay. The query fails due to a syntax error but that's expectable.
F041-08	All comparison operators are supported	select * from t where 0=1 or 0>1 or 0<1 or 0<>1;	Okay.
F051 Basic date and time			
F051-01	DATE data type (including support of DATE literal)	create table dates (s1 date);	Fail. Tarantool does not support DATE data type.
F051-02	TIME data type (including support of TIME literal)	create table times (s1 time default time '1:2:3');	Fail. Syntax error.
F051-03	TIMESTAMP data type (including support of TIMESTAMP literal)	create table timestamps (s1 timestamp);	Fail. Syntax error.
F051-04	Comparison predicate on DATE, TIME and TIMESTAMP data types	select * from dates where s1 = s1;	Fail. The data types are not supported.

Continued on next page

Table 1 – continued from previous page

Feature ID	Feature	Example	Test
F051-05	Explicit CAST between date-time types and character string types	select cast(s1 as varchar(10)) from dates;	Fail. The data types are not supported.
F051-06	CURRENT_DATE	select current_date from t;	Fail. Syntax error.
F051-07	CURRENT_TIME	select * from t where current_time < '23:23:23';	Fail. Syntax error.
F051-08	LOCALTIME	select localtime from t;	Fail. Syntax error.
F051-09	LOCALTIMESTAMP	select localtimestamp from t;	Fail. Syntax error.
F081	UNION and EXCEPT in views	create view vv as select * from t7 except select * from t15;	Okay.
F131	Grouped operations		
F131-01	WHERE, GROUP BY, and HAVING clauses supported in queries with grouped views	create view vv2 as select * from vv group by s1;	Okay.
F131-02	Multiple tables supported in queries with grouped views	create view vv3 as select * from vv2,t30;	Okay.
F131-03	Set functions supported in queries with grouped views	create view vv4 as select count(*) from vv2;	Okay.
F131-04	Subqueries with GROUP BY and HAVING clauses and grouped views	create view vv5 as select count(*) from vv2 group by s1 having count(*) > 0;	Okay.
F181	Multiple module support		Fail. Tarantool doesn't have modules.
F201	CAST function	select cast(s1 as int) from t;	Okay.
F221	Explicit defaults	update t set s1 = default;	Fail. Syntax error.
F261	CASE expression		
F261-01	Simple CASE	select case when 1 = 0 then 5 else 7 end from t;	Okay.
F261-02	Searched CASE	select case 1 when 0 then 5 else 7 end from t;	Okay.
F261-03	NULLIF	select nullif(s1,7) from t;	Okay.
F261-04	COALESCE	select coalesce(s1,7) from t;	Okay.
F311	Schema definition statement		
F311-01	CREATE SCHEMA		Fail. Tarantool doesn't have schemas or databases.

Continued on next page

Table 1 – continued from previous page

Feature ID	Feature	Example	Test
F311-02	CREATE TABLE for persistent base tables		Fail. Tarantool doesn't have CREATE TABLE inside CREATE SCHEMA.
F311-03	CREATE VIEW		Fail. Tarantool doesn't have CREATE VIEW inside CREATE SCHEMA.
F311-04	CREATE VIEW: WITH CHECK OPTION		Fail. Tarantool doesn't have CREATE VIEW inside CREATE SCHEMA.
F311-05	GRANT statement		Fail. Tarantool doesn't have GRANT inside CREATE SCHEMA.
F471	Scalar subquery values	select s1 from t where s1 = (select count(*) from t);	Okay.
F481	Expanded NULL Predicate	select * from t where row(s1,s1) is not null;	Fail. Syntax error.
F812	Basic flagging		Fail. Tarantool doesn't support any flagging.
S011	Distinct types	create type x as float;	Fail. Tarantool doesn't support distinct types.
T321	Basic SQL-invoked routines		
T321-01	User-defined functions with no overloading	create function f () returns int return 5;	Fail. Tarantool doesn't support user-defined SQL functions.
T321-02	User-defined procedures with no overloading	create procedure p () begin end;	Fail. Tarantool doesn't support user-defined procedures.
T321-03	Function invocation	select f(1) from t;	Okay. Tarantool can invoke Lua user-defined functions.
T321-04	CALL statement.	call p();	Fail. Tarantool doesn't support user-defined procedures.
T321-05	RETURN statement.	create function f() returns int return 5;	Fail. Tarantool doesn't support user-defined functions.
T631	IN predicate with one list element	select * from t where 1 in (1);	Okay.
F021	Basic information schema	select * from information_schema.tables;	Fail. There is no schema with that name (not counted in the final score).

Total number of items marked "Fail": 69

Total number of items marked "Okay": 77

3.9 FAQ

Q Why Tarantool?

A Tarantool is the latest generation of a family of in-memory data servers developed for web applications. It is the result of practical experience and trials within Mail.Ru since development began in 2008.

Q Why Lua?

A Lua is a lightweight, fast, extensible multi-paradigm language. Lua also happens to be very easy to embed. Lua coroutines relate very closely to Tarantool fibers, and Lua architecture works well with Tarantool internals. Lua acts well as a stored program language for Tarantool, although connecting with other languages is also easy.

Q What's the key advantage of Tarantool?

A

Tarantool provides a rich database feature set (HASH, TREE, RTREE, BITSET indexes, secondary indexes, composite indexes, transactions, triggers, asynchronous replication) in a flexible environment of a Lua interpreter.

These two properties make it possible to be a fast, atomic and reliable in-memory data server which handles non-trivial application-specific logic. The advantage over traditional SQL servers is in performance: low-overhead, lock-free architecture means Tarantool can serve an order of magnitude more requests per second, on comparable hardware. The advantage over NoSQL alternatives is in flexibility: Lua allows flexible processing of data stored in a compact, denormalized format.

Q Who is developing Tarantool?

A There is an engineering team employed by Mail.Ru – check out our commit logs on github.com/tarantool. The development is fully open. Most of the connectors' authors, and the maintainers for different distributions, come from the wider community.

Q Are there problems associated with being an in-memory server?

A The principal storage engine (memtx) is designed for RAM plus persistent storage. It is immune to data loss because there is a write-ahead log. Its memory-allocation and compression techniques ensure there is no waste. And if Tarantool runs out of memory, then it will stop accepting updates until more memory is available, but will continue to handle read and delete requests without difficulty. However, for databases which are much larger than the available RAM space, Tarantool has a second storage engine (vinyl) which is only limited by the available disk space.

Q Can I store (large) BLOBs in Tarantool?

A Starting with Tarantool 1.7, there is no "hard" limit for the maximal tuple size. Tarantool, however, is designed for high-velocity workload with a lot of small chunks. For example, when you change an existing tuple, Tarantool creates a new version of the tuple in memory. Thus, an optimal tuple size is within kilobytes.

Q I delete data from vinyl, but disk usage stays the same. What gives?

A Data you write to vinyl is persisted in append-only run files. These files are immutable, and to perform a delete, a deletion marker (tombstone) is written to a newer run file instead. On compaction, new and old run files are merged, and a new run file is produced. Independently, the checkpoint manager keeps track of all run files involved in a checkpoint, and deletes obsolete files once they are no longer needed.

CHAPTER 4

Reference

4.1 SQL reference

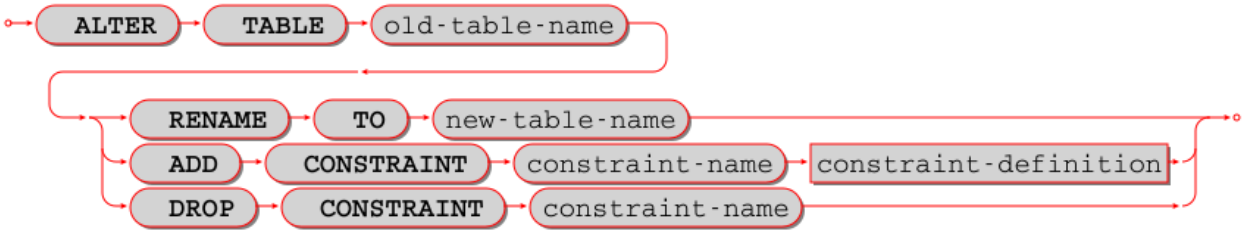
This reference covers all the SQL statements and clauses supported by Tarantool.

4.1.1 SQL statements and clauses

ALTER TABLE

Syntax:

- ALTER TABLE table-name RENAME TO new-table-name;
- ALTER TABLE table-name ADD CONSTRAINT constraint-name constraint-definition;
- ALTER TABLE table-name DROP CONSTRAINT constraint-name;



ALTER is used to change a table's name or to add new constraints or to drop old constraints.

Examples:

```
-- renaming a table:
ALTER TABLE t1 RENAME TO t2;
```

For ALTER ... RENAME, the old-table must exist, the new-table must not exist.

```
-- adding a foreign-key constraint definition:
ALTER TABLE t1 ADD CONSTRAINT c FOREIGN KEY (s1) REFERENCES t1;
```

For ALTER ... ADD CONSTRAINT, the table must exist, table must be empty, the constraint name must not already exist for the table.

It is not possible to say CREATE TABLE table_a ... REFERENCES table_b ... if table b does not exist yet. This is a situation where ALTER TABLE is handy – users can CREATE TABLE table_a without the foreign key, then CREATE TABLE table_b, then ALTER TABLE table_a ... REFERENCES table_b ....

```
-- adding a primary-key constraint definition:
-- This is unusual because primary keys are created automatically
-- and it is illegal to have two primary keys for the same table.
-- However, it is possible to drop a primary-key index, and this
-- is a way to restore the primary key if that happens.
ALTER TABLE t1 ADD CONSTRAINT primary_key PRIMARY KEY (s1);

-- adding a unique-constraint definition:
-- Alternatively, you can say CREATE UNIQUE INDEX unique_key ON t1 (s1);
ALTER TABLE t1 ADD CONSTRAINT unique_key UNIQUE (s1);

-- Adding a check-constraint definition:
ALTER TABLE t1 ADD CONSTRAINT check_ CHECK (s1 > 0);
```

For ALTER ... DROP CONSTRAINT, it is only legal to drop a named constraint, and Tarantool only looks for names of foreign-key constraints. (Tarantool generates the constraint names automatically if the user does not provide them.)

To remove a unique constraint, use DROP INDEX, which will drop the constraint as well.

```
-- dropping a constraint:
ALTER TABLE t1 DROP CONSTRAINT c;
```

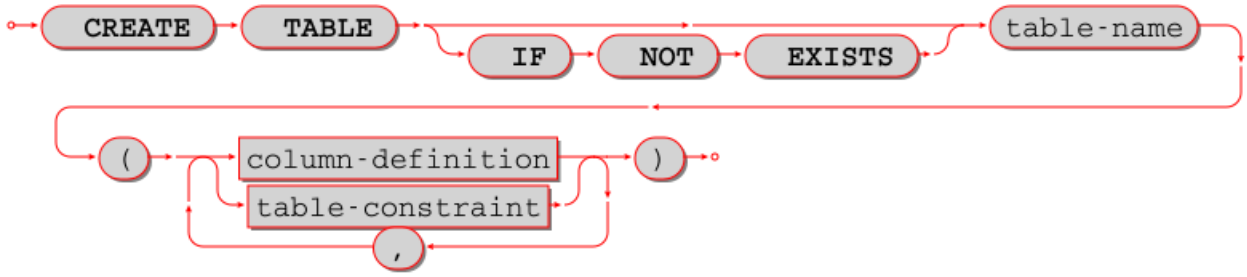
Limitations:

- It is not possible to add or drop a column.
- It is not possible to modify NOT NULL constraints or column properties DEFAULT and data type. However, it is possible to modify them with Tarantool/NOSQL, for example by calling space_object:format() with a different is_nullable value.

CREATE TABLE

Syntax:

CREATE TABLE [IF NOT EXISTS] table-name ((column-definition or table-constraint list);



Create a new base table, usually called a “table”.

Note: A table is a base table if it is created with CREATE TABLE and contains data in persistent storage.

A table is a viewed table, or just "view", if it is created with CREATE VIEW and gets its data from other views or from base tables.

The table-name must be an identifier which is valid according to the rules for identifiers, and must not be the name of an already existing base table or view.

The column-definition or table-constraint list is a comma-separated list of column definitions or table constraints.

A table-element-list must be a comma-separated list of table elements; each table element may be either a column definition or a table constraint definition.

Rules:

- A primary key is necessary; it can be specified with a table constraint PRIMARY KEY.
- There must be at least one column.
- When IF NOT EXISTS is specified, and there is already a table with the same name, the statement is ignored.

Actions:

1. Tarantool evaluates each column definition and table-constraint, and returns an error if any of the rules is violated.
2. Tarantool makes a new definition in the schema.
3. Tarantool makes new indexes for PRIMARY KEY or UNIQUE constraints. A unique index name is created automatically.
4. Tarantool effectively executes a COMMIT statement.

Examples:

```
-- the simplest form, with one column and one constraint:
CREATE TABLE t1 (s1 INTEGER, PRIMARY KEY (s1));

-- you can see the effect of the statement by querying
-- Tarantool system spaces:
SELECT * FROM "_space" WHERE "name" = 'T1';
SELECT * FROM "_index" JOIN "_space" ON "_index"."id" = "_space"."id"
       WHERE "_space"."name" = 'T1';

-- variation of the simplest form, with delimited identifiers
-- and an inline comment:
CREATE TABLE "T1" ("S1" INT /* synonym of INTEGER */, PRIMARY KEY ("S1"));

-- two columns, one named constraint
CREATE TABLE t1 (s1 INTEGER, s2 STRING, CONSTRAINT c1 PRIMARY KEY (s1, s2));
```

Limitations:

- The maximum number of columns is 2000.
- The maximum length of a row depends on the memtx_max_tuple_size or vinyl_max_tuple_size configuration option.

Column definition

Syntax:

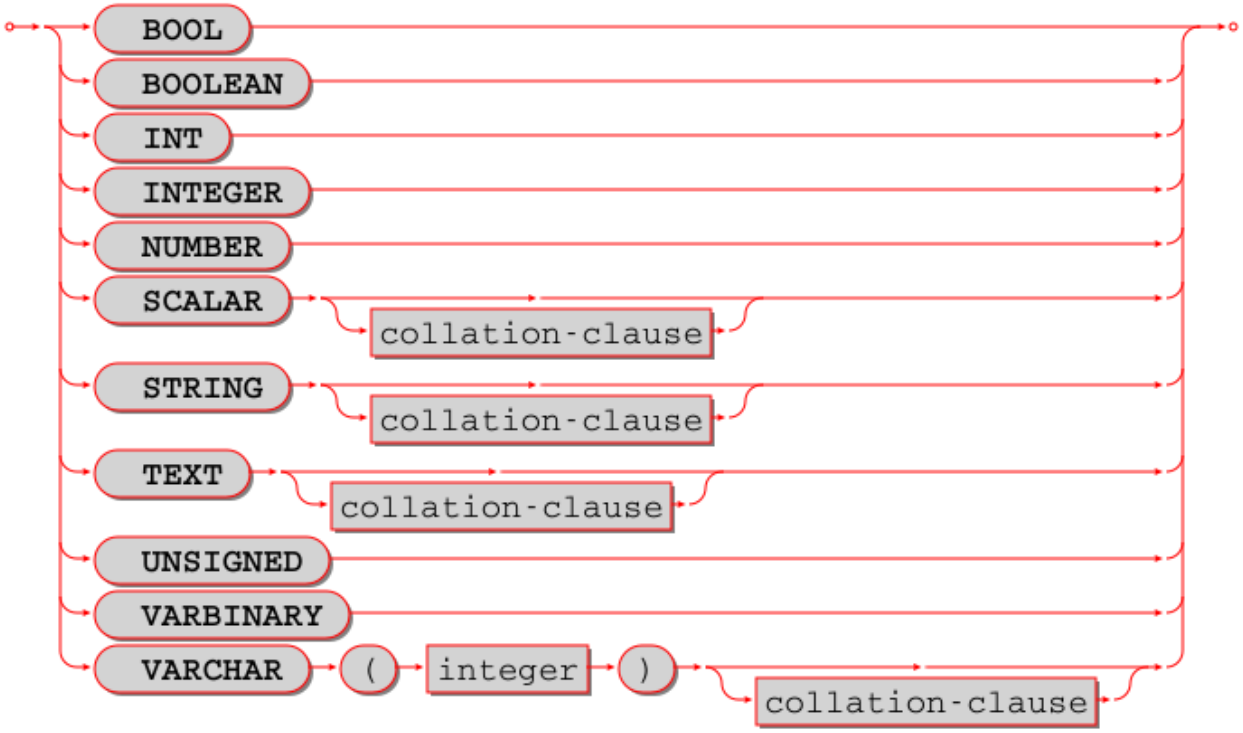
column-name data-type [, column-constraint]

Define a column, which is a table-element used in a CREATE TABLE statement.

The column-name must be an identifier which is valid according to the rules for identifiers.

Each column-name must be unique within a table.

Column definition – data type



Every operand has a data type.

For literals, the data type is usually determined by the format.

For identifiers, the data type is usually determined by the definition.

The usual determination may change because of context or because of explicit casting.

For some SQL data type names there are aliases. An alias may be used for data definition. For example VARCHAR(5) and TEXT are aliases of STRING and may appear in CREATE TABLE table_name (column_name VARCHAR(5) PRIMARY KEY); but Tarantool, if asked, will report that the data type of column_name is STRING.

For every SQL data type there is a corresponding NoSQL type, for example an SQL STRING is stored in a NoSQL space as type = 'string'.

To avoid confusion in this manual, all references to SQL data type names are in upper case and all similar words which refer to NoSQL types or to other kinds of object are in lower case, for example:

- STRING is a data type name, but string is a general term;

- NUMBER is a data type name, but number is a general term.

Although it is common to say that a VARBINARY value is a "binary string", this manual will not use that term and will instead say "byte sequence".

Here are all the SQL data types, their corresponding NoSQL types, their aliases, and minimum / maximum literal examples.

Data types

SQL type	NoSQL type	Aliases	Minimum	Maximum
BOOLEAN	boolean	BOOL	FALSE	TRUE
INTEGER	integer	INT	-9223372036854775808	18446744073709551615
UNSIGNED	unsigned	(none)	0	18446744073709551615
NUMBER	number	(none)	-1.79769e308	1.79769e308
STRING	string	TEXT, VARCHAR(n)	''	'many-characters'
VARBINARY	varbinary	(none)	X''	'X'many-hex-digits'
SCALAR	scalar	(none)	FALSE	X'many-hex-digits'

BOOLEAN values are FALSE, TRUE, and UNKNOWN (which is the same as NULL). FALSE is less than TRUE.

INTEGER values are numbers that do not contain decimal points and are not expressed with exponential notation. The range of possible values is between -2^63 and +2^64, or NULL.

UNSIGNED values are numbers that do not contain decimal points and are not expressed with exponential notation. The range of possible values is between 0 and +2^64, or NULL.

NUMBER values are numbers that do contain decimal points (for example 0.5) or are expressed with exponential notation (for example 5E-1). The range of possible values is the same as for the IEEE 754 floating-point standard, or NULL. Numbers outside the range of NUMBER literals may be displayed as -inf or inf.

STRING values are any sequence of zero or more characters encoded with UTF-8, or NULL. The possible character values are the same as for the Unicode standard. Byte sequences which are not valid UTF-8 characters are allowed but not recommended. STRING literal values are enclosed within single quotes, for example 'literal'. If the VARCHAR alias is used for column definition, it must include a maximum length, for example column_1 VARCHAR(40). However, the maximum length is ignored. The data-type may be followed by [COLLATE collation-name]. .. // see section COLLATE clause.

VARBINARY values are any sequence of zero or more octets (bytes), or NULL. VARBINARY literal values are expressed as X followed by pairs of hexadecimal digits enclosed within single quotes, for example X'0044'. VARBINARYs NoSQL equivalent is 'varbinary' but not character string – the MessagePack storage is MP_BIN (MsgPack binary).

SCALAR can be used for column definitions but the individual column values have one of the preceding types – BOOLEAN, INTEGER, UNSIGNED, NUMBER, STRING, or VARBINARY. See more about SCALAR in the next section. The data-type may be followed by [COLLATE collation-name]. .. // see section COLLATE clause.

Any value of any data type may be NULL. Ordinarily NULL will be cast to the data type of any operand it is being compared to or to the data type of the column it is in. If the data type of NULL cannot be determined from context, it is BOOLEAN.

Column definition – the rules for the SCALAR data type

SCALAR is a "complex" data type, unlike all the other data types which are "primitive". Two column values in a SCALAR column can have two different primitive data types.

1. Any item defined as SCALAR has an underlying primitive type. For example, here:

```
CREATE TABLE t (s1 SCALAR PRIMARY KEY);
INSERT INTO t VALUES (55),('41');
```

   the underlying primitive type of the item in the first row is INTEGER because literal 55 has data type INTEGER, and the underlying primitive type in the second row is STRING (the data type of a literal is always clear from its format).

   An item's primitive type is far more important than its defined type. Incidentally Tarantool might find the primitive type by looking at the way MsgPack stores it, but that is an implementation detail.

2. A SCALAR definition may not include a maximum length, as there is no suggested restriction.
3. A SCALAR definition may include a COLLATE clause, which affects any items whose primitive data type is STRING. The default collation is "binary".
4. Some assignments are illegal when data types differ, but legal when the target is a SCALAR item. For example UPDATE ... SET column1 = `'a'` is illegal if column1 is defined as INTEGER, but is legal if column1 is defined as SCALAR – values which happen to be INTEGER will be changed so their data type is STRING.
5. There is no literal syntax which implies data type SCALAR.
6. TYPEOF(x) is never SCALAR, it is always the underlying data type. This is true even if x is null (in that case the data type is BOOLEAN). In fact there is no function that is guaranteed to return the defined data type. For example, TYPEOF(CAST(1 AS SCALAR)); returns INTEGER, not SCALAR.
7. For any operation that requires implicit casting from an item defined as SCALAR, the syntax is legal but the operation may fail at runtime. At runtime, Tarantool detects the underlying primitive data type and applies the rules for that. For example, if a definition is:

```
CREATE TABLE t (s1 SCALAR PRIMARY KEY, s2 INTEGER);
```

   and within any row s1 = `'a'`, that is, its underlying primitive type is STRING to indicate character strings, then UPDATE t SET s2 = s1; is illegal. Tarantool usually does not know that in advance.

8. For any dyadic operation that requires implicit casting for comparison, the syntax is legal and the operation will not fail at runtime. Take this situation: comparison with a primitive type VARBINARY and a primitive type STRING.

```
CREATE TABLE t (s1 SCALAR PRIMARY KEY);
INSERT INTO t VALUES (X'41');
SELECT * FROM t WHERE s1 > 'a';
```

   The comparison is valid, because Tarantool knows the ordering of X'41' and 'a' in Tarantool/NoSQL 'scalar'. This would be true even if s1 was not defined as SCALAR.

9. The result data type of min/max operation on a column defined as SCALAR is the data type of the minimum/maximum operand, unless the result value is NULL. For example:

```
CREATE TABLE t (s1 INT, s2 SCALAR PRIMARY KEY);
INSERT INTO t VALUES (1,X'44'),(2,11),(3,1E4),(4,'a');
SELECT MIN(s2), HEX(MAX(s2)) FROM t;
```

The result is: - - [11, '44',]

That is only possible with Tarantool/NoSQL scalar rules, but SELECT SUM(s2) would not be legal because addition would in this case require implicit casting from VARBINARY to integer, which is not sensible.

10. The result data type of a primitive combination is never SCALAR because we in effect use TYPEOF(item) not the defined data type. (Here we use the word "combination" in the way that the standard document uses it for section "Result of data type combinations".) Therefore for MAX(1E308, 'a', 0, X'00') the result is X'00'.

Column definition – relation to NoSQL

All the SQL data types correspond to Tarantool/NoSQL types with the same name. For example an SQL STRING is stored in a NoSQL space as type = 'string'.

Therefore specifying an SQL data type X determines that the storage will be in a space with a format column saying that the NoSQL type is 'x'.

The rules for that NoSQL type are applicable to the SQL data type.

If two items have SQL data types that have the same underlying type, then they are compatible for all assignment or comparison purposes.

If two items have SQL data types that have different underlying types, then the rules for explicit casts, or implicit (assignment) casts, or implicit (comparison) casts, apply.

There is one floating-point value which is not handled by SQL: -nan is seen as NULL.

There are also some Tarantool/NoSQL data types which have no corresponding SQL data types. For example, SELECT "flags" FROM "_space"; will return a column whose data type is 'map'. Such columns can only be manipulated in SQL by invoking Lua functions.

Column definition – column-constraint or default clause

The column-constraint or default clause may be as follows:

Data types

Type	Comment
NOT NULL	means "it is illegal to assign a NULL to this column"
PRIMARY KEY	explained in the later section "Constraint definition"
UNIQUE	explained in the later section "Constraint definition"
CHECK (expression)	explained in the later section "Constraint definition"
DEFAULT expression	means "if INSERT does not assign to this column then assign expression result to this column" – if there is no DEFAULT clause then DEFAULT NULL is assumed.

If column-constraint is PRIMARY KEY, this is a shorthand for a separate table-constraint definition: "PRIMARY KEY (column-name)".

If column-constraint is UNIQUE, this is a shorthand for a separate table-constraint definition: "UNIQUE (column-name)".

Columns defined with PRIMARY KEY are automatically NOT NULL.

To enforce some restrictions that Tarantool does not enforce automatically, add CHECK clauses, like these:

```
CREATE TABLE t ("smallint" INTEGER PRIMARY KEY, CHECK ("smallint" <= 32767 AND "smallint" >=
↪-32768));
CREATE TABLE t ("shorttext" CHAR(10) PRIMARY KEY, CHECK (length("shorttext") <= 10));
```

but this may cause inserts or updates to be slow.

Column definition – examples

These are shown within CREATE TABLE statements. Data types may also appear in CAST functions.

```
-- the simple form with column-name and data-type
CREATE TABLE t (column1 INTEGER ...);
-- with column-name and data-type and column-constraint
CREATE TABLE t (column1 STRING PRIMARY KEY ...);
-- with column-name and data-type and collate-clause and two column-constraints
CREATE TABLE t (column1 SCALAR COLLATE "unicode" ...);
```

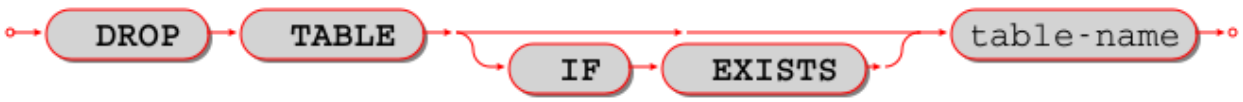
```
-- with all possible data types and aliases
CREATE TABLE t
(column1 BOOLEAN, column2 BOOL,
 column3 INT PRIMARY KEY, column4 INTEGER,
 column4 NUMBER,
 column7 STRING, column8 STRING COLLATE "unicode",
 column9 TEXT, columna TEXT COLLATE "unicode_sv_s1",
 columnb VARCHAR(0), columnc VARCHAR(100000) COLLATE "binary",
 columnd VARBINARY,
 columne SCALAR, columnf SCALAR COLLATE "unicode_uk_s2");
```

```
-- with all possible column constraints and a default clause
CREATE TABLE t
(column1 INT PRIMARY KEY,
 column2 INT UNIQUE,
 column3 INT CHECK (column3 > column2),
 column4 INT REFERENCES t,
 column6 INT DEFAULT NULL);
```

DROP TABLE

Syntax:

DROP TABLE [IF EXISTS] table-name;



Drop a table.

The table-name must identify a table that was created earlier with the CREATE TABLE statement.

Rules:

- If there is a view that references the table, the drop will fail. Please drop the referencing view with DROP VIEW first.
- If there is a foreign key that references the table, the drop will fail. Please drop the referencing constraint with ALTER TABLE ... DROP first.

Actions:

1. Tarantool returns an error if the table does not exist.
2. The table and all its data are dropped.
3. All indexes for the table are dropped.
4. All triggers for the table are dropped.
5. Tarantool effectively executes a COMMIT statement.

Examples:

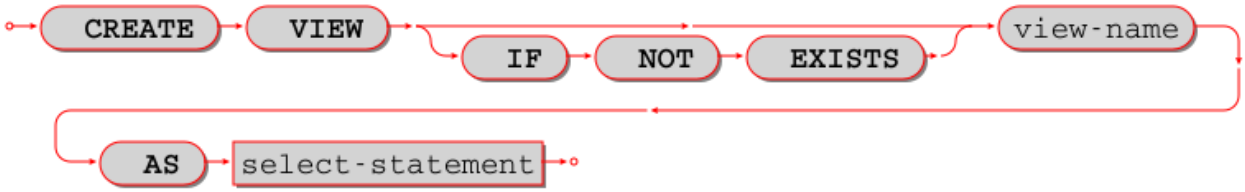
```
-- the simple case:
DROP TABLE t31;
-- with an IF EXISTS clause:
DROP TABLE IF EXISTS t31;
```

See also: DROP VIEW.

CREATE VIEW

Syntax:

CREATE VIEW [IF NOT EXISTS] view-name [(column-list)] AS subquery;



Create a new viewed table, usually called a "view".

The view-name must be valid according to the rules for identifiers.

The optional column-list must be a comma-separated list of names of columns in the view.

The syntax of the subquery must be the same as the syntax of a SELECT statement, or of a VALUES clause.

Rules:

- There must not already be a base table or view with the same name as view-name.
- If column-list is specified, the number of columns in column-list must be the same as the number of columns in the select-list of the subquery.

Actions:

1. Tarantool will throw an error if a rule is violated.
2. Tarantool will create a new persistent object with column-names equal to the names in the column-list or the names in the subquery's select-list.
3. Tarantool effectively executes a COMMIT statement.

Examples:

```
-- the simple case:
CREATE VIEW v AS SELECT column1, column2 FROM t;
-- with a column-list:
CREATE VIEW v (a,b) AS SELECT column1, column2 FROM t;
```

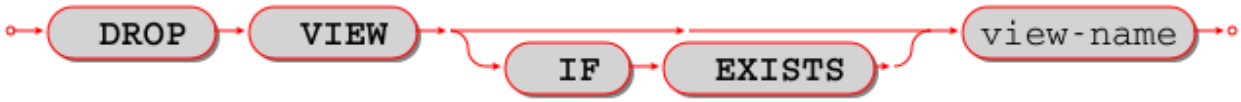
Limitations:

- It is not possible to insert or update or delete from a view, although sometimes a possible substitution is to create an INSTEAD OF trigger.

DROP VIEW

Syntax:

DROP VIEW [IF EXISTS] view-name;



Drop a view.

The view-name must identify a view that was created earlier with the CREATE VIEW statement.

Rules: none

Actions:

1. Tarantool returns an error if the view does not exist.
2. The view is dropped.
3. All triggers for the view are dropped.
4. Tarantool effectively executes a COMMIT statement.

Examples:

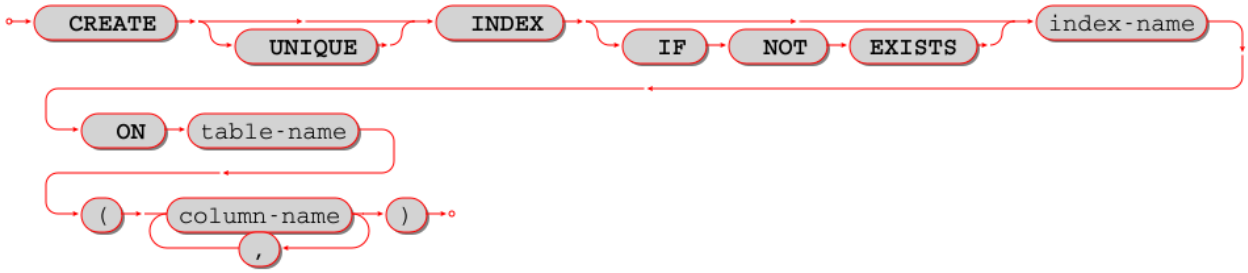
```
-- the simple case:
DROP VIEW v31;
-- with an IF EXISTS clause:
DROP VIEW IF EXISTS v31;
```

See also: DROP TABLE.

CREATE INDEX

Syntax:

CREATE [UNIQUE] INDEX [IF NOT EXISTS] index-name ON table-name (column-list);



Create an index.

The index-name must be valid according to the rules for identifiers.

The table-name must refer to an existing table.

The column-list must be a comma-separated list of names of columns in the table.

Rules:

- There must not already be, for the same table, an index with the same name as index-name.
- An index name is local to the table the index is defined on.
- The maximum number of indexes per table is 128.

Actions:

1. Tarantool will throw an error if a rule is violated.
2. If the new index is UNIQUE, Tarantool will throw an error if any row exists with columns that have duplicate values.
3. Tarantool will create a new index.
4. Tarantool effectively executes a COMMIT statement.

Automatic indexes:

Indexes may be created automatically for columns mentioned in the PRIMARY KEY or UNIQUE clauses of a CREATE TABLE statement. If an index was created automatically, then the index-name is based on four items:

1. pk if this is for a PRIMARY KEY clause, unique if this is for a UNIQUE clause;
2. _unnamed_;
3. the name of the table;
4. _ and an ordinal number; the first index is 1, the second index is 2, and so on.

For example, after CREATE TABLE t (s1 INT PRIMARY KEY, s2 INT, UNIQUE (s2)); there are two indexes named pk_unnamed_T_1 and unique_unnamed_T_2. You can confirm this by saying SELECT * FROM "_index"; which will list all indexes on all tables. There is no need to say CREATE INDEX for columns that already have automatic indexes.

Examples:

```
-- the simple case
CREATE INDEX i ON t (column1);
-- with IF NOT EXISTS clause
CREATE INDEX IF NOT EXISTS i ON t (column1);
-- with UNIQUE specifier and more than one column
CREATE UNIQUE INDEX i ON t (column1, column2);
```

Dropping an automatic index created for a unique constraint will drop the unique constraint as well.

DROP INDEX

Syntax:

DROP INDEX [IF EXISTS] index-name ON table-name;

The index-name must be the name of an existing index, which was created with CREATE INDEX. Or, the index-name must be the name of an index that was created automatically due to a PRIMARY KEY or UNIQUE clause in the CREATE TABLE statement. To see what a table's indexes are, use PRAGMA index_list (table-name).

Rules: none

Actions:

1. Tarantool throws an error if the index does not exist, or is an automatically created index.
2. Tarantool will drop the index.
3. Tarantool effectively executes a COMMIT statement.

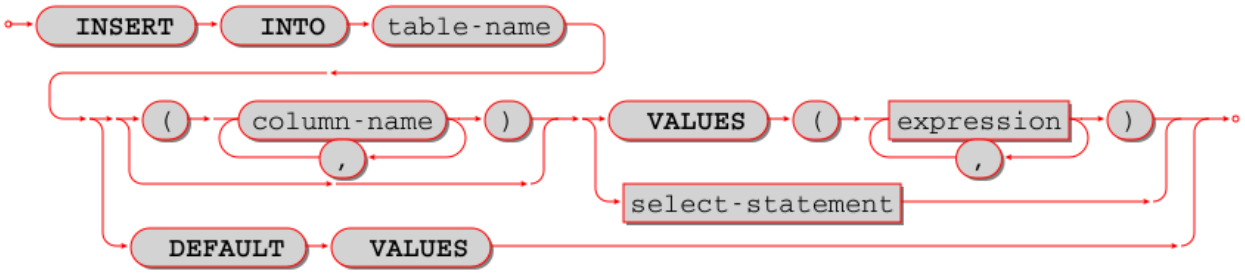
Example:

```
-- the simplest form:
DROP INDEX i ON t;
```

INSERT

Syntax:

- INSERT INTO table-name [(column-list)] VALUES (expression-list) [, (expression-list)];
- INSERT INTO table-name [(column-list)] select-statement;
- INSERT INTO table-name DEFAULT VALUES;



Insert one or more new rows into a table.

The table-name must be a name of a table defined earlier with CREATE TABLE.

The optional column-list must be a comma-separated list of names of columns in the table.

The expression-list must be a comma-separated list of expressions; each expression may contain literals and operators and subqueries and function invocations.

Rules:

- The values in the expression-list are evaluated from left to right.
- The order of the values in the expression-list must correspond to the order of the columns in the table, or (if a column-list is specified) to the order of the columns in the column-list.
- The data type of the value should correspond to the data type of the column, that is, the data type that was specified with CREATE TABLE.
- If a column-list is not specified, then the number of expressions must be the same as the number of columns in the table.
- If a column-list is specified, then some columns may be omitted; omitted columns will get default values.
- The parenthesized expression-list may be repeated – (expression-list),(expression-list),... – for multiple rows.

Actions:

1. Tarantool evaluates each expression in expression-list, and returns an error if any of the rules is violated.
2. Tarantool creates zero or more new rows containing values based on the values in the VALUES list or based on the results of the select-expression or based on the default values.
3. Tarantool executes constraint checks and trigger actions and the actual insertion.
4. Tarantool inserts values into the table.

Examples:

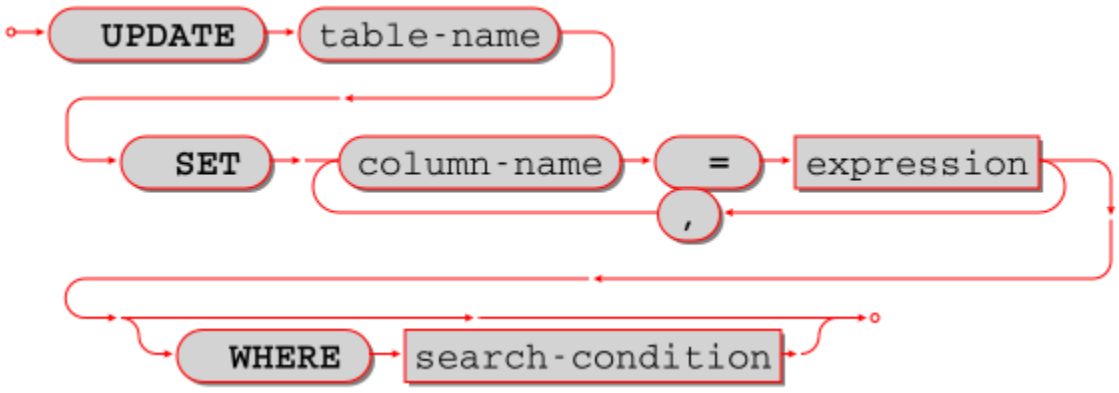
```
-- the simplest form:
INSERT INTO table1 VALUES (1, 'A');
-- with a column list:
INSERT INTO table1 (column1, column2) VALUES (2, 'B');
-- with an arithmetic operator in the first expression:
INSERT INTO table1 VALUES (2 + 1, 'C');
-- put two rows in the table:
INSERT INTO table1 VALUES (4, 'D'), (5, 'E');
```

See also: REPLACE statement.

UPDATE

Syntax:

UPDATE table-name SET column-name = expression [, column-name = expression ...] [WHERE search-condition];



Update zero or more existing rows in a table.

The table-name must be a name of a table defined earlier with CREATE TABLE or CREATE VIEW.

The column-name must be an updatable column in the table.

The expression may contain literals and operators and subqueries and function invocations and column names.

Rules:

- The values in the SET clause are evaluated from left to right.
- The data type of the value should correspond to the data type of the column, that is, the data type that was specified with CREATE TABLE.
- If a search-condition is not specified, then all rows in the table will be updated; otherwise only those rows which match the search-condition will be updated.

Actions:

1. Tarantool evaluates each expression in the SET clause, and returns an error if any of the rules is violated. For each row that is found by the WHERE clause, a temporary new row is formed based on the original contents and the modifications caused by the SET clause.
2. Tarantool executes constraint checks and trigger actions and the actual update.

Examples:

```
-- the simplest form:
UPDATE t SET column1 = 1;
-- with more than one assignment in the SET clause:
UPDATE t SET column1 = 1, column2 = 2;
-- with a WHERE clause:
UPDATE t SET column1 = 5 WHERE column2 = 6;
```

Special cases:

It is legal to say SET (list of columns) = (list of values). For example:

```
UPDATE t SET (column1, column2, column3) = (1,2,3);
```

It is not legal to assign to a column more than once. For example:

```
INSERT INTO t (column1) VALUES (0);
UPDATE t SET column1 = column1 + 1, column1 = column1 + 1;
```

The result is an error: “duplicate column name”.

It is not legal to assign to a primary-key column.

DELETE

Syntax:

DELETE FROM table-name [WHERE search-condition];

Delete zero or more existing rows in a table.

The table-name must be a name of a table defined earlier with CREATE TABLE or CREATE VIEW.

The search-condition may contain literals and operators and subqueries and function invocations and column names.

Rules:

- If a search-condition is not specified, then all rows in the table will be deleted; otherwise only those rows which match the search-condition will be deleted.

Actions:

1. Tarantool evaluates each expression in the search-condition, and returns an error if any of the rules is violated.
2. Tarantool finds the set of rows that are to be deleted.
3. Tarantool executes constraint checks and trigger actions and the actual deletion.
4. Tarantool deletes the set of matching rows from the table.

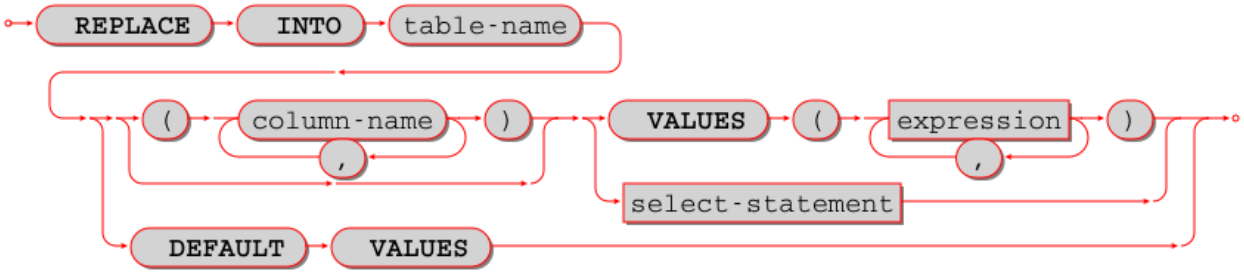
Examples:

```
-- the simplest form:
DELETE FROM t;
-- with a WHERE clause:
DELETE FROM t WHERE column2 = 6;
```

REPLACE

Syntax:

- REPLACE INTO table-name [(column-list)] VALUES (expression-list) [, (expression-list)];
- REPLACE INTO table-name [(column-list)] select-statement;
- REPLACE INTO table-name DEFAULT VALUES;



Insert one or more new rows into a table, or update existing rows.

If a row already exists (as determined by the primary key or any unique key), then the action is delete + insert, and the rules are the same as for a DELETE statement followed by an INSERT statement. Otherwise the action is insert, and the rules are the same as for the INSERT statement.

Examples:

```
-- the simplest form:
REPLACE INTO table1 VALUES (1, 'A');
-- with a column list:
REPLACE INTO table1 (column1, column2) VALUES (2, 'B');
-- with an arithmetic operator in the first expression:
```

(continues on next page)

(continued from previous page)

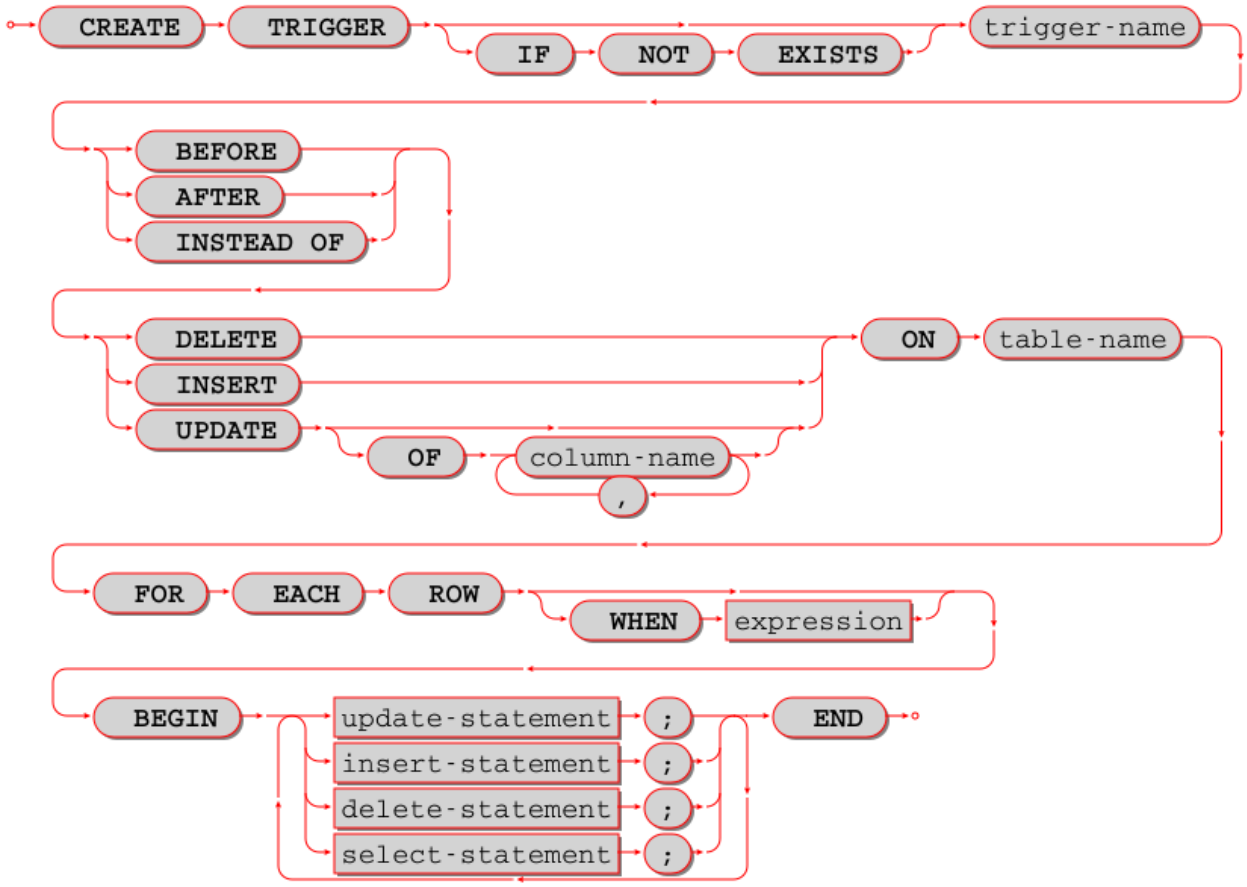
```
REPLACE INTO table1 VALUES (2 + 1, 'C');
-- put two rows in the table:
REPLACE INTO table1 VALUES (4, 'D'), (5, 'E');
```

See also: INSERT Statement, UPDATE Statement.

CREATE TRIGGER

Syntax:

CREATE TRIGGER [IF NOT EXISTS] trigger-name BEFORE|AFTER|INSTEAD OF INSERT|UPDATE|DELETE ON table-name FOR EACH ROW [WHEN (search-condition)] BEGIN update-statement | insert-statement | delete-statement | select-statement; [update-statement | insert-statement | delete-statement | select-statement; ...] END;



The trigger-name must be valid according to the rules for identifiers.

If the trigger action time is BEFORE or AFTER, then the table-name must refer to an existing base table.

If the trigger action time is INSTEAD OF, then the table-name must refer to an existing view.

Rules:

- There must not already be a trigger with the same name as trigger-name.
- Triggers on different tables or views share the same namespace.

- The statements between BEGIN and END should not refer to the table-name mentioned in the ON clause.
- The statements between BEGIN and END should not contain an INDEXED BY clause.

SQL triggers are not fired upon Tarantool/NoSQL requests. This will change in version 2.2.

On a replica, effects of trigger execution are applied, and the SQL triggers themselves are not fired upon replication events.

NoSQL triggers are fired both on replica and master, thus if you have a NoSQL trigger on replica, it is fired when applying effects of an SQL trigger.

Actions:

1. Tarantool will throw an error if a rule is violated.
2. Tarantool will create a new trigger.
3. Tarantool effectively executes a COMMIT statement.

Examples:

```
-- the simple case:
CREATE TRIGGER delete_if_insert BEFORE INSERT ON stores FOR EACH ROW
  BEGIN DELETE FROM warehouses; END;
-- with IF NOT EXISTS clause:
CREATE TRIGGER IF NOT EXISTS delete_if_insert BEFORE INSERT ON stores FOR EACH ROW
  BEGIN DELETE FROM warehouses; END;
-- with FOR EACH ROW and WHEN clauses:
CREATE TRIGGER delete_if_insert BEFORE INSERT ON stores FOR EACH ROW WHEN a=5
  BEGIN DELETE FROM warehouses; END;
-- with multiple statements between BEGIN and END:
CREATE TRIGGER delete_if_insert BEFORE INSERT ON stores FOR EACH ROW
  BEGIN DELETE FROM warehouses; INSERT INTO inventories VALUES (1); END;
```

Trigger extra clauses

- UPDATE OF column-list

  After BEFORE|AFTER UPDATE it is optional to add OF column-list. If any of the columns in column-list is affected at the time the row is processed, then the trigger will be activated for that row. For example:

  ```
  CREATE TRIGGER trigger_on_table1
   BEFORE UPDATE  OF column1, column2 ON table1
   FOR EACH ROW
   BEGIN UPDATE table2 SET column1 = column1 + 1; END;
  UPDATE table1 SET column3 = column3 + 1; -- Trigger will not be activated
  UPDATE table1 SET column2 = column2 + 0; -- Trigger will be activated
  ```

- WHEN

  After table-name FOR EACH ROW it is optional to add [WHEN expression]. If the expression is true at the time the row is processed, only then the trigger will be activated for that row. For example:

  ```
  CREATE TRIGGER trigger_on_table1 BEFORE UPDATE ON table1 FOR EACH ROW
   WHEN (SELECT COUNT(*) FROM table1) > 1
   BEGIN UPDATE table2 SET column1 = column1 + 1; END;
  ```

This trigger will not be activated unless there is more than one row in table1.

- OLD and NEW

  The keywords OLD and NEW have special meaning in the context of trigger action:

  – OLD.column-name refers to the value of column-name before the change.

  – NEW.column-name refers to the value of column-name after the change.

  For example:

```
CREATE TABLE table1 (column1 VARCHAR(15), column2 INT PRIMARY KEY);
CREATE TABLE table2 (column1 VARCHAR(15), column2 VARCHAR(15), column3 INT PRIMARY␣
↪KEY);
INSERT INTO table1 VALUES ('old value', 1);
INSERT INTO table2 VALUES ('', '', 1);
CREATE TRIGGER trigger_on_table1 BEFORE UPDATE ON table1 FOR EACH ROW
 BEGIN UPDATE table2 SET column1 = old.column1, column2 = new.column1; END;
UPDATE table1 SET column1 = 'new value';
SELECT * FROM table2;
```

  At the beginning of the UPDATE for the single row of table1, the value in column1 is 'old value' – so that is what is seen as old.column1.

  At the end of the UPDATE for the single row of table1, the value in column1 is 'new value' – so that is what is seen as new.column1. (OLD and NEW are qualifiers for table1, not table2.)

  Therefore, SELECT * FROM table2; returns ['old value', 'new value'].

  OLD.column-name does not exist for an INSERT trigger.

  NEW.column-name does not exist for a DELETE trigger.

  OLD and NEW are read-only; you cannot change their values.

- Deprecated or illegal statements:

  It is legal for the trigger action to include a SELECT statement or a REPLACE statement, but not recommended.

  It is illegal for the trigger action to include a qualified column reference other than OLD.column-name or NEW.column-name. For example, CREATE TRIGGER ... BEGIN UPDATE table1 SET table1.column1=5; END; is illegal.

  It is illegal for the trigger action to include statements that include a WITH clause, a DEFAULT VALUES clause, or an INDEXED BY clause.

  It is usually not a good idea to have a trigger on table1 which causes a change on table2, and at the same time have a trigger on table2 which causes a change on table1. For example:

```
CREATE TRIGGER trigger_on_table1
 BEFORE UPDATE ON table1
 FOR EACH ROW
 BEGIN UPDATE table2 SET column1 = column1 + 1; END;
CREATE TRIGGER trigger_on_table2
 BEFORE UPDATE ON table2
 FOR EACH ROW
 BEGIN UPDATE table1 SET column1 = column1 + 1; END;
```

  Luckily UPDATE table1 ... will not cause an infinite loop, because Tarantool recognizes when it has already updated so it will stop. However, not every DBMS acts this way.

Trigger activation

These are remarks concerning trigger activation.

Standard terminology:

- "trigger action time" = BEFORE or AFTER or INSTEAD OF
- "trigger event" = INSERT or DELETE or UPDATE
- "triggered statement" = BEGIN ... INSERT|DELETE|UPDATE ... END
- "triggered when clause" = WHEN (search condition)
- "activate" = execute a triggered statement
- some vendors use the word "fire" instead of "activate"

If there is more than one trigger for the same trigger event, Tarantool may execute the triggers in any order.

It is possible for a triggered statement to cause activation of another triggered statement. For example, this is legal:

```
CREATE TRIGGER on_t1 BEFORE DELETE ON t1 BEGIN DELETE FROM t2; END;
CREATE TRIGGER on_t2 BEFORE DELETE ON t2 BEGIN DELETE FROM t3; END;
```

Activation occurs FOR EACH ROW, not FOR EACH STATEMENT. Therefore, if no rows are candidates for insert or update or delete, then no triggers are activated.

The BEFORE trigger is activated even if the trigger event fails.

If an UPDATE trigger event does not make a change, the trigger is activated anyway. For example, if row 1 column1 contains 'a', and the trigger event is UPDATE ... SET column1 = `'a'`;, the trigger is activated.

The triggered statement may refer to a function: RAISE(FAIL, error-message). If a triggered statement invokes a RAISE(FAIL, error-message) function, or if a triggered statement causes an error, then statement execution stops immediately.

The triggered statement may refer to column values within the rows being changed. in this case:

- The row "as of before" the change is called the "old" row (which makes sense only for UPDATE and DELETE statements).
- The row "as of after" the change is called the "new" row (which makes sense only for UPDATE and INSERT statements).

This example shows how an INSERT can be done to a view by referring to the "new" row:

```
CREATE TABLE t (s1 INT PRIMARY KEY, s2 INT);
CREATE VIEW v AS SELECT s1, s2 FROM t;
CREATE TRIGGER tv INSTEAD OF INSERT ON v
  FOR EACH ROW
  BEGIN INSERT INTO t VALUES (new.s1, new.s2); END;
INSERT INTO v VALUES (1,2);
```

Ordinarily saying INSERT INTO view_name ... is illegal in Tarantool, so this is a workaround.

It is possible to generalize this so that all data-change statements on views will change the base tables, provided that the view contains all the columns of the base table, and provided that the triggers refer to those columns when necessary, as in this example:

```
CREATE TABLE base_table (primary_key_column INT PRIMARY KEY, value_column INT);
CREATE VIEW viewed_table AS SELECT primary_key_column, value_column FROM base_table;
CREATE TRIGGER viewed_insert INSTEAD OF INSERT ON viewed_table FOR EACH ROW
  BEGIN
    INSERT INTO base_table VALUES (new.primary_key_column, new.value_column);
  END;
CREATE TRIGGER viewed_update INSTEAD OF UPDATE ON viewed_table FOR EACH ROW
  BEGIN
    UPDATE base_table
    SET primary_key_column = new.primary_key_column, value_column = new.value_column
    WHERE primary_key_column = old.primary_key_column;
  END;
CREATE TRIGGER viewed_delete INSTEAD OF DELETE ON viewed_table FOR EACH ROW
  BEGIN
    DELETE FROM base_table WHERE primary_key_column = old.primary_key_column;
  END;
```

When INSERT or UPDATE or DELETE occurs for table X, Tarantool usually operates in this order (a basic scheme):

```
For each row
  Perform constraint checks
  For each BEFORE trigger that refers to table X
    Check that the trigger's WHEN condition is true.
    Execute what is in the trigger's BEGIN|END block.
  Insert or update or delete the row in table X.
  Perform more constraint checks
  For each AFTER trigger that refers to table X
    Check that the trigger's WHEN condition is true.
    Execute what is in the trigger's BEGIN|END block.
```

However, Tarantool does not guarantee execution order when there are multiple constraints, or multiple triggers for the same event (including NoSQL on_replace triggers or SQL INSTEAD OF triggers that affect a view of table X).

The maximum number of trigger activations per statement is 32.

INSTEAD OF triggers

A trigger which is created with the clause INSTEAD OF INSERT|UPDATE|DELETE ON view-name is an INSTEAD OF trigger. For each affected row, the trigger action is performed "instead of" the INSERT or UPDATE or DELETE statement that causes trigger activation.

For example, ordinarily it is illegal to INSERT rows in a view, but it is legal to create a trigger which intercepts attempts to INSERT, and puts rows in the underlying base table:

```
CREATE TABLE t1 (column1 INT PRIMARY KEY, column2 INT);
CREATE VIEW v1 AS SELECT column1, column2 FROM t1;
CREATE TRIGGER t1 INSTEAD OF INSERT ON v1 FOR EACH ROW BEGIN
 INSERT INTO t1 VALUES (NEW.column1, NEW.column2); END;
INSERT INTO v1 VALUES (1, 1);
-- ... The result will be: table t1 will contain a new row.
```

INSTEAD OF triggers are only legal for views, while BEFORE or AFTER triggers are not legal for views.

It is legal to create INSTEAD OF triggers with triggered WHEN clauses.

Limitations:

- It is legal to create INSTEAD OF triggers with UPDATE OF column-list clauses, but they are not standard SQL.

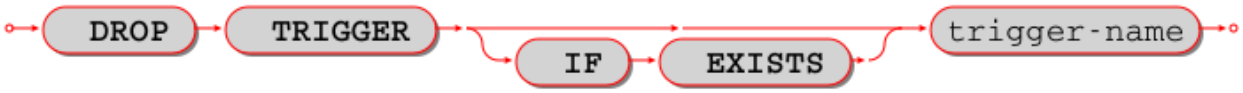
  Example:

```
CREATE TRIGGER et1
  INSTEAD OF UPDATE OF column2,column1 ON ev1
  FOR EACH ROW BEGIN
  INSERT INTO et2 VALUES (NEW.column1, NEW.column2); END;
```

DROP TRIGGER

Syntax:

DROP TRIGGER [IF EXISTS] trigger-name;



Drop a trigger.

The trigger-name must identify a trigger that was created earlier with the CREATE TRIGGER statement.

Rules: none

Actions:

1. Tarantool returns an error if the trigger does not exist.
2. The trigger is dropped.
3. Tarantool effectively executes a COMMIT statement.

Examples:

```
-- the simple case:
DROP TRIGGER tr;
-- with an IF EXISTS clause:
DROP TRIGGER IF EXISTS tr;
```

TRUNCATE

Syntax:

TRUNCATE TABLE table-name;

Remove all rows in the table.

TRUNCATE is considered to be a schema-change rather than a data-change statement, so it does not work within transactions (it cannot be rolled back).

Rules:

- It is illegal to truncate a table which is referenced by a foreign key.
- It is illegal to truncate a table which is also a system space, such as _space.

- The table must be a base table rather than a view.

Actions:

1. All rows in the table are removed. Usually this is faster than DELETE FROM table-name;.
2. If the table has an autoincrement primary key, its sequence is reset to zero.
3. There is no effect for any triggers associated with the table.
4. There is no effect on the counts for the row_count() function.
5. Only one action is written to the write-ahead log (with DELETE FROM table-name; there would be one action for each deleted row).

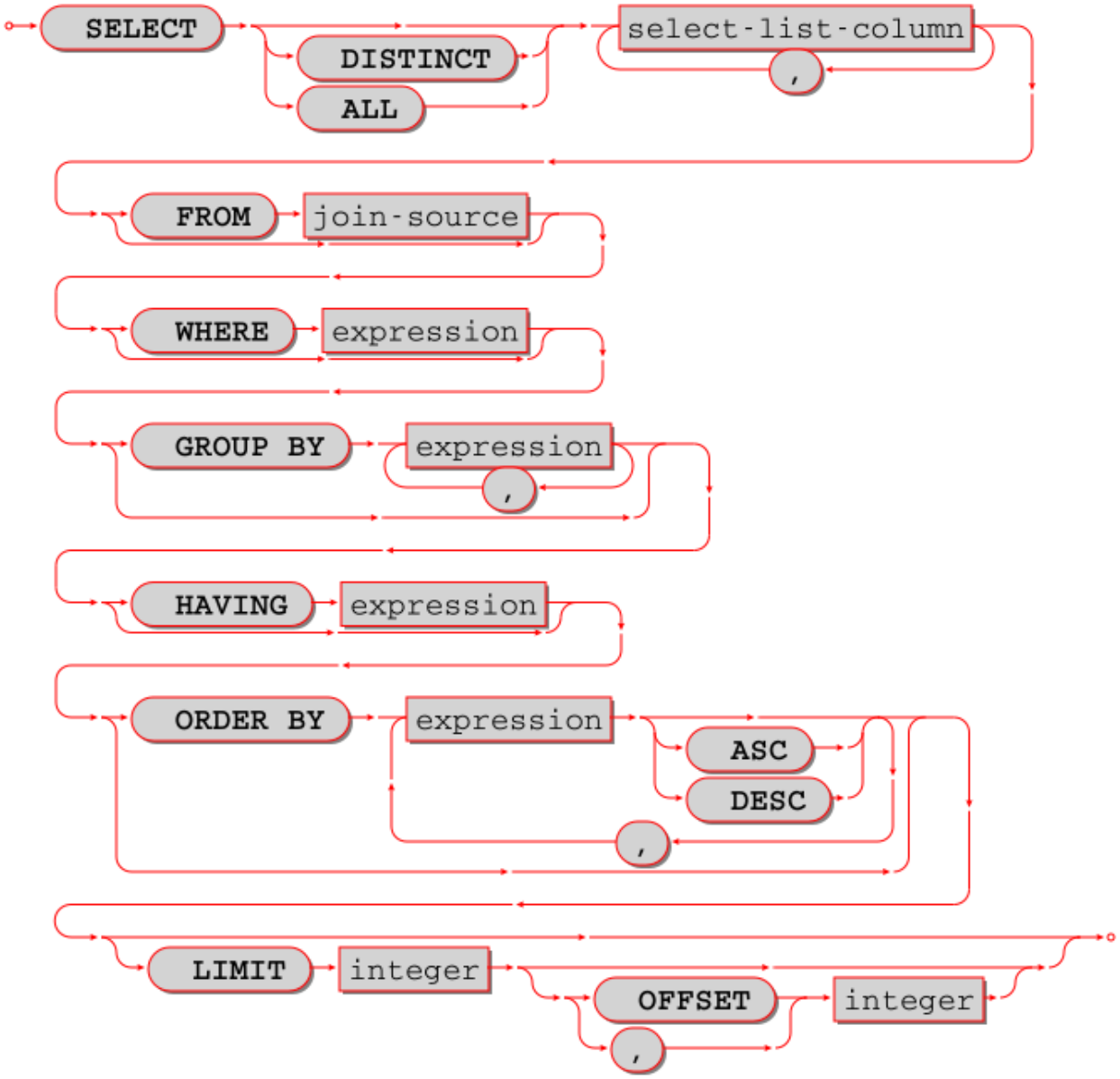
Example:

```
TRUNCATE TABLE t;
```

SELECT

Syntax:

SELECT [ALL|DISTINCT] select-list [from clause] [where clause] [group-by clause] [having clause] [order-by clause];



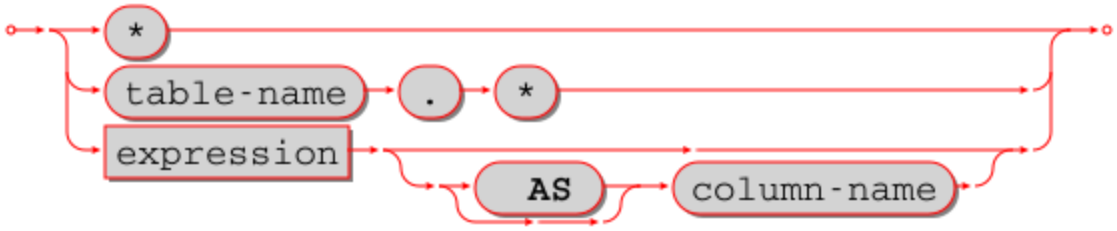
Select zero or more rows.

The clauses of the SELECT statement are discussed in the following five sections.

Select-list

Syntax:

select-list-column [, select-list-column ...] select-list-column:



Define what will be in a result set; this is a clause in a SELECT statement.

The select-list is a comma-delimited list of expressions, or * (asterisk). An expression can have an alias provided with [AS [column-name]] clause.

The * "asterisk" shorthand is valid if and only if the SELECT statement also contains a FROM clause which specifies the table or tables (details about the FROM clause are in the next section). The simple form is * which means "all columns" – for example, if the select is done for a table which contains three columns s1 s2 s3, then SELECT * ... is equivalent to SELECT s1, s2, s3 .... The qualified form is table-name.* which means "all columns in the specified table", which again must be a result of the FROM clause – for example, if the table is named table1, then table1.* is equivalent to a list of the columns of table1.

The [AS [column-name]] clause determines the column name. The column name is useful for two reasons:

- in a tabular display, the column names are the headings
- if the results of the SELECT are used in CREATE TABLE new-table-name ... AS SELECT select-list ..., then the column names in the new table will be the column names in the select-list.

If [AS [column-name]] is missing, Tarantool makes a name equal to the expression, for example SELECT 5*88 will cause the column name to be 5*88, but such names may be ambiguous or illegal in other contexts, so it is better to say, for example, SELECT 5 * 88 AS column1.

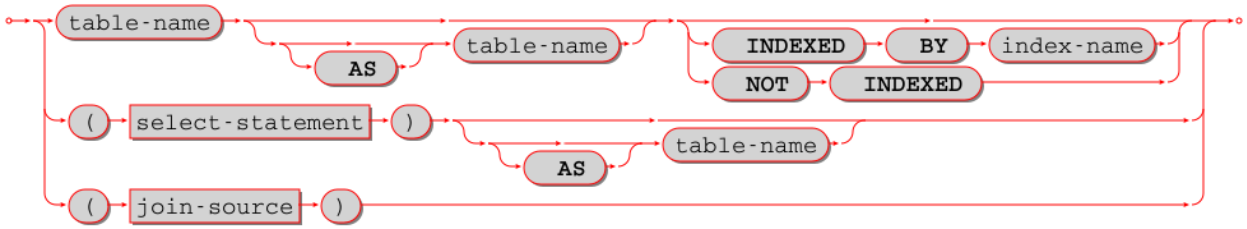
Examples:

```
-- the simple form:
SELECT 5;
-- with multiple expressions including operators:
SELECT 1, 2 * 2, 'Three' || 'Four';
-- with [[AS] column-name] clause:
SELECT 5 AS column1;
-- * which must be eventually followed by a FROM clause:
SELECT * FROM table1;
-- as a list:
SELECT 1 AS a, 2 AS b, table1.* FROM table1;
```

FROM clause

Syntax:

FROM table-reference [, table-reference ...]



Specify the table or tables for the source of a SELECT statement.

The table-reference must be a name of an existing table, or a subquery, or a joined table.

A joined table looks like this:

table-reference-or-joined-table join-operator table-reference-or-joined-table [join-specification]

A join-operator must be any of the standard types:

- [NATURAL] LEFT [OUTER] JOIN,
- [NATURAL] INNER JOIN, or
- CROSS JOIN

A join-specification must be any of:

- ON expression, or
- USING (column-name [, column-name ...])

Parentheses are allowed, and [[AS] correlation-name] is allowed.

The maximum number of joins in a FROM clause is 64.

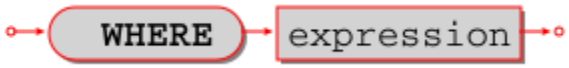
Examples:

```
-- the simplest form:
SELECT * FROM t;
-- with two tables, making a Cartesian join:
SELECT * FROM t1, t2;
-- with one table joined to itself, requiring correlation names:
SELECT a.*, b.* FROM t1 AS a, t1 AS b;
-- with a left outer join:
SELECT * FROM t1 LEFT JOIN t2;
```

WHERE clause

Syntax:

WHERE condition;



Specify the condition for filtering rows from a table; this is a clause in a SELECT or UPDATE or DELETE statement.

The condition may contain any expression that returns a BOOLEAN (TRUE or FALSE or UNKNOWN) value, or returns a value that can be interpreted as BOOLEAN (for example 1 or 0).

For each row in the table:

- if the condition is true, then the row is kept;
- if the condition is false or unknown, then the row is ignored.

In effect, WHERE condition takes a table with n rows and returns a table with n or fewer rows.

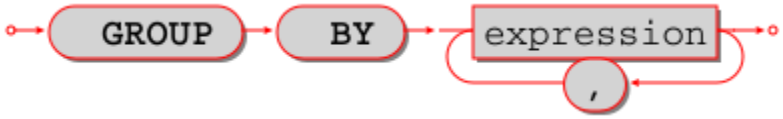
Examples:

```
-- with a simple condition:
SELECT 1 FROM t WHERE column1 = 5;
-- with a condition that contains AND and OR and parentheses:
SELECT 1 FROM t WHERE column1 = 5 AND (x > 1 OR y < 1);
```

GROUP BY clause

Syntax:

GROUP BY expression [, expression ...]



Make a grouped table; this is a clause in a SELECT statement.

The expressions should be column names in the table, and each column should be specified only once.

In effect, GROUP BY clause takes a table with rows that may have matching values, combines rows that have matching values into single rows, and returns a table which, because it is the result of GROUP BY, is called a grouped table.

Thus, if the input is a table:

```
a   b     c
-   -     -
1   'a'   'b
1   'b'   'b'
2   'a'   'b'
3   'a'   'b'
1   'b'   'b'
```

then GROUP BY a, b will produce a grouped table:

```
a   b     c
-   -     -
1   'a'   'b'
1   'b'   'b'
2   'a'   'b'
3   'a'   'b'
```

The rows where column a and column b have the same value have been merged; column c has been preserved but its value should not be depended on – if the rows were not all 'b', Tarantool could pick any value.

It is useful to envisage a grouped table as having hidden extra columns for the aggregation of the values, for example:

```
a   b     c    COUNT(a) SUM(a) MIN(c)
-   -     -    -------- ------ ------
1   'a'  'b'        2      2    'b'
1   'b'  'b'        1      1    'b'
2   'a'  'b'        1      2    'b'
    'a'  'b'        1      3    'b'
```

These extra columns are what aggregate functions are for.

Examples:

```
-- with a single column:
SELECT 1 FROM t GROUP BY column1;
-- with two columns:
SELECT 1 FROM t GROUP BY column1, column2;
```

Limitations:

- SELECT s1,s2 FROM t GROUP BY s1; is legal.
- SELECT s1 AS q FROM t GROUP BY q; is legal.
- SELECT s1 FROM t GROUP by 1; is legal.

Aggregate functions

Syntax:

function-name (one or more expressions)

Apply a built-in aggregate function to one or more expressions and return a scalar value.

Aggregate functions are only legal in certain clauses of SELECT for grouped tables. (A table is a grouped table if a GROUP BY clause is present.) Also, if an aggregate function is used in a select-list and GROUP BY clause is omitted, then Tarantool assumes SELECT ... GROUP BY [all columns];.

NULLs are ignored for all aggregate functions except COUNT(*).

AVG([DISTINCT] expression) Return the average value of expression.

Example: AVG(column1)

COUNT([DISTINCT] expression) Return the number of occurrences of expression.

Example: COUNT(column1)

COUNT(*) Return the number of occurrences of a row.

Example: COUNT(*)

GROUP_CONCAT(expression-1 [, expression-2]) Return a list of expression-1 values, separated by commas if expression-2 is omitted, or separated by the expression-2 value if expression-2 is not omitted.

Example: GROUP_CONCAT(column1)

MAX([DISTINCT] expression) Return the maximum value of expression.

Example: MAX(column1)

MIN([DISTINCT] expression) Return the minimum value of expression.

Example: MIN(column1)

SUM([DISTINCT] expression) Return the sum of values of expression.

Example: SUM(column1)

TOTAL([DISTINCT] expression) Return the sum of values of expression.

Example: TOTAL(column1)

HAVING clause

Syntax:

HAVING condition;

Specify the condition for filtering rows from a grouped table; this is a clause in a SELECT statement.

The clause preceding the HAVING clause may be a GROUP BY clause. HAVING operates on the table that the GROUP BY produces, which may contain grouped columns and aggregates.

If the preceding clause is not a GROUP BY clause, then there is only one group and the HAVING clause may only contain aggregate functions or literals.

For each row in the table:

- if the condition is true, then the row is kept;
- if the condition is false or unknown, then the row is ignored.

In effect, HAVING condition takes a table with n rows and returns a table with n or fewer rows.

Examples:

```
-- with a simple condition:
SELECT 1 FROM t GROUP BY column1 HAVING column2 > 5;
-- with a more complicated condition:
SELECT 1 FROM t GROUP BY column1 HAVING column2 > 5 OR column2 < 5;
-- with an aggregate:
SELECT x, SUM(y) FROM t GROUP BY x HAVING SUM(y) > 0;
-- with no GROUP BY and an aggregate:
SELECT SUM(y) FROM t GROUP BY x HAVING MIN(y) < MAX(y);
```

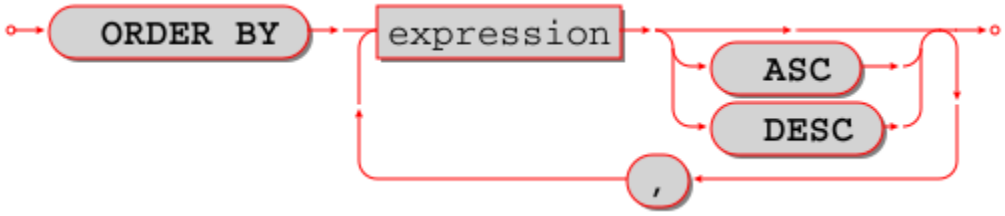
Limitations:

- HAVING without GROUP BY is not supported for multiple tables.

ORDER BY clause

Syntax:

ORDER BY expression [ASC|DESC] [, expression [ASC|DESC] ...]



Put rows in order; this is a clause in a SELECT statement.

An ORDER BY expression has one of three types which are checked in order:

1. Expression is a positive integer, representing the ordinal position of the column in the select list. For example, in the statement SELECT x, y, z FROM t ORDER BY 2; ORDER BY 2 means “order by the second column in the select list”, which is y.

2. Expression is a name of a column in the select list, which is determined by an AS clause. For example, in the statement SELECT x, y AS x, z FROM t ORDER BY x; ORDER BY x means "order by the column explicitly named x in the select list", which is the second column.
3. Expression contains a name of a column in a table of the FROM clause. For example, in the statement SELECT x, y FROM t1 JOIN t2 ORDER BY z; ORDER BY z means "order by a column named z which is expected to be in table t1 or table t2".

If both tables contain a column named z, then Tarantool will choose the first column that it finds.

The expression may also contain operators and function names and literals. For example, in the statement SELECT x, y FROM t ORDER BY UPPER(z); ORDER BY UPPER(z) means "order by the uppercase form of column t.z", which may be similar to doing ordering in a case-insensitive manner.

Type 3 is illegal if the SELECT statement contains UNION or EXCEPT or INTERSECT.

If an ORDER BY clause contains multiple expressions, then expressions on the left are processed first and expressions on the right are processed only if necessary for tie-breaking. For example, in the statement SELECT x, y FROM t ORDER BY x, y; if there are two rows which both have the same values for column x, then an additional check is made to see which row has a greater value for column y.

In effect, ORDER BY clause takes a table with rows that may be out of order, and returns a table with rows in order.

Sorting order:

- The default order is ASC (ascending), the optional order is DESC (descending).
- NULLs come first, then numbers (INTEGER or NUMBER), then STRINGs, then VARBINARYs.
- Within STRINGs, ordering is according to collation.
- Collation may be specified within the ORDER BY column-list, or may be default.

Examples:

```
-- with a single column:
SELECT 1 FROM t ORDER BY column1;
-- with two columns:
SELECT 1 FROM t ORDER BY column1, column2;
-- with a variety of data:
CREATE TABLE h (s1 INT PRIMARY KEY, s2 INT);
INSERT INTO h VALUES (7,'A'),(4,'A '),(-4,'AZ'),(17,17),(23,NULL);
INSERT INTO h VALUES (17.5,'Д'),(1e+300,'a'),(0,''),(-1,'');
SELECT * FROM h ORDER BY s2, s1;
-- The result of the above SELECT will be:
- - [23, null]
  - [17, 17]
  - [-1, '']
  - [0, '']
  - [7, 'A']
  - [4, 'A ']
  - [-4, 'AZ']
  - [1e+300, 'a']
  - [17.5, 'Д']
...
```

Limitations:

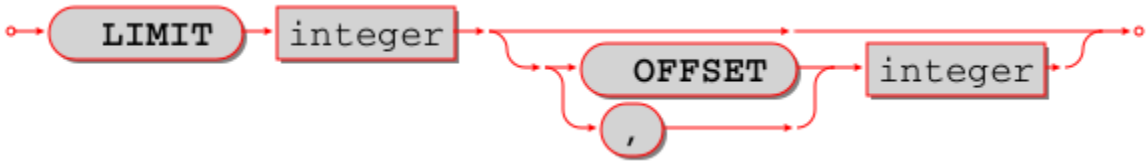
- ORDER BY 1 is legal. This is common but is not standard SQL nowadays.

LIMIT clause

Syntax:

- LIMIT limit-expression [OFFSET offset-expression]
- LIMIT offset-expression, limit-expression

Note: The above is not a typo: offset-expression and limit-expression are in reverse order if a comma is used.



Specify a maximum number of rows and a start row; this is a clause in a SELECT statement.

Expressions may contain integers and arithmetic operators or functions, for example ABS(-3/1). However, the result must be an integer value greater than or equal to zero.

Usually the LIMIT clause follows an ORDER BY clause, because otherwise Tarantool does not guarantee that rows are in order.

Examples:

```
-- simple case:
SELECT * FROM t LIMIT 3;
-- both limit and order:
SELECT * FROM t LIMIT 3 OFFSET 1;
-- applied to a UNIONed result (LIMIT clause must be the final clause):
SELECT column1 FROM table1 UNION SELECT column1 FROM table2 ORDER BY 1 LIMIT 1;
```

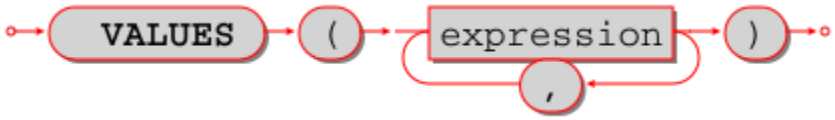
Limitations:

- If ORDER BY ... LIMIT is used, then all order-by columns must be ASC or all must be DESC.

VALUES

Syntax:

VALUES (expression [, expression ...]) [, (expression [, expression ...])



Select one or more rows.

VALUES has the same effect as SELECT, that is, it returns a result set, but VALUES statements may not have FROM or GROUP or ORDER BY or LIMIT clauses.

VALUES may be used wherever SELECT may be used, for example in subqueries.

Examples:

```
-- simple case:
VALUES (1);
-- equivalent to SELECT 1, 2, 3:
VALUES (1, 2, 3);
-- two rows:
VALUES (1, 2, 3), (4, 5, 6);
```

Subquery

Syntax:

- SELECT-statement syntax
- VALUES-statement syntax

A subquery has the same syntax as a SELECT statement or VALUES statement embedded inside a main statement.

Note: The SELECT and VALUES statements are called "queries" because they return answers, in the form of result sets.

Subqueries may be the second part of INSERT statements. For example:

```
INSERT INTO t2 SELECT a,b,c FROM t1;
```

Subqueries may be in the FROM clause of SELECT statements.

Subqueries may be expressions, or be inside expressions. In this case they must be parenthesized, and usually the number of rows must be 1. For example:

```
SELECT 1, (SELECT 5), 3 FROM t WHERE c1 * (SELECT COUNT(*) FROM t2) > 5;
```

Subqueries may be expressions on the right side of certain comparison operators, and in this unusual case the number of rows may be greater than 1. The comparison operators are: [NOT] EXISTS and [NOT] IN. For example:

```
DELETE FROM t WHERE s1 NOT IN (SELECT s2 FROM t);
```

Subqueries may refer to values in the outer query. In this case, the subquery is called a "correlated subquery".

Subqueries may refer to rows which are being updated or deleted by the main query. In that case, the subquery finds the matching rows first, before starting to update or delete. For example, after:

```
CREATE TABLE t (s1 INT PRIMARY KEY, s2 INT);
INSERT INTO t VALUES (1,3),(2,1);
DELETE FROM t WHERE s2 NOT IN (SELECT s1 FROM t);
```

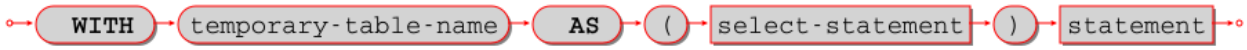
only one of the rows is deleted, not both rows.

WITH clause

WITH clause (common table expression)

Syntax:

WITH temporary-table-name AS (subquery) [, temporary-table-name AS (subquery)] SELECT statement | INSERT statement | DELETE statement | UPDATE statement | REPLACE statement;




```
WITH v AS (SELECT * FROM t) SELECT * FROM v;
```

is equivalent to creating a view and selecting from it:

```
CREATE VIEW v AS SELECT * FROM t;
SELECT * FROM v;
```

The difference is that a WITH-clause "view" is temporary and only useful within the same statement. No CREATE privilege is required.

The WITH-clause can also be thought of as a subquery that has a name. This is useful when the same subquery is being repeated. For example:

```
SELECT * FROM t WHERE a < (SELECT s1 FROM x) AND b < (SELECT s1 FROM x);
```

can be replaced with:

```
WITH S AS (SELECT s1 FROM x) SELECT * FROM t,S WHERE a < S.s1 AND b < S.s1;
```

This "factoring out" of a repeated expression is regarded as good practice.

Examples:

```
WITH cte AS (VALUES (7,'') INSERT INTO j SELECT * FROM cte;
WITH cte AS (SELECT s1 AS x FROM k) SELECT * FROM cte;
WITH cte AS (SELECT COUNT(*) FROM k WHERE s2 < 'x' GROUP BY s3)
  UPDATE j SET s2 = 5
  WHERE s1 = (SELECT s1 FROM cte) OR s3 = (SELECT s1 FROM cte);
```

WITH can only be used at the beginning of a statement, therefore it cannot be used at the beginning of a subquery or after a set operator or inside a CREATE statement.

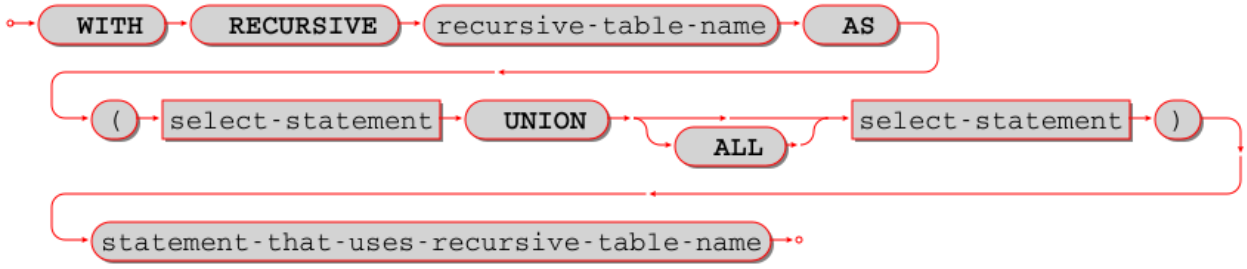
A WITH-clause "view" is read-only because Tarantool does not support updatable views.

WITH RECURSIVE

WITH RECURSIVE clause (iterative common table expression)

The real power of WITH lies in the WITH RECURSIVE clause, which is useful when it is combined with UNION or UNION ALL:

WITH RECURSIVE recursive-table-name AS (SELECT ... FROM non-recursive-table-name ... UNION [ALL] SELECT ... FROM recursive-table-name ...) statement-that-uses-recursive-table-name;



In non-SQL this can be read as: starting with a seed value from a non-recursive table, produce a recursive viewed table, UNION that with itself, UNION that with itself, UNION that with itself ... forever, or until a condition in the WHERE clause says "stop".

For example:

```
CREATE TABLE ts (s1 INT PRIMARY KEY);
INSERT INTO ts VALUES (1);
WITH RECURSIVE w AS (
  SELECT s1 FROM ts
  UNION ALL
  SELECT s1+1 FROM w WHERE s1 < 4)
SELECT * FROM w;
```

First, table w is seeded from t1, so it has one row: [1].

Then, UNION ALL (SELECT s1+1 FROM w) takes the row from w – which contains [1] – adds 1 because the select list says "s1+1", and so it has one row: [2].

Then, UNION ALL (SELECT s1+1 FROM w) takes the row from w – which contains [2] – adds 1 because the select list says "s1+1", and so it has one row: [3].

Then, UNION ALL (SELECT s1+1 FROM w) takes the row from w – which contains [3] – adds 1 because the select list says "s1+1", and so it has one row: [4].

Then, UNION ALL (SELECT s1+1 FROM w) takes the row from w – which contains [4] – and now the importance of the WHERE clause becomes evident, because "s1 < 4" is false for this row, and therefore we have reached the "stop" condition.

So, before the "stop", table w got 4 rows – [1], [2], [3], [4] – and the result of the statement looks like:

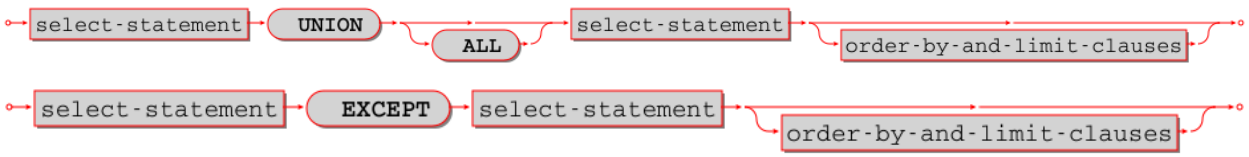
```
tarantool> WITH RECURSIVE w AS (
         >   SELECT s1 FROM ts
         >   UNION ALL
         >   SELECT s1+1 FROM w WHERE s1 < 4)
         > SELECT * FROM w;
---
- - [1]
  - [2]
  - [3]
  - [4]
...
```

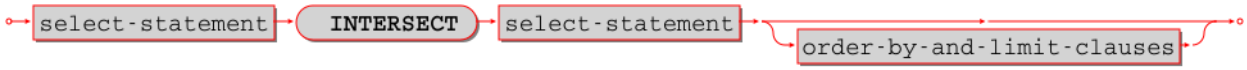
In other words, this WITH RECURSIVE ... SELECT produces a table of auto-incrementing values.

UNION, EXCEPT, and INTERSECT clauses

Syntax:

- select-statement UNION [ALL] select-statement [ORDER BY clause] [LIMIT clause];
- select-statement EXCEPT select-statement [ORDER BY clause] [LIMIT clause];
- select-statement INTERSECT select-statement [ORDER BY clause] [LIMIT clause];





UNION, EXCEPT, and INTERSECT are collectively called “set operators” or “table operators”. In particular:

- a UNION b means “take rows which occur in a OR b”.
- a EXCEPT b means “take rows which occur in a AND NOT b”.
- a INTERSECT b means “take rows which occur in a AND b”.

Duplicate rows are eliminated unless ALL is specified.

The select-statements may be chained: SELECT ... SELECT ... SELECT ...;

Each select-statement must result in the same number of columns.

The select-statements may be replaced with VALUES statements.

The maximum number of set operations is 50.

Example:

```
CREATE TABLE t1 (s1 INT PRIMARY KEY, s2 VARCHAR(1));
CREATE TABLE t2 (s1 INT PRIMARY KEY, s2 VARCHAR(1));
INSERT INTO t1 VALUES (1,'A'),(2,'B'),(3,NULL);
INSERT INTO t2 VALUES (1,'A'),(2,'C'),(3,NULL);
SELECT s2 FROM t1 UNION SELECT s2 FROM t2;
SELECT s2 FROM t1 UNION ALL SELECT s2 FROM t2 ORDER BY s2;
SELECT s2 FROM t1 EXCEPT SELECT s2 FROM t2;
SELECT s2 FROM t1 INTERSECT SELECT s2 FROM t2;
```

In this example:

- The UNION query returns 4 rows: NULL, ‘A’, ‘B’, ‘C’.
- The UNION ALL query returns 6 rows: NULL, NULL, ‘A’, ‘A’, ‘B’, ‘C’.
- The EXCEPT query returns 1 row: ‘B’.
- The INTERSECT query returns 2 rows: NULL, ‘A’.

Limitations:

- Parentheses are not allowed.
- Evaluation is left to right, INTERSECT does not have precedence.

Example:

```
CREATE TABLE t01 (s1 INT PRIMARY KEY, s2 VARCHAR(1));
CREATE TABLE t02 (s1 INT PRIMARY KEY, s2 VARCHAR(1));
CREATE TABLE t03 (s1 INT PRIMARY KEY, s2 VARCHAR(1));
INSERT INTO t01 VALUES (1,'A');
INSERT INTO t02 VALUES (1,'B');
INSERT INTO t03 VALUES (1,'A');
SELECT s2 FROM t01 INTERSECT SELECT s2 FROM t03 UNION SELECT s2 FROM t02;
SELECT s2 FROM t03 UNION SELECT s2 FROM t02 INTERSECT SELECT s2 FROM t03;
-- ... results are different.
```

INDEXED BY clause

Syntax:

INDEXED BY index-name

The INDEXED BY clause may be used in a SELECT, DELETE, or UPDATE statement, immediately after the table-name. For example:

```
DELETE FROM table7 INDEXED BY index7 WHERE column1 = 'a';
```

In this case the search for 'a' will take place within index7. For example:

```
SELECT * FROM table7 NOT INDEXED WHERE column1 = 'a';
```

In this case the search for 'a' will be done via a search of the whole table, what is sometimes called a "full table scan", even if there is an index for column1.

Ordinarily Tarantool chooses the appropriate index or lookup method depending on a complex set of "optimizer" rules; the INDEXED BY clause overrides the optimizer choice.

Example:

Suppose a table has two columns:

- The first column is the primary key and therefore it has an automatic index named pk_unnamed_T_1.
- The second column has an index created by the user.

The user selects with INDEXED BY the-index-on-column1, then selects with INDEXED BY the-index-on-column-2.

```
CREATE TABLE t (column1 INT PRIMARY KEY, column2 INT);
CREATE INDEX i ON t (column2);
INSERT INTO t VALUES (1,2),(2,1);
SELECT * FROM t INDEXED BY "pk_unnamed_T_1";
SELECT * FROM t INDEXED BY i;
-- Result for the first select: (1,2),(2,1)
-- Result for the second select: (2,1),(1,2).
```

Transactions

START TRANSACTION

Syntax:

START TRANSACTION;

Start a transaction. After START TRANSACTION;, a transaction is "active". If a transaction is already active, then START TRANSACTION; is illegal.

Transactions should be active for fairly short periods of time, to avoid concurrency issues. To end a transaction, say COMMIT; or ROLLBACK;.

Just like in NoSQL, transaction control statements are subject to limitations set by the storage engine involved:

- For memtx storage engine, if a yield happens within an active transaction, the transaction is rolled back.
- For vinyl engine, yields are allowed.

However,transaction control statements still may not work as you expect when run over a network connection: a transaction is associated with a fiber, not a network connection, and different transaction control statements sent via the same network connection may be executed by different fibers from the fiber pool.

In order to ensure that all statements are part of the intended transaction, put all of them between START TRANSACTION; and COMMIT; or ROLLBACK; then send as a single batch. For example:

- Enclose each separate SQL statement in a box.execute() function.
- Pass all the box.execute() functions to the server in a single message.

  If you are using a console, you can do this by writing everything on a single line.

  If you are using net.box, you can do this by putting all the function calls in a single string and calling eval(string).

Example:

```
START TRANSACTION;
```

Example of a whole transaction sent to a server on localhost:3301 with eval(string):

```
net_box = require('net.box')
conn = net_box.new('localhost', 3301)
s = 'box.execute([[START TRANSACTION; ]]) '
s = s .. 'box.execute([[INSERT INTO t VALUES (1); ]]) '
s = s .. 'box.execute([[ROLLBACK; ]]) '
conn:eval(s)
```

COMMIT

Syntax:

COMMIT;

Commit an active transaction, so all changes are made permanent and the transaction ends.

COMMIT is illegal unless a transaction is active. If a transaction is not active then SQL statements are committed automatically.

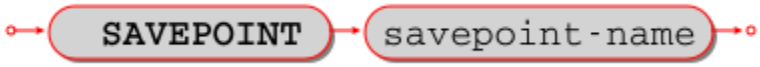
Example:

```
COMMIT;
```

SAVEPOINT

Syntax:

SAVEPOINT savepoint-name;



Set a savepoint, so that ROLLBACK TO savepoint-name is possible.

SAVEPOINT is illegal unless a transaction is active.

If a savepoint with the same name already exists, it is released before the new savepoint is set.

Example:

```
SAVEPOINT x;
```

RELEASE SAVEPOINT

Syntax:

RELEASE SAVEPOINT savepoint-name;

Release (destroy) a savepoint created by SAVEPOINT statement.

RELEASE is illegal unless a transaction is active.

Savepoints are released automatically when a transaction ends.

Example:

```
RELEASE SAVEPOINT x;
```

ROLLBACK

Syntax:

ROLLBACK [TO [SAVEPOINT] savepoint-name];

If ROLLBACK does not specify a savepoint-name, rollback an active transaction, so all changes since START TRANSACTION are cancelled, and the transaction ends.

If ROLLBACK does specify a savepoint-name, rollback an active transaction, so all changes since savepoint-name are cancelled, and the transaction does not end.

ROLLBACK is illegal unless a transaction is active.

Examples:

```
-- the simple form:
ROLLBACK;
-- the form so changes before a savepoint are not cancelled:
ROLLBACK TO SAVEPOINT x;
```

```
-- An example of a Lua function that will do a transaction
-- containing savepoint and rollback to savepoint.
function f()
box.execute([[DROP TABLE IF EXISTS t;]]) -- commits automatically
box.execute([[CREATE TABLE t (s1 VARCHAR(20) PRIMARY KEY);]]) -- commits automatically
box.execute([[START TRANSACTION;]]) -- after this succeeds, a transaction is active
box.execute([[INSERT INTO t VALUES ('Data change #1');]])
box.execute([[SAVEPOINT "1";]])
box.execute([[INSERT INTO t VALUES ('Data change #2');]])
box.execute([[ROLLBACK TO SAVEPOINT "1";]]) -- rollback Data change #2
box.execute([[ROLLBACK TO SAVEPOINt "1";]]) -- this is legal but does nothing
box.execute([[COMMIT;]]) -- make Data change #1 permanent, end the transaction
end
```

Functions

Syntax:

function-name (one or more expressions)

Apply a built-in function to one or more expressions and return a scalar value.

Tarantool supports 32 built-in functions.

CHAR([numeric-expression [,numeric-expression...]) Return the characters whose Unicode code point values are equal to the numeric expressions.

Short example: The first 128 Unicode characters are the "ASCII" characters, so CHAR(65,66,67) is 'ABC'.

Long example: For the current list of Unicode characters, in order by code point, see www.unicode.org/Public/UCD/latest/ucd/UnicodeData.txt. In that list, there is a line for a Linear B ideogram 100CC;LINEAR B IDEOGRAM B240 WHEELED CHARIOT ... Therefore, for a string with a chariot in the middle, use the concatenation operator || and the CHAR function 'start of string ' || CHAR(0X100CC) || ' end of string'.

COALESCE(expression, expression [, expression ...]) Return the value of the first non-NULL expression, or, if all expression values are NULL, return NULL.

Example: COALESCE(NULL, 17, 32) is 17

HEX(expression) Return the hexadecimal code for each byte in expression, which may be either a string or a byte sequence. For ASCII characters, this is straightforward because the encoding is the same as the code point value. For non-ASCII characters, since character strings are usually encoded in UTF-8, each character will require two or more bytes.

Examples:

- HEX('A') will return '41'
- HEX('Д') will return 'D094'

IFNULL(expression, expression) Return the value of the first non-NULL expression, or, if both expression values are NULL, return NULL. Thus IFNULL(expression,expression) is the same as COALESCE(expression, expression).

Example: IFNULL(NULL, 17) is 17

LENGTH(expression) Return the number of characters in the expression, or the number of bytes in the expression. It depends on the data type: strings with data type STRING are counted in characters, byte sequences with data type VARBINARY are counted in bytes and are not ended by the

nul character. There are two aliases for LENGTH(expression) – CHAR_LENGTH(expression) and CHARACTER_LENGTH(expression) do the same thing.

Examples:

- LENGTH('ДД') is 2, the string has 2 characters
- LENGTH(CAST('ДД' AS VARBINARY)) is 4, the string has 4 bytes
- LENGTH(CHAR(0,65)) is 2, '0' does not mean 'end of string'
- LENGTH(X'410041') is 3, X'...' byte sequences have type VARBINARY

NULLIF(expression-1, expression-2) Return expression-1 if expression-1 <> expression-2, otherwise return NULL.

Examples:

- NULLIF('a','A') is 'a'
- NULLIF(1.00, 1) is NULL

PRINTF(string-expression [, expression ...]) Return a string formatted according to the rules of the C sprintf() function, where %d%s means the next two arguments are a number and a string, etc.

If an argument is missing or is NULL, it becomes:

- '0' if the format requires an integer,
- '0.0' if the format requires a decimal number,
- '' if the format requires a string.

Example: PRINTF('%da', 5) is '5a'

QUOTE(string-literal) Return a string with enclosing quotes if necessary, and with quotes inside the enclosing quotes if necessary. This function is useful for creating strings which are part of SQL statements, because of SQL's rules that string literals are enclosed by single quotes, and single quotes inside such strings are shown as two single quotes in a row.

Example: QUOTE('a') is ''a''

SOUNDEX(string-expression) Return a four-character string which represents the sound of string-expression. Often words and names which have different spellings will have the same Soundex representation if they are pronounced similarly, so it is possible to search by what they sound like. The algorithm works with characters in the Latin alphabet and works best with English words.

Example: SOUNDEX('Crater') and SOUNDEX('Creature') both return 'C636'.

UNICODE(string-expression) Return the Unicode code point value of the first character of string-expression. If string-expression is empty, the return is NULL. This is the reverse of CHAR(integer).

Example: UNICODE('Щ') is 1065 (hexadecimal 0429)

UPPER(string-expression) Return the expression, with lower-case characters converted to upper case. This is the reverse of LOWER(string-expression).

Example: UPPER('-4щl') is '-4ЩL'

VERSION() Return the Tarantool version.

Example: for a March 2019 build VERSION() is '2.1.1-374-g27283debc'

4.2 Built-in modules reference

This reference covers Tarantool's built-in Lua modules.

Note: Some functions in these modules are analogs to functions from standard Lua libraries. For better results, we recommend using functions from Tarantool's built-in modules.

4.2.1 Module box

As well as executing Lua chunks or defining your own functions, you can exploit Tarantool's storage functionality with the box module and its submodules.

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).

To catch errors that functions in box submodules may throw, use pcall.

The contents of the box module can be inspected at runtime with box, with no arguments. The box module contains:

Submodule box.cfg

The box.cfg submodule is for administrators to specify all the server configuration parameters.

Say box.cfg without braces to view the current configuration, for example:

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

To set the parameters, say box.cfg{...}, for example:

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

If you say box.cfg{} with no parameters, Tarantool applies default settings:

```
tarantool> box.cfg{}
tarantool> box.cfg -- sorted in the alphabetic order
---
- background                = false
  checkpoint_count          = 2
  checkpoint_interval       = 3600
  checkpoint_wal_threshold  = 1000000000000000000
  coredump                  = false
  custom_proc_title         = nil
  feedback_enabled          = true
  feedback_host             = 'https://feedback.tarantool.io'
```

(continues on next page)

(continued from previous page)

```
feedback_interval           = 3600
force_recovery              = false
hot_standby                 = false
io_collect_interval         = nil
listen                    = nil
log                       = nil
log_format                  = plain
log_level                 = 5
log_nonblock                = true
memtx_dir                   = '.'
memtx_max_tuple_size        = 1024 * 1024
memtx_memory                = 256 * 1024 *1024
memtx_min_tuple_size        = 16
net_msg_max               = 768
pid_file                  = nil
readahead                   = 16320
read_only                   = false
replication               = nil
replication_connect_timeout  = 4
replication_skip_conflict    = false
replication_sync_lag        = 10
replication_sync_timeout    = 300
replication_timeout         = 1
rows_per_wal                = 500000
slab_alloc_factor           = 1.05
snap_io_rate_limit          = nil
strip_core                = true
too_long_threshold          = 0.5
username                  = nil
vinyl_bloom_fpr             = 0.05
vinyl_cache               = 128
vinyl_dir                 = '.'
vinyl_max_tuple_size        = 1024 * 1024* 1024 * 1024
vinyl_memory                = 128 * 1024 * 1024
vinyl_page_size             = 8 * 1024
vinyl_range_size            = nil
vinyl_read_threads          = 1
vinyl_run_count_per_level   = 2
vinyl_run_size_ratio        = 3.5
vinyl_timeout             = 60
vinyl_write_threads         = 2
wal_dir                   = '.'
wal_dir_rescan_delay        = 2
wal_max_size              = 256 * 1024 * 1024
wal_mode                  = 'write'
worker_pool_threads         = 4
work_dir                  = nil
```

The first call to box.cfg{...} (with or without parameters) initiates Tarantool's database module box. Before Tarantool 2.0, you needed to call box.cfg{...} prior to performing any database operations. Now you can start working with the database outright, without calling box.cfg{...}. In this case, Tarantool initiates the database module and applies default settings, as if you said box.cfg{} (without parameters).

box.cfg{...} is also the command that reloads persistent data files into RAM upon restart once we have data.

Submodule box.ctl

The box.ctl submodule contains two wait functions and two functions related to triggers.

The wait functions wait_ro (wait until read-only) and wait_rw (wait until read-write) are useful during initialization of a server.

A particular use is for box_once(). For example, when a replica is initializing, it may call a box.once() function while the server is still in read-only mode, and fail to make changes that are necessary only once before the replica is fully initialized. This could cause conflicts between a master and a replica if the master is in read-write mode and the replica is in read-only mode. Waiting until "read only mode = false" solves this problem.

To see whether a function is already in read-only or read-write mode, check box.info.ro.

box.ctl.wait_ro([timeout])
Wait until box.info.ro is true.

Parameters

- timeout (number) – maximum number of seconds to wait

Return nil, or error (errors may be due to timeout or fiber cancellation)

Example:

```
tarantool> box.info().ro
---
- false
...

tarantool> n = box.ctl.wait_ro(0.1)
---
- error: timed out
...
```

box.ctl.wait_rw([timeout])
Wait until box.info.ro is false.

Parameters

- timeout (number) – maximum number of seconds to wait

Return nil, or error (errors may be due to timeout or fiber cancellation)

Example:

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

The box.ctl submodule also contains two functions for the two server trigger definitions: on_schema_init and on_shutdown.

box.ctl.on_schema_init(trigger-function[, old-trigger-function])
Create a "schema_init trigger". The trigger-function will be executed when box.cfg{} happens for the first time. That is, the schema_init trigger is called before the server's configuration and recovery begins, and therefore box.ctl.on_schema_init must be called before box.cfg is called.

Parameter: trigger-function (function) – function which will become the trigger function

Parameter: old-trigger-function (function) – existing trigger function which will be replaced by trigger-function

Return: nil or function pointer

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

A common use is: make a schema_init trigger function which creates a before_replace trigger function on a system space. Thus, since system spaces are created when the server starts, the before_replace triggers will be activated for each tuple in each system space. For example, such a trigger could change the storage engine of a given space, or make a given space replica-local while a replica is being bootstrapped. Making such a change after box.cfg is not reliable because other connections might use the database before the change can be made.

Details about trigger characteristics are in the triggers section.

Example:

Suppose that, before the server is fully up and ready for connections, you want to make sure that the engine of space space_name is vinyl. So you want to make a trigger that will be activated when a tuple is inserted in the _space system space. In this case you could end up with a master that has space-name with engine='memtx' and a replica that has space_name with engine='vinyl', with the same contents.

```
function function_for_before_replace(old, new)
  if new[3] == 'space_name' and new[4] ~= 'vinyl' then
    return new:update{{'=', 4, 'vinyl'}}
  end
end

box.ctl.on_schema_init(function()
  box.space._space:before_replace(function_for_before_replace)
end)

box.cfg{replication='master_uri', ...}
```

box.ctl.on_shutdown(trigger-function[, old-trigger-function])

Create a "shutdown trigger". The trigger-function will be executed whenever os.exit() happens, or when the server is shut down after receiving a SIGTERM or SIGINT or SIGHUP signal (but not after SIGSEGV or SIGABORT or any signal that causes immediate program termination). The trigger function is actually called just before shutdown, so the trigger function can still refer to any methods and members in other Tarantool modules.

Like box.ctl.on_schema_init(), box.ctl.on_shutdown() may be done before box.cfg{} is invoked.

Parameters

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

Return nil or function pointer

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

Details about trigger characteristics are in the triggers section.

Submodule box.index

Overview

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.

Index

Below is a list of all box.index functions and members.

Name	Use
index_object.unique	Flag, true if an index is unique
index_object.type	Index type
index_object.parts	Array of index key fields
index_object:pairs()	Prepare for iterating
index_object:select()	Select one or more tuples via index
index_object:get()	Select a tuple via index
index_object:min()	Find the minimum value in index
index_object:max()	Find the maximum value in index
index_object:random()	Find a random value in index
index_object:count()	Count tuples matching key value
index_object:update()	Update a tuple
index_object:delete()	Delete a tuple by key
index_object:alter()	Alter an index
index_object:drop()	Drop an index
index_object:rename()	Rename an index
index_object:bsize()	Get count of bytes for an index
index_object:stat()	Get statistics for an index
index_object:compact()	Remove unused index space
index_object:user_defined()	Any function / method that any user wants to add

object index_object

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

  Rtype boolean

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

index_object.parts
: An array describing the index fields. To learn more about the index field types, refer to this table.

  Rtype table

  Example:

```
tarantool> box.space.tester.index.primary
---
- unique: true
  parts:
```

(continues on next page)

(continued from previous page)

```
  - type: unsigned
    is_nullable: false
    fieldno: 1
  id: 0
  space_id: 513
  name: primary
  type: TREE
...
```

index_object:pairs([key[, iterator-type]])

Search for a tuple or a set of tuples via the given index, and allow iterating over one tuple at a time.

The key parameter specifies what must match within the index.

Note: key is only used to find the first match. Do not assume all matched tuples will contain the key.

The iterator 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.

For information about iterators' internal structures see the "Lua Functional library" documentation.

Parameters

- index_object (index_object) – an object reference.
- key (scalar/table) – value to be matched against the index key, which may be multi-part
- iterator – as defined in tables below. The default iterator type is 'EQ'

Return iterator which can be used in a for/end loop or with totable()

Possible errors:

- no such space; wrong type;
- selected iteration type is not supported for the index type;
- key is not supported for the iteration type.

Complexity factors: Index size, Index type; Number of tuples accessed.

A search-key-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 key will be compared to each part of an index key.

The returned tuples will be in order by index key value, or by the hash of the index key value if index type = 'hash'. If the index is non-unique, then duplicates will be secondarily in order by primary key value. The order will be reversed if the iterator type is 'LT' or 'LE' or 'REQ'.

Iterator types for TREE indexes

Type	Arguments	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 box.index.EQ. 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 box.index.GE.
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.

Informally, we can state that searches with TREE indexes are generally what users will find is intuitive, provided that there are no nils and no missing parts. Formally, the logic is as follows. A search key has zero or more parts, for example {}, {1,2,3},{1,nil,3}. An index key has one or more parts, for example {1}, {1,2,3},{1,2,3}. A search key may contain nil (but not msgpack.NULL, which is the wrong type). An index key may not contain nil or msgpack.NULL, although a later version of Tarantool will have different rules – the behavior of searches with nil is subject to change. Possible iterators are LT, LE, EQ, REQ, GE, GT. A search key is said to "match" an index key if the following statements, which are pseudocode for the comparison operation, return TRUE.

```
If (number-of-search-key-parts > number-of-index-key-parts) return ERROR
If (number-of-search-key-parts == 0) return TRUE
for (i = 1; ; ++i)
{
  if (i > number-of-search-key-parts) OR (search-key-part[i] is nil)
  {
    if (iterator is LT or GT) return FALSE
    return TRUE
  }
  if (type of search-key-part[i] is not compatible with type of index-key-part[i])
  {
    return ERROR
  }
  if (search-key-part[i] == index-key-part[i])
  {
```

(continues on next page)

(continued from previous page)

```
    continue
  }
  if (search-key-part[i] > index-key-part[i])
  {
    if (iterator is EQ or REQ or LE or LT) return FALSE
    return TRUE
  }
  if (search-key-part[i] < index-key-part[i])
  {
    if (iterator is EQ or REQ or GE or GT) return FALSE
    return TRUE
  }
}
```

Iterator types for HASH indexes

Type	Arguments	Description
box.index.ALL	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'	hash 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'	hash 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	Arguments	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'	bit-set 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_SET	bit-set 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_SET	bit-set 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_ALL_NOT_SET	bit-set value	If all 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	Arguments	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 RTREE". 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 values	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.NEIGHBOR or 'NEIGHBOR'	search value	If 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', '!']
```

(continues on next page)

(continued from previous page)

```
...
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 (index_object) – an object reference.
- key (scalar/table) – values to be matched against the index key
- options (table/nil) – none, any or all of next parameters
- options.iterator – type of iterator
- options.limit (number) – maximum number of tuples
- options.offset (number) – start tuple number

Return the tuple or tuples that match the field values.

Rtype array of tuples

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']
...
```

Note: 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.

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

Note: 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.

Note: 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 throws 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]
```

(continues on next page)

(continued from previous page)

```
...
```

index_object:get(key)

Search for a tuple via the given index, as described earlier.

Parameters

- index_object (index_object) – an object reference.
- key (scalar/table) – values to be matched against the index key

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.tester.index.primary:get(2)
---
- [2, 'Music']
...
```

index_object:min([key])

Find the minimum value in the specified index.

Parameters

- index_object (index_object) – an object reference.
- key (scalar/table) – values to be matched against the index key

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. Starting with Tarantool version 2.0, index_object:min(key value) will return nothing if key value is not equal to a value in the index.

Rtype tuple

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

Complexity factors: Index size, Index type.

Example:

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

index_object:max([key])

Find the maximum value in the specified index.

Parameters

- index_object (index_object) – an object reference.
- key (scalar/table) – values to be matched against the index key

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. Starting with Tarantool version 2.0, index_object:max(key value) will return nothing if key value is not equal to a value in the index.

Rtype tuple

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

Complexity factors: Index size, Index type.

Example:

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

index_object:random(seed)

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 (index_object) – an object reference.
- seed (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.tester.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 (index_object) – an object reference.
- key (scalar/table) – values to be matched against the index key
- iterator – comparison method

Return the number of matching index keys.

Rtype number

Example:

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

index_object:update(key, {{operator, field_no, value}, ...})
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 (index_object) – an object reference.
- key (scalar/table) – values to be matched against the index key
- operator (string) – operation type represented in string
- field_no (number) – what field the operation will apply to. The field number can be negative, meaning the position from the end of tuple. (#tuple + negative field number + 1)
- value (lua_value) – what value will be applied

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 (index_object) – an object reference.
- key (scalar/table) – values 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. It is legal in some circumstances to change one or more of the index characteristics, for example its type, its sequence options, its parts, and whether it is unique, Usually this causes rebuilding of the space, except for the simple case where a part's is_nullable flag is changed from false to true.

Parameters

- index_object (index_object) – an object reference.
- options (table) – options list, same as the options list for create_index, see the chart named Options for space_object:create_index().

Return nil

Possible errors:

- index does not exist,
- the primary-key index cannot be changed to {unique = false}.

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

Example:

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

tarantool> box.space.vinyl_space.index.i:alter({page_size=4096})
---
...
```

index_object:drop()

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

Parameters

- index_object (index_object) – an object reference.

Return nil.

Possible errors:

- index does not exist,
- 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 (index_object) – an object reference.
- index-name (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 (index_object) – an object reference.

Return number of bytes

Rtype number

index_object:stat()

Return statistics about actions taken that affect the index.

This is for use with the vinyl engine.

Some detail items in the output from index_object:stat() are:

- index_object:stat().latency – timings subdivided by percentages;
- index_object:stat().bytes – the number of bytes total;
- index_object:stat().disk.rows – the approximate number of tuples in each range;
- index_object:stat().disk.statement – counts of inserts|updates|upserts|deletes;
- index_object:stat().disk.compaction – counts of compactions and their amounts;
- index_object:stat().disk.dump – counts of dumps and their amounts;
- index_object:stat().disk.iterator.bloom – counts of bloom filter hits|misses;
- index_object:stat().disk.pages – the size in pages;
- index_object:stat().disk.last_level – size of data in the last LSM tree level;
- index_object:stat().cache.evict – number of evictions from the cache;
- index_object:stat().range_size – maximum number of bytes in a range;
- index_object:stat().dumps_per_compaction – average number of dumps required to trigger major compaction in any range of the LSM tree.

Summary index statistics are also available via box.stat.vinyl().

Parameters

- index_object (index_object) – an object reference.

Return statistics

Rtype table

index_object:compact()

Remove unused index space. For the memtx storage engine this method does nothing; index_object:compact() is only for the vinyl storage engine. For example, with vinyl, if a tuple is deleted, the space is not immediately reclaimed. There is a scheduler for reclaiming space automatically based on factors such as lsm shape and amplification as discussed in the section Storing data with vinyl, so calling index_object:compact() manually is not always necessary.

Return nil (Tarantool returns without waiting for compaction to complete)

index_object:user_defined()

Users can define any functions they want, and associate them with indexes: in effect they can make their own index methods. They do this by:

(1) creating a Lua function,

(2) adding the function name to a predefined global variable which has type = table, and

(3) invoking the function any time thereafter, as long as the server is up, by saying index_object:function-name([parameters]).

There are three predefined global variables:

- Adding to box_schema.index_mt makes the method available for all indexes.
- Adding to box_schema.memtx_index_mt makes the method available for all memtx indexes.
- Adding to box_schema.vinyl_index_mt makes the method available for all vinyl indexes.

Alternatively, user-defined methods can be made available for only one index, by calling getmetatable(index_object) and then adding the function name to the meta table.

Parameters

- index_object (index_object) – an object reference.
- any-name (any-type) – whatever the user defines

Example:

```
-- Visible to any index of a memtx space, no parameters.
-- After these requests, the value of global_variable will be 6.
box.schema.space.create('t', {engine='memtx'})
box.space.t:create_index('i')
global_variable = 5
function f() global_variable = global_variable + 1 end
box.schema.memtx_index_mt.counter = f
box.space.t.index.i:counter()
```

Example:

```
-- Visible to index box.space.t.index.i only, 1 parameter.
-- After these requests, the value of X will be 1005.
box.schema.space.create('t', {engine='memtx', id = 1000})
box.space.t:create_index('i')
X = 0
i = box.space.t.index.i
function f(i_arg, param) X = X + param + i_arg.space_id end
box.schema.memtx_index_mt.counter = f
meta = getmetatable(i)
meta.counter = f
i:counter(5)
```

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;
- raise 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
```

(continues on next page)

(continued from previous page)

```
  search_key = page[#page][1]
  iterator_string = "GT"
  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.

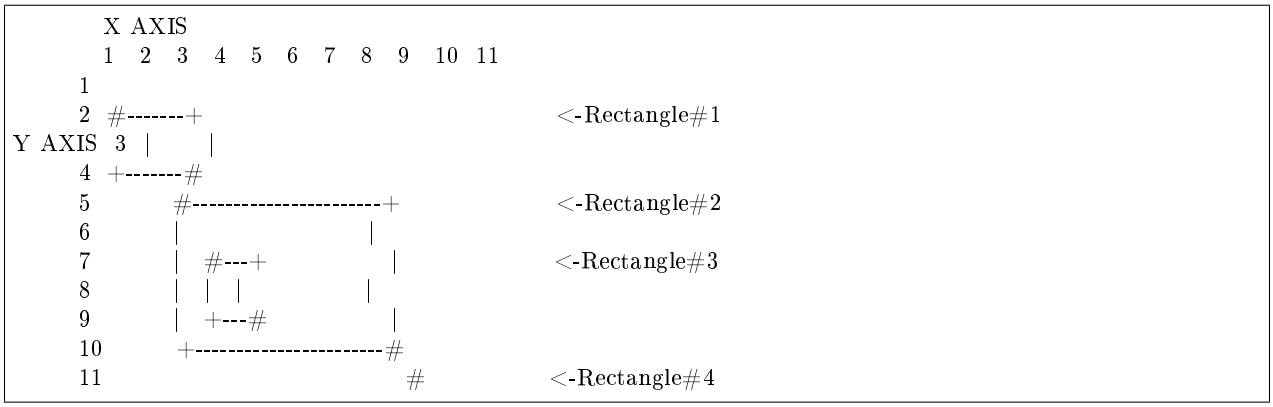
In this example the tuples are merely printed, a page at a time. But it should be simple to change the functionality, for example by yielding after each retrieval, or by breaking when the tuples fail to match some additional criteria.

```
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 option 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 option 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.

Submodule box.info

The box.info submodule provides access to information about server instance variables.

- cluster.uuid is the UUID of the replica set. Every instance in a replica set will have the same cluster.uuid value. This value is also stored in box.space._schema system space.
- gc() returns the state of the Tarantool garbage collector including the checkpoints and their consumers (users); see details below.
- id corresponds to replication.id (see below).
- lsn corresponds to replication.lsn (see below).
- memory() returns the statistics about memory (see below).
- pid is the process ID. This value is also shown by tarantool module and by the Linux command ps -A.
- ro is true if the instance is in "read-only" mode (same as read_only in box.cfg{}), or if status is 'orphan'.
- signature is the sum of all lsn values from the vector clocks (vclock) of all instances in the replica set.
- status corresponds to replication.upstream.status (see below).
- uptime is the number of seconds since the instance started. This value can also be retrieved with tarantool.uptime().
- uuid corresponds to replication.uuid (see below).
- vclock corresponds to replication.downstream.vclock (see below).
- version is the Tarantool version. This value is also shown by tarantool -V.

- vinyl() returns runtime statistics for the vinyl storage engine. This function is deprecated, use box.stat.vinyl() instead.

box.info.memory()
The memory function of box.info gives the admin user a picture of the whole Tarantool instance.

Note: To get a picture of the vinyl subsystem, use box.stat.vinyl() instead.

- memory().cache – number of bytes used for caching user data. The memtx storage engine does not require a cache, so in fact this is the number of bytes in the cache for the tuples stored for the vinyl storage engine.
- memory().data – number of bytes used for storing user data (the tuples) with the memtx engine and with level 0 of the vinyl engine, without taking memory fragmentation into account.
- memory().index – number of bytes used for indexing user data, including memtx and vinyl memory tree extents, the vinyl page index, and the vinyl bloom filters.
- memory().lua – number of bytes used for Lua runtime.
- memory().net – number of bytes used for network input/output buffers.
- memory().tx – number of bytes in use by active transactions. For the vinyl storage engine, this is the total size of all allocated objects (struct txv, struct vy_tx, struct vy_read_interval) and tuples pinned for those objects.

An example with a minimum allocation while only the memtx storage engine is in use:

```
tarantool> box.info.memory()
---
- cache: 0
  data: 6552
  tx: 0
  lua: 1315567
  net: 98304
  index: 1196032
...
```

box.info.gc()
The gc function of box.info gives the admin user a picture of the factors that affect the Tarantool garbage collector. The garbage collector compares vclock (vector clock) values of users and checkpoints, so a look at box.info.gc() may show why the garbage collector has not removed old WAL files, or show what it may soon remove.

- gc().consumers – a list of users whose requests might affect the garbage collector.
- gc().checkpoints – a list of preserved checkpoints.
- gc().checkpoints[n].references – a list of references to a checkpoint.
- gc().checkpoints[n].vclock – a checkpoint's vclock value.
- gc().checkpoints[n].signature – a sum of a checkpoint's vclock's components.
- gc().checkpoint_is_in_progress – true if a checkpoint is in progress, otherwise false
- gc().vclock – the garbage collector's vclock.
- gc().signature – the sum of the garbage collector's checkpoint's components.

box.info.replication

The replication section of box.info() contains statistics for all instances in the replica set in regard to the current instance (see also "Monitoring a replica set"):

- replication.id is a short numeric identifier of the instance within the replica set.
- replication.uuid is a globally unique identifier of the instance. This value is also stored in box.space._cluster system space.
- replication.lsn is the log sequence number (LSN) for the latest entry in the instance's write ahead log (WAL).
- replication.upstream contains statistics for the replication data uploaded by the instance.
- replication.upstream.status is the replication status of the instance:
  - auth means that the instance is getting authenticated to connect to a replication source.
  - connecting means that the instance is trying to connect to the replications source(s) listed in its replication parameter.
  - disconnected means that the instance is not connected to the replica set (due to network problems, not replication errors).
  - follow means that replication is in progress.
  - running means the instance's role is "master" (non read-only) and replication is in progress.
  - stopped means that replication was stopped due to a replication error (e.g. duplicate key).
  - orphan means that the instance has not (yet) succeeded in joining the required number of masters (see orphan status).
  - synch means that the master and replica are synchronizing to have the same data.
- replication.upstream.idle is the time (in seconds) since the instance received the last event from a master. This is the primary indicator of replication health. See more in Monitoring a replica set.
- replication.upstream.peer contains the replication user name, host IP adress and port number used for the instance. See more in Monitoring a replica set.
- replication.upstream.lag is the time difference between the local time at the instance, recorded when the event was received, and the local time at another master recorded when the event was written to the write ahead log on that master. See more in Monitoring a replica set.
- replication.upstream.message contains an error message in case of a degraded state, empty otherwise.
- replication.downstream contains statistics for the replication data requested and downloaded from the instance.
- replication.downstream.vclock contains the vector clock, which is a table of 'id, lsn' pairs, for example vclock: {1: 3054773, 4: 8938827, 3: 285902018}. Even if an instance is removed, its values will still appear here.
- replication.downstream.status = disconnected is displayed if the downstream instance disconnects from the upstream instance. Otherwise the status is not reported.

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:

This example is for a master-replica set that contains one master instance and one replica instance. The request was issued at the replica instance.

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

Function 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.

See an example of using box.once() while bootstrapping a replica set.

If an error occurs inside box.once() when initializing a database, you can re-execute the failed box.once() block without stopping the database. The solution is to delete the once object from the system space _schema. Say box.space._schema:select{}, find your once object there and delete it. For example, re-executing a block with key='hello' :

When box.once() is used for initialization, it may be useful to wait until the database is in an appropriate state (read-only or read-write). In that case, see the functions in the box.ctl submodule.

```
tarantool> box.space._schema:select{}
---
- - ['cluster', 'b4e15788-d962-4442-892e-d6c1dd5d13f2']
  - ['max_id', 512]
  - ['oncebye']
  - ['oncehello']
  - ['version', 1, 7, 2]
...

tarantool> box.space._schema:delete('oncehello')
---
- ['oncehello']
...

tarantool> box.once('hello', function() end)
---
...
```

Parameters

- key (string) – a value that will be checked
- function (function) – a function
- ... – arguments that must be passed to function

Submodule box.schema

Overview

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

Index

Below is a list of all box.schema functions.

Name	Use
box.schema.space.create()	Create a space
box.schema.user.create()	Create a user
box.schema.user.drop()	Drop a user
box.schema.user.exists()	Check if a user exists
box.schema.user.grant()	Grant privileges to a user or a role
box.schema.user.revoke()	Revoke privileges from a user or a role
box.schema.user.password()	Get a hash of a user's password
box.schema.user.passwd()	Associate a password with a user
box.schema.user.info()	Get a description of a user's privileges
box.schema.role.create()	Create a role
box.schema.role.drop()	Drop a role
box.schema.role.exists()	Check if a role exists
box.schema.role.grant()	Grant privileges to a role
box.schema.role.revoke()	Revoke privileges from a role
box.schema.role.info()	Get a description of a role's privileges
box.schema.func.create()	Create a function tuple
box.schema.func.drop()	Drop a function tuple
box.schema.func.exists()	Check if a function tuple exists
box.schema.sequence.create()	Create a new sequence generator
sequence_object:next()	Generate and return the next value
sequence_object:alter()	Change sequence options
sequence_object:reset()	Reset sequence state
sequence_object:set()	Set the new value
sequence_object:drop()	Drop the sequence
space_object:create_index()	Create an index

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

Create a space.

Parameters

- space-name (string) – name of space, which should conform to the rules for object names
- 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
engine	'memtx' or 'vinyl'	string	'memtx'
field_count	fixed count of fields: for example if field_count=5, it is illegal to insert a tuple with fewer than or more than 5 fields	number	0 i.e. not fixed
format	field names and types: See the illustrations of format clauses in the space_object:format() description and in the box.space._space example. Optional and usually not specified.	table	(blank)
id	unique identifier: users can refer to spaces with the id instead of the name	number	last space's id, +1
if_not_exists	create space only if a space with the same name does not exist already, otherwise do nothing but do not cause an error	boolean	false
is_local	space contents are replication-local: changes are stored in the write-ahead log of the local node but there is no replication.	boolean	false
temporary	space contents are temporary: changes are not stored in the write-ahead log and there is no replication. Note re storage engine: vinyl does not support temporary spaces.	boolean	false
user	name of the user who is considered to be the space's owner for authorization purposes	string	current user's name

There are three 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().

Example

```
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
         > })
---
...
```

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 reference on _user space.

The possible options are:

- if_not_exists = true|false (default = false) - boolean; true means there should be no error if the user already exists,

- password (default = '') - string; 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.

Note: The maximum number of users is 32.

Parameters

- user-name (string) – name of user, which should conform to the rules for object names
- options (table) – if_not_exists, password

Return nil

Examples:

```
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 reference on _user space.

Parameters

- user-name (string) – the name of the user
- options (table) – if_exists = true|false (default = false) - boolean; true means there should be no error if the user does not exist.

Examples:

```
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. For explanation of how Tarantool maintains user data, see section Users and reference on _user space.

Parameters

- 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 or to another role.

Parameters

- user-name (string) – the name of the user.

- privileges (string) – ‘read’ or ‘write’ or ‘execute’ or ‘create’ or ‘alter’ or ‘drop’ or a combination.
- object-type (string) – ‘space’ or ‘function’ or ‘sequence’ or ‘role’.
- object-name (string) – name of object to grant permissions for.
- 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’. In this case, object name is omitted.

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

The possible options are:

- grantor = grantor_name_or_id – string or number, for custom grantor,
- if_not_exists = true|false (default = false) - boolean; true means there should be no error if the user already has the privilege.

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

Revoke privileges from a user or from another role.

Parameters

- user-name (string) – the name of the user.
- privilege (string) – ‘read’ or ‘write’ or ‘execute’ or ‘create’ or ‘alter’ or ‘drop’ or a combination.
- object-type (string) – ‘space’ or ‘function’ or ‘sequence’.
- object-name (string) – the name of a function or space or sequence.

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 user's password. For explanation of how Tarantool maintains passwords, see section Passwords and reference on _user space.

Note:

- If a non-'guest' user has no password, it's impossible to connect to Tarantool using this user. The user is regarded as "internal" only, not usable from a remote connection. Such users can be useful if they have defined some procedures with the SETUID option, on which privileges are granted to externally-connectable users. This way, external users cannot create/drop objects, they can only invoke procedures.
- For the 'guest' user, it's impossible to set a password: that would be misleading, since 'guest' is the default user on a newly-established connection over a binary port, and Tarantool does not require a password to establish a binary connection. It is, however, possible to change the current user to 'guest' by providing the AUTH packet with no password at all or an empty password. This feature is useful for connection pools, which want to reuse a connection for a different user without re-establishing it.

Parameters

- password (string) – password to be hashed

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 the user specified by user-name. The user must exist and must not be 'guest'.

Users who wish to change their own passwords should use box.schema.user.passwd(password) syntax.

Administrators who wish to change passwords of other users should use box.schema.user.passwd(user-name, password) syntax.

Parameters

- 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.

box.schema.role.create(role-name[, {options}])
Create a role. For explanation of how Tarantool maintains role data, see section Roles.

Parameters

- role-name (string) – name of role, which should conform to the rules for object names

- options (table) – if_not_exists = true|false (default = false) - boolean; true means there should be no error if the role already exists

Return nil

Example:

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

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

Drop a role. For explanation of how Tarantool maintains role data, see section Roles.

Parameters

- role-name (string) – the name of the role
- options (table) – if_exists = true|false (default = false) - boolean; true means there should be no error if the role does not exist.

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.

Parameters

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

Rtype bool

Example:

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

box.schema.role.grant(role-name, privilege, object-type, object-name[, option])
box.schema.role.grant(role-name, privilege, 'universe'[, nil, option])
box.schema.role.grant(role-name, role-name[, nil, nil, option])

Grant privileges to a role.

Parameters

- role-name (string) – the name of the role.
- privilege (string) – 'read' or 'write' or 'execute' or 'create' or 'alter' or 'drop' or a combination.
- object-type (string) – 'space' or 'function' or 'sequence' or 'role'.
- object-name (string) – the name of a function or space or sequence or role.
- option (table) – if_not_exists = true|false (default = false) - boolean; true means there should be no error if the role already has the privilege.

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'. In this case, object name is omitted.

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(role-name, privilege, object-type, object-name)

Revoke privileges from a role.

Parameters

- role-name (string) – the name of the role.
- privilege (string) – 'read' or 'write' or 'execute' or 'create' or 'alter' or 'drop' or a combination.
- object-type (string) – 'space' or 'function' or 'sequence' or 'role'.
- object-name (string) – the name of a function or space or sequence or role.

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.

Parameters

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

Example:

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

box.schema.func.create(func-name[, {options-without-body}])

Create a function tuple. without including the body option. (For functions created without the body option, see box.schema.func.create(func-name [, {options-with-body}]).

This is called a "not persistent" function because functions without bodies are not persistent. This does not create the function itself – that is done with Lua – but if it is necessary to grant privileges for a function, box.schema.func.create must be done first. For explanation of how Tarantool maintains function data, see the reference for the box.space._func space.

The possible options are:

- if_not_exists = true|false (default = false) - boolean; true means there should be no error if the _func tuple already exists.
- setuid = true|false (default = false) - boolean; true means that Tarantool should treat the function's caller as the function's owner, with owner privileges. setuid works only over binary ports, setuid does not work if the function is invoked via an admin console or inside a Lua script.

- language = 'LUA'|'C' (default = 'LUA') - string.

  Parameters

  - func-name (string) – name of function, which should conform to the rules for object names
  - 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.create(func-name[, {options-with-body}])

Create a function tuple. including the body option. (For functions created without the body option, see box.schema.func.create(func-name [, {options-without-body}]).

This is called a "persistent" function because only functions with bodies are persistent. This does create the function itself, the body is a function definition. For explanation of how Tarantool maintains function data, see the reference for the box.space._func space.

The possible options are:

- if_not_exists = true|false (default = false) - boolean; same as for box.schema.func.create(func-name [, {options-without-body}]).
- setuid = true|false (default = false) - boolean; same as for box.schema.func.create(func-name [, {options-without-body}]).
- language = 'LUA'|'C' (default = 'LUA') - string. same as for box.schema.func.create(func-name [, {options-without-body}]).
- is_sandboxed = true|false (default = false) - boolean; whether the function should be executed in a sandbox.
- is_deterministic = true|false (default = false) - boolean; true means that the function should be deterministic, false means that the function may or may not be deterministic.
- body = function definition (default = nil) - string; the function definition.

  Parameters

  - func-name (string) – name of function, which should conform to the rules for object names
  - options (table) – if_not_exists, setuid, language, is_sandboxed, is_deterministic, body.

  Return nil

C functions are imported from .so files, Lua functions can be defined within body. We will only describe Lua functions in this section.

A function tuple with a body is "persistent" because the tuple is stored in a snapshot and is recoverable if the server restarts. All of the option values described in this section are visible in the box.space._func system space.

If is_sandboxed is true, then the function will be executed in an isolated environment: any operation that accesses the world outside the sandbox will be forbidden or will have no effect. Therefore a sandboxed function can only use modules and functions which cannot affect isolation: assert error ipairs math.* next pairs pcall print select string.* table.* tonumber tostring type unpack utf8.* xpcall. Also a sandboxed function cannot refer to global variables – they will be treated as local variables because the sandbox is established with setfenv. So a sandboxed function will happen to be stateless and deterministic.

If is_deterministic is true, there is no immediate effect. Tarantool plans to use the is_deterministic value in a future version. A function is deterministic if it always returns the same outputs given the same inputs. It is the function creator's responsibility to ensure that a function is truly deterministic.

Using a persistent Lua function

After a persistent Lua function is created, it can be found in the box.space._func system space, and it can be shown with box.func.func-name and it can be invoked by any user with authorization to 'execute' it. The syntax for invoking is: box.func.func-name:call([parameters]) or, if the connection is remote, the syntax is as in net_box:call().

Example:

```
tarantool> lua_code = [[function(a, b) return a + b end]]
tarantool> box.schema.func.create('sum', {body = lua_code})

tarantool> box.func.sum
---
- is_sandboxed: false
  is_deterministic: false
  id: 2
  setuid: false
  body: function(a, b) return a + b end
  name: sum
  language: LUA
...

tarantool> box.func.sum.call({1, 2})
---
- 3
...
```

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

Drop a function tuple. For explanation of how Tarantool maintains function data, see reference on _func space.

Parameters

- func-name (string) – the name of the function
- options (table) – if_exists = true|false (default = false) - boolean; true means there should be no error if the _func tuple does not exist.

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.

Parameters

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

Rtype bool

Example:

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

box.schema.func.reload([name])

Reload a C module with all its functions without restarting the server.

Under the hood, Tarantool loads a new copy of the module (*.so shared library) and starts routing all new request to the new version. The previous version remains active until all started calls are finished. All shared libraries are loaded with RTLD_LOCAL (see "man 3 dlopen"), therefore multiple copies can co-exist without any problems.

Note: Reload will fail if a module was loaded from Lua script with ffi.load().

Parameters

- name (string) – the name of the module to reload

Example:

```
-- reload the entire module contents
box.schema.func.reload('module')
```

Sequences

An introduction to sequences is in the Sequences section of the "Data model" chapter. Here are the details for each function and option.

All functions related to sequences require appropriate privileges.

box.schema.sequence.create(name[, options])

Create a new sequence generator.

Parameters

- name (string) – the name of the sequence
- options (table) – see a quick overview in the "Options for box.schema.sequence.create()" chart (in the Sequences section of the "Data model" chapter), and see more details below.

Return a reference to a new sequence object.

Options:

- start – the STARTS WITH value. Type = integer, Default = 1.
- min – the MINIMUM value. Type = integer, Default = 1.
- max - the MAXIMUM value. Type = integer, Default = 9223372036854775807.

  There is a rule: min <= start <= max. For example it is illegal to say {start=0} because then the specified start value (0) would be less than the default min value (1).

  There is a rule: min <= next-value <= max. For example, if the next generated value would be 1000, but the maximum value is 999, then that would be considered "overflow".

There is a rule: start and min and max must all be <= 9223372036854775807 which is 2^63 - 1 (not 2^64).

- cycle – the CYCLE value. Type = bool. Default = false.

  If the sequence generator's next value is an overflow number, it causes an error return – unless cycle == true.

  But if cycle == true, the count is started again, at the MINIMUM value or at the MAXIMUM value (not the STARTS WITH value).

- cache – the CACHE value. Type = unsigned integer. Default = 0.

  Currently Tarantool ignores this value, it is reserved for future use.

- step – the INCREMENT BY value. Type = integer. Default = 1.

  Ordinarily this is what is added to the previous value.

sequence_object:next()

Generate the next value and return it.

The generation algorithm is simple:

- If this is the first time, then return the STARTS WITH value.
- If the previous value plus the INCREMENT value is less than the MINIMUM value or greater than the MAXIMUM value, that is "overflow", so either raise an error (if cycle = false) or return the MAXIMUM value (if cycle = true and step < 0) or return the MINIMUM value (if cycle = true and step > 0).

If there was no error, then save the returned result, it is now the "previous value".

For example, suppose sequence 'S' has:

- min == -6,
- max == -1,
- step == -3,
- start = -2,
- cycle = true,
- previous value = -2.

Then box.sequence.S:next() returns -5 because -2 + (-3) == -5.

Then box.sequence.S:next() again returns -1 because -5 + (-3) < -6, which is overflow, causing cycle, and max == -1.

This function requires a 'write' privilege on the sequence.

Note: This function should not be used in "cross-engine" transactions (transactions which use both the memtx and the vinyl storage engines).

To see what the previous value was, without changing it, you can select from the _sequence_data system space.

---

sequence_object:alter(options)

The alter() function can be used to change any of the sequence's options. Requirements and restrictions are the same as for box.schema.sequence.create().

sequence_object:reset()
: Set the sequence back to its original state. The effect is that a subsequent next() will return the start value. This function requires a 'write' privilege on the sequence.

sequence_object:set(new-previous-value)
: Set the "previous value" to new-previous-value. This function requires a 'write' privilege on the sequence.

sequence_object:drop()
: Drop an existing sequence.

  Example:

  Here is an example showing all sequence options and operations:

```
s = box.schema.sequence.create(
            'S2',
            {start=100,
            min=100,
            max=tonumber64('9223372036854775807'),
            cache=100000,
            cycle=false,
            step=100
            })
s:alter({step=6})
s:next()
s:reset()
s:set(150)
s:drop()
```

space_object:create_index(... [sequence='...' option] ...)
: You can use the sequence option when creating or altering a primary-key index. The sequence becomes associated with the index, so that the next insert() will put the next generated number into the primary-key field, if the field value would otherwise be nil.

  The syntax may be any of: sequence = sequence identifier or sequence = {id = sequence identifier } or sequence = {field = field number } or sequence = {id = sequence identifier , field = field number } or sequence = true or sequence = {}. The sequence identifier may be either a number (the sequence id) or a string (the sequence name). The field number may be the ordinal number of any field in the index; default = 1. Examples of all possibilities: sequence = 1 or sequence = 'sequence_name' or sequence = {id = 1} or sequence = {id = 'sequence_name'} or sequence = {id = 1, field = 1} or sequence = {id = 'sequence_name', field = 1} or sequence = {field = 1} or sequence = true or sequence = {}. Notice that the sequence identifier can be omitted, if it is omitted then a new sequence is created automatically with default name = space-name_seq. Notice that the field number does not have to be 1, that is, the sequence can be associated with any field in the primary-key index.

  For example, if 'Q' is a sequence and 'T' is a new space, then this will work:

```
tarantool> box.space.T:create_index('Q',{sequence='Q'})
---
- unique: true
  parts:
  - type: unsigned
    is_nullable: false
    fieldno: 1
  sequence_id: 8
  id: 0
  space_id: 514
  name: Q
```

(continues on next page)

(continued from previous page)

```
  type: TREE
...
```

(Notice that the index now has a sequence_id field.)

And this will work:

```
tarantool> box.space.T:insert{nil,0}
---
- [1, 0]
...
```

Note: The index key type may be either 'integer' or 'unsigned'. If any of the sequence options is a negative number, then the index key type should be 'integer'.

Users should not insert a value greater than 9223372036854775807, which is 2^63 - 1, in the indexed field. The sequence generator will ignore it.

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

If a sequence was created automatically because the sequence identifier was omitted, then it will be dropped automatically if the index is altered so that sequence=false, or if the index is dropped.

index_object:alter() can also be used to associate a sequence with an existing index, with the same syntax for options.

When a sequence is used with an index based on a JSON path, inserted tuples must have all components of the path preceding the autoincrement field, and the autoincrement field. To achieve that use box.NULL rather than nil. Example:

```
s = box.schema.space.create('test')
s:create_index('pk', {parts = {{'[1].a.b[1]', 'unsigned'}}, sequence = true})
s:replace{} -- error
s:replace{{c = {}}} -- error
s:replace{{a = {c = {}}}} -- error
s:replace{{a = {b = {}}}} -- error
s:replace{{a = {b = {nil}}}} -- error
s:replace{{a = {b = {box.NULL}}}} -- ok
```

Submodule box.session

Overview

The box.session submodule allows querying the session state, writing to a session-specific temporary Lua table, or sending out-of-band messages, or setting up triggers which will fire when a session starts or ends.

A session is an object associated with each client connection.

Index

Below is a list of all box.session functions and members.

Name	Use
box.session.id()	Get the current session's ID
box.session.exists()	Check if a session exists
box.session.peer()	Get the session peer's host address and port
box.session.sync()	Get the sync integer constant
box.session.user()	Get the current user's name
box.session.type()	Get the connection type or cause of action
box.session.su()	Change the current user
box.session.uid()	Get the current user's ID
box.session.euid()	Get the current effective user's ID
box.session.storage	Table with session-specific names and values
box.session.on_connect()	Define a connect trigger
box.session.on_disconnect()	Define a disconnect trigger
box.session.on_auth()	Define an authentication trigger
box.session.push()	Send an out-of-band message

box.session.id()

Return the unique identifier (ID) for the current session. The result can be 0 or -1 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 Tarantool instance.

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 instance, the return is null. The command is executed on the server instance, so the "local name" is the server instance'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. This value becomes invalid when the session is disconnected.

Rtype number

This function is local for the request, i.e. not global for the session. If the connection behind the session is multiplexed, this function can be safely used inside the request processor.

box.session.user()

Return the name of the current user

Rtype string

box.session.type()

Return the type of connection or cause of action.

Rtype string

Possible return values are:

- 'binary' if the connection was done via the binary protocol, for example to a target made with box.cfg{listen=...};
- 'console' if the connection was done via the administrative console, for example to a target made with console.listen;
- 'repl' if the connection was done directly, for example when using Tarantool as a client;
- 'applier' if the action is due to replication, regardless of how the connection was done;
- 'background' if the action is in a background fiber, regardless of whether the Tarantool server was started in the background.

box.session.type() is useful for an on_replace() trigger on a replica – the value will be 'applier' if and only if the trigger was activated because of a request that was done on the master.

box.session.su(user-name[, function-to-execute])

Change Tarantool's current user – this is analogous to the Unix command su.

Or, if function-to-execute is specified, change Tarantool's current user temporarily while executing the function – this is analogous to the Unix command sudo.

Parameters

- user-name (string) – name of a target user
- function-to-execute – name of a function, or definition of a function. Additional parameters may be passed to box.session.su, they will be interpreted as parameters of function-to-execute.

Example

```
tarantool> function f(a) return box.session.user() .. a end
---
...

tarantool> box.session.su('guest', f, '-xxx')
---
- guest-xxx
...

tarantool> box.session.su('guest',function(...) return ... end,1,2)
---
- 1
- 2
...
```

box.session.uid()

Return the user ID of the current user.

Rtype number

Every user has a unique name (seen with box.session.user()) and a unique ID (seen with box.session.uid()). The values are stored together in the _user space.

box.session.euid()

Return the effective user ID of the current user.

This is the same as box.session.uid(), except in two cases:

- The first case: if the call to box.session.euid() is within a function invoked by box.session.su(user-name, function-to-execute) – in that case, box.session.euid() returns the ID of the changed user (the user who is specified by the user-name parameter of the su function) but box.session.uid() returns the ID of the original user (the user who is calling the su function).
- The second case: if the call to box.session.euid() is within a function specified with box.schema.func.create(function-name, {setuid= true}) and the binary protocol is in use – in that case, box.session.euid() returns the ID of the user who created "function-name" but box.session.uid() returns the ID of the the user who is calling "function-name".

  Rtype number

Example

```
tarantool> box.session.su('admin')
---
...
tarantool> box.session.uid(), box.session.euid()
---
- 1
- 1
...
tarantool> function f() return {box.session.uid(),box.session.euid()} end
---
...
tarantool> box.session.su('guest', f)
---
- - 1
  - 0
...
```

box.session.storage

A Lua table that can hold arbitrary unordered session-specific names and values, which will last until the session ends. For example, this table could be useful to store current tasks when working with a Tarantool queue manager.

Example

```
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
```

(continues on next page)

(continued from previous page)

```
---
- 'radius_of_mars=3396 random_memorandum=Don''t forget the eggs. '
...
```

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

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 execution fails and raises an error, the error is sent to the client and the connection is closed.

Parameters

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

Return nil or function pointer

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

If both parameters are omitted, then the response is a list of existing trigger functions.

Details about trigger characteristics are in the triggers section.

Example

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

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

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

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().

Since version 1.10, the trigger function is invoked immediately after the disconnect, even if requests that were made during the session have not finished.

Parameters

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

Return nil or function pointer

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

If both parameters are omitted, then the response is a list of existing trigger functions.

Details about trigger characteristics are in the triggers section.

Example #1

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

Example #2

After the following series of requests, a Tarantool instance 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
```

box.session.on_auth([trigger-function[, old-trigger-function]])

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.

Parameters

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

Return nil or function pointer

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

If both parameters are omitted, then the response is a list of existing trigger functions.

Details about trigger characteristics are in the triggers section.

Example 1

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

Example 2

This is a more complex example, with two server instances.

The first server instance listens on port 3301; its default user name is 'admin'. There are three on_auth triggers:

- The first trigger has a function with no arguments, it can only look at box.session.user().
- The second trigger has a function with a user_name argument, it can look at both of: box.session.user() and user_name.
- The third trigger has a function with a user_name argument and a status argument, it can look at all three of: box.session.user() and user_name and status.

The second server instance will connect with console.connect, and then will cause a display of the variables that were set by the trigger functions.

```
-- On the first server instance, which listens on port 3301
box.cfg{listen=3301}
function function1()
  print('function 1, box.session.user()='..box.session.user())
  end
function function2(user_name)
  print('function 2, box.session.user()='..box.session.user())
  print('function 2, user_name='..user_name)
  end
function function3(user_name, status)
  print('function 3, box.session.user()='..box.session.user())
  print('function 3, user_name='..user_name)
  if status == true then
    print('function 3, status = true, authorization succeeded')
    end
  end
box.session.on_auth(function1)
box.session.on_auth(function2)
box.session.on_auth(function3)
box.schema.user.passwd('admin')
```

```
-- On the second server instance, that connects to port 3301
console = require('console')
console.connect('admin:admin@localhost:3301')
```

The result looks like this:

```
function 3, box.session.user()=guest
function 3, user_name=admin
function 3, status = true, authorization succeeded
function 2, box.session.user()=guest
```

(continues on next page)

(continued from previous page)

```
function 2, user_name=admin
function 1, box.session.user()=guest
```

box.session.push(message[, sync])

Generate an out-of-band message. By "out-of-band" we mean an extra message which supplements what is passed in a network via the usual channels. Although box.session.push() can be called at any time, in practice it is used with networks that are set up with module net.box, and it is invoked by the server (on the "remote database system" to use our terminology for net.box), and the client has options for getting such messages.

This function returns an error if the session is disconnected.

Parameters

- message (any-Lua-type) – what to send
- sync (int) – an optional argument to indicate what the session is, as taken from an earlier call to box_session:sync(). If it is omitted, the default is the current box.session.sync() value.

Rtype {nil, error} or true:

- If the result is an error, then the first part of the return is nil and the second part is the error object.
- If the result is not an error, then the return is the boolean value true.
- When the return is true, the message has gone to the network buffer as a packet with the code IPROTO_CHUNK (0x80).

The server's sole job is to call box.session.push(), there is no automatic mechanism for showing that the message was received.

The client's job is to check for such messages after it sends something to the server. The major client methods – conn:call, conn:eval, conn:select, conn:insert, conn:replace, conn:update, conn:upsert, delete – may cause the server to send a message.

Situation 1: when the client calls synchronously with the default {async=false} option. There are two optional additional options: on_push=function-name, and on_push_ctx=function-argument. When the client receives an out-of-band message for the session, it invokes "function-name(function-argument)". For example, with options {on_push=table.insert, on_push_ctx=messages}, the client will insert whatever it receives into a table named 'messages'.

Situation 2: when the client calls asynchronously with the non-default {async=true} option. Here on_push and on_push_ctx are not allowed, but the messages can be seen by calling pairs() in a loop.

Situation 2 complication: pairs() is subject to timeout. So there is an optional argument = timeout per iteration. If timeout occurs before there is a new message or a final response, there is an error return. To check for an error one can use the first loop parameter (if the loop starts with "for i, message in future:pairs()" then the first loop parameter is i). If it is box.NULL then the second parameter (in our example, "message") is the error object.

Example

```
-- Make two shells. On Shell#1 set up a "server", and
-- in it have a function that includes box.session.push:
box.cfg{listen=3301}
box.schema.user.grant('guest','read,write,execute','universe')
x = 0
fiber = require('fiber')
```

(continues on next page)

(continued from previous page)

```
function server_function() x=x+1; fiber.sleep(1); box.session.push(x); end

-- On Shell#2 connect to this server as a "client" that
-- can handle Lua (such as another Tarantool server operating
-- as a client), and initialize a table where we'll get messages:
net_box = require('net.box')
conn = net_box.connect(3301)
messages_from_server = {}

-- On Shell#2 remotely call the server function and receive
-- a SYNCHRONOUS out-of-band message:
conn:call('server_function', {},
          {is_async = false,
           on_push = table.insert,
           on_push_ctx = messages_from_server})
messages_from_server
-- After a 1-second pause that is caused by the fiber.sleep()
-- request inside server_function, the result in the
--  messages_from_server table will be: 1. Like this:
-- tarantool> messages_from_server
-- ---
-- - - 1
-- ...
-- Good. That shows that box.session.push(x) worked,
-- because we know that x was 1.

-- On Shell#2 remotely call the same server function and
-- get an ASYNCHRONOUS out-of-band message. For this we cannot
-- use on_push and on_push_ctx options, but we can use pairs():
future = conn:call('server_function', {}, {is_async = true})
messages = {}
keys = {}
for i, message in future:pairs() do
    table.insert(messages, message) table.insert(keys, i) end
messages
future:wait_result(1000)
for i, message in future:pairs() do
    table.insert(messages, message) table.insert(keys, i) end
messages
-- There is no pause because conn:call does not wait for
-- server_function to finish. The first time that we go through
-- the pairs() loop, we see the messages table is empty. Like this:
-- tarantool> messages
-- ---
-- - - 2
--   - []
-- ...
-- That is okay because the server hasn't yet called
-- box.session.push(). The second time that we go through
-- the pairs() loop, we see the value of x at the time of
-- the second call to box.session.push(). Like this:
-- tarantool> messages
-- ---
-- - - 2
--   - &0 []
--   - 2
```

(continues on next page)

(continued from previous page)

```
--   - *0
-- ...
-- Good. That shows that the message was asynchronous, and
-- that box.session.push() did its job.
```

Submodule box.slab

Overview

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 usage and memory fragmentation.

Index

Below is a list of all box.slab functions.

Name	Use
box.runtime.info()	Show a memory usage report for Lua runtime
box.slab.info()	Show an aggregated memory usage report for slab allocator
box.slab.stats()	Show a detailed memory usage report for slab allocator

box.runtime.info()

Show a memory usage report (in bytes) for the Lua runtime.

Return

- lua is the heap size of the Lua garbage collector;
- maxalloc is the maximal memory quota that can be allocated for Lua;
- used is the current memory size used by Lua.

Rtype table

Example:

```
tarantool> box.runtime.info()
---
- lua: 913710
  maxalloc: 4398046510080
  used: 12582912
...
tarantool> box.runtime.info().used
---
- used: 12582912
...
```

box.slab.info()

Show an aggregated memory usage report (in bytes) for the slab allocator.

This report is useful for assessing out-of-memory risks: the risks are high if both arena_used_ratio and quota_used_ratio are high (90-95%).

If quota_used_ratio is low, then high arena_used_ratio and/or items_used_ratio indicate that the memory fragmentation is low (i.e. the memory is used efficiently).

If quota_used_ratio is high (approaching 100%), then low arena_used_ratio (50-60%) indicates that the memory is heavily fragmentized. Most probably, there is no immediate out-of-memory risk in this case, but generally this is an issue to consider. For example, probable risks are that the entire memory quota is used for tuples, and there is are no slabs left for a piece of an index. Or that all slabs are allocated for storing tuples, but in fact all the slabs are half-empty.

Return

- items_size is the total amount of memory (including allocated, but currently free slabs) used only for tuples, no indexes;
- items_used_ratio = items_used / slab_count * slab_size (these are slabs used only for tuples, no indexes);
- quota_size is the maximum amount of memory that the slab allocator can use for both tuples and indexes (as configured in the memtx_memory parameter, the default is 2^28 bytes = 268,435,456 bytes);
- quota_used_ratio = quota_used / quota_size;
- arena_used_ratio = arena_used / arena_size;
- items_used is the efficient amount of memory (omitting allocated, but currently free slabs) used only for tuples, no indexes;
- quota_used is the amount of memory that is already distributed to the slab allocator;
- arena_size is the total memory used for tuples and indexes together (including allocated, but currently free slabs);
- arena_used is the efficient memory used for storing tuples and indexes together (omitting allocated, but currently free slabs).

Rtype table

Example:

```
tarantool> box.slab.info()
---
- items_size: 228128
  items_used_ratio: 1.8%
  quota_size: 1073741824
  quota_used_ratio: 0.8%
  arena_used_ratio: 43.2%
  items_used: 4208
  quota_used: 8388608
  arena_size: 2325176
  arena_used: 1003632
...

tarantool> box.slab.info().arena_used
---
- 1003632
...
```

box.slab.stats()

Show a detailed memory usage report (in bytes) for the slab allocator. The report is broken down into groups by data item size as well as by slab size (64-byte, 136-byte, etc). The report includes the memory allocated for storing both tuples and indexes.

return

- mem_free is the allocated, but currently unused memory;
- mem_used is the memory used for storing data items (tuples and indexes);
- item_count is the number of stored items;
- item_size is the size of each data item;
- slab_count is the number of slabs allocated;
- slab_size is the size of each allocated slab.

rtype table

Example:

Here is a sample report for the first group:

```
tarantool> box.slab.stats()[1]
---
- mem_free: 16232
  mem_used: 48
  item_count: 2
  item_size: 24
  slab_count: 1
  slab_size: 16384
...
```

This report is saying that there are 2 data items (item_count = 2) stored in one (slab_count = 1) 24-byte slab (item_size = 24), so mem_used = 2 * 24 = 48 bytes. Also, slab_size is 16384 bytes, of which 16384 - 48 = 16232 bytes are free (mem_free).

A complete report would show memory usage statistics for all groups:

```
tarantool> box.slab.stats()
---
- - mem_free: 16232
    mem_used: 48
    item_count: 2
    item_size: 24
    slab_count: 1
    slab_size: 16384
  - mem_free: 15720
    mem_used: 560
    item_count: 14
    item_size: 40
    slab_count: 1
    slab_size: 16384
  <...>
  - mem_free: 32472
    mem_used: 192
    item_count: 1
    item_size: 192
    slab_count: 1
    slab_size: 32768
  - mem_free: 1097624
    mem_used: 999424
    item_count: 61
    item_size: 16384
    slab_count: 1
```

(continues on next page)

(continued from previous page)

```
  slab_size: 2097152
...
```

The total mem_used for all groups in this report equals arena_used in box.slab.info() report.

Submodule box.space

Overview

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/lua/schema.lua.

Index

Below is a list of all box.space functions and members.

Name	Use
space_object:auto_increment()	Generate key + Insert a tuple
space_object:bsize()	Get count of bytes
space_object:count()	Get count of tuples
space_object:create_index()	Create an index
space_object:delete()	Delete a tuple
space_object:drop()	Destroy a space
space_object:format()	Declare field names and types
space_object:frommap()	Convert from map to tuple or table
space_object:get()	Select a tuple
space_object:insert()	Insert a tuple
space_object:len()	Get count of tuples
space_object:on_replace()	Create a replace trigger with a function that cannot change the tuple
space_object:before_replace()	Create a replace trigger with a function that can change the tuple
space_object:pairs()	Prepare for iterating
space_object:put()	Insert or replace a tuple
space_object:rename()	Rename a space
space_object:replace()	Insert or replace a tuple
space_object:run_triggers()	Enable/disable a replace trigger
space_object:select()	Select one or more tuples
space_object:truncate()	Delete all tuples
space_object:update()	Update a tuple
space_object:upsert()	Update a tuple
space_object:user_defined()	Any function / method that any user wants to add
space_object:create_check_constraint()	Create a check constraint
space_object.enabled	Flag, true if space is enabled
space_object.field_count	Required number of fields
space_object.id	Numeric identifier of space
space_object.index	Container of space's indexes
box.space._cluster	(Metadata) List of replica sets
box.space._func	(Metadata) List of function tuples

Continued on next page

Table 1 – continued from previous page

Name	Use
box.space._index	(Metadata) List of indexes
box.space._vindex	(Metadata) List of indexes accessible for the current user
box.space._priv	(Metadata) List of privileges
box.space._vpriv	(Metadata) List of privileges accessible for the current user
box.space._schema	(Metadata) List of schemas
box.space._sequence	(Metadata) List of sequences
box.space._sequence_data	(Metadata) List of sequences
box.space._space	(Metadata) List of spaces
box.space._vspace	(Metadata) List of spaces accessible for the current user
box.space._user	(Metadata) List of users
box.space._ck_constraint	(Metadata) List of check constraints
box.space._vuser	(Metadata) List of users accessible for the current user
box.space._collation	(Metadata) List of collations
box.space._vcollation	(Metadata) List of collations accessible for the current user

object space_object

space_object:auto_increment(tuple)

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

Since version 1.7.5 this method is deprecated – it is better to use a sequence.

Parameters

- space_object (space_object) – an object reference
- tuple (table/tuple) – 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;
- primary-key indexed field is not a number.

Example:

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

space_object:bsize()

Parameters

- space_object (space_object) – an object reference

Return Number of bytes in the space. This number, which is stored in Tarantool's internal memory, represents the total number of bytes in all tuples, not including index keys. For a measure of index size, see index_object:bsize().

Example:

```
tarantool> box.space.tester:bsize()
---
- 22
...
```

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

Return the number of tuples. If compared with len(), this method works slower because count() scans the entire space to count the tuples.

Parameters

- space_object (space_object) – an object reference
- key (scalar/table) – primary-key field values, must be passed as a Lua table if key is multi-part
- iterator – comparison method

Return Number of tuples.

Example:

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

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

Create an index. It is mandatory to create an index for a space 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 (space_object) – an object reference
- index_name (string) – name of index, which should conform to the rules for object names
- options (table) – see "Options for space_object:create_index()", below

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') Note re storage engine: vinyl only supports 'TREE'	'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 'boolean' or 'varbinary' or 'array' or 'scalar', and optional collation or is_nullable value or path}	{1, 'unsigned'}
dimension	affects RTREE only	number	2
distance	affects RTREE only	string ('euclid' or 'manhattan')	'euclid'
bloom_fpr	affects vinyl only	number	vinyl_bloom_fpr
page_size	affects vinyl only	number	vinyl_page_size
range_size	affects vinyl only	number	vinyl_range_size
run_count_per_level	affects vinyl only	number	vinyl_run_count_per_lev
run_size_ratio	affects vinyl only	number	vinyl_run_size_ratio
sequence	see section regarding specifying a sequence in create_index()	string or number	not present
func	functional index	string	not present

The options in the above chart are also applicable for index_object:alter().

Note re storage engine: vinyl has extra options which by default are based on configuration parameters vinyl_bloom_fpr, vinyl_page_size, vinyl_range_size, vinyl_run_count_per_level, and vinyl_run_size_ratio – see the description of those parameters. The current values can be seen by selecting from box.space._index.

Building or rebuilding a large index will cause occasional yields so that other requests will not be blocked. If the other requests cause an illegal situation such as a duplicate key in a unique index, the index building or rebuilding will fail.

Possible errors:

- too many parts;
- index '. . . ' already exists;
- primary key must be unique.

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

Details about index field types:

The eight index field types (unsigned | string | integer | number | boolean | varbinary | 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. A string may have a collation.
- 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.
- boolean: true or false. Legal in memtx TREE or HASH indexes, and in vinyl TREE indexes.
- varbinary: any set of octets, up to the maximum length. Legal in memtx TREE or HASH indexes, and in vinyl TREE indexes. A varbinary byte sequence does not have a collation because its contents are not UTF-8 characters.
- array: array of numbers. Legal in memtx RTREE indexes.
- scalar: null (input with msgpack.NULL or yaml.NULL or json.NULL), booleans (true or false), or integers between -9223372036854775808 and 18446744073709551615, or single-precision floating point numbers, or double-precison floating-point numbers, or strings. When there is a mix of types, the key order is: null, then booleans, then numbers, then strings. Legal in memtx TREE or HASH indexes, and in vinyl TREE indexes.

Additionally, nil is allowed with any index field type if is_nullable=true is specified.

Index field types to use in space_object:create_index()

Index field type	What can be in it	Where is it legal	Examples
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'
varbinary	byte sequences – any set of octets	memtx TREE or HASH indexes vinyl TREE indexes	'\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
boolean	true or false	memtx TREE or HASH indexes, vinyl TREE indexes	false true
array	array of integers between -9223372036854775808 and 9223372036854775807	memtx RTREE indexes	{10, 11} {3, 5, 9, 10}
scalar	null, booleans (true or false), integers between -9223372036854775808 and 18446744073709551615, single-precision floating point numbers, double-precision floating point numbers, strings	memtx TREE or HASH indexes, vinyl TREE indexes	null true -1 1.234 '' 'py'

Allowing null for an indexed key: If the index type is TREE, and the index is not the primary index, then the parts={...} clause may include is_nullable=true or is_nullable=false (the default). If is_nullable is true, then it is legal to insert nil or an equivalent such as msgpack.NULL (or it is legal to insert nothing at all for trailing nullable fields). Within indexes, such "null values" are always treated as equal to other null values, and are always treated as less than non-null values. Nulls may appear multiple times even in a unique index. Example:

```
box.space.tester:create_index('I',{unique=true,parts={{2,'number',is_nullable=true}}})
```

Warning: It is legal to create multiple indexes for the same field with different is_nullable values, or to call space_object:format() with a different is_nullable value from what is used for an index. When there is a contradiction, the rule is: null is illegal unless is_nullable=true for every index and for the space format.

Using field names instead of field numbers: create_index() can use field names and/or field types described by the optional space_object:format() clause. In the following example, we show format() for a space that has two columns named 'x' and 'y', and then we show five variations of the parts={} clause of create_index(), first for the 'x' column, second for both the 'x' and 'y' columns. The variations include omitting the type, using numbers, and adding extra braces.

```
box.space.tester:format({{name='x', type='scalar'}, {name='y', type='integer'}})
box.space.tester:create_index('I2',{parts={{'x','scalar'}}})
box.space.tester:create_index('I3',{parts={{'x','scalar'},{'y','integer'}}})
box.space.tester:create_index('I4',{parts={1,'scalar'}})
box.space.tester:create_index('I5',{parts={1,'scalar',2,'integer'}})
box.space.tester:create_index('I6',{parts={1}})
box.space.tester:create_index('I7',{parts={1,2}})
box.space.tester:create_index('I8',{parts={'x'}})
box.space.tester:create_index('I9',{parts={'x','y'}})
box.space.tester:create_index('I10',{parts={{'x'}}})
box.space.tester:create_index('I11',{parts={{'x'},{'y'}}})
```

Using the path option for map fields: To create an index for a field that is a map (a path string and a scalar value), specify the path string during index_create, that is, parts={ field-number,'data-type', path = 'path-name' }. The index type must be 'tree' or 'hash' and the field's contents must always be maps with the same path.

```
-- Example 1 -- The simplest use of path:
-- Result will be - - [{'age': 44}]
box.schema.space.create('T')
box.space.T:create_index('I',{parts={{1, 'scalar', path='age'}}})
box.space.T:insert{{age=44}}
box.space.T:select(44)
-- Example 2 -- path plus format() plus JSON syntax to add clarity
-- Result will be: - [1, {'FIO': {'surname': 'Xi', 'firstname': 'Ahmed'}}]
s = box.schema.space.create('T')
format = {{'id', 'unsigned'}, {'data', 'map'}}
s:format(format)
parts = {{'data.FIO["firstname"]', 'str'}, {'data.FIO["surname"]', 'str'}}
i = s:create_index('info', {parts = parts})
s:insert({1, {FIO={firstname='Ahmed', surname='Xi'}}})
```

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

Using the path option with [*] The string in a path option can contain '[*]' which is called an array index placeholder. Indexes defined with this are useful for JSON documents that all have the same structure. For example, when creating an index on field#2 for a string document that will start with {'data': [{'name': '...'}, {'name': '...'}], the parts section in the create_index request could look like: parts = {{2, 'str', path = 'data[*].name'}}. Then tuples containing names can be retrieved quickly with index_object:select({key-value}). In fact a single field can have multiple keys, as in this example which retrieves the same tuple twice because there are two keys 'A' and 'B' which both match the request:

```
s = box.schema.space.create('json_documents')
s:create_index('primarykey')
i = s:create_index('multikey', {parts = {{2, 'str', path = 'data[*].name'}}})
s:insert({1,
        {data = {{name='A'},
                 {name='B'}},
         extra_field = 1}})
```

(continues on next page)

(continued from previous page)

```
i:select({''},{iterator='GE'})
--  The result of the select request looks like this:
--  tarantool> i:select({''},{iterator='GE'})
--  ---
--  - - [1, {'data': [{'name': 'A'}, {'name': 'B'}], 'extra_field': 1}]
--    - [1, {'data': [{'name': 'A'}, {'name': 'B'}], 'extra_field': 1}]
--  ...
```

Some restrictions exist: () '[*]' must be alone or must be at the end of a name in the path; () '[*]' must not appear twice in the path; () if an index has a path with x[*] then no other index can have a path with x.component; () '[*]' must not appear in the path of a primary-key ; () if an index has unique=true and has a path with '[*]' then duplicate keys from different tuples are disallowed but duplicate keys for the same tuple are allowed; () As with Using the path option for map fields, the field's value must have the structure that the path definition implies, or be nil (nil is not indexed).

Making a functional index with space_object:create_index()

Functional indexes are indexes that call a user-defined function for forming the index key, rather than depending entirely on the Tarantool default formation. Functional indexes are useful for condensing or truncating or reversing or any other way that users want to customize the index.

The function definition must expect a tuple (which has the contents of fields at the time a data-change request happens) and must return a tuple (which has the contents that will actually be put in the index).

The space must have a memtx engine. The function must be persistent and deterministic. The key parts must not depend on JSON paths. The create_index definition must include specification of all key parts, and the function must return a table which has the same number of key parts with the same types. The function must access key-part values by index, not by field name. Functional indexes must not be primary-key indexes. Functional indexes cannot be altered and the function cannot be changed if it is used for an index, so the only way to change them is to drop the index and create it again.

Example:

A function could make a key using only the first letter of a string field.

```
-- Step 1: Make the space.
-- The space needs a primary-key field, which is not the field that we
-- will use for the functional index.
box.schema.space.create('x', {engine = 'memtx'})
box.space.x:create_index('i',{parts={1, 'string'}})
-- Step 2: Make the function.
-- The function expects a tuple. In this example it will work on tuple[2]
-- because the key souce is field number 2 in what we will insert.
-- Use string.sub() from the string module to get the first character.
lua_code = [[function(tuple) return {string.sub(tuple[2],1,1)} end]]
-- Step 3: Make the function persistent.
-- Use the box.schema.func.create function for this.
box.schema.func.create('F',
    {body = lua_code, is_deterministic = true, is_sandboxed = true})
-- Step 4: Make the functional index.
-- Specify the fields whose values will be passed to the function.
-- Specify the function.
box.space.x:create_index('j',{parts={1, 'string'},func = 'F'})
-- Step 5: Test.
-- Insert a few tuples.
-- Select using only the first letter, it will work because that is the key
```

(continues on next page)

(continued from previous page)

```
-- Or, select using the same function as was used for insertion
box.space.x:insert{'a', 'wombat'}
box.space.x:insert{'b', 'rabbit'}
box.space.x.index.j:select('w')
box.space.x.index.j:select(box.func.F:call({{'x', 'wombat'}}));
```

The results of the two select requests will look like this:

```
tarantool>     box.space.x.index.j:select('w')
---
- - ['a', 'wombat']
...

tarantool>     box.space.x.index.j:select(box.func.F:call({{'x','wombat'}}));
---
- - ['a', 'wombat']
...
```

Functions for functional indexes can return multiple keys. Such functions are called "multikey" functions. The box.func.create options must include opts = {is_multikey = true}. The return value must be a table of tuples. If a multikey function returns N tuples, then N keys will be added to the index.

Example:

```
s = box.schema.space.create('withdata')
s:format({{name = 'name', type = 'string'},
        {name = 'address', type = 'string'}})
pk = s:create_index('name', {parts = {1, 'string'}})
lua_code = [[function(tuple)
             local address = string.split(tuple[2])
             local ret = {}
             for _, v in pairs(address) do
               table.insert(ret, {utf8.upper(v)})
             end
             return ret
           end]]
box.schema.func.create('address',
                       {body = lua_code,
                        is_deterministic = true,
                        is_sandboxed = true,
                        opts = {is_multikey = true}})
idx = s:create_index('addr', {unique = false,
                              func = 'address',
                              parts = {{1, 'string',
                                       collation = 'unicode_ci'}}})
s:insert({"James", "SIS Building Lambeth London UK"})
s:insert({"Sherlock", "221B Baker St Marylebone London NW1 6XE UK"})
idx:select('Uk')
-- Both tuples will be returned.
```

space_object:delete(key)

Delete a tuple identified by a primary key.

Parameters

- space_object (space_object) – an object reference
- key (scalar/table) – primary-key field values, must be passed as a Lua table if key

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

For more usage scenarios and typical errors see Example: using data operations further in this section.

space_object:drop()

Drop a space.

Parameters

- space_object (space_object) – an object reference

Return nil

Possible errors: 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:format([format-clause])

Declare field names and types.

Parameters

- space_object (space_object) – an object reference
- format-clause (table) – a list of field names and types

Return nil, unless format-clause is omitted

Possible errors:

- space_object does not exist;
- field names are duplicated;
- type is not legal.

Ordinarily Tarantool allows unnamed untyped fields. But with format users can, for example, document that the Nth field is the surname field and must contain strings. It is also possible to specify a format clause in box.schema.space.create().

The format clause contains, for each field, a definition within braces: {name='...',type='...'[, is_nullable=...]}, where:

- the name value may be any string, provided that two fields do not have the same name;
- the type value may be any of those allowed for indexed fields: unsigned | string | varbinary | integer | number | boolean | array | scalar (the same as the requirement in "Options for space_object:create_index");
- the optional is_nullable value may be either true or false (the same as the requirement in "Options for space_object:create_index"). See also the warning notice in section Allowing null for an indexed key.

It is not legal for tuples to contain values that have the wrong type; for example after box.space.tester:format({{' ',type='number'}}) the request box.space.tester:insert{'string-which-is-not-a-number'} will cause an error.

It is not legal for tuples to contain null values if is_nullable=false, which is the default; for example after box.space.tester:format({{' ',type='number',is_nullable=false}}) the request box.space.tester:insert{nil,2} will cause an error.

It is legal for tuples to have more fields than are described by a format clause. The way to constrain the number of fields is to specify a space's field_count member.

It is legal for tuples to have fewer fields than are described by a format clause, if the omitted trailing fields are described with is_nullable=true; for example after box.space.tester:format({{'a',type='number'},{'b',type='number',is_nullable=true}}) the request box.space.tester:insert{2} will not cause a format-related error.

It is legal to use format on a space that already has a format, thus replacing any previous definitions, provided that there is no conflict with existing data or index definitions.

It is legal to use format to change the is_nullable flag; for example after box.space.tester:format({{' ',type='scalar',is_nullable=false}}) the request box.space.tester:format({{' ',type='scalar',is_nullable=true}}) will not cause an error – and will not cause rebuilding of the space. But going the other way and changing is_nullable from true to false might cause rebuilding and might cause an error if there are existing tuples with nulls.

Example:

```
box.space.tester:format({{name='surname',type='string'},{name='IDX',type='array'}})
box.space.tester:format({{name='surname',type='string',is_nullable=true}})
```

There are legal variations of the format clause:

- omitting both 'name=' and 'type=',
- omitting 'type=' alone, and
- adding extra braces.

The following examples show all the variations, first for one field named 'x', second for two fields named 'x' and 'y'.

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

(continues on next page)

(continued from previous page)

```
box.space.tester:format({{name='x',type='scalar'},{name='y',type='unsigned'}})
box.space.tester:format({{name='x'}})
box.space.tester:format({{name='x'},{name='y'}})
box.space.tester:format({{'x',type='scalar'}})
box.space.tester:format({{'x',type='scalar'},{'y',type='unsigned'}})
box.space.tester:format({{'x','scalar'}})
box.space.tester:format({{'x','scalar'},{'y','unsigned'}})
```

The following example shows how to create a space, format it with all possible types, and insert into it.

```
tarantool> box.schema.space.create('t')
--- ...
tarantool> box.space.t:format({{name='1',type='any'},
         >                     {name='2',type='unsigned'},
         >                     {name='3',type='string'},
         >                     {name='4',type='number'},
         >                     {name='5',type='integer'},
         >                     {name='6',type='boolean'},
         >                     {name='7',type='scalar'},
         >                     {name='8',type='array'},
         >                     {name='9',type='map'}})
--- ...
tarantool> box.space.t:create_index('i',{parts={2,'unsigned'}})
--- ...
tarantool> box.space.t:insert{{'a'},      -- any
         >                    1,          -- unsigned
         >                    'W?',       -- string
         >                    5.5,        -- number
         >                    -0,         -- integer
         >                    true,       -- boolean
         >                    true,       -- scalar
         >                    {{'a'}},    -- array
         >                    {val=1}}    -- map
---
- [['a'], 1, 'W?', 5.5, 0, true, true, [['a']], {'val': 1}]
...
```

Names specified with the format clause can be used in space_object:get() and in space_object:create_index() and in tuple_object[field-name] and in tuple_object[field-path].

If the format clause is omitted, then the returned value is the table that was used in a previous space_object:format(format-clause) invocation. For example, after box.space.tester:format({{'x','scalar'}}), box.space.tester:format() will return [{'name': 'x', 'type': 'scalar'}].

Formatting or reformatting a large space will cause occasional yields so that other requests will not be blocked. If the other requests cause an illegal situation such as a field value of the wrong type, the formatting or reformatting will fail.

space_object:frommap(map[, option])

Convert a map to a tuple instance or to a table. The map must consist of "field name = value" pairs. The field names and the value types must match names and types stated previously for the space, via space_object:format().

Parameters

- space_object (space_object) – an object reference

- map (field-value-pairs) – a series of “field = value” pairs, in any order.
- option (boolean) – the only legal option is {table = true|false}; if the option is omitted or if {table = false}, then return type will be ‘cdata’ (i.e. tuple); if {table = true}, then return type will be ‘table’.

Return a tuple instance or table.

Rtype tuple or table

Possible errors: space_object does not exist or has no format; “unknown field”.

Example:

```
-- Create a format with two fields named 'a' and 'b'.
-- Create a space with that format.
-- Create a tuple based on a map consistent with that space.
-- Create a table based on a map consistent with that space.
tarantool> format1 = {{name='a',type='unsigned'},{name='b',type='scalar'}}
---
...
tarantool> s = box.schema.create_space('test', {format = format1})
---
...
tarantool> s:frommap({b = 'x', a = 123456})
---
- [123456, 'x']
...
tarantool> s:frommap({b = 'x', a = 123456}, {table = true})
---
- - 123456
  - x
...
```

space_object:get(key)

Search for a tuple in the given space.

Parameters

- space_object (space_object) – an object reference
- key (scalar/table) – value to be matched against the index key, which may be multi-part.

Return the tuple whose index key matches key, or nil.

Rtype tuple

Possible errors: 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 space by appending [1]. Therefore box.space.tester:get{1} has the same effect as box.space.tester:select{1}[1], if exactly one tuple is found.

Example:

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

Using field names instead of field numbers: get() can use field names described by the optional space_object:format() clause. This is similar to a standard Lua feature, where a component can

be referenced by its name instead of its number. For example, we can format the tester space with a field named x and use the name x in the index definition:

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

Then, if get or select retrieve a single tuple, we can reference the field 'x' in the tuple by its name:

```
box.space.tester:get{1}['x']
box.space.tester:select{1}[1]['x']
```

space_object:insert(tuple)

Insert a tuple into a space.

Parameters

- space_object (space_object) – an object reference
- tuple (tuple/table) – tuple to be inserted.

Return the inserted tuple

Rtype tuple

Possible errors: ER_TUPLE_FOUND if a tuple with the same unique-key value already exists.

Example:

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

For more usage scenarios and typical errors see Example: using data operations further in this section.

space_object:len()

Return the number of tuples in the space. If compared with count(), this method works faster because len() does not scan the entire space to count the tuples.

Parameters

- space_object (space_object) – an object reference

Return Number of tuples in the space.

Example:

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

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

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

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

Parameters

- trigger-function (function) – function which will become the trigger function

- old-trigger-function (function) – existing trigger function which will be replaced by trigger-function

Return nil or function pointer

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

If both parameters are omitted, then the response is a list of existing trigger functions.

If it is necessary to know whether the trigger activation happened due to replication or on a specific connection type, the function can refer to box.session.type().

Details about trigger characteristics are in the triggers section.

See also space_object:before_replace().

Example #1:

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

The trigger-function can have up to four parameters: (tuple) old value which has the contents before the request started, (tuple) new value which has the contents after the request ended, (string) space name, (string) type of request which is 'INSERT' or 'DELETE' or 'UPDATE' or 'REPLACE'. For example, the following code causes nil and 'INSERT' to be printed when the insert request is processed, and causes [1, 'Hi'] and 'DELETE' to be printed when the delete request is processed:

```
box.schema.space.create('space_1')
box.space.space_1:create_index('space_1_index',{})
function on_replace_function (old, new, s, op) print(old) print(op) end
box.space.space_1:on_replace(on_replace_function)
box.space.space_1:insert{1,'Hi'}
box.space.space_1:delete{1}
```

Example #2:

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
```

space_object:before_replace([trigger-function[, old-trigger-function]])

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

Parameters

- trigger-function (function) – function which will become the trigger function; for the trigger function's optional parameters see the description of on_replace.
- old-trigger-function (function) – existing trigger function which will be replaced by trigger-function

Return nil or function pointer

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

If both parameters are omitted, then the response is a list of existing trigger functions.

If it is necessary to know whether the trigger activation happened due to replication or on a specific connection type, the function can refer to box.session.type().

Details about trigger characteristics are in the triggers section.

See also space_object:on_replace().

Administrators can make replace triggers with on_replace(), or make triggers with before_replace(). If they make both types, then all before_replace triggers are executed before all on_replace triggers. The functions for both on_replace and before_replace triggers can make changes to the database, but only the functions for before_replace triggers can change the tuple that is being replaced.

Since a before_replace trigger function has the extra capability of making a change to the old tuple, it also can have extra overhead, to fetch the old tuple before making the change. Therefore an on_replace trigger is better if there is no need to change the old tuple. However, this only applies for the memtx engine – for the vinyl engine, the fetch will happen for either kind of trigger. (With memtx the tuple data is stored along with the index key so no extra search is necessary; with vinyl that is not the case so the extra search is necessary.)

Where the extra capability is not needed, on_replace should be used instead of before_replace. Usually before_replace is used only for certain replication scenarios – it is useful for conflict resolution.

The value that a before_replace trigger function can return affects what will happen after the return. Specifically:

- if there is no return value, then execution proceeds, inserting|replacing the new value;
- if the value is nil, then the tuple will be deleted;
- if the value is the same as the old parameter, then no on_replace‘ function will be called and the data change will be skipped
- if the value is the same as the new parameter, then it's as if the before_replace function wasn't called;
- if the value is something else, then execution proceeds, inserting|replacing the new value.

However, if a trigger function returns an old tuple, or if a trigger function calls run_triggers(false), that will not affect other triggers that are activated for the same insert|update|replace request.

Example:

The following are before_replace functions that have no return value, or that return nil, or the same as the old parameter, or the same as the new parameter, or something else.

```
function f1 (old, new) return end
function f2 (old, new) return nil end
function f3 (old, new) return old end
function f4 (old, new) return new end
function f5 (old, new) return box.tuple.new({new[1],'b'}) end
```

space_object:pairs([key[, iterator]])

Search for a tuple or a set of tuples in the given space, and allow iterating over one tuple at a time.

Parameters

- space_object (space_object) – an object reference
- key (scalar/table) – value to be matched against the index key, which may be multi-part
- iterator – see index_object:pairs

Return iterator which can be used in a for/end loop or with totable()

Possible errors:

- no such space;
- wrong type.

Complexity factors: Index size, Index type.

For examples of complex pairs requests, where one can specify which index to search and what condition to use (for example "greater than" instead of "equal to"), see the later section index_object:pairs.

For information about iterators' internal structures see the "Lua Functional library" documentation.

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
...
```

space_object:rename(space-name)

Rename a space.

Parameters

- space_object (space_object) – an object reference

- space-name (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 (space_object) – an object reference
- tuple (table/tuple) – tuple to be inserted

Return the inserted tuple.

Rtype tuple

Possible errors: ER_TUPLE_FOUND if a different tuple with the same unique-key value already exists. (This will only happen if there is a unique secondary index.)

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

Example:

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

For more usage scenarios and typical errors see Example: using data operations further in this section.

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:

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)
```

space_object:select([key[, options]])
Search for a tuple or a set of tuples in the given space.

Parameters

- space_object (space_object) – an object reference
- key (scalar/table) – value to be matched against the index key, which may be multi-part.
- options (table/nil) – none, any or all of the same options that index_object:select allows:
  - options.iterator (type of iterator)
  - options.limit (maximum number of tuples)
  - options.offset (number of tuples to skip)

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

Rtype array of tuples

A select request can also be done with a specific index and index options, which are the subject of index_object:select.

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'}
```

(continues on next page)

(continued from previous page)

```
---
- - [1, 'B']
...
tarantool> -- must equal only one primary-key field
tarantool> s:select{1}
---
- - [1, 'A']
  - [1, 'B']
  - [1, 'C']
...
tarantool> -- must equal 0 fields, so returns all tuples
tarantool> s:select{}
---
- - [1, 'A']
  - [1, 'B']
  - [1, 'C']
  - [2, 'D']
...
tarantool> -- the first field must be greater than 0, and
tarantool> -- skip the first tuple, and return up to
tarantool> -- 2 tuples. This example's options all
tarantool> -- depend on index characteristics so see
tarantool> -- more explanation in index_object:select().
tarantool> s:select({0},{iterator='GT',offset=1,limit=2})
---
- - [1, 'B']
  - [1, 'C']
...
```

As the last request in the above example shows: to make complex select requests, where you 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, it will be necessary to become familiar with index_object:select.

For more usage scenarios and typical errors see Example: using data operations further in this section.

space_object:truncate()

Deletes all tuples.

Parameters

- space_object (space_object) – an object reference

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

Return nil

The truncate method can only be called by the user who created the space, or from within a setuid function created by the user who created the space. Read more about setuid functions in the reference for box.schema.func.create().

The truncate method cannot be called from within a transaction.

Example:

```
tarantool> box.space.tester:truncate()
---
...
```

(continues on next page)

(continued from previous page)

```
tarantool> box.space.tester:len()
---
- 0
...
```

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 (space_object) – an object reference
- key (scalar/table) – primary-key field values, must be passed as a Lua table if key is multi-part
- operator (string) – operation type represented in string
- field_no (number) – what field the operation will apply to. The field number can be negative, meaning the position from the end of tuple. (#tuple + negative field number + 1)
- value (lua_value) – what value will be applied

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 initially there is a space named tester with a primary-key index 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.tester: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.tester: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!!Z'.

For more usage scenarios and typical errors see Example: using data operations further in this section.

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 (space_object) – an object reference
- tuple (table/tuple) – default tuple to be inserted, if analogue isn't found
- operator (string) – operation type represented in string
- field_no (number) – what field the operation will apply to. The field number can be negative, meaning the position from the end of tuple. (#tuple + negative field number + 1)
- value (lua_value) – what value will be applied

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

For more usage scenarios and typical errors see Example: using data operations further in this section.

space_object:user_defined()

Users can define any functions they want, and associate them with spaces: in effect they can make their own space methods. They do this by:

(1) creating a Lua function,

(2) adding the function name to a predefined global variable which has type = table, and

(3) invoking the function any time thereafter, as long as the server is up, by saying space_object:function-name([parameters]).

The predefined global variable is box.schema.space_mt. Adding to box.schema.space_mt makes the method available for all spaces.

Alternatively, user-defined methods can be made available for only one space, by calling getmetatable(space_object) and then adding the function name to the meta table. See also the example for index_object:user_defined().

Parameters

- index_object (index_object) – an object reference.
- any-name (any-type) – whatever the user defines

Example:

```
-- Visible to any space, no parameters.
-- After these requests, the value of global_variable will be 6.
box.schema.space.create('t')
box.space.t:create_index('i')
global_variable = 5
function f(space_arg) global_variable = global_variable + 1 end
box.schema.space_mt.counter = f
box.space.t:counter()
```

space_object:create_check_constraint(check_constraint_name, expression)

Create a check constraint. A check constraint is a requirement that must be met when a tuple is inserted or updated in a space. Check constraints created with space_object:create_check_constraint have the same effect as check constraints created with an SQL CHECK() clause in a CREATE TABLE statement.

Parameters

- space_object (space_object) – an object reference
- check_constraint_name (string) – name of check constraint, which should conform to the rules for object names
- expression (string) – SQL code of an expression which must return a boolean result

Return check constraint object

Rtype check_constraint_object

The space must be formatted with space_object:format() so that the expression can contain field names. The space must be empty. The space must not be a system space.

The expression must return true or false and should be deterministic. The expresion may be any SQL (not Lua) expression containing field names, built-in function names, literals, and operators. Not subqueries. If a field name contains lower case characters, it must be enclosed in "double quotes".

Check constraints are checked before the request is performed, at the same time as Lua before_replace triggers. If there is more than one check constraint or before_replace trigger, then they are ordered according to time of creation. (This is a change from the earlier behavior of check constraints, which caused checking before the tuple was formed.)

Check constraints can be dropped with space_object:check_constraint_name:drop().

Example:

```
box.schema.space.create('t')
box.space.t:format({{name = 'f1', type = 'unsigned'},
                    {name = 'f2', type = 'string'},
                    {name = 'f3', type = 'string'}})
box.space.t:create_index('i')
box.space.t:create_check_constraint('c1', [["f2" > 'A']])
box.space.t:create_check_constraint('c2',
                      [["f2"=UPPER("f3") AND NOT "f2" LIKE '__']])
-- This insert will fail, constraint c1 expression returns false
box.space.t:insert{1, 'A', 'A'}
-- This insert will fail, constraint c2 expression returns false
box.space.t:insert{1, 'B', 'c'}
-- This insert will succeed, both constraint expressions return true
box.space.t:insert{1, 'B', 'b'}
-- This update will fail, constraint c2 expression returns false
box.space.t:update(1, {{'=', 2, 'xx'}, {'=', 3, 'xx'}})
```

A list of check constraints is in space_object._ck_constraint.

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

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.

Example:

```
tarantool> box.space.tester.field_count
---
- 0
...
```

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.tester:insert{0} and box.space[800]:insert{0} are equivalent requests.

Example:

```
tarantool> box.space.tester.id
---
- 512
...
```

box.space.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.

To reset, use box.stat.reset().

Rtype table

Example:

```
# checking the number of indexes for space 'tester'
tarantool> local counter=0; for i=0,#box.space.tester.index do
  if box.space.tester.index[i]~=nil then counter=counter+1 end
  end; print(counter)
1
---
...
# checking the type of index 'primary'
tarantool> box.space.tester.index.primary.type
---
- TREE
...
```

box.space._cluster
: _cluster is a system space for support of the replication feature.

box.space._func
: _func is a system space with function tuples made by box.schema.func.create() or

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

Tuples in this space contain the following fields:

- id (integer identifier),
- owner (integer identifier),
- the function name,
- the setuid flag,
- a language name (optional): 'LUA' (default) or 'C'.
- the body
- the is_deterministic flag
- the is_sandboxed flag
- options

If the function tuple was made in the older way without specification of body, then the _func space will contain default values for the body and the is_deterministic flag and the is_sandboxed flag. Such function tuples are called "not persistent". You continue to create Lua functions in the usual way, by saying function function_name () ... end, without adding anything in the _func space. The _func space only exists for storing function tuples so that their names can be used within grant/revoke functions.

If the function tuple was made the newer way with specification of body, then all the fields may contain non-default values. Such functions are called "persistent". They should be invoked with box.func.func-name:call([parameters]).

You can:

- Create a _func tuple with box.schema.func.create(),
- Drop a _func tuple with box.schema.func.drop(),
- Check whether a _func tuple exists with box.schema.func.exists().

Example:

In the following example, we create a function named 'f7', put it into Tarantool's _func space and grant 'execute' privilege for this function to 'guest' user.

```
tarantool> function f7()
         >   box.session.uid()
         > end
---
...
tarantool> box.schema.func.create('f7')
---
...
tarantool> box.schema.user.grant('guest', 'execute', 'function', 'f7')
---
...
tarantool> box.schema.user.revoke('guest', 'execute', 'function', 'f7')
---
...
```

box.space._index
: _index is a system space.

  Tuples in this space contain the following fields:

- id (= id of space),
- iid (= index number within space),
- name,
- type,
- opts (e.g. unique option), [tuple-field-no, tuple-field-type . . .].

Here is what _index contains in a typical installation:

```
tarantool> box.space._index:select{}
---
- - [272, 0, 'primary', 'tree', {'unique': true}, [[0, 'string']]]
  - [280, 0, 'primary', 'tree', {'unique': true}, [[0, 'unsigned']]]
  - [280, 1, 'owner', 'tree', {'unique': false}, [[1, 'unsigned']]]
  - [280, 2, 'name', 'tree', {'unique': true}, [[2, 'string']]]
  - [281, 0, 'primary', 'tree', {'unique': true}, [[0, 'unsigned']]]
  - [281, 1, 'owner', 'tree', {'unique': false}, [[1, 'unsigned']]]
  - [281, 2, 'name', 'tree', {'unique': true}, [[2, 'string']]]
  - [288, 0, 'primary', 'tree', {'unique': true}, [[0, 'unsigned'], [1, 'unsigned']]]
  - [288, 2, 'name', 'tree', {'unique': true}, [[0, 'unsigned'], [2, 'string']]]
  - [289, 0, 'primary', 'tree', {'unique': true}, [[0, 'unsigned'], [1, 'unsigned']]]
  - [289, 2, 'name', 'tree', {'unique': true}, [[0, 'unsigned'], [2, 'string']]]
  - [296, 0, 'primary', 'tree', {'unique': true}, [[0, 'unsigned']]]
  - [296, 1, 'owner', 'tree', {'unique': false}, [[1, 'unsigned']]]
  - [296, 2, 'name', 'tree', {'unique': true}, [[2, 'string']]]
---
...
```

box.space._vindex

_vindex is a system space that represents a virtual view. The structure of its tuples is identical to that of _index, but permissions for certain tuples are limited in accordance with user privileges. _vindex contains only those tuples that are accessible to the current user. See Access control for details about user privileges.

If the user has the full set of privileges (like 'admin'), the contents of _vindex match the contents of _index. If the user has limited access, _vindex contains only tuples accessible to this user.

Note:

- _vindex is a system view, so it allows only read requests.
- While the _index space requires proper access privileges, any user can always read from _vindex.

---

box.space._priv

_priv is a system space where privileges are stored.

Tuples in this space contain the following fields:

- 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', 'function', 'sequence' or 'universe',
- the numeric id of the object,
- the type of operation: "read" = 1, "write" = 2, "execute" = 4, "create" = 32, "drop" = 64, "alter" = 128, or a combination such as "read,write,execute".

You can:

- Grant a privilege with box.schema.user.grant().
- Revoke a privilege with box.schema.user.revoke().

Note:

- 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, make sure that all their associated privileges have been revoked.
- Only the 'admin' user can grant privileges for the 'universe'.
- Only the 'admin' user or the creator of a space can drop, alter, or truncate the space.
- Only the 'admin' user or the creator of a user can change a different user's password.

---

box.space._vpriv

_vpriv is a system space that represents a virtual view. The structure of its tuples is identical to that of _priv, but permissions for certain tuples are limited in accordance with user privileges. _vpriv contains only those tuples that are accessible to the current user. See Access control for details about user privileges.

If the user has the full set of privileges (like 'admin'), the contents of _vpriv match the contents of _priv. If the user has limited access, _vpriv contains only tuples accessible to this user.

Note:

- _vpriv is a system view, so it allows only read requests.
- While the _priv space requires proper access privileges, any user can always read from _vpriv.

---

box.space._schema

_schema is a system space.

This space contains the following tuples:

- version tuple with version information for this Tarantool instance,
- cluster tuple with the instance's replica set ID,
- max_id tuple with the maximal space ID,
- once... tuples that correspond to specific box.once() blocks from the instance's initialization file. The first field in these tuples contains the key value from the corresponding box.once() block prefixed with 'once' (e.g. oncehello), so you can easily find a tuple that corresponds to a specific box.once() block.

Example:

Here is what _schema contains in a typical installation (notice the tuples for two box.once() blocks, 'oncebye' and 'oncehello'):

```
tarantool> box.space._schema:select{}
---
- - ['cluster', 'b4e15788-d962-4442-892e-d6c1dd5d13f2']
  - ['max_id', 512]
```

(continues on next page)

(continued from previous page)

```
- ['oncebye']
- ['oncehello']
- ['version', 1, 7, 2]
```

box.space._sequence
: _sequence is a system space for support of the sequence feature. It contains persistent information that was established by box.schema.sequence.create() or box.schema.sequence.alter().

box.space._sequence_data
: _sequence_data is a system space for support of the sequence feature.

  Each tuple in _sequence_data contains two fields:

  - the id of the sequence, and
  - the last value that the sequence generator returned (non-persistent information).

  There is no guarantee that this space will be updated immediately after every data-change request.

box.space._space
: _space is a system space.

  Tuples in this space contain the following fields:

  - id,
  - owner (= id of user who owns the space),
  - name, engine, field_count,
  - flags (e.g. temporary),
  - format (as made by a format clause).

  These fields are established by space.create().

  Example #1:

  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
    i = i + 1
    end
    table.insert(ta, line)
  end
  return ta
end
```

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

```
tarantool> example()
---
```

(continues on next page)

(continued from previous page)

```
- - '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 #2:

The following requests will create a space using box.schema.space.create() with a format clause, then retrieve 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', {
         >   id = 12345,
         >   format = {
         >     [1] = {["name"] = "field_1"},
         >     [2] = {["type"] = "unsigned"}
         >   }
         > })
---
- index: []
  on_replace: 'function: 0x41c67338'
  temporary: false
  id: 12345
  engine: memtx
  enabled: false
  name: TM
  field_count: 0
- created
...
tarantool> box.space._space:select(12345)
---
- - [12345, 1, 'TM', 'memtx', 0, {}, [{'name': 'field_1'}, {'type': 'unsigned'}]]
...
```

box.space._vspace

_vspace is a system space that represents a virtual view. The structure of its tuples is identical to that of _space, but permissions for certain tuples are limited in accordance with user privileges. _vspace contains only those tuples that are accessible to the current user. See Access control for details about user privileges.

If the user has the full set of privileges (like 'admin'), the contents of _vspace match the contents of _space. If the user has limited access, _vspace contains only tuples accessible to this user.

Note:

- _vspace is a system view, so it allows only read requests.

- While the _space space requires proper access privileges, any user can always read from _vspace.

box.space._user
_user is a system space where user-names and password hashes are stored.

Tuples in this space contain the following fields:

- the numeric id of the tuple ("id"),
- the numeric id of the tuple's creator,
- the name,
- the type: 'user' or 'role',
- optional password.

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

Name	ID	Type	Description
guest	0	user	Default user when connecting remotely. Usually an untrusted user with few privileges.
admin	1	user	Default user when using Tarantool as a console. Usually an administrative user with all privileges.
public	2	role	Pre-defined role, automatically granted 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 with box.schema.role.grant('public', 'read','space','t').
replication	3	role	Pre-defined role, which the 'admin' user can grant to users who need to use replication features.
super	31	role	Pre-defined role, which the 'admin' user can grant to users who need all privileges on all objects. The 'super' role has these privileges on 'universe': read, write, execute, create, drop, alter.

To select a tuple 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']
...
```

Warning: 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, use box.schema.user.create():
box.schema.user.create(user-name)
box.schema.user.create(user-name, {if_not_exists = true})
box.schema.user.create(user-name, {password = password})
To change the user's password, use box.schema.user.password():
-- To change the current user's password
box.schema.user.passwd(password)

-- To change a different user's password
-- (usually only 'admin' can do it)
box.schema.user.passwd(user-name, password)
To drop a user, use box.schema.user.drop():
box.schema.user.drop(user-name)
To check whether a user exists, use box.schema.user.exists(), which returns true or false:
box.schema.user.exists(user-name)
To find what privileges a user has, use box.schema.user.info():
box.schema.user.info(user-name)

Note: The maximum number of users is 32.

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': 't3xjUpQdrt857O+YRvGbMY5py8Q='}]
...
tarantool> box.schema.user.drop('JeanMartin')
---
...
```

box.space._ck_constraint
_ck_constraint is a system space where check constraints are stored.

Tuples in this space contain the following fields:

- the numeric id of the space ("space_id"),
- the name,
- whether the check is deferred ("is_deferred"),
- the language of the expression, such as 'SQL',
- the expression ("code")

Example:

```
tarantool> box.space._ck_constraint:select()
---
- - [527, 'c1', false, 'SQL', '"f2" > ''A''']
  - [527, 'c2', false, 'SQL', '"f2" == UPPER("f3") AND NOT "f2" LIKE ''__''']
...
```

Example: using 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 space, 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: using 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' }
```

However, the retrieved tuple is not easy to read:

```
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:

```
tarantool> do
         >   local tuple_of_space = box.space._space.index.name:get{'_space'}
         >   for _, field in ipairs(tuple_of_space[7]) do
         >     print(field.name .. ', ' .. field.type)
         >   end
         > end
id, num
owner, num
name, str
engine, str
field_count, num
flags, str
format, *
---
...
```

box.space._vuser

_vuser is a system space that represents a virtual view. The structure of its tuples is identical to that of _user, but permissions for certain tuples are limited in accordance with user privileges. _vuser contains only those tuples that are accessible to the current user. See Access control for details about user privileges.

If the user has the full set of privileges (like 'admin'), the contents of _vuser match the contents of _user. If the user has limited access, _vuser contains only tuples accessible to this user.

To see how _vuser works, connect to a Tarantool database remotely via tarantoolctl and select all tuples from the _user space, both when the 'guest' user is and is not allowed to read from the database.

First, start Tarantool and grant the 'guest' user with read, write and execute privileges:

```
tarantool> box.cfg{listen = 3301}
---
...
tarantool> box.schema.user.grant('guest', 'read,write,execute', 'universe')
---
...
```

Switch to the other terminal, connect to the Tarantool instance and select all tuples from the _user space:

```
$ tarantoolctl connect 3301
localhost:3301> box.space._user:select{}
---
- - [0, 1, 'guest', 'user', {}]
  - [1, 1, 'admin', 'user', {}]
  - [2, 1, 'public', 'role', {}]
  - [3, 1, 'replication', 'role', {}]
  - [31, 1, 'super', 'role', {}]
...
```

This result contains the same set of users as if you made the request from your Tarantool instance as 'admin'.

Switch to the first terminal and revoke the read privileges from the 'guest' user:

```
tarantool> box.schema.user.revoke('guest', 'read', 'universe')
---
...
```

Switch to the other terminal, stop the session (to stop tarantoolctl, type Ctrl+C or Ctrl+D) and repeat the box.space._user:select{} request. The access is denied:

```
$ tarantoolctl connect 3301
localhost:3301> box.space._user:select{}
---
- error: Read access to space '_user' is denied for user 'guest'
...
```

However, if you select from _vuser instead, the users' data available for the 'guest' user is displayed:

```
localhost:3301> box.space._vuser:select{}
---
- - [0, 1, 'guest', 'user', {}]
...
```

Note:

- _vuser is a system view, so it allows only read requests.
- While the _user space requires proper access privileges, any user can always read from _vuser.

---

box.space._collation

_collation is a system space with a list of collations. There are over 270 built-in collations and users may add more. Here is one example:

```
localhost:3301> box.space._collation:select(239)
---
- - [239, 'unicode_uk_s2', 1, 'ICU', 'uk', {'strength': 'secondary'}]
...
```

Explanation of the fields in the example: id = 239 i.e. Tarantool's primary key is 239, name = 'unicode_uk_s2' i.e. according to Tarantool's naming convention this is a Unicode collation + it is for the uk locale + it has secondary strength, owner = 1 i.e. the admin user, type = 'ICU' i.e. the rules are according to International Components for Unicode, locale = 'uk' i.e. Ukrainian, opts = 'strength:secondary' i.e. with this collation comparisons use both primary and secondary weights.

box.space._vcollation
: _vcollation is a system space with a list of collations. The structure of its tuples is identical to that of box.space._collation, but permissions for certain tuples are limited in accordance with user privileges.

Example: using data operations

This example demonstrates all legal scenarios – as well as typical errors – for each data operation in Tarantool: INSERT, DELETE, UPDATE, UPSERT, REPLACE, and SELECT.

```
-- Bootstrap the database --
box.cfg{}
format = {}
format[1] = {'field1', 'unsigned'}
format[2] = {'field2', 'unsigned'}
format[3] = {'field3', 'unsigned'}
s = box.schema.create_space('test', {format = format})
-- Create a primary index --
pk = s:create_index('pk', {parts = {{'field1'}}})
-- Create a unique secondary index --
sk_uniq = s:create_index('sk_uniq', {parts = {{'field2'}}})
-- Create a non-unique secondary index --
sk_non_uniq = s:create_index('sk_non_uniq', {parts = {{'field3'}}, unique = false})
```

INSERT

insert accepts a well-formatted tuple and checks all keys for duplicates.

```
tarantool> -- Unique indexes: ok --
tarantool> s:insert({1, 1, 1})
---
- [1, 1, 1]
...
tarantool> -- Conflicting primary key: error --
tarantool> s:insert({1, 1, 1})
---
- error: Duplicate key exists in unique index 'pk' in space 'test'
...
tarantool> -- Conflicting unique secondary key: error --
tarantool> s:insert({2, 1, 1})
---
- error: Duplicate key exists in unique index 'sk_uniq' in space 'test'
...
tarantool> -- Key {1} exists in sk_non_uniq index, but it is not unique: ok --
tarantool> s:insert({2, 2, 1})
---
- [2, 2, 1]
...
tarantool> s:truncate()
---
...
```

DELETE

delete accepts a full key of any unique index.

space:delete is an alias for “delete by primary key”.

```
tarantool> -- Insert some test data --
tarantool> s:insert{3, 4, 5}
---
- [3, 4, 5]
...
tarantool> s:insert{6, 7, 8}
---
- [6, 7, 8]
...
tarantool> s:insert{9, 10, 11}
---
- [9, 10, 11]
...
tarantool> s:insert{12, 13, 14}
---
- [12, 13, 14]
...
tarantool> -- Nothing done here: no {4} key in pk index --
tarantool> s:delete{4}
---
...
tarantool> s:select{}
---
- - [3, 4, 5]
  - [6, 7, 8]
  - [9, 10, 11]
  - [12, 13, 14]
...
tarantool> -- Delete by a primary key: ok --
tarantool> s:delete{3}
---
- [3, 4, 5]
...
tarantool> s:select{}
---
- - [6, 7, 8]
  - [9, 10, 11]
  - [12, 13, 14]
...
tarantool> -- Explicitly delete by a primary key: ok --
tarantool> s.index.pk:delete{6}
---
- [6, 7, 8]
...
tarantool> s:select{}
---
- - [9, 10, 11]
  - [12, 13, 14]
...
tarantool> -- Delete by a unique secondary key: ok --
s.index.sk_uniq:delete{10}
---
- [9, 10, 11]
...
s:select{}
---
```

(continues on next page)

(continued from previous page)

```
- - [12, 13, 14]
...
tarantool> -- Delete by a non-unique secondary index: error --
tarantool> s.index.sk_non_uniq:delete{14}
---
- error: Get() doesn't support partial keys and non-unique indexes
...
tarantool> s:select{}
---
- - [12, 13, 14]
...
tarantool> s:truncate()
---
...
```

The key must be full: delete cannot work with partial keys.

```
tarantool> s2 = box.schema.create_space('test2')
---
...
tarantool> pk2 = s2:create_index('pk2', {parts = {{1, 'unsigned'}, {2, 'unsigned'}}})
---
...
tarantool> s2:insert{1, 1}
---
- [1, 1]
...
tarantool> -- Delete by a partial key: error --
tarantool> s2:delete{1}
---
- error: Invalid key part count in an exact match (expected 2, got 1)
...
tarantool> -- Delete by a full key: ok --
tarantool> s2:delete{1, 1}
---
- [1, 1]
...
tarantool> s2:select{}
---
- []
...
tarantool> s2:drop()
---
...
```

UPDATE

Similarly to delete, update accepts a full key of any unique index, and also the operations to execute.

space:update is an alias for “update by primary key”.

```
tarantool> -- Insert some test data --
tarantool> s:insert{3, 4, 5}
---
- [3, 4, 5]
```

(continues on next page)

(continued from previous page)

```
...
tarantool> s:insert{6, 7, 8}
---
- [6, 7, 8]
...
tarantool> s:insert{9, 10, 11}
---
- [9, 10, 11]
...
tarantool> s:insert{12, 13, 14}
---
- [12, 13, 14]
...
tarantool> -- Nothing done here: no {4} key in pk index --
s:update({4}, {{'=', 2, 400}})
---
...
tarantool> s:select{}
---
- - [3, 4, 5]
  - [6, 7, 8]
  - [9, 10, 11]
  - [12, 13, 14]
...
tarantool> -- Update by a primary key: ok --
tarantool> s:update({3}, {{'=', 2, 400}})
---
- [3, 400, 5]
...
tarantool> s:select{}
---
- - [3, 400, 5]
  - [6, 7, 8]
  - [9, 10, 11]
  - [12, 13, 14]
...
tarantool> -- Explicitly update by a primary key: ok --
tarantool> s.index.pk:update({6}, {{'=', 2, 700}})
---
- [6, 700, 8]
...
tarantool> s:select{}
---
- - [3, 400, 5]
  - [6, 700, 8]
  - [9, 10, 11]
  - [12, 13, 14]
...
tarantool> -- Update by a unique secondary key: ok --
tarantool> s.index.sk_uniq:update({10}, {{'=', 2, 1000}})
---
- [9, 1000, 11]
...
tarantool> s:select{}
---
- - [3, 400, 5]
```

(continues on next page)

(continued from previous page)

```
  - [6, 700, 8]
  - [9, 1000, 11]
  - [12, 13, 14]
...
tarantool> -- Update by a non-unique secondary key: error --
tarantool> s.index.sk_non_uniq:update({14}, {{'=', 2, 1300}})
---
- error: Get() doesn't support partial keys and non-unique indexes
...
tarantool> s:select{}
---
- - [3, 400, 5]
  - [6, 700, 8]
  - [9, 1000, 11]
  - [12, 13, 14]
...
tarantool> s:truncate()
---
...
```

UPSERT

upsert accepts a well-formatted tuple and update operations.

If an old tuple is found by the primary key of the specified tuple, then the update operations are applied to the old tuple, and the new tuple is ignored.

If no old tuple is found, then the new tuple is inserted, and the update operations are ignored.

Indexes have no upsert method - this is a method of a space.

```
tarantool> s.index.pk.upsert == nil
---
- true
...
tarantool> s.index.sk_uniq.upsert == nil
---
- true
...
tarantool> s.upsert ~= nil
---
- true
...
tarantool> -- As the first argument, upsert accepts --
tarantool> -- a well-formatted tuple, NOT a key! --
tarantool> s:insert{1, 2, 3}
---
- [1, 2, 3]
...
tarantool> s:upsert({1}, {{'=', 2, 200}})
---
- error: Tuple field count 1 is less than required by space format or defined indexes
    (expected at least 3)
...
tarantool> s:select{}
---
```

(continues on next page)

(continued from previous page)

```
- - [1, 2, 3]
...
tarantool> s:delete{1}
---
- [1, 2, 3]
...
```

upsert turns into insert when no old tuple is found by the primary key.

```
tarantool> s:upsert({1, 2, 3}, {{'=', 2, 200}})
---
...
tarantool> -- As you can see, {1, 2, 3} were inserted, --
tarantool> -- and the update operations were not applied. --
s:select{}
---
- - [1, 2, 3]
...
tarantool> -- Performing another upsert with the same primary key, --
tarantool> -- but different values in the other fields. --
s:upsert({1, 20, 30}, {{'=', 2, 200}})
---
...
tarantool> -- The old tuple was found by the primary key {1} --
tarantool> -- and update operations were applied. --
tarantool> -- The new tuple was ignored. --
tarantool> s:select{}
---
- - [1, 200, 3]
...
```

upsert searches for an old tuple by the primary index, NOT by a secondary index. This can lead to a duplication error if the new tuple ruins the uniqueness of a secondary index.

```
tarantool> s:upsert({2, 200, 3}, {{'=', 3, 300}})
---
- error: Duplicate key exists in unique index 'sk_uniq' in space 'test'
...
s:select{}
---
- - [1, 200, 3]
...
tarantool> -- But this works, when uniqueness is preserved. --
tarantool> s:upsert({2, 0, 0}, {{'=', 3, 300}})
---
...
tarantool> s:select{}
---
- - [1, 200, 3]
  - [2, 0, 0]
...
tarantool> s:truncate()
---
...
```

REPLACE

replace accepts a well-formatted tuple and searches for an old tuple by the primary key of the new tuple.

If the old tuple is found, then it is deleted, and the new tuple is inserted.

If the old tuple was not found, then just the new tuple is inserted.

```
tarantool> s:replace{1, 2, 3}
---
- [1, 2, 3]
...
tarantool> s:select{}
---
- - [1, 2, 3]
...
tarantool> s:replace{1, 3, 4}
---
- [1, 3, 4]
...
tarantool> s:select{}
---
- - [1, 3, 4]
...
tarantool> s:truncate()
---
...
```

replace can ruin unique constraints, like upsert does.

```
tarantool> s:insert{1, 1, 1}
---
- [1, 1, 1]
...
tarantool> s:insert{2, 2, 2}
---
- [2, 2, 2]
...
tarantool> -- This replace fails, because if the new tuple {1, 2, 0} replaces --
tarantool> -- the old tuple by the primary key from 'pk' index {1, 1, 1}, --
tarantool> -- this results in a duplicate unique secondary key in 'sk_uniq' index: --
tarantool> -- key {2} is used both in the new tuple and in {2, 2, 2}. --
tarantool> s:replace{1, 2, 0}
---
- error: Duplicate key exists in unique index 'sk_uniq' in space 'test'
...
tarantool> s:truncate()
---
...
```

SELECT

select works with any indexes (primary/secondary) and with any keys (unique/non-unique, full/partial).

If a key is partial, then select searches by all keys, where the prefix matches the specified key part.

```
tarantool> s:insert{1, 2, 3}
---
- [1, 2, 3]
...
tarantool> s:insert{4, 5, 6}
---
- [4, 5, 6]
...
tarantool> s:insert{7, 8, 9}
---
- [7, 8, 9]
...
tarantool> s:insert{10, 11, 9}
---
- [10, 11, 9]
...
tarantool> s:select{1}
---
- - [1, 2, 3]
...
tarantool> s:select{}
---
- - [1, 2, 3]
  - [4, 5, 6]
  - [7, 8, 9]
  - [10, 11, 9]
...
tarantool> s.index.pk:select{4}
---
- - [4, 5, 6]
...
tarantool> s.index.sk_uniq:select{8}
---
- - [7, 8, 9]
...
tarantool> s.index.sk_non_uniq:select{9}
---
- - [7, 8, 9]
  - [10, 11, 9]
...
```

Submodule box.stat

The box.stat submodule provides access to request and network statistics.

Use box.stat() to show the average number of requests per second, and the total number of requests since startup, broken down by request type.

Use box.stat.net() to see network activity: the number of packets sent and received, the count of active iproto connections, and the average number of requests per second.

Use box.stat.vinyl() to see vinyl-storage-engine activity, for example box.stat.vinyl().tx has the number of commits and rollbacks. See details at the end of this section.

Use box.stat.reset() to reset the statistics of box.stat(), box.stat.net(), box.stat.vinyl() and box.space.index.

In the tables that box.stat() and box.stat.net() return: rps stands for "[average number of] requests per second [in the last 5 seconds]", total stands for "total [number processed since the server began]", current

stands for "[number of] current [requests in progress, which can be limited by box.cfg.net_msg_max]".

```
tarantool> box.stat() -- return 10 tables
---
- DELETE:
    total: 1873949
    rps: 123
  SELECT:
    total: 1237723
    rps: 4099
  INSERT:
    total: 0
    rps: 0
  EVAL:
    total: 0
    rps: 0
  CALL:
    total: 0
    rps: 0
  REPLACE:
    total: 1239123
    rps: 7849
  UPSERT:
    total: 0
    rps: 0
  AUTH:
    total: 0
    rps: 0
  ERROR:
    total: 0
    rps: 0
  UPDATE:
    total: 0
    rps: 0
...
tarantool> box.stat().DELETE -- total + requests per second from one table
---
- total: 0
  rps: 0
...
tarantool> box.stat.net() -- 4 tables
---
- SENT:
    total: 0
    rps: 0
  CONNECTIONS:
    current: 0
    rps: 0
    total: 0
  REQUESTS:
    current: 0
    rps: 0
    total: 0
  RECEIVED:
    total: 0
    rps: 0
...
tarantool> box.stat.vinyl().tx.commit -- one item of the vinyl table
```

(continues on next page)

(continued from previous page)

```
---
- 1047632
...
```

Here are details about the box.stat.vinyl() items.

Details about box.stat.vinyl().regulator: The vinyl regulator decides when to take or delay actions for disk IO, grouping activity in batches so that it is consistent and efficient. The regulator is invoked by the vinyl scheduler, once per second, and updates related variables whenever it is invoked.

- box.stat.vinyl().regulator.dump_bandwidth is the estimated average rate at which dumps are done. Initially this will appear as 10485760 (10 megabytes per second). Only significant dumps (larger than one megabyte) are used for estimating.
- box.stat.vinyl().regulator.dump_watermark is the point when dumping must occur. The value is slightly smaller than the amount of memory that is allocated for vinyl trees, which is the vinyl_memory parameter.
- box.stat.vinyl().regulator.write_rate is the actual average rate at which recent writes to disk are done. Averaging is done over a 5-second time window, so if there has been no activity for 5 seconds then regulator.write_rate = 0. The write_rate may be slowed when a dump is in progress or when the user has set snap_io_rate_limit.
- box.stat.vinyl().regulator.rate_limit is the write rate limit, in bytes per second, imposed on transactions by the regulator based on the observed dump/compaction performance.

Details about box.stat.vinyl().disk: Since vinyl is an on-disk storage engine (unlike memtx which is an in-memory storage engine), it can handle large databases – but if a database is larger than the amount of memory that is allocated for vinyl, then there will be more disk activity.

- box.stat.vinyl().disk.data and box.stat.vinyl().disk.index are the amount of data that has gone into files in a subdirectory of vinyl_dir, with names like {lsn}.run and {lsn}.index. The size of the run will be related to the output of scheduler.dump_*.
- box.stat.vinyl().disk.data_compacted Sum size of data stored at the last LSM tree level, in bytes, without taking disk compression into account. It can be thought of as the size of disk space that the user data would occupy if there were no compression, indexing, or space increase caused by the LSM tree design.

Details about box.stat.vinyl().memory: Although the vinyl storage engine is not "in-memory", Tarantool does need to have memory for write buffers and for caches:

- box.stat.vinyl().memory.tuple_cache is the number of bytes that are being used for tuples (data).
- box.stat.vinyl().memory.tx is transactional memory. This will usually be 0.
- box.stat.vinyl().memory.level0 is the "level0" memory area, sometimes abbreviated "L0", which is the area that vinyl can use for in-memory storage of an LSM tree.

Therefore we can say that "L0 is becoming full" when the amount in memory.level0 is close to the maximum, which is regulator.dump_watermark. We can expect that "L0 = 0" immediately after a dump. box.stat.vinyl().memory.page_index and box.stat.vinyl().memory.bloom_filter have the current amount being used for index-related structures. The size is a function of the number and size of keys, plus page_size, plus bloom_fpr. This is not a count of bloom filter "hits" (the number of reads that could be avoided because the bloom filter predicts their presence in a run file) – that statistic can be found with index_object:stat().

Details about box.stat.vinyl().tx: This is about requests that affect transactional activity ("tx" is used here as an abbreviation for "transaction"):

- box.stat.vinyl().tx.conflict counts conflicts that caused a transaction to roll back.

- box.stat.vinyl().tx.commit is the count of commits (successful transaction ends). It includes implicit commits, for example any insert causes a commit unless it is within a begin-end block.
- box.stat.vinyl().tx.rollback is the count of rollbacks (unsuccessful transaction ends). This is not merely a count of explicit box.rollback requests – it includes requests that ended in errors. For example, after an attempted insert request that causes a "Duplicate key exists in unique index" error, tx.rollback is incremented.
- box.stat.vinyl().tx.statements will usually be 0.
- box.stat.vinyl().tx.transactions is the number of transactions that are currently running.
- box.stat.vinyl().tx.gap_locks is the number of gap locks that are outstanding during execution of a request. For a low-level description of Tarantool's implementation of gap locking, see Gap locks in Vinyl transaction manager.
- box.stat.vinyl().tx.read_views shows whether a transaction has entered a read-only state to avoid conflict temporarily. This will usually be 0.

Details about box.stat.vinyl().scheduler: This primarily has counters related to tasks that the scheduler has arranged for dumping or compaction: (most of these items are reset to 0 when the server restarts or when box.stat.reset() occurs):

- box.stat.vinyl().scheduler.compaction_* is the amount of data from recent changes that has been compacted. This is divided into scheduler.compaction_input (the amount that is being compacted), scheduler.compaction_queue (the amount that is waiting to be compacted), scheduler.compaction_time (total time spent by all worker threads performing compaction, in seconds), and scheduler.compaction_output (the amount that has been compacted, which is presumably smaller than scheduler.compaction_input).
- box.stat.vinyl().scheduler.tasks_* is about dump/compaction tasks, in three categories, scheduler.tasks_inprogress (currently running), scheduler.tasks_completed (successfully completed) scheduler.tasks_failed (aborted due to errors).
- box.stat.vinyl().scheduler_dump_* has the amount of data from recent changes that has been dumped, including dump_time (total time spent by all worker threads performing dumps, in seconds), and dump_count (the count of completed dumps), dump_input and dump_output.

  A "dump" is explained in section Storing data with vinyl:

  > Sooner or later the number of elements in an LSM tree exceeds the L0 size and that is when L0 gets written to a file on disk (called a 'run') and then cleared for storing new elements. This operation is called a 'dump'.

  Thus it can be predicted that a dump will occur if the size of L0 (which is memory.level0) is approaching the maximum (which is regulator.dump_watermark) and a dump is not already in progress. In fact Tarantool will try to arrange a dump before this hard limit is reached.

  A dump will also occur during a snapshot operation.

Function box.snapshot

box.snapshot()
: Take a snapshot of all data and store it in memtx_dir/<latest-lsn>.snap. To take a snapshot, Tarantool first enters the delayed garbage collection mode for all data. In this mode, the Tarantool garbage collector will not remove files which were created before the snapshot started, it will not remove them 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. Users may restrict the speed by changing snap_io_rate_limit.

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 you need to have some extra free memory to run this command. 10% of memtx_memory is, on average, sufficient. This statement waits until a snapshot is taken and returns operation result.

Note: 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 this view is written to the snapshot file by a separate thread (the "Write Ahead Log" 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 checkpoint 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 replicated data is 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 a SIGUSR1 signal to the instance. 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.

Submodule box.tuple

Overview

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.

Index

Below is a list of all box.tuple functions.

Name	Use
box.tuple.new()	Create a tuple
#tuple_object	Count tuple fields
tuple_object:bsize()	Get count of bytes in a tuple
tuple_object[field-number]	Get a tuple's field by specifying a number
tuple_object[field-name]	Get a tuple's field by specifying a name
tuple_object[field-path]	Get a tuple's fields or parts by specifying a path
tuple_object:find()	Get the number of the first field matching the search value
tuple_object:findall()	Get the number of all fields matching the search value
tuple_object:transform()	Remove (and replace) a tuple's fields
tuple_object:unpack()	Get a tuple's fields
tuple_object:totable()	Get a tuple's fields as a table
tuple_object:tomap()	Get a tuple's fields as a table along with key:value pairs
tuple_object:pairs()	Prepare for iterating
tuple_object:update()	Update a tuple

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.

Parameters

- 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.tester:insert{
         >   33,
         >   tonumber('1'),
         >   tonumber64('2')
         > }:totable()
---
...
tarantool> t = box.tuple.new{'abc', 'def', 'ghi', 'abc'}
---
...
tarantool> t
---
- ['abc', 'def', 'ghi', 'abc']
...
```

object tuple_object

#<tuple_object>
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. With both the memtx storage engine and the vinyl storage engine the default maximum is one megabyte (memtx_max_tuple_size or vinyl_max_tuple_size). 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.

The value does not include the size of "struct tuple" (for the current size of this structure look in the tuple.h file in Tarantool's source code).

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 then there is one more byte to store a count of the number of fields, so bsize() returns 3*(1+3)+1. This is the same as the size of the string that msgpack.encode({'aaa','bbb','ccc'}) would return.

```
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>(field-name)

If t is a tuple instance, t['field-name'] will return the field named 'field-name' in the tuple. Fields have names if the tuple has been retrieved from a space that has an associated format.

Return field value.

Rtype lua-value

In the following example, a tuple named t is returned from replace and then the second field in t named 'field2' is returned.

```
tarantool> format = {}
---
...
tarantool> format[1] = {name = 'field1', type = 'unsigned'}
---
...
tarantool> format[2] = {name = 'field2', type = 'string'}
---
...
tarantool> s = box.schema.space.create('test', {format = format})
---
...
tarantool> pk = s:create_index('pk')
---
...
tarantool> t = s:replace{1, 'Я'}
---
...
tarantool> t['field2']
---
- Я
...
```

<tuple_object>(field-path)

If t is a tuple instance, t['path'] will return the field or subset of fields that are in path. path must be a well formed JSON specification. path may contain field names if the tuple has been retrieved from a space that has an associated format.

To prevent ambiguity, Tarantool first tries to interpret the request as tuple_object[field-number] or tuple_object[field-name]. If and only if that fails, Tarantool tries to interpret the request as tuple_object[field-path].

The path must be a well formed JSON specification, but it may be preceded by '.'. The '.' is a signal that the path acts as a suffix for the tuple.

The advantage of specifying a path is that Tarantool will use it to search through a tuple body and get only the tuple part, or parts, that are actually necessary.

In the following example, a tuple named t is returned from replace and then only the relevant part (in this case, matching a name) of a relevant field is returned. Namely: the second field, the sixth part, the value following 'value='.

```
tarantool> format = {}
---
...
tarantool> format[1] = {name = 'field1', type = 'unsigned'}
---
...
```

(continues on next page)

(continued from previous page)

```
tarantool> format[2] = {name = 'field2', type = 'array'}
---
...
tarantool> format[3] = {name = 'field4', type = 'string' }
---
...
tarantool> format[4] = {name = "[2][6]['пw']['Я']", type = 'string'}
---
...
tarantool> s = box.schema.space.create('test', {format = format})
---
...
tarantool> pk = s:create_index('pk')
---
...
tarantool> field2 = {1, 2, 3, "4", {5,6,7}, {пw={Я="п"}, key="V!", value="K!"}}
---
...
tarantool> t = s:replace{1, field2, "123456", "Not K!"}
---
...
tarantool> t["[2][6]['value']"]
---
- K!
...
```

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

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

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')
---
```

(continues on next page)

(continued from previous page)

```
- 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.

If the original tuple comes from a space that has been formatted with a format clause, the formatting will not be preserved for the result tuple.

Parameters

- 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:totable([start-field-number[, end-field-number]])

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

It is preferable to use t:totable() rather than t:unpack().

Return field(s) from the tuple

Rtype lua-table

In the following example, a tuple named t is created, 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:totable()
---
- ['Fld#1', 'Fld#2', 'Fld#3', 'Fld#4', 'Fld#5']
...
```

tuple_object:tomap([options])

A Lua table can have indexed values, also called key:value pairs. For example, here:

```
a = {}; a['field1'] = 10; a['field2'] = 20
```

a is a table with "field1: 10" and "field2: 20".

The tuple_object:totable() function only returns a table containing the values. But the tuple_object:tomap() function returns a table containing not only the values, but also the key:value pairs.

This only works if the tuple comes from a space that has been formatted with a format clause.

Parameters

- options (table) – the only possible option is names_only.

  If names_only is false or omitted (default), then all the fields will appear twice, first with numeric headings and second with name headings.

  If names_only is true, then all the fields will appear only once, with name headings.

Return field-number:value pair(s) and key:value pair(s) from the tuple

Rtype lua-table

In the following example, a tuple named t1 is returned from a space that has been formatted, then tables named t1map1 and t1map2 are produced from t1.

```
format = {{'field1', 'unsigned'}, {'field2', 'unsigned'}}
s = box.schema.space.create('test', {format = format})
s:create_index('pk',{parts={1,'unsigned',2,'unsigned'}})
t1 = s:insert{10, 20}
t1map = t1:tomap()
t1map_names_only = t1:tomap({names_only=true})
```

t1map will contain "1: 10", "2: 20", "field1: 10", "field2: 20".

t1map_names_only will contain "field1: 10", "field2: 20".

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({{operator, field_no, 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_no, value}, ...}) which updates a tuple in a space.

For details: see the description for operator, field_no, and value in the section box.space.space-name:update{key, format, {field_number, value}...).

If the original tuple comes from a space that has been formatted with a format clause, the formatting will be preserved for the result tuple.

Parameters

- operator (string) – operation type represented in string (e.g. '=' for 'assign new value')
- field_no (number) – what field the operation will apply to. The field number can be negative, meaning the position from the end of tuple. (#tuple + negative field number + 1)
- value (lua_value) – what value will be applied

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'}
---
```

(continues on next page)

(continued from previous page)

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

Example

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
lua_table = tuple:totable()                             -- 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:totable()
  scalar1, scalar2, scalar3 = tuple1:unpack()
  tuple2 = box.tuple.new(luatable1[1],luatable1[2],luatable1[3])
  field_number = tuple2:find('b')
  tuple2 = tuple2:transform(field_number, 1)
  return 'tuple2 = ' , tuple2 , ' # of bytes = ' , tuple2:bsize()
end
```

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

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

Submodule box.error

Overview

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.

Index

Below is a list of all box.error functions.

Name	Use
box.error()	Throw an error
box.error.last()	Get a description of the last error
box.error.clear()	Clear the record of errors
box.error.new()	Create an error but do not throw

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.

Parameters

- 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.

Parameters

- 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" – it 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/buildd/tarantool-1.7.0.252.g1654e31~precise/src/box/key_def.cc
...
tarantool> box.error.clear()
---
...
tarantool> box.error.last()
---
- null
...
```

box.error.new(code, errtext[, errtext ...])

Create an error object, but do not throw. This is useful when error information should be saved for later retrieval. The parameters are the same as for box.error(), see the description there.

Parameters

- code (number) – number of a pre-defined error
- errtext(s) (string) – part of the message which will accompany the error

Example:

```
tarantool> e = box.error.new{code = 555, reason = 'Arbitrary message'}
---
...
tarantool> e:unpack()
---
- type: ClientError
  code: 555
  message: Arbitrary message
  trace:
  - file: '[string "e = box.error.new{code = 555, reason = ''Arbit..."]'
    line: 1
...
```

Functions for transaction management

Overview

For general information and examples, see section Transaction control.

Observe the following rules when working with transactions:

Rule #1

The requests in a transaction must be sent to a 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.

Rule #2

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.

Rule #3

Requests which cause changes to the data definition – create, alter, drop, truncate – are only allowed with Tarantool version 2.1 or later. Data-definition requests which change an index or change a format, such as space_object:create_index() and space_object:format(), are not allowed inside transactions except as the first request after box.begin().

Index

Below is a list of all functions for transaction management.

Name	Use
box.begin()	Begin the transaction
box.commit()	End the transaction and save all changes
box.rollback()	End the transaction and discard all changes
box.savepoint()	Get a savepoint descriptor
box.rollback_to_savepoint()	Do not end the transaction and discard all changes made after a savepoint
box.atomic()	Execute a function, treating it as a transaction
box.on_commit()	Define a trigger that will be activated by box.commit
box.on_rollback()	Define a trigger that will be activated by box.rollback
is_in_txn()	State whether a transaction is in progress
is_in_txn()	State whether a transaction is in progress

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.

Possible errors: error if this operation is not permitted because there is already an active transaction. error if for some reason memory cannot be allocated.

box.commit()
End the transaction, and make all its data-change operations permanent.

Possible errors: error and abort the transaction in case of a conflict. error if the operation fails to write to disk. error if for some reason memory cannot be allocated.

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.sleep() or fiber.yield(), will have the same effect.

box.savepoint()
Return a descriptor of a savepoint (type = table), which can be used later by box.rollback_to_savepoint(savepoint). Savepoints can only be created while a transaction is active, and they are destroyed when a transaction ends.

Return savepoint table

Rtype Lua object

Return error if the savepoint cannot be set in absence of active transaction.

Possible errors: error if for some reason memory cannot be allocated.

box.rollback_to_savepoint(savepoint)
Do not end the transaction, but cancel all its data-change and box.savepoint() operations that were done after the specified savepoint.

Return error if the savepoint cannot be set in absence of active transaction.

Possible errors: error if the savepoint does not exist.

Example:

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

box.atomic(tx-function[, function-arguments])
Execute a function, acting as if the function starts with an implicit box.begin() and ends with an implicit box.commit() if successful, or ends with an implicit box.rollback() if there is an error.

Return the result of the function passed to atomic() as an argument.

Possible errors: any error that box.begin() and box.commit() can return.

box.on_commit(trigger-function[, old-trigger-function])
Define a trigger for execution when a transaction ends due to an event such as box.commit.

The trigger function may take an iterator parameter, as described in an example for this section.

The trigger function should not access any database spaces.

If the trigger execution fails and raises an error, the effect is severe and should be avoided – use Lua's pcall() mechanism around code that might fail.

box.on_commit() must be invoked within a transaction, and the trigger ceases to exist when the transaction ends.

Parameters

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

Return nil or function pointer

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

Details about trigger characteristics are in the triggers section.

Simple and useless example: this will display 'commit happened':

```
function f()
function f() print('commit happened') end
box.begin() box.on_commit(f) box.commit()
```

But of course there is more to it: the function parameter can be an ITERATOR.

The iterator goes through the effects of every request that changed a space during the transaction.

The iterator will have:

- an ordinal request number,
- the old value of the tuple before the request (this will be nil for an insert request),
- the new value of the tuple after the request (this will be nil for a delete request),
- and the id of the space.

Less simple more useful example: this will display the effects of two replace requests:

```
box.space.test:drop()
s = box.schema.space.create('test')
i = box.space.test:create_index('i')
function f(iterator)
  for request_number, old_tuple, new_tuple, space_id in iterator() do
    print('request_number ' .. tostring(request_number))
    print('  old_tuple ' .. tostring(old_tuple[1]) .. ' ' .. old_tuple[2])
    print('  new_tuple ' .. tostring(new_tuple[1]) .. ' ' .. new_tuple[2])
    print('  space_id ' .. tostring(space_id))
  end
end
s:insert{1,'-'}
box.begin() s:replace{1,'x'} s:replace{1,'y'} box.on_commit(f) box.commit()
```

The result will look like this:

```
tarantool> box.begin() s:replace{1,'x'} s:replace{1,'y'} box.on_commit(f) box.commit()
request_number 1
  old_tuple 1 -
  new_tuple 1 x
  space_id 517
request_number 2
  old_tuple 1 x
  new_tuple 1 y
  space_id 517
```

box.on_rollback(trigger-function[, old-trigger-function])

Define a trigger for execution when a transaction ends due to an event such as box.rollback.

The parameters and warnings are exactly the same as for box.on-commit.

box.is_in_txn()
If a transaction is in progress (for example the user has called box.begin and has not yet called either box.commit or box.rollback, return true. Otherwise return false.

Functions for SQL

The box module contains two functions related to SQL:

- box.internal.sql_create_function – for making Lua functions callable from SQL statements. This, or an SQL statement with the same effect, will be part of the documentation regarding SQL Plus Lua.
- box.execute – for making SQL statements callable from Lua functions.

Some SQL statements are illustrated in the SQL tutorial.

box.execute(sql-statement[, extra-parameters])
Execute the SQL statement contained in the sql-statement parameter.

Parameters

- sql-statement (string) – statement, which should conform to the rules for SQL grammar
- extra-parameters (table) – optional list for placeholders in the statement

Return depends on statement

There are two ways to pass extra parameters for box.execute():

- The first way is to concatenate strings. For example, this Lua script will insert 10 rows with different primary-key values into table t:

```
for i=1,10,1 do
  box.execute("insert into t values (" .. i .. ")")
end
```

- The second way is to put one or more placeholder "?" tokens inside the string, and pass a second argument, which must be a table containing values for each placeholder. For example these two requests are equivalent:

```
box.execute([[INSERT INTO tt VALUES (1,'x');]]);
x = {1,'x'}; box.execute([[INSERT INTO tt VALUES (?,?);]], x);
```

Since box.execute() is an invocation of a Lua function, it either causes an error message or returns a value.

For some statements the returned value will contain a field named rowcount. For example;

```
tarantool> box.execute([[INSERT INTO tt VALUES (8,8),(9,9);]])
tarantool> box.execute([[CREATE TABLE table1 (column1 INT PRIMARY key, column2 VARCHAR(10));
 ↪]])
---
- rowcount: 1
...
tarantool> box.execute([[INSERT INTO table1 VALUES (55,'Hello SQL world!');]])
---
- rowcount: 1
...
```

For statements that cause generation of values for PRIMARY KEY AUTOINCREMENT columns, there will also be a field named "autoincrement_ids".

For SELECT statements the returned value will contain a field named metadata (a table with column names and data types) and a field named "rows" (a table with the result set). For example:

```
tarantool> box.execute([[SELECT * FROM table1 WHERE column1 > 0;]])
---
- metadata:
  - name: COLUMN1
    type: integer
  - name: COLUMN2
    type: string
  rows:
  - [55, 'Hello SQL world!']
...
```

The result structure contains Tarantool/NoSQL data type names in MsgPack format. For example, for a statement SELECT "x" FROM t WHERE "x"=5; where "x" is an integer column and there is one row, the raw data for the result set will look like this:

```
dd 00 00 00 01              1-element array
82                          2-element map (for metadata + rows)
a8 6d 65 74 61 64 61 74 61      string = "metadata"
91                          1-element array (for column count)
82                          2-element map (for name + type)
a4 6e 61 6d 65                  string = "name"
a1 78                       string = "x"
a4 74 79 70 6                   string = "type"
a7 69 6e 74 65 67 65 72         string = "integer"
a4 72 6f 77 73                  string = "rows"
91                          1-element array (for row count)
91                          1-element array (for field count)
05                          contents
```

The order of components within a map is not guaranteed.

Alternative: if you are using the Tarantool server as a client, you can switch languages thus:

```
\set language sql
\set delimiter ;
```

Afterwards, you can enter any SQL statement directly without needing box.execute().

There is also an execute() function available via module net.box, for example after conn = net_box.connect(url-string) one can say conn:execute(sql-statement]).

4.2.2 Module buffer

The buffer module returns a dynamically resizable buffer which is solely for use as an option for methods of the net.box module.

Ordinarily the net.box methods return a Lua table. If a buffer option is used, then the net.box methods return a raw MsgPack string. This saves time on the server, if the client application has its own routine for decoding MsgPack strings.

buffer.ibuf()

Return a descriptor of a buffer.

Rtype cdata

Example:

Assume a Tarantool server is listening on farhost:3301. Assume it has a space T with one tuple: 'ABCDE', 12345. In this example we start up a server on localhost:3302 and then use net.box routines to connect to farhost. Then we create a buffer, and use it as an option for a conn.space...select() call. The result will be in MsgPack format. To show this, we will use msgpack.decode_unchecked() on ibuf.rpos (the "read position" of the buffer). Thus we do not decode on the remote server, but we do decode on the local server.

```
box.cfg{listen=3302}
buffer = require('buffer')
ibuf = buffer.ibuf()
net_box = require('net.box')
conn = net_box.connect('farhost:3301')
buffer = require('buffer')
conn.space.T:select({},{buffer=ibuf})
msgpack = require('msgpack')
msgpack.decode_unchecked(ibuf.rpos)
```

The result of the final request looks like this:

```
tarantool> msgpack.decode_unchecked(ibuf.rpos)
---
- {48: [['ABCDE', 12345]]}
- 'cdata<char *>: 0x7f97ba10c041'
...
```

Note: Before Tarantool version 1.7.7, the function to use for this case is msgpack.ibuf_decode(ibuf.rpos). Starting with Tarantool version 1.7.7, ibuf_decode is deprecated.

Module buffer and skip-header

The example in the previous section

```
tarantool> msgpack.decode_unchecked(ibuf.rpos)
---
- {48: [['ABCDE', 12345]]}
- 'cdata<char *>: 0x7f97ba10c041'
...
```

showed that, ordinarily, the response from net.box includes a header – 48 (hexadecimal 30) is the key for IPROTO_DATA. But in some situations, for example when passing the buffer to a C function that expects a MsgPack byte array without a header, the header can be skipped. This is done by specifying skip-header=true as an option to conn.space.space-name:select{...} or conn.space.space-name:insert{...} or conn.space.space-name:replace{...} or conn.space.space-name:update{...} or conn.space.space-name:upsert{...} or conn.space.space-name:delete{...}. The default is skip-header=false.

Now here is the same example, except that skip_header=true is used.

```
box.cfg{listen=3302}
buffer = require('buffer')
ibuf = buffer.ibuf()
net_box = require('net.box')
conn = net_box.connect('farhost:3301')
```

(continues on next page)

(continued from previous page)

```
buffer = require('buffer')
conn.space.T:select({},{buffer=ibuf, skip_header=true})
msgpack = require('msgpack')
msgpack.decode_unchecked(ibuf.rpos)
```

The result of the final request looks like this:

```
tarantool>        msgpack.decode_unchecked(ibuf.rpos)
---
- [['ABCDE', 12345]]
- 'cdata<char *>: 0x7f8fd102803f'
...
```

Notice that the IPROTO_DATA header (48) is gone.

The result is still inside an array, as is clear from the fact that it is shown inside square brackets. It is possible to skip the array header too, with msgpack.decode_array_header().

4.2.3 Module clock

Overview

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.

Index

Below is a list of all clock functions.

Name	Use
clock.time() clock.realtime()	Get the wall clock time in seconds
clock.time64() clock.realtime64()	Get the wall clock time in nanoseconds
clock.monotonic()	Get the monotonic time in seconds
clock.monotonic64()	Get the monotonic time in nanoseconds
clock.proc()	Get the processor time in seconds
clock.proc64()	Get the processor time in nanoseconds
clock.thread()	Get the thread time in seconds
clock.thread64()	Get the thread time in nanoseconds
clock.bench()	Measure the time a function takes within a processor

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 the standard Lua function 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 the transaction processor thread started.

Rtype number or number64

Example:

```
-- This will print seconds in the thread since the start.
clock = require('clock')
print(clock.thread64())
```

clock.bench(function[, ...])

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) – function or function reference
- ... – whatever values are required by the function.

Return table. first element - seconds of CPU time, second element - whatever the function returns.

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)
```

4.2.4 Module console

Overview

The console module allows one Tarantool instance to access another Tarantool instance, and allows one Tarantool instance to start listening on an admin port.

Index

Below is a list of all console functions.

Name	Use
console.connect()	Connect to an instance
console.listen()	Listen for incoming requests
console.start()	Start the console
console.ac()	Set the auto-completion flag
console.delimiter()	Set a delimiter

console.connect(uri)

Connect to the instance 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 instance, in interactive mode, to access another Tarantool instance. Subsequent requests will appear to be handled locally, but in reality the requests are being sent to the remote instance and the local instance is acting as a client. Once connection is successful, the prompt will change and subsequent requests are sent to, and executed on, the remote instance. Results are displayed on the local instance. To return to local mode, enter control-D.

If the Tarantool instance 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 instance is done with user name = 'guest'. The remote instance could allow for this by granting at least one privilege: box.schema.user.grant('guest', 'execute','universe').

Parameters

- uri (string) – the URI of the remote instance

Return nil

Possible errors: the connection will fail if the target Tarantool instance 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 instance 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.

Parameters

- uri (string) – the URI of the local instance

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 instance with tarantool initialization file there is no console. This can be remedied by adding these lines at the end of the initialization file:

```
local 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 or the user is using Tarantool via console.connect(), then hitting the TAB key may cause tarantool to complete a word automatically. The default auto-completion value is true.

console.delimiter(marker)
: Set a custom end-of-request marker for Tarantool console.

  The default end-of-request marker is a newline (line feed). Custom markers are 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). Nonetheless for special needs, or for entering multi-line requests in older Tarantool versions, you can change the end-of-request marker. As a result, newline alone is not treated as end of request.

  To go back to normal mode, say: console.delimiter('')<marker>

  Parameters

  - marker (string) – a custom end-of-request marker for Tarantool console

  Example:

```
tarantool> console = require('console'); console.delimiter('!')
---
...
tarantool> function f ()
         > statement_1 = 'a'
         > statement_2 = 'b'
         > end!
---
...
tarantool> console.delimiter('')!
---
...
```

4.2.5 Module crypto

Overview

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

Index

Below is a list of all crypto functions.

Name	Use
crypto.cipher.{algorithm}.{cipher_mode}.encrypt()	Encrypt a string
crypto.cipher.{algorithm}.{cipher_mode}.decrypt()	Decrypt a string
crypto.digest.{algorithm}()	Get a digest

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, read the article about Encryption Modes

Example:

```
_16byte_iv='1234567890123456'
_16byte_pass='1234567890123456'
e=crypto.cipher.aes128.cbc.encrypt('string', _16byte_pass, _16byte_iv)
crypto.cipher.aes128.cbc.decrypt(e, _16byte_pass, _16byte_iv)
```

crypto.digest.{dss|dss1|md4|md5|mdc2|ripemd160}(string)
crypto.digest.{sha1|sha224|sha256|sha384|sha512}(string)

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

- dss - dss (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 - ripemd (with 160-bit binary strings using RIPEMD-160)
- 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')
```

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

-- 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()
```

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('x',b32,b16)==digest.aes256cbc.encrypt('x',b32,b16)
crypto.digest.md4('string') == digest.md4('string')
crypto.digest.md5('string') == digest.md5('string')
crypto.digest.sha1('string') == digest.sha1('string')
crypto.digest.sha224('string') == digest.sha224('string')
crypto.digest.sha256('string') == digest.sha256('string')
crypto.digest.sha384('string') == digest.sha384('string')
crypto.digest.sha512('string') == digest.sha512('string')
```

4.2.6 Module csv

Overview

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 (default: comma) – single-byte character to designate end-of-field
- quote_char = string (default: quote mark) – single-byte character to designate encloser of string
- chunk_size = number (default: 4096) – number of characters to read at once (usually for file-IO efficiency)
- skip_head_lines = number (default: 0) – number of lines to skip at the start (usually for a header)

Index

Below is a list of all csv functions.

Name	Use
csv.load()	Load a CSV file
csv.dump()	Transform input into a CSV-formatted string
csv.iterate()	Iterate over CSV records

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.

Parameters

- 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()
---
- 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().

Parameters

- 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')
---
```

(continues on next page)

(continued from previous page)

```
...
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. Use of an iterator is strongly recommended if the amount of data is large (ten or more megabytes).

Parameters

- 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
...
```

4.2.7 Module decimal

The decimal module has functions for working with exact numbers. This is important when numbers are large or even the slighest inaccuracy is unacceptable. For example Lua calculates 0.16666666666667 * 6 with floating-point so the result is 1. But with the decimal module (using decimal.new to convert the number to decimal type) decimal.new('0.16666666666667') * 6 is 1.00000000000002.

To construct a decimal number, bring in the module with require('decimal') and then use decimal.new(n) or any function in the decimal module: abs(n) exp(n) ln(n) log10(n) new(n) precision(n) rescale(decimal-number, new-scale) scale(n) sqrt(n) trim(decimal-number), where n can be a string or a non-decimal number or a decimal number. If it is a string or a non-decimal number, Tarantool converts it to a decimal number before working with it. It is best to construct from strings, and to convert back to strings after calculations, because Lua numbers have only 15 digits of precision. Decimal numbers have 38 digits of precision, that is, the total number of digits before and after the decimal point can be 38. Tarantool supports the usual arithmetic and comparison operators + - * / % ^ < > <= >= ~= ==. If an operation has both decimal and non-decimal operands, then the non-decimal operand is converted to decimal before the operation happens.

Use tostring(decimal-number) to convert back to a string.

A decimal operation will fail if overflow happens (when a number is greater than 10^38 - 1 or less than -10^38 - 1). A decimal operation will fail if arithmetic is impossible (such as division by zero or square root of minus 1). A decimal operation will not fail if rounding of post-decimal digits is necessary to get 38-digit precision.

decimal.abs(string-or-number-or-decimal-number)
: Returns absolute value of a decimal number. For example if a is -1 then decimal.abs(a) returns 1.

decimal.exp(string-or-number-or-decimal-number)
: Returns e raised to the power of a decimal number. For example if a is 1 then decimal.exp(a) returns 2.71828182845904523536028747135266249 78. Compare math.exp(1) from the Lua math library, which returns 2.718281828459.

decimal.ln(string-or-number-or-decimal-number)
: Returns natural logarithm of a decimal number. For example if a is 1 then decimal.ln(a) returns 0.

decimal.log10(string-or-number-or-decimal-number)
: Returns base-10 logarithm of a decimal number. For example if a is 100 then decimal.log10(a) returns 2.

decimal.new(string-or-number-or-decimal-number)
: Returns the value of the input as a decimal number. For example if a is 1E-1 then decimal.new(a) returns 0.1.

decimal.precision(string-or-number-or-decimal-number)
: Returns the number of digits in a decimal number. For example if a is 123.4560 then decimal. precision(a) returns 7.

decimal.rescale(decimal-number, new-scale)
: Returns the number after possible rounding or padding. If the number of post-decimal digits is greater than new-scale, then rounding occurs. The rounding rule is: round half away from zero. If the number of post-decimal digits is less than new-scale, then padding of zeros occurs. For example if a is -123.4550 then decimal.rescale(a, 2) returns -123.46, and decimal.rescale(a, 5) returns -123.45500.

decimal.scale(string-or-number-or-decimal-number)
: Returns the number of post-decimal digits in a decimal number. For example if a is 123.4560 then decimal.scale(a) returns 4.

decimal.sqrt(string-or-number-or-decimal-number)
: Returns the square root of a decimal number. For example if a is 2 then decimal.sqrt(a) returns 1.41421356237309504880168872420969807 86.

decimal.trim(decimal-number)
: Returns a decimal number after possible removing of trailing post-decimal zeros. For example if a is 2.20200 then decimal.trim(a) returns 2.202.

4.2.8 Module digest

Overview

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

Index

Below is a list of all digest functions.

Name	Use
digest.aes256cbc.encrypt()	Encrypt a string using AES
digest.aes256cbc.decrypt()	Decrypt a string using AES
digest.md4()	Get a digest made with MD4
digest.md4_hex()	Get a hexadecimal digest made with MD4
digest.md5()	Get a digest made with MD5
digest.md5_hex()	Get a hexadecimal digest made with MD5
digest.pbkdf2()	Get a digest made with PBKDF2
digest.sha1()	Get a digest made with SHA-1
digest.sha1_hex()	Get a hexadecimal digest made with SHA-1
digest.sha224()	Get a 224-bit digest made with SHA-2
digest.sha224_hex()	Get a 56-byte hexadecimal digest made with SHA-2
digest.sha256()	Get a 256-bit digest made with SHA-2
digest.sha256_hex()	Get a 64-byte hexadecimal digest made with SHA-2
digest.sha384()	Get a 384-bit digest made with SHA-2
digest.sha384_hex()	Get a 96-byte hexadecimal digest made with SHA-2
digest.sha512()	Get a 512-bit digest made with SHA-2
digest.sha512_hex()	Get a 128-byte hexadecimal digest made with SHA-2
digest.base64_encode()	Encode a string to Base64
digest.base64_decode()	Decode a Base64-encoded string
digest.urandom()	Get an array of random bytes
digest.crc32()	Get a 32-bit checksum made with CRC32
digest.crc32.new()	Initiate incremental CRC32
digest.guava()	Get a number made with a consistent hash
digest.murmur()	Get a digest made with MurmurHash
digest.murmur.new()	Initiate incremental MurmurHash

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.pbkdf2(string, salt[, iterations[, digest-length]])
: Returns binary string = digest made with PBKDF2. For effective encryption the iterations value should be at least several thousand. The digest-length value determines the length of the resulting binary string.

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 tring = digest made with SHA-2.

digest.sha512_hex(string)
: Returns 128-byte string = hexadecimal of a digest calculated with sha512.

digest.base64_encode()
: Returns base64 encoding from a regular string.

  The possible options are:

  - nopad – result must not include '=' for padding at the end,
  - nowrap – result must not include line feed for splitting lines after 72 characters,
  - urlsafe – result must not include '=' or line feed, and may contain '-' or '_' instead of '+' or '/' for positions 62 and 63 in the index table.

  Options may be true or false, the default value is false.

  For example:

```
digest.base64_encode(string_variable,{nopad=true})
```

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 Cyclic Redundancy Check polynomial value: 0x1EDC6F41 / 4812730177. (Other settings are: input = reflected, output = reflected, initial value = 0xFFFFFFFF, final xor value = 0x0.) 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.

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())
```

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.sha1('^S^e^c^ret Wordpass')}
         >   return 'OK'
         > end
---
...
tarantool> function password_check(password)
         >   local t = box.space.tester:select{12345}
         >   if digest.sha1(password) == t[2] then
         >     return 'Password is valid'
         >   else
         >     return 'Password is not valid'
         >   end
         > end
---
...
tarantool> password_insert()
---
- 'OK'
...
```

If a later user calls the password_check() function and enters the wrong password, the result is an error.

```
tarantool> password_check('Secret Password')
---
- 'Password is not valid'
...
```

4.2.9 Module errno

Overview

The errno module is typically used within a function or within a Lua program, in association with a module whose functions can return operating-system errors, such as fio.

Index

Below is a list of all errno functions.

Name	Use
errno()	Get an error number for the last OS-related function
errno.strerror()	Get an error message for the corresponding error number

errno()
: Return an error number for the last operating-system-related function, or 0. To invoke it, simply say errno(), without the module name.

  Rtype integer

errno.strerror([code])
: Return a string, given an error number. The string will contain the text of the conventional error message for the current operating system. If code is not supplied, the error message will be for the last operating-system-related function, or 0.

  Parameters

  - code (integer) – number of an operating-system error

  Rtype string

Example:

This function displays the result of a call to fio.open() which causes error 2 (errno.ENOENT). The display includes the error number, the associated error string, and the error name.

```
tarantool> function f()
         >   local fio = require('fio')
         >   local errno = require('errno')
         >   fio.open('no_such_file')
         >   print('errno() = ' .. errno())
         >   print('errno.strerror() = ' .. errno.strerror())
         >   local t = getmetatable(errno).__index
         >   for k, v in pairs(t) do
         >     if v == errno() then
         >       print('errno() constant = ' .. k)
         >     end
         >   end
         > end
---
...

tarantool> f()
errno() = 2
errno.strerror() = No such file or directory
errno() constant = ENOENT
---
...
```

To see all possible error names stored in the errno metatable, say getmetatable(errno) (output abridged):

```
tarantool> getmetatable(errno)
---
- __newindex: 'function: 0x41666a38'
  __call: 'function: 0x41666890'
  __index:
  ENOLINK: 67
  EMSGSIZE: 90
  EOVERFLOW: 75
  ENOTCONN: 107
  EFAULT: 14
  EOPNOTSUPP: 95
  EEXIST: 17
  ENOSR: 63
  ENOTSOCK: 88
  EDESTADDRREQ: 89
  <...>
...
```

4.2.10 Module fiber

Overview

With the fiber module, you can:

- create, run and manage fibers,
- send and receive messages between different processes (i.e. different connections, sessions, or fibers) via channels, and
- use a synchronization mechanism for fibers, similar to "condition variables" and similar to operating-system functions such as pthread_cond_wait() plus pthread_cond_signal().

Index

Below is a list of all fiber functions and members.

Name	Use
fiber.create()	Create and start a fiber
fiber.new()	Create but do not start a fiber
fiber.self()	Get a fiber object
fiber.find()	Get a fiber object by ID
fiber.sleep()	Make a fiber go to sleep
fiber.yield()	Yield control
fiber.status()	Get the current fiber's status
fiber.info()	Get information about all fibers
fiber.kill()	Cancel a fiber
fiber.testcancel()	Check if the current fiber has been cancelled
fiber_object:id()	Get a fiber's ID
fiber_object:name()	Get a fiber's name
fiber_object:name(name)	Set a fiber's name
fiber_object:status()	Get a fiber's status

Continued on next page

Table 2 – continued from previous page

Name	Use
fiber_object:cancel()	Cancel a fiber
fiber_object.storage	Local storage within the fiber
fiber_object:set_joinable()	Make it possible for a new fiber to join
fiber_object:join()	Wait for a fiber's state to become 'dead'
fiber.time()	Get the system time in seconds
fiber.time64()	Get the system time in microseconds
fiber.channel()	Create a communication channel
channel_object:put()	Send a message via a channel
channel_object:close()	Close a channel
channel_object:get()	Fetch a message from a channel
channel_object:is_empty()	Check if a channel is empty
channel_object:count()	Count messages in a channel
channel_object:is_full()	Check if a channel is full
channel_object:has_readers()	Check if an empty channel has any readers waiting
channel_object:has_writers()	Check if a full channel has any writers waiting
channel_object:is_closed()	Check if a channel is closed
fiber.cond()	Create a condition variable
cond_object:wait()	Make a fiber go to sleep until woken by another fiber
cond_object:signal()	Wake up a single fiber
cond_object:broadcast()	Wake up all fibers

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 is created with fiber.new() or 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 Lua 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.

Parameters

- 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.new(function[, function-arguments])

Create but do not start a fiber: the fiber is created but does not begin to run immediately – it starts after the fiber creator (that is, the job that is calling fiber.new()) yields, under transaction control. The initial fiber state is 'suspended'. Thus fiber.new() differs slightly from fiber.create().

Ordinarily fiber.new() is used in conjunction with fiber_object:set_joinable() and fiber_object:join().

Parameters

- 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.new(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)

Parameters

- 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 scheduler and sleep for the specified number of seconds. Only the current fiber can be made to sleep.

Parameters

- time – number of seconds to sleep.

Example:

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

fiber.yield()

Yield control to the scheduler. Equivalent to fiber.sleep(0), except that fiber.sleep(0) depends on a timer, fiber.yield() does not.

Example:

```
tarantool> fiber.yield()
---
...
```

fiber.status([fiber_object])

Return the status of the current fiber. Or, if optional fiber_object is passed, return the status of the specified 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().

Parameters

- 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()) -- kill self, may make program end
---
- 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()

Parameters

- fiber_object – generally this is an object referenced in the return from fiber.create or fiber.self or fiber.find

Return id of the fiber.

Rtype number

fiber.self():id() can also be expressed as fiber.id().

Example:

```
tarantool> fiber_object = fiber.self()
---
...
tarantool> fiber_object:id()
---
- 101
...
```

fiber_object:name()

Parameters

- fiber_object – generally this is an object referenced in the return from fiber.create or fiber.self or fiber.find

Return name of the fiber.

Rtype string

fiber.self():name() can also be expressed as fiber.name().

Example:

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

fiber_object:name(name)

Change the fiber name. By default a 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.

Parameters

- fiber_object – generally this is an object referenced in the return from fiber.create or fiber.self or fiber.find
- 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.

Parameters

- fiber_object – generally this is an object referenced in the return from fiber.create or fiber.self or fiber.find

Return the status of fiber. One of: “dead”, “suspended”, or “running”.

Rtype string

fiber.self():status(can also be expressed as fiber.status().

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.

Parameters

- fiber_object – generally this is an object referenced in the return from fiber.create or fiber.self or fiber.find

Return nil

Possible errors: cancel is not permitted for the specified fiber object.

Example:

```
tarantool> fiber.self():cancel() -- kill self, may make program end
---
- 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 Lua garbage collector will mark or free the local storage when fiber_object:cancel() happens.

Example:

```
tarantool> fiber = require('fiber')
---
...
tarantool> function f () fiber.sleep(1000); end
---
...
tarantool> fiber_function = fiber.create(f)
---
...
tarantool> fiber_function.storage.str1 = 'string'
---
...
tarantool> fiber_function.storage['str1']
---
- string
```

(continues on next page)

(continued from previous page)

```
...
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_object:set_joinable(true_or_false)

fiber_object:set_joinable(true) makes a fiber joinable; fiber_object:set_joinable(false) makes a fiber not joinable; the default is false.

A joinable fiber can be waited for, with fiber_object:join().

Best practice is to call fiber_object:set_joinable() before the fiber function begins to execute, because otherwise the fiber could become 'dead' before fiber_object:set_joinable() takes effect. The usual sequence could be:

1. Call fiber.new() instead of fiber.create() to create a new fiber_object.

   Do not yield at this point, because that will cause the fiber function to begin.

2. Call fiber_object:set_joinable(true) to make the new fiber_object joinable.

   Now it is safe to yield.

3. Call fiber_object:join().

   Usually fiber_object:join() should be called, otherwise the fiber's status may become 'suspended' when the fiber function ends, instead of 'dead'.

   Parameters

   - true_or_false – the boolean value that changes the set_joinable flag

   Return nil

Example:

The result of the following sequence of requests is:

- the global variable d will be 6 (which proves that the function was not executed until after d was set to 1, when fiber.sleep(1) caused a yield);
- fiber.status(fi2) will be 'suspended' (which proves that after the function was executed the fiber status did not change to 'dead').

```
fiber=require('fiber')
d=0
function fu2() d=d+5 end
fi2=fiber.new(fu2) fi2:set_joinable(true) d=1 fiber.sleep(1)
print(d)
fiber.status(fi2)
```

fiber_object:join()

"Join" a joinable fiber. That is, let the fiber's function run and wait until the fiber's status is 'dead' (normally a status becomes 'dead' when the function execution finishes). Joining will cause

a yield, therefore, if the fiber is currently in a suspended state, execution of its fiber function will resume.

This kind of waiting is more convenient than going into a loop and periodically checking the status; however, it works only if the fiber was created with fiber.new() and was made joinable with fiber_object:set_joinable().

Return two values. The first value is boolean. If the first value is true, then the join succeeded because the fiber's function ended normally and the second result has the return value from the fiber's function. If the first value is false, then the join succeeded because the fiber's function ended abnormally and the second result has the details about the error, which one can unpack in the same way that one unpacks a pcall result.

Rtype boolean +result type, or boolean + struct error

Example:

The result of the following sequence of requests is:

- the first fiber.status() call returns 'suspended',
- the join() call returns true,
- the elapsed time is usually 5 seconds, and
- the second fiber.status() call returns 'dead'.

This proves that the join() does not return until the function – which sleeps 5 seconds – is 'dead'.

```
fiber=require('fiber')
function fu2() fiber.sleep(5) end
fi2=fiber.new(fu2) fi2:set_joinable(true)
start_time = os.time()
fiber.status(fi2)
fi2:join()
print('elapsed = ' .. os.time() - start_time)
fiber.status(fi2)
```

fiber.time()

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
...
```

Example

Make the function which will be associated with the fiber. This function contains an infinite loop. 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 true 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)
```

(continues on next page)

(continued from previous page)

```
# 102 .  dead . gvar= 421
---
...
```

Channels

Call fiber.channel() to allocate space and get a channel object, which will be called channel for examples in this section.

Call the other routines, via channel, to send messages, receive messages, or check channel status.

Message exchange is synchronous. The Lua garbage collector will mark or free the channel 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.

  Parameters
  - capacity (int) – the maximum number of slots (spaces for channel:put messages) that can be in use at once. The default is 0.

  Return new channel.

  Rtype userdata, possibly including the string "channel . . . ".

object channel_object

channel_object:put(message[, timeout])
: Send a message using a channel. If the channel is full, channel:put() waits until there is a free slot in the channel.

  Parameters
  - message (lua-value) – what will be sent, usually a string or number or table
  - timeout (number) – maximum number of seconds to wait for a slot to become free

  Return If timeout is specified, and there is no free slot in the channel for the duration of the timeout, then the return value is false. If the channel is closed, then the return value is false. Otherwise, the return value is true, indicating success.

  Rtype boolean

channel_object:close()
: Close the channel. All waiters in the channel will stop waiting. All following channel:get() operations will return nil, and all following channel:put() operations will return false.

channel_object:get([timeout])
: Fetch and remove a message from a channel. If the channel is empty, channel:get() waits for a message.

  Parameters
  - timeout (number) – maximum number of seconds to wait for a message

  Return If timeout is specified, and there is no message in the channel for the duration of the timeout, then the return value is nil. If the channel is closed, then the return

value is nil. Otherwise, the return value is the message placed on the channel by channel:put().

Rtype usually string or number or table, as determined by channel:put

channel_object:is_empty()
Check whether the channel is empty (has no messages).

Return true if the channel is empty. Otherwise false.

Rtype boolean

channel_object:count()
Find out how many messages are in the channel.

Return the number of messages.

Rtype number

channel_object:is_full()
Check whether the channel is full.

Return true if the channel is full (the number of messages in the channel equals the number of slots so there is no room for a new message). Otherwise false.

Rtype boolean

channel_object:has_readers()
Check whether readers are waiting for a message because they have issued channel:get() and the channel is empty.

Return true if readers are waiting. Otherwise false.

Rtype boolean

channel_object:has_writers()
Check whether writers are waiting because they have issued channel:put() and the channel is full.

Return true if writers are waiting. Otherwise false.

Rtype boolean

channel_object:is_closed()

Return true if the channel is already closed. Otherwise false.

Rtype boolean

Example

This example should give a rough idea of what some functions for fibers should look like. It's assumed that the functions would be referenced in fiber.create().

```
fiber = require('fiber')
channel = fiber.channel(10)
function consumer_fiber()
    while true do
        local task = channel:get()
        ...
    end
end

function consumer2_fiber()
```

(continues on next page)

(continued from previous page)

```
    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

        ...
        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
```

Condition variables

Call fiber.cond() to create a named condition variable, which will be called 'cond' for examples in this section.

Call cond:wait() to make a fiber wait for a signal via a condition variable.

Call cond:signal() to send a signal to wake up a single fiber that has executed cond:wait().

Call cond:broadcast() to send a signal to all fibers that have executed cond:wait().

fiber.cond()
Create a new condition variable.

Return new condition variable.

Rtype Lua object

object cond_object

cond_object:wait([timeout])
Make the current fiber go to sleep, waiting until another fiber invokes the signal() or broadcast() method on the cond object. The sleep causes an implicit fiber.yield().

Parameters

- timeout – number of seconds to wait, default = forever.

Return If timeout is provided, and a signal doesn't happen for the duration of the timeout, wait() returns false. If a signal or broadcast happens, wait() returns true.

Rtype boolean

cond_object:signal()
Wake up a single fiber that has executed wait() for the same variable.

Rtype nil

cond_object:broadcast()
Wake up all fibers that have executed wait() for the same variable.

Rtype nil

Example

Assume that a tarantool instance is running and listening for connections on localhost port 3301. Assume that guest users have privileges to connect. We will use the tarantoolctl utility to start two clients.

On terminal #1, say

```
$ tarantoolctl connect '3301'
tarantool> fiber = require('fiber')
tarantool> cond = fiber.cond()
tarantool> cond:wait()
```

The job will hang because cond:wait() – without an optional timeout argument – will go to sleep until the condition variable changes.

On terminal #2, say

```
$ tarantoolctl connect '3301'
tarantool> cond:signal()
```

Now look again at terminal #1. It will show that the waiting stopped, and the cond:wait() function returned true.

This example depended on the use of a global conditional variable with the arbitrary name cond. In real life, programmers would make sure to use different conditional variable names for different applications.

4.2.11 Module fio

Overview

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.

The fio module contains:

- functions for common pathname manipulations,
- functions for directory or file existence and type checks,
- functions for common file manipulations, and
- constants which are the same as POSIX flag values (for example fio.c.flag.O_RDONLY = POSIX O_RDONLY).

Index

Below is a list of all fio functions and members.

Common pathname manipulations

fio.pathjoin(partial-string[, partial-string ...])
Concatenate partial string, separated by '/' to form a path name.

Parameters

- 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.

Parameters

- 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).

Parameters

- 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/'
```

fio.abspath(file-name)

Given a final part (the file name), return the full path name.

Parameters

- file-name (string) – file name

Return directory name, that is, path name including file name.

Rtype string

Example:

```
tarantool> fio.abspath('my.lua')
---
- 'path/to/my.lua'
...
```

Directory or file existence and type checks

Functions in this section are similar to some Python os.path functions.

path.exists(path-name)

Parameters

- path-name (string) – path to directory or file.

Return true if path-name refers to a directory or file that exists and is not a broken symbolic link; otherwise false

Rtype boolean

path.is_dir(path-name)

Parameters

- path-name (string) – path to directory or file.

Return true if path-name refers to a directory; otherwise false

Rtype boolean

path.is_file(path-name)

Parameters

- path-name (string) – path to directory or file.

Return true if path-name refers to a file; otherwise false

Rtype boolean

path.is_link(path-name)

Parameters

- path-name (string) – path to directory or file.

Return true if path-name refers to a symbolic link; otherwise false

Rtype boolean

path.lexists(path-name)

Parameters

- path-name (string) – path to directory or file.

Return true if path-name refers to a directory or file that exists or is a broken symbolic link; otherwise false

Rtype boolean

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.

Parameters

- 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.

Parameters

- path-name (string) – path name of file.

Return (If no error) table of fields which describe the file's block size, creation time, size, and other attributes. (If error) two return values: null, error message.

Rtype table.

Additionally, the result of fio.stat('file-name') will include methods equivalent to POSIX macros:

- is_blk() = POSIX macro S_ISBLK,
- is_chr() = POSIX macro S_ISCHR,
- is_dir() = POSIX macro S_ISDIR,
- is_fifo() = POSIX macro S_ISFIFO,
- is_link() = POSIX macro S_ISLINK,
- is_reg() = POSIX macro S_ISREG,
- is_sock() = POSIX macro S_ISSOCK.

For example, fio.stat('/'):is_dir() will return true.

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.

Parameters

- 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 (If no error) true. (If error) two return values: false, error message.

Rtype boolean

Example:

```
tarantool> fio.mkdir('/etc')
---
- false
...
```

fio.chdir(path-name)

Change working directory. For details type man 2 chdir.

Parameters

- path-name (string) – path of directory.

Return (If success) true. (If failure) false.

Rtype boolean

Example:

```
tarantool> fio.chdir('/etc')
---
- true
...
```

fio.listdir(path-name)

List files in directory. The result is similar to the result from the ls command.

Parameters

- path-name (string) – path of directory.

Return (If no error) a list of files. (If error) two return values: null, error message.

Rtype table

Example:

```
tarantool> fio.listdir('/usr/lib/tarantool')
---
- - mysql
...
```

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.

Parameters

- 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
...
```

fio.tempdir()

Return the name of a directory that can be used to store temporary files.

Example:

```
tarantool> fio.tempdir()
---
- /tmp/lG31e7
...
```

fio.cwd()
Return the name of the current working directory.

Example:

```
tarantool> fio.cwd()
---
- /home/username/tarantool_sandbox
...
```

fio.copytree(from-path, to-path)
Copy everything in the from-path, including subdirectory contents, to the to-path. The result is similar to the result that one gets from the cp -r command. The to-path should be empty.

Parameters

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

Return (If no error) true. (If error) two return values: false, error message.

Rtype boolean

Example:

```
tarantool> fio.copytree('/home/original','/home/archives')
---
- true
...
```

fio.mktree(path-name)
Create the path, including subdirectories, but without file contents. The result is similar to the result that one gets from the mkdir command.

Parameters

- path-name (string) – path-name.

Return (If no error) true. (If error) two return values: false, error message.

Rtype boolean

Example:

```
tarantool> fio.mktree('/home/archives')
---
- true
...
```

fio.rmtree(path-name)
Remove the directory indicated by path-name, including subdirectories. The result is similar to the result that one gets from the rmdir command, recursively. The directory must be empty.

Parameters

- path-name (string) – path-name.

Return (If no error) true. (If error) two return values: null, error message.

Rtype boolean

Example:

```
tarantool> fio.rmtree('/home/archives')
---
- true
...
```

fio.link(src, dst)
fio.symlink(src, dst)
fio.readlink(src)
fio.unlink(src)

Functions to create and delete links. For details type man readlink, man 2 link, man 2 symlink, man 2 unlink.

Parameters

- src (string) – existing file name.
- dst (string) – linked name.

Return (If no error) fio.link and fio.symlink and fio.unlink return true, fio.readlink returns the link value. (If error) two return values: false|null, error message.

Example:

```
tarantool> fio.link('/home/username/tmp.txt', '/home/username/tmp.txt2')
---
- true
...
tarantool> fio.unlink('/home/username/tmp.txt2')
---
- true
...
```

fio.rename(path-name, new-path-name)

Rename a file or directory. For details type man 2 rename.

Parameters

- path-name (string) – original name.
- new-path-name (string) – new name.

Return (If no error) true. (If error) two return values: false, error message.

Rtype boolean

Example:

```
tarantool> fio.rename('/home/username/tmp.txt', '/home/username/tmp.txt2')
---
- true
...
```

fio.utime(file-name[, accesstime[, updatetime]])

Change the access time and possibly also change the update time of a file. For details type man 2 utime. Times should be expressed as number of seconds since the epoch.

Parameters

- file-name (string) – name.
- accesstime (number) – time of last access. default current time.
- updatetime (number) – time of last update. default = access time.

Return (If no error) true. (If error) two return values: false, error message.

Rtype boolean

Example:

```
tarantool> fio.utime('/home/username/tmp.txt')
---
- true
...
```

fio.copyfile(path-name, new-path-name)

Copy a file. The effect is similar to the effect that one gets with the cp command.

Parameters

- path-name (string) – path to original file.
- new-path-name (string) – path to new file.

Return (If no error) true. (If error) two return values: false, error message.

Rtype boolean

Example:

```
tarantool> fio.copyfile('/home/user/tmp.txt', '/home/usern/tmp.txt2')
---
- true
...
```

fio.chown(path-name, owner-user, owner-group)
fio.chmod(path-name, new-rights)

Manage the rights to file objects, or ownership of file objects. For details type man 2 chown or man 2 chmod.

Parameters

- owner-user (string) – new user uid.
- owner-group (string) – new group uid.
- new-rights (number) – new permissions

Return null

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.

Parameters

- path-name (string) –

- new-size (number) –

Return (If no error) true. (If error) two return values: false, error message.

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.

Parameters

- path-name (string) – Full path to the file to open.
- 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. On Linux the full set of flags as described on the Linux man page is:
  - O_APPEND (start at end of file),
  - O_ASYNC (signal when IO is possible),
  - O_CLOEXEC (enable a flag related to closing),
  - O_CREAT (create file if it doesn't exist),
  - O_DIRECT (do less caching or no caching),
  - O_DIRECTORY (fail if it's not a directory),
  - O_EXCL (fail if file cannot be created),
  - O_LARGEFILE (allow 64-bit file offsets),
  - O_NOATIME (no access-time updating),
  - O_NOCTTY (no console tty),
  - O_NOFOLLOW (no following symbolic links),
  - O_NONBLOCK (no blocking),
  - O_PATH (get a path for low-level use),
  - O_SYNC (force writing if it's possible),

– O_TMPFILE (the file will be temporary and nameless),

– O_TRUNC (truncate)

... and, always, one of:

– O_RDONLY (read only),

– O_WRONLY (write only), or

– O_RDWR (either read or write).

- 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_CREAT or O_TMPFILE. Mode bits can be combined by enclosing them in braces.

Return (If no error) file handle (abbreviated as 'fh' in later description). (If error) two return values: null, error message.

Rtype userdata

Possible errors: nil.

Example 1:

```
tarantool> fh = fio.open('/home/username/tmp.txt', {'O_RDWR', 'O_APPEND'})
---
...
tarantool> fh -- display file handle returned by fio.open
---
- fh: 11
...
```

Example 2:

Using fio.open() with tonumber('N', 8) to set permissions as an octal number:

```
tarantool> fio.open('x.txt', {'O_WRONLY', 'O_CREAT'}, tonumber('644',8))
---
- fh: 12
...
```

object file-handle

file-handle:close()

Close a file that was opened with fio.open. For details type man 2 close.

Parameters

- fh (userdata) – file-handle as returned by fio.open().

Return true if success, false if failure.

Rtype boolean

Example:

```
tarantool> fh:close() -- where fh = file-handle
---
- true
...
```

file-handle:pread(count, offset)

file-handle:pread(buffer, count, offset)

Perform random-access read operation on a file, without affecting the current seek position of the file. For details type man 2 pread.

Parameters

- fh (userdata) – file-handle as returned by fio.open().
- buffer – where to read into (if the format is pread(buffer, count, offset))
- count (number) – number of bytes to read
- offset (number) – offset within file where reading begins

If the format is pread(count, offset) then return a string containing the data that was read from the file, or nil if failure.

If the format is pread(buffer, count, offset) then return the data to the buffer. (Buffers can be acquired with buffer.ibuf.)

Example:

```
tarantool> fh:pread(25, 25)
---
- |
  elete from t8//
  insert in
...
```

file-handle:pwrite(new-string, offset)
file-handle:pwrite(buffer, count, offset)

Perform random-access write operation on a file, without affecting the current seek position of the file. For details type man 2 pwrite.

Parameters

- fh (userdata) – file-handle as returned by fio.open().
- new-string or buffer (string) – value to write
- count (number) – number of bytes to write (if the format is pwrite(buffer, count, offset))
- offset (number) – offset within file where writing begins

Return true if success, false if failure.

Rtype boolean

If the format is pwrite(new-string, offset) then the returned string is written to the file, as far as the end of the string.

If the format is pwrite(buffer, count, offset) then the buffer contents are written to the file, for count bytes. (Buffers can be acquired with buffer.ibuf.)

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

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

file-handle:read([count])
file-handle:read(buffer, count)
Perform non-random-access read on a file. For details type man 2 read or man 2 write.

Note: 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.

Parameters

- fh (userdata) – file-handle as returned by fio.open().
- buffer – where to read into (if the format is read(buffer, count))
- count (number) – number of bytes to read

If the format is read() – omitting count – then read all bytes in the file.

If the format is read() or read([count]) then return a string containing the data that was read from the file, or nil if failure.

If the format is read(buffer, count) then return the data to the buffer. (Buffers can be acquired with buffer.ibuf.)

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

tarantool> fh:read(ibuf:reserve(5), 5)
---
- 5
...

tarantool> require('ffi').string(ibuf:alloc(5),5)
---
- abcde
```

file-handle:write(new-string)
file-handle:write(buffer, count)
Perform non-random-access write on a file. For details type man 2 write.

Note: 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.

Parameters

- fh (userdata) – file-handle as returned by fio.open().
- new-string or buffer (string) – value to write
- count (number) – number of bytes to write (if the format is write(buffer, count))

Return true if success, false if failure.

Rtype boolean

If the format is write(new-string) then the returned string is written to the file, as far as the end of the string.

If the format is write(buffer, count) then the buffer contents are written to the file, for count bytes. (Buffers can be acquired with buffer.ibuf.)

Example:

```
tarantool> fh:write("new data")
---
- true
...
```

file-handle:truncate(new-size)

Change the size of an open file. Differs from fio.truncate, which changes the size of a closed file.

Parameters

- fh (userdata) – file-handle as returned by fio.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.

Parameters

- fh (userdata) – file-handle as returned by fio.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 fio.stat which return statistics about a closed file. For details type man 2 stat.

Parameters

- fh (userdata) – file-handle as returned by fio.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 fio.sync, which is for all files. For details type man 2 fsync or man 2 fdatasync.

Parameters

- fh (userdata) – file-handle as returned by fio.open().

Return true if success, false if failure.

Example:

```
tarantool> fh:fsync()
---
- true
...
```

FIO constants

fio.c

Table with constants which are the same as POSIX flag values on the target platform (see man 2 stat).

Example:

```
tarantool> fio.c
---
- seek:
    SEEK_SET: 0
    SEEK_END: 2
    SEEK_CUR: 1
  mode:
    S_IWGRP: 16
    S_IXGRP: 8
    S_IROTH: 4
    S_IXOTH: 1
    S_IRUSR: 256
```

(continues on next page)

(continued from previous page)

```
    S_IXUSR: 64
    S_IRWXU: 448
    S_IRWXG: 56
    S_IWOTH: 2
    S_IRWXO: 7
    S_IWUSR: 128
    S_IRGRP: 32
  flag:
    O_EXCL: 2048
    O_NONBLOCK: 4
    O_RDONLY: 0
    <...>
...
```

4.2.12 Module fun

Luafun, 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
---
...
```

4.2.13 Module http

Overview

The http module, specifically the http.client submodule, provides the functionality of an HTTP client with support for HTTPS and keepalive. It uses routines in the libcurl library.

Index

Below is a list of all http functions.

Name	Use
http.client.new()	Create an HTTP client instance
client_object:request()	Perform an HTTP request
client_object:stat()	Get a table with statistics

http.client.new([options])
Construct a new HTTP client instance.

Parameters

- options (table) – the maximum number of entries in the connection cache.

Return a new HTTP client instance

Rtype userdata

Example:

```
tarantool> http_client = require('http.client').new({max_connections = 5})
---
...
```

object client_object

client_object:request(method, url, body, opts)
If http_client is an HTTP client instance, http_client:request() will perform an HTTP request and, if there is a successful connection, will return a table with connection information.

Parameters

- method (string) – HTTP method, for example 'GET' or 'POST' or 'PUT'
- url (string) – location, for example 'https://tarantool.org/doc'
- body (string) – optional initial message, for example 'My text string!'
- opts (table) – table of connection options, with any of these components:
  - timeout - number of seconds to wait for a curl API read request before timing out
  - ca_path - path to a directory holding one or more certificates to verify the peer with
  - ca_file - path to an SSL certificate file to verify the peer with
  - verify_host - set on/off verification of the certificate's name (CN) against host. See also CURLOPT_SSL_VERIFYHOST
  - verify_peer - set on/off verification of the peer's SSL certificate. See also CURLOPT_SSL_VERIFYPEER
  - ssl_key - path to a private key file for a TLS and SSL client certificate. See also CURLOPT_SSLKEY
  - ssl_cert - path to a SSL client certificate file. See also CURLOPT_SSLCERT
  - headers - table of HTTP headers
  - keepalive_idle - delay, in seconds, that the operating system will wait while the connection is idle before sending keepalive probes. See also CURLOPT_TCP_KEEPIDLE and the note below about keepalive_interval.

- keepalive_interval - the interval, in seconds, that the operating system will wait between sending keepalive probes. See also CURLOPT_TCP_KEEPINTVL. If both keepalive_idle and keepalive_interval are set, then Tarantool will also set HTTP keepalive headers: Connection:Keep-Alive and Keep-Alive:timeout=<keepalive_idle>. Otherwise Tarantool will send Connection:close
- low_speed_time - set the "low speed time" – the time that the transfer speed should be below the "low speed limit" for the library to consider it too slow and abort. See also CURLOPT_LOW_SPEED_TIME
- low_speed_limit - set the "low speed limit" – the average transfer speed in bytes per second that the transfer should be below during "low speed time" seconds for the library to consider it to be too slow and abort. See also CURLOPT_LOW_SPEED_LIMIT
- verbose - set on/off verbose mode
- unix_socket - a socket name to use instead of an Internet address, for a local connection. The Tarantool server must be built with libcurl 7.40 or later. See the second example later in this section.
- max_header_name_len - the maximal length of a header name. If a header name is bigger than this value, it is truncated to this length. The default value is '32'.

Return connection information, with all of these components:

- status - HTTP response status
- reason - HTTP response status text
- headers - a Lua table with normalized HTTP headers
- body - response body
- proto - protocol version

Rtype table

The following "shortcuts" exist for requests:

- http_client:get(url, options) - shortcut for http_client:request("GET", url, nil, opts)
- http_client:post (url, body, options) - shortcut for http_client:request("POST", url, body, opts)
- http_client:put(url, body, options) - shortcut for http_client:request("PUT", url, body, opts)
- http_client:patch(url, body, options) - shortcut for http_client:request("PATCH", url, body, opts)
- http_client:options(url, options) - shortcut for http_client:request("OPTIONS", url, nil, opts)
- http_client:head(url, options) - shortcut for http_client:request("HEAD", url, nil, opts)
- http_client:delete(url, options) - shortcut for http_client:request("DELETE", url, nil, opts)
- http_client:trace(url, options) - shortcut for http_client:request("TRACE", url, nil, opts)
- http_client:connect:(url, options) - shortcut for http_client:request("CONNECT", url, nil, opts)

Requests may be influenced by environment variables, for example users can set up an http proxy by setting HTTP_PROXY=proxy before initiating any requests. See the web page document Environment variables libcurl understands.

client_object:stat()
The http_client:stat() function returns a table with statistics:

- active_requests - number of currently executing requests
- sockets_added - total number of sockets added into an event loop
- sockets_deleted - total number of sockets sockets from an event loop
- total_requests - total number of requests
- http_200_responses - total number of requests which have returned code HTTP 200
- http_other_responses - total number of requests which have not returned code HTTP 200
- failed_requests - total number of requests which have failed including system errors, curl errors, and HTTP errors

Example 1:

Connect to an HTTP server, look at the size of the response for a 'GET' request, and look at the statistics for the session.

```
tarantool> http_client = require('http.client').new()
---
...
tarantool> r = http_client:request('GET','http://tarantool.org')
---
...
tarantool> string.len(r.body)
---
- 21725
...
tarantool> http_client:stat()
---
- total_requests: 1
  sockets_deleted: 2
  failed_requests: 0
  active_requests: 0
  http_other_responses: 0
  http_200_responses: 1
  sockets_added: 2
```

Example 2:

Start two Tarantool instances on the same computer.

On the first Tarantool instance, listen on a Unix socket:

```
box.cfg{listen='/tmp/unix_domain_socket.sock'}
```

On the second Tarantool instance, send via http_client:

```
box.cfg{}
http_client = require('http.client').new({5})
http_client:put('http://localhost/','body',{unix_socket = '/tmp/unix_domain_socket.sock'})
```

Terminal #1 will show an error message: "Invalid MsgPack". This is not useful but demonstrates the syntax and demonstrates that was sent was received.

4.2.14 Module iconv

Overview

The iconv module provides a way to convert a string with one encoding to a string with another encoding, for example from ASCII to UTF-8. It is based on the POSIX iconv routines.

An exact list of the available encodings may depend on environment. Typically the list includes ASCII, BIG5, KOI8R, LATIN8, MS-GREEK, SJIS, and about 100 others. For a complete list, type iconv --list on a terminal.

Index

Below is a list of all iconv functions.

Name	Use
iconv.new()	Create an iconv instance
iconv.converter()	Perform conversion on a string

iconv.new(to, from)

Construct a new iconv instance.

Parameters

- to (string) – the name of the encoding that we will convert to.
- from (string) – the name of the encoding that we will convert from.

Return a new iconv instance – in effect, a callable function

Rtype userdata

If either parameter is not a valid name, there will be an error message.

Example:

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

iconv.converter(input-string)

Convert.

param string input-string the string to be converted (the "from" string)

return the string that results from the conversion (the "to" string)

If anything in input-string cannot be converted, there will be an error message and the result string will be unchanged.

Example:

We know that the Unicode code point for "Д" (CYRILLIC CAPITAL LETTER DE) is hexadecimal 0414 according to the character database of Unicode. Therefore that is what it will look like in UTF-16. We know that Tarantool typically uses the UTF-8 character set. So make a from-UTF-8-to-UTF-16 converter, use string.hex('Д') to show what Д's encoding looks like in the UTF-8 source, and use string.hex('Д'-after-conversion) to show what it looks like in the UTF-16 target. Since the result is 0414, we see that iconv conversion works.

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

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

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

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

4.2.15 Module json

Overview

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.

Index

Below is a list of all json functions and members.

Name	Use
json.encode()	Convert a Lua object to a JSON string
json.decode()	Convert a JSON string to a Lua object
json.NULL	Analog of Lua's "nil"
json.cfg()	Set persistent flags

json.encode(lua-value[, configuration])
Convert a Lua object to a JSON string.

Parameters

- lua_value – either a scalar value or a Lua table value.
- configuration – see json.cfg

Return the original value reformatted as a JSON string.

Rtype string

Example:

```
tarantool> json=require('json')
---
...
tarantool> json.encode(123)
---
```

(continues on next page)

(continued from previous page)

```
- '123'
...
tarantool> json.encode({123})
---
- '[123]'
...
tarantool> json.encode({123, 234, 345})
---
- '[123,234,345]'
...
tarantool> json.encode({abc = 234, cde = 345})
---
- '{"cde":345,"abc":234}'
...
tarantool> json.encode({hello = {'world'}})
---
- '{"hello":["world"]}'
...
```

json.decode(string[, configuration])

Convert a JSON string to a Lua object.

Parameters

- string (string) – a string formatted as JSON.
- configuration – see json.cfg

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
...
```

See the tutorial Sum a JSON field for all tuples to see how json.decode() can fit in an application.

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'}
```

(continues on next page)

(continued from previous page)

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

json.cfg(list of parameter assignments)

Set values affecting behavior of json.encode and json.decode. The values are all either integers or boolean true/false values.

- encode_invalid_as_nil – use null for unrecognizable types; default = false
- encode_invalid_numbers – allow nan and inf; default = true
- encode_load_metatables – load metatables; default = true
- encode_max_depth – how deep nesting can be; default = 32
- encode_number_precision – maximum post-decimal digits; default = 14
- encode_sparse_ratio – how sparse an array can be; default = 2
- encode_sparse_safe – how much can safely be sparse; default = 10

- encode_use_tostring – use tostring for unrecognizable types; default = false
- decode_invalid_numbers – like encode_invalid_numbers; default = true
- decode_max_depth – like encode_max_depth; default = 32
- decode_save_metatables – like encode_load_metatables; default = true
- decode_sparse_convert – like encode_sparse_convert; default = true

For example, the following code will encode 0/0 as nan ("not a number") and 1/0 as inf ("infinity"), rather than returning nil or an error message:

```
json = require('json')
json.cfg{encode_invalid_numbers = true}
x = 0/0
y = 1/0
json.encode({1, x, y, 2})
```

The result of the json.encode() request will look like this:

```
tarantool> json.encode({1, x, y, 2})
---
- '[1,nan,inf,2]
...
```

To achieve the same effect for only one call to json.encode() without changing the configuration persistently, one could say json.encode({1, x, y, 2}, {encode_invalid_numbers = true}).

The same configuration settings exist for json, for MsgPack, and for YAML.

4.2.16 Module key_def

The key_def module has a function for making a definition of the field numbers and types of a tuple. The definition is usually used in conjunction with an index definition to extract or compare the index key values.

key_def.new(parts)
Create a new key_def instance.

Parameters

- parts (table) – field numbers and types. There must be at least one part and it must have at least fieldno and type.

Returns key_def-object a key_def object

The parts table has components which are the same as the parts option in Options for space_object:create_index().

fieldno (integer) for example fieldno=1

type (string) for example type='string'

Other components are optional.

Example: key_def.new({{type = 'unsigned', fieldno = 1}})

object key_def_object
A key_def object is an object returned by key_def.new(). It has methods extract_key(), compare(), compare_with_key(), merge(), totable().

key_def_object:extract_key(tuple)
Return a tuple containing only the fields of the key_def object.

Parameters

- tuple (table) – tuple or Lua table with field contents

Return the fields that were defined for the key_def object

Example #1:

```
-- Suppose that an item has five fields
-- 1, 99.5, 'X', nil, 99.5
-- and the fields that we care about are
-- #3 (a string) and #1 (an integer).
-- We can define those fields with k = key_def.new
-- and extract the values with k:extract_key.

tarantool> key_def = require('key_def')
---
...

tarantool> k = key_def.new({{type = 'string', fieldno = 3},
>                           {type = 'unsigned', fieldno =1 }})
---
...

tarantool> k:extract_key({1, 99.5, 'X', nil, 99.5})
---
- ['X', 1]
...
```

Example #2

```
-- Now suppose that the item is a tuple in a space which
-- has an index on field #3 plus field #1.
-- We can use key_def.new with the index definition
-- instead of filling it out as we did with Example #1.
-- The result will be the same.
key_def = require('key_def')
box.schema.space.create('T')
i = box.space.T:create_index('I',{parts={3,'string',1,'unsigned'}})
box.space.T:insert{1, 99.5, 'X', nil, 99.5}
k = key_def.new(i.parts)
k:extract_key(box.space.T:get({'X', 1}))
```

Example #3

```
-- Iterate through the tuples in a secondary non-unique index.
-- extracting the tuples' primary-key values so they can be deleted
-- using a unique index. This code should be part of a Lua function.
local key_def_lib = require('key_def')
local s = box.schema.space.create('test')
local pk = s:create_index('pk')
local sk = s:create_index('test', {unique = false, parts = {
    {2, 'number', path = 'a'}, {2, 'number', path = 'b'}}})
s:insert{1, {a = 1, b = 1}}
s:insert{2, {a = 1, b = 2}}
local key_def = key_def_lib.new(pk.parts)
for _, tuple in sk:pairs({1}) do
    local key = key_def:extract_key(tuple)
```

(continues on next page)

(continued from previous page)

```
    pk:delete(key)
end
```

key_def_object:compare(tuple_1, tuple_2)
Compare the key fields of tuple_1 to the key fields of tuple_2. This is a tuple-by-tuple comparison so users do not have to write code which compares a field at a time. Each field's type and collation wll be taken into account. In effect it is a comparison of extract_key(tuple_1) with extract_key(tuple_2).

Parameters

- tuple1 (table) – tuple or Lua table with field contents
- tuple2 (table) – tuple or Lua table with field contents

Return > 0 if tuple_1 key fields > tuple_2 key fields, = 0 if tuple_1 key fields = tuple_2 key fields, < 0 if tuple_1 key fields < tuple_2 key fields

Example:

```
-- This will return 0
key_def = require('key_def')
k = key_def.new({{type='string',fieldno=3,collation='unicode_ci'},
                 {type='unsigned',fieldno=1}})
k:compare({1, 99.5, 'X', nil, 99.5}, {1, 99.5, 'x', nil, 99.5})
```

key_def_object:compare_with_key(tuple_1, tuple_2)
Compare the key fields of tuple_1 to all the fields of tuple_2. This is the same as key_def_object:compare() except that tuple_2 contains only the key fields. In effect it is a comparison of extract_key(tuple_1) with tuple_2.

Parameters

- tuple1 (table) – tuple or Lua table with field contents
- tuple2 (table) – tuple or Lua table with field contents

Return > 0 if tuple_1 key fields > tuple_2 fields, = 0 if tuple_1 key fields = tuple_2 fields, < 0 if tuple_1 key fields < tuple_2 fields

Example:

```
-- This will return 0
key_def = require('key_def')
k = key_def.new({{type='string',fieldno=3,collation='unicode_ci'},
                 {type='unsigned',fieldno=1}})
k:compare_with_key({1, 99.5, 'X', nil, 99.5}, {'x', 1})
```

key_def_object:merge(other_key_def_object)
Combine the main key_def_object with other_key_def_object. The return value is a new key_def_object containing all the fields of the main key_def_object, then all the fields of other_key_def_object which are not in the main key_def_object.

Parameters

- other_key_def_object (key_def_object) – definition of fields to add

Return key_def_object

Example:

```
-- This will return a key definition with fieldno=3 and fieldno=1.
key_def = require('key_def')
k = key_def.new({{type = 'string', fieldno = 3}})
k2= key_def.new({{type = 'unsigned', fieldno = 1},
                 {type = 'string', fieldno = 3}})
k:merge(k2)
```

key_def_object:totable()
: Return a table containing what is in the key_def_object. This is the reverse of key_def.new() – key_def.new() takes a table and returns a key_def object, key_def_object:totable() takes a key_def object and returns a table. This is useful for input to _serialize methods.

  Return table

  Example:

```
-- This will return a table with type='string', fieldno=3
key_def = require('key_def')
k = key_def.new({{type = 'string', fieldno = 3}})
k:totable()
```

4.2.17 Module log

Overview

The Tarantool server puts all diagnostic messages in a log file specified by the log 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.

As explained in the description of log_format configuration setting, there are two possible formats for log entries:

- 'plain' (the default), or
- 'json' (with more detail and with JSON labels).

Here is what a log entry looks like after box.cfg{log_format='plain'}:

```
2017-10-16 11:36:01.508 [18081] main/101/interactive I> set 'log_format' configuration option to "plain"
```

Here is what a log entry looks like after box.cfg{log_format='json'}:

```
{"time": "2017-10-16T11:36:17.996-0600",
"level": "INFO",
"message": "set 'log_format' configuration option to \"json\"",
"pid": 18081,|
"cord_name": "main",
"fiber_id": 101,
"fiber_name": "interactive",
"file": "builtin\/box\/load_cfg.lua",
"line": 317}
```

Index

Below is a list of all log functions.

Name	Use
log.error() log.warn() log.info() log.verbose() log.debug()	Write a user-generated message to a log file
log.logger_pid()	Get the PID of a logger
log.rotate()	Rotate a log file

log.error(message)
log.warn(message)
log.info(message)
log.verbose(message)
log.debug(message)

Output a user-generated message to the log file, given log_level_function_name = error or warn or info or verbose or debug.

As explained in the description of the configuration setting for log_level, there are seven levels of detail:

- 1 – SYSERROR
- 2 – ERROR – this corresponds to log.error(...)
- 3 – CRITICAL
- 4 – WARNING – this corresponds to log.warn(...)
- 5 – INFO – this corresponds to log.info(...)
- 6 – VERBOSE – this corresponds to log.verbose(...)
- 7 – DEBUG – this corresponds to log.debug(...)

For example, if box.cfg.log_level is currently 5 (the default value), then log.error(...), log.warn(...) and log.info(...) messages will go to the log file. However, log.verbose(...) and log.debug(...) messages will not go to the log file, because they correspond to higher levels of detail.

Parameters

- message (string) – The actual output will be a line containing:
  - the current timestamp,
  - a module name,
  - 'E', 'W', 'I', 'V' or 'D' 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()

Return PID of a logger

log.rotate()

Rotate the log.

Return nil

Example

```
$ tarantool
tarantool> box.cfg{log_level=3, log='tarantool.txt'}
tarantool> log = require('log')
tarantool> log.error('Error')
tarantool> log.info('Info %s', box.info.version)
tarantool> os.exit()
```

```
$ less tarantool.txt
2017-09-20 ... [68617] main/101/interactive C> version 1.7.5-31-ge939c6ea6
2017-09-20 ... [68617] main/101/interactive C> log level 3
2017-09-20 ... [68617] main/101/interactive [C]:-1 E> Error
```

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.

4.2.18 Module msgpack

Overview

The msgpack module takes strings in MsgPack format and decodes them, or takes a series of non-MsgPack values and encodes them. Tarantool makes heavy internal use of MsgPack because tuples in Tarantool are stored as MsgPack arrays.

Index

Below is a list of all msgpack functions and members.

Name	Use
msgpack.encode()	Convert a Lua object to an MsgPack string
msgpack.decode()	Convert a MsgPack string to a Lua object
msgpack.decode_unchecked()	Convert a MsgPack string to a Lua object
msgpack.NULL	Analog of Lua's "nil"
msgpack.decode_array_header	Skip array header in a MsgPack string
msgpack.decode_map_header	Skip map header in a MsgPack string

msgpack.encode(lua_value)
Convert a Lua object to a MsgPack string.

Parameters

- lua_value – either a scalar value or a Lua table value.

Return the original value reformatted as a MsgPack string.

Rtype string

msgpack.decode(msgpack_string[, start_position])
Convert a MsgPack string to a Lua object.

Parameters

- msgpack_string (string) – a string formatted as MsgPack.

- start_position (integer) – where to start, minimum = 1, maximum = string length, default = 1.

Return

- (if msgpack_string is in valid MsgPack format) the original contents of msgpack_string, formatted as a Lua table, (otherwise) a scalar value, such as a string or a number;
- "next_start_position". If decode stops after parsing as far as byte N in msgpack_string, then "next_start_position" will equal N + 1, and decode(msgpack_string, next_start_position) will continue parsing from where the previous decode stopped, plus 1. Normally decode parses all of msgpack_string, so "next_start_position" will equal string.len(msgpack_string) + 1.

Rtype table and number

msgpack.decode_unchecked(string)

Convert a MsgPack string to a Lua object. Because checking is skipped, decode_unchecked() can operate with string pointers to buffers which decode() cannot handle. For an example see the buffer module.

Parameters

- string – a string formatted as MsgPack.

Return

- the original contents formatted as a Lua table;
- the number of bytes that were decoded.

Rtype lua object

msgpack.NULL

A value comparable to Lua "nil" which may be useful as a placeholder in a tuple.

msgpack.decode_array_header(byte-array, size)

Call the mp_decode_array function in the MsgPuck library and return the array size and a pointer to the first array component. A subsequent call to msgpack_decode can decode the component instead of the whole array.

Parameters

- byte-array – a pointer to a byte array formatted as MsgPack.
- size – a number greater than or equal to the string's length

Return

- the size of the array;
- a pointer to after the array header.

```
-- Example of decode_array_header
-- Suppose we have the raw data '\x93\x01\x02\x03'.
-- \x93 is MsgPack encoding for a header of a three-item array.
-- We want to skip it and decode the next three items.
msgpack=require('msgpack'); ffi=require('ffi')
x,y=msgpack.decode_array_header(ffi.cast('char*','\x93\x01\x02\x03'),4)
a=msgpack.decode(y,1);b=msgpack.decode(y+1,1);c=msgpack.decode(y+2,1);
a,b,c
-- The result will be: 1,2,3.
```

msgpack.decode_map_header(byte-array, size)

Call the mp_decode_map function in the MsgPuck library and return the map size and a pointer to the first map component. A subsequent call to msgpack_decode can decode the component instead of the whole map.

Parameters

- byte-array – a pointer to a byte array formatted as MsgPack.
- size – a number greater than or equal to the byte array's length

Return

- the size of the map;
- a pointer to after the map header.

```
-- Example of decode_map_header
-- Suppose we have the raw data '\x81\xa2\x41\x41\xc3'.
-- \x81 is MsgPack encoding for a header of a one-item map.
-- We want to skip it and decode the next map item.
msgpack=require('msgpack'); ffi=require('ffi')
x,y=msgpack.decode_map_header(ffi.cast('char*','\x81\xa2\x41\x41\xc3'),5)
a=msgpack.decode(y,3);b=msgpack.decode(y+3,1)
x,a,b
-- The result will be: 1,"AA", true.
```

Example

```
tarantool> msgpack = require('msgpack')
---
...
tarantool> y = msgpack.encode({'a',1,'b',2})
---
...
tarantool> z = msgpack.decode(y)
---
...
tarantool> z[1], z[2], z[3], z[4]
---
- a
- 1
- b
- 2
...
tarantool> box.space.tester:insert{20, msgpack.NULL, 20}
---
- [20, null, 20]
...
```

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

{}	'fixmap' if metatable is 'map' = 80 otherwise 'fixarray' = 90
'a'	'fixstr' = a1 61
false	'false' = c2
true	'true' = c3
127	'positive fixint' = 7f
65535	'uint 16' = cd ff ff
4294967295	'uint 32' = ce ff ff ff ff
nil	'nil' = c0
msg-pack.NULL	same as nil
[0] = 5	'fixmap(1)' + 'positive fixint' (for the key) + 'positive fixint' (for the value) = 81 00 05
[0] = nil	'fixmap(0)' = 80 – nil is not stored when it is a missing map value
1.5	'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.

4.2.19 Module net.box

Overview

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 (see SQL DBMS modules reference). The other variant, which is discussed in this section, is for connecting to Tarantool server instances via a network.

You can call the following methods:

- require('net.box') to get a net.box object (named net_box for examples in this section),
- net_box.connect() to connect and get a connection object (named conn for examples in this section),
- other net.box() routines, passing conn:, to execute requests on the remote database system,
- 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 that is 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. However for some cases a single connection is not enough — for example, when it is necessary to prioritize requests or to use different authentication IDs.

Most net.box methods allow a final {options} argument, which can be:

- {timeout=...}. For example, a method whose final argument is {timeout=1.5} will stop after 1.5 seconds on the local node, although this does not guarantee that execution will stop on the remote server node.
- {buffer=...}. For an example see buffer module.
- {is_async=...}. For example, a method whose final argument is {is_async=true} will not wait for the result of a request. See the is_async description.
- {on_push=... on_push_ctx=...}. For receiving out-of-band messages. See the box.session.push description.

The diagram below shows possible connection states and transitions:

On this diagram:

- The state machine starts in the 'initial' state.
- net_box.connect() method changes the state to 'connecting' and spawns a worker fiber.
- If authentication and schema upload are required, it's possible later on to re-enter the 'fetch_schema' state from 'active' if a request fails due to a schema version mismatch error, so schema reload is triggered.
- conn.close() method sets the state to 'closed' and kills the worker. If the transport is already in the 'error' state, close() does nothing.

Index

Below is a list of all net.box functions.

Name	Use
net_box.connect() net_box.new()	Create a connection
conn:ping()	Execute a PING command
conn:wait_connected()	Wait for a connection to be active or closed
conn:is_connected()	Check if a connection is active or closed
conn:wait_state()	Wait for a target state
conn:close()	Close a connection
conn.space.space-name:select{field-value}	Select one or more tuples
conn.space.space-name:get{field-value}	Select a tuple
conn.space.space-name:insert{field-value}	Insert a tuple
conn.space.space-name:replace{field-value}	Insert or replace a tuple
conn.space.space-name:update{field-value}	Update a tuple
conn.space.space-name:upsert{field-value}	Update a tuple
conn.space.space-name:delete{field-value}	Delete a tuple
conn:eval()	Evaluate and execute the expression in a string
conn:call()	Call a stored procedure
conn:timeout()	Set a timeout

net_box.connect(URI[, {option[s]}])

net_box.new(URI[, {option[s]}])

Note: The names connect() and new() are synonyms: connect() is preferred; new() is retained for backward compatibility.

Create a new connection. The connection is established on demand, at the time of the first request. It can be re-established automatically after a disconnect (see reconnect_after option below). The returned conn object supports methods for making remote requests, such as select, update or delete.

For a 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.connect('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.

Possible options:

- wait_connected: by default, connection creation is blocked until the connection is established, but passing wait_connected=false makes it return immediately. Also, passing a timeout makes it wait before returning (e.g. wait_connected=1.5 makes it wait at most 1.5 seconds).

  Note: In the presence of reconnect_after, wait_connected ignores transient failures. The wait completes once the connection is established or is closed explicitly.

- reconnect_after: a net.box instance automatically reconnects any time the connection is broken or if a connection attempt fails. This makes transient network failures become transparent to the application. Reconnect happens automatically in the background, so queries/requests that suffered due to connectivity loss are transparently retried. The number of retries is unlimited,

connection attempts are done over the specified timeout (e.g. reconnect_after=5 for 5 secs). Once a connection is explicitly closed, or once the Lua garbage collector removes it, reconnects stop.

- call_16: [since 1.7.2] by default, net.box connections comply with a new binary protocol command for CALL, which is not backward compatible with previous versions. The new CALL no longer restricts a function to returning an array of tuples and allows returning an arbitrary MsgPack/JSON result, including scalars, nil and void (nothing). The old CALL is left intact for backward compatibility. It will be removed in the next major release. All programming language drivers will be gradually changed to use the new CALL. To connect to a Tarantool instance that uses the old CALL, specify call_16=true.
- console: depending on the option's value, the connection supports different methods (as if instances of different classes were returned). With console = true, you can use conn methods close(), is_connected(), wait_state(), eval() (in this case, both binary and Lua console network protocols are supported). With console = false (default), you can also use conn database methods (in this case, only the binary protocol is supported). Deprecation notice: console = true is deprecated, users should use console.connect() instead.
- connect_timeout: number of seconds to wait before returning "error: Connection timed out".

Parameters

- URI (string) – the URI of the target for the connection
- options – possible options are wait_connected, reconnect_after, call_16 and console

Return conn object

Rtype userdata

Examples:

```
conn = net_box.connect('localhost:3301')
conn = net_box.connect('127.0.0.1:3302', {wait_connected = false})
conn = net_box.connect('127.0.0.1:3303', {reconnect_after = 5, call_16 = true})
```

object conn

conn:ping()

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.

Parameters

- timeout (number) – in seconds

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:wait_state(state[s][, timeout])

[since 1.7.2] Wait for a target state.

Parameters

- states (string) – target states
- timeout (number) – in seconds

Return true when a target state is reached, false on timeout or connection closure

Rtype boolean

Examples:

```
-- wait infinitely for 'active' state:
conn:wait_state('active')

-- wait for 1.5 secs at most:
conn:wait_state('active', 1.5)

-- wait infinitely for either `active` or `fetch_schema` state:
conn:wait_state({active=true, fetch_schema=true})
```

conn:close()

Close a connection.

Connection objects are destroyed by the Lua garbage collector, 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, ...} [, {options}])

conn.space.space-name:select({...}) is the remote-call equivalent of the local call box.space.space-name:select{...}. For an additional option see Module buffer and skip-header.

Example:

```
conn.space.testspace:select({1,'B'}, {timeout=1})
```

Note: 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, ...} [, {options}])
conn.space.space-name:get(...) is the remote-call equivalent of the local call box.space.space-name:get(...).

Example:

```
conn.space.testspace:get({1})
```

conn.space.<space-name>:insert({field-value, ...} [, {options}])
conn.space.space-name:insert(...) is the remote-call equivalent of the local call box.space.space-name:insert(...). For an additional option see Module buffer and skip-header.

Example:

```
conn.space.testspace:insert({2,3,4,5}, {timeout=1.1})
```

conn.space.<space-name>:replace({field-value, ...} [, {options}])
conn.space.space-name:replace(...) is the remote-call equivalent of the local call box.space.space-name:replace(...). For an additional option see Module buffer and skip-header.

Example:

```
conn.space.testspace:replace({5,6,7,8})
```

conn.space.<space-name>:update({field-value, ...} [, {options}])
conn.space.space-name:update(...) is the remote-call equivalent of the local call box.space.space-name:update(...). For an additional option see Module buffer and skip-header.

Example:

```
conn.space.Q:update({1},{{'=',2,5}}, {timeout=0})
```

conn.space.<space-name>:upsert({field-value, ...} [, {options}])
conn.space.space-name:upsert(...) is the remote-call equivalent of the local call box.space.space-name:upsert(...). For an additional option see Module buffer and skip-header.

conn.space.<space-name>:delete({field-value, ...} [, {options}])
conn.space.space-name:delete(...) is the remote-call equivalent of the local call box.space.space-name:delete(...). For an additional option see Module buffer and skip-header.

conn:eval(Lua-string[, {arguments}[, {options}]])
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').

To ensure that the return from conn:eval is whatever the Lua expression returns, begin the Lua-string with the word "return".

Examples:

```
tarantool> --Lua-string
tarantool> conn:eval('function f5() return 5+5 end; return f5();')
---
- 10
...
tarantool> --Lua-string, {arguments}
```

(continues on next page)

(continued from previous page)

```
tarantool> conn:eval('return ...', {1,2,{3,'x'}})
---
- 1
- 2
- [3, 'x']
...
tarantool> --Lua-string, {arguments}, {options}
tarantool> conn:eval('return {nil,5}', {}, {timeout=0.1})
---
- [null, 5]
...
```

conn:call(function-name[, {arguments}[, {options}]])
: 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. The return from conn:call is whatever the function returns.

  Limitation: the called function cannot return a function, for example if func2 is defined as function func2 () return func end then conn:call(func2) will return "error: unsupported Lua type 'function'".

  Examples:

```
tarantool> -- create 2 functions with conn:eval()
tarantool> conn:eval('function f1() return 5+5 end;')
tarantool> conn:eval('function f2(x,y) return x,y end;')
tarantool> -- call first function with no parameters and no options
tarantool> conn:call('f1')
---
- 10
...
tarantool> -- call second function with two parameters and one option
tarantool> conn:call('f2',{1,'B'},{timeout=99})
---
- 1
- B
...
```

conn:timeout(timeout)
: timeout(...) is a wrapper which sets a timeout for the request that follows it. Since version 1.7.4 this method is deprecated – it is better to pass a timeout value for a method's {options} parameter.

  Example:

```
conn:timeout(0.5).space.tester:update({1}, {{'=', 2, 15}})
```

  Although timeout(...) is deprecated, all remote calls support its use. 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.

conn:request(... {is_async=...})
: {is_async=true|false} is an option which is applicable for all net_box requests including conn:call, conn:eval, and the conn.space.space-name requests.

  The default is is_async=false, meaning requests are synchronous for the fiber. The fiber is blocked, waiting until there is a reply to the request or until timeout expires. Before Tarantool version 1.10, the only way to make asynchronous requests was to put them in separate fibers.

The non-default is is_async=true, meaning requests are asynchronous for the fiber. The request causes a yield but there is no waiting. The immediate return is not the result of the request, instead it is an object that the calling program can use later to get the result of the request.

This immediately-returned object, which we'll call "future", has its own methods:

- future:is_ready() which will return true when the result of the request is available,
- future:result() to get the result of the request,
- future:wait_result(timeout) to wait until the result of the request is available and then get it,
- future:discard() to abandon the object.

Typically a user would say future=request-name(...{is_async=true}), then either loop checking future:is_ready() until it is true and then say request_result=future:result(), or say request_result=future:wait_result(...). Alternatively the client could check for "out-of-band" messages from the server by calling pairs() in a loop – see box.session.push().

Example:

```
tarantool> future = conn.space.tester:insert({900},{is_async=true})
---
...
tarantool> future
---
- method: insert
  response: [900]
  cond: cond
  on_push_ctx: []
  on_push: 'function: builtin#91'
...
tarantool> future:is_ready()
---
- true
...
tarantool> future:result()
---
- [900]
...
```

Typically {is_async=true} is used only if the load is large (more than 100,000 requests per second) and latency is large (more than 1 second), or when it is necessary to send multiple requests in parallel then collect responses (sometimes called a "map-reduce" scenario).

Note: Although the final result of an async request is the same as the result of a sync request, it is structured differently: as a table, instead of as the unpacked values.

Example

This example shows the use of most of the net.box methods.

The sandbox configuration for this example assumes that:

- the Tarantool instance is running on localhost 127.0.0.1:3301,
- there is a space named tester with a numeric primary key and with a tuple that contains a key value = 800,

- the current user has read, write and execute privileges.

Here are commands for a quick sandbox setup:

```
box.cfg{listen = 3301}
s = box.schema.space.create('tester')
s:create_index('primary', {type = 'hash', parts = {1, 'unsigned'}})
t = s:insert({800, 'TEST'})
box.schema.user.grant('guest', 'read,write,execute', 'universe')
```

And here starts the example:

```
tarantool> net_box = require('net.box')
---
...
tarantool> function example()
         >   local conn, wtuple
         >   if net_box.self:ping() then
         >     table.insert(ta, 'self:ping() succeeded')
         >     table.insert(ta, '  (no surprise -- self connection is pre-established)')
         >   end
         >   if box.cfg.listen == '3301' then
         >     table.insert(ta,'The local server listen address = 3301')
         >   else
         >     table.insert(ta, 'The local server listen address is not 3301')
         >     table.insert(ta, '(  (maybe box.cfg{...listen="3301"...} was not stated)')
         >     table.insert(ta, '(  (so connect will fail)')
         >   end
         >   conn = net_box.connect('127.0.0.1:3301')
         >   conn.space.tester:delete({800})
         >   table.insert(ta, 'conn delete done on tester.')
         >   conn.space.tester:insert({800, 'data'})
         >   table.insert(ta, 'conn insert done on tester, index 0')
         >   table.insert(ta, '  primary key value = 800.')
         >   wtuple = conn.space.tester:select({800})
         >   table.insert(ta, 'conn select done on tester, index 0')
         >   table.insert(ta, '  number of fields = ' .. #wtuple)
         >   conn.space.tester:delete({800})
         >   table.insert(ta, 'conn delete done on tester')
         >   conn.space.tester:replace({800, 'New data', 'Extra data'})
         >   table.insert(ta, 'conn:replace done on tester')
         >   conn.space.tester:update({800}, {{'=', 2, 'Fld#1'}})
         >   table.insert(ta, 'conn update done on tester')
         >   conn:close()
         >   table.insert(ta, 'conn close done')
         > end
---
...
tarantool> ta = {}
---
...
tarantool> example()
---
...
tarantool> ta
---
- - self:ping() succeeded
  - '  (no surprise -- self connection is pre-established)'
```

(continues on next page)

(continued from previous page)

```
- 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
...
```

4.2.20 Module os

Overview

The os module contains the functions execute(), rename(), getenv(), remove(), date(), exit(), time(), clock(), tmpname(), environ(), setenv(), setlocale(), difftime(). Most of these functions are described in the Lua manual Chapter 22 The Operating System Library.

Index

Below is a list of all os functions.

Name	Use
os.execute()	Execute by passing to the shell
os.rename()	Rename a file or directory
os.getenv()	Get an environment variable
os.remove()	Remove a file or directory
os.date()	Get a formatted date
os.exit()	Exit the program
os.time()	Get the number of seconds since the epoch
os.clock()	Get the number of CPU seconds since the program start
os.tmpname()	Get the name of a temporary file
os.environ()	Get a table with all environment variables
os.setenv()	Set an environment variable
os.setlocale()	Change the locale
os.difftime()	Get the number of seconds between two times

os.execute(shell-command)

Execute by passing to the shell.

Parameters

- 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
```

(continues on next page)

(continued from previous page)

```
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.

Parameters

- 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")
---
```

(continues on next page)

(continued from previous page)

```
- Sunday April 24
...
```

os.exit()
: Exit the program. If this is done on a server instance, then the instance 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
...
```

os.environ()
: Return a table containing all environment variables.

  Example:

```
tarantool> os.environ()['TERM']..os.environ()['SHELL']
---
- xterm/bin/bash
...
```

os.setenv(variable-name, variable-value)
: Set an environment variable.

  Example:

```
tarantool> os.setenv('VERSION','99')
---
-
...
```

os.setlocale([new-locale-string])
: Change the locale. If new-locale-string is not specified, return the current locale.

  Example:

```
tarantool> require('string').sub(os.setlocale(),1,20)
---
- LC_CTYPE=en_US.UTF-8
...
```

os.difftime(time1, time2)
: Return the number of seconds between two times.

  Example:

```
tarantool> os.difftime(os.time() - 0)
---
- 1486594859
...
```

4.2.21 Module pickle

Index

Below is a list of all pickle functions.

Name	Use
pickle.pack()	Convert Lua variables to binary format
pickle.unpack()	Convert Lua variables back from binary format

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 scalar value to a 1-byte integer, and stores the integer in the resulting string
s, S	converts Lua scalar value to a 2-byte integer, and stores the integer in the resulting string, low byte first
i, I	converts Lua scalar value to a 4-byte integer, and stores the integer in the resulting string, low byte first
l, L	converts Lua scalar value to an 8-byte integer, and stores the integer in the resulting string, low byte first
n	converts Lua scalar value to a 2-byte integer, and stores the integer in the resulting string, big endian,
N	converts Lua scalar value to a 4-byte integer, and stores the integer in the resulting string, big
q, Q	converts Lua scalar value to an 8-byte integer, and stores the integer in the resulting string, big endian,
f	converts Lua scalar value to a 4-byte float, and stores the float in the resulting string
d	converts Lua scalar value to a 8-byte double, and stores the double in the resulting string
a, A	converts Lua scalar value to a sequence of bytes, and stores the sequence in the resulting string

Parameters

- 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

A scalar value can be either a variable or a literal. Remember that large integers should be entered with tonumber64() or LL or ULL suffixes.

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]
...
```

(continues on next page)

(continued from previous page)

```
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.

Parameters

- format (string) –
- binary-string (string) –

Return A list of strings or numbers.

Rtype table

Example:

```
tarantool> pickle = require('pickle')
---
...
tarantool> tuple = box.space.tester:replace{0}
---
...
tarantool> string.len(tuple[1])
---
- 1
...
tarantool> pickle.unpack('b', tuple[1])
---
- 48
...
tarantool> pickle.unpack('bsi', pickle.pack('bsi', 255, 65535, 4294967295))
---
- 255
- 65535
- 4294967295
...
tarantool> pickle.unpack('ls', pickle.pack('ls', tonumber64('18446744073709551615'), 65535))
---
...
tarantool> num, num64, str = pickle.unpack('slA', pickle.pack('slA', 666,
         > tonumber64('666666666666666'), 'string'))
---
...
```

4.2.22 Module socket

Overview

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.

Index

Below is a list of all socket functions.

Name	Use
socket()	Create a socket
socket.tcp_connect()	Connect a socket to a remote host
socket.getaddrinfo()	Get information about a remote site
socket.tcp_server()	Make Tarantool act as a TCP server
socket_object:sysconnect()	Connect a socket to a remote host
socket_object:send() socket_object:write()	Send data over a connected socket
socket_object:syswrite()	Write data to the socket buffer if non-blocking
socket_object:recv()	Read from a connected socket
socket_object:sysread()	Read data from the socket buffer if non-blocking
socket_object:bind()	Bind a socket to the given host/port
socket_object:listen()	Start listening for incoming connections
socket_object:accept()	Accept a client connection + create a connected socket
socket_object:sendto()	Send a message on a UDP socket to a specified host
socket_object:recvfrom()	Receive a message on a UDP socket
socket_object:shutdown()	Shut down a reading end, a writing end, or both
socket_object:close()	Close a socket
socket_object:error() socket_object:errno()	Get information about the last error on a socket
socket_object:setsockopt()	Set socket flags
socket_object:getsockopt()	Get socket flags
socket_object:linger()	Set/clear the SO_LINGER flag
socket_object:nonblock()	Set/get the flag value
socket_object:readable()	Wait until something is readable
socket_object:writable()	Wait until something is writable
socket_object:wait()	Wait until something is either readable or writable
socket_object:name()	Get information about the connection's near side
socket_object:peer()	Get information about the connection's far side
socket.iowait()	Wait for read/write activity

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[, timeout]])

Connect a socket to a remote host.

Parameters

- host (string) – URL or IP address
- port (number) – port number
- timeout (number) – timeout

Return a connected socket, if no error.

Rtype userdata

Example:

```
socket.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. This function may use the worker_pool_threads configuration parameter.

Return A table containing these fields: “host”, “family”, “type”, “protocol”, “port”.

Rtype table

Example:

```
tarantool> socket.getaddrinfo('tarantool.org', 'http')
---
- - host: 188.93.56.70
    family: AF_INET
    type: SOCK_STREAM
    protocol: tcp
    port: 80
  - host: 188.93.56.70
    family: AF_INET
    type: SOCK_DGRAM
    protocol: udp
    port: 80
...
```

socket.tcp_server(host, port, handler-function-or-table[, timeout])

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=...}.

Parameters

- host (string) – host name or IP

- port (number) – host port, may be 0
- handler-function-or-table (function/table) – what to execute when a connection occurs
- timeout (number) – number of seconds to wait before timing out

The handler-function-or-table parameter may be simply a function name / function declaration: handler_function. Or it may be a table: {handler = handler_function [, prepare = prepare_function] [, name = name] }. handler_function is mandatory; it may have a single parameter = the socket; it is for continuous operation after the connection is made. prepare_function is optional; it is executed once before any connection is made. Examples:

```
socket.tcp_server('localhost', 3302, function (s) loop_loop() end)
socket.tcp_server('localhost', 3302, {handler=hfunc, name='name'})
socket.tcp_server('localhost', 3302, {handler=hfunc, prepare=pfunc})
```

For a fuller example see Use tcp_server to accept file contents sent with socat.

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.

Parameters

- data (string) – what is to be sent

Return the number of bytes sent.

Rtype number

Possible errors: nil on error.

socket_object:syswrite(size)
Write as much data as possible 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.

Parameters

- size (integer) – maximum number of bytes to receive. See Recommended size.

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

Parameters

- 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".
- options (table) – chunk=limit and/or delimiter=delimiter, for example {chunk=5, delimiter='x'}.

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 data from the socket buffer if non-blocking. In case the socket is blocking, sysread() can block the calling process. Rarely used. For details, see also this description.

Parameters

- size (integer) – maximum number of bytes to read, for example 50 means "stop after 50 bytes"

Return an empty string if there is nothing more to read, or a nil value if error, or a string up to size bytes long.

Rtype string

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.

Parameters

- host (string) – URL or IP address
- port (number) – port number

Return true for success, false for error. If return is false, use socket_object:errno() or socket_object:error() to see details.

Rtype boolean

socket_object:listen(backlog)
Start listening for incoming connections.

Parameters

- 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.

Parameters

- host (string) – URL or IP address
- port (number) – port number
- data (string) – what is to be sent

Return the number of bytes sent.

Rtype number

Possible errors: on error, returns nil and may return status, errno, errstr.

socket_object:recvfrom(size)
Receive a message on a UDP socket.

Parameters

- size (integer) – maximum number of bytes to receive. See Recommended size.

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.

Parameters

- 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 the Lua garbage collector removes its user data.

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_PEERCRED
- SO_PRIORITY
- SO_PROTOCOL
- SO_RCVBUF
- SO_RCVBUFFORCE
- SO_RCVLOWAT
- SO_SNDLOWAT
- SO_RCVTIMEO
- SO_SNDTIMEO
- SO_REUSEADDR
- SO_SNDBUF
- SO_SNDBUFFORCE
- SO_TIMESTAMP
- SO_TYPE

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.

Parameters

- 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

socket.iowait(fd, read-or-write-flags[, timeout])
The socket.iowait() function is used to wait until read-or-write activity occurs for a file descriptor.

Parameters

- fd – file descriptor
- read-or-write-flags – ‘R’ or 1 = read, ‘W’ or 2 = write, ‘RW’ or 3 = read|write.
- timeout – number of seconds to wait

If the fd parameter is nil, then there will be a sleep until the timeout. If the timeout parameter is nil or unspecified, then timeout is infinite.

Ordinarily the return value is the activity that occurred (‘R’ or ‘W’ or ‘RW’ or 1 or 2 or 3). If the timeout period goes by without any reading or writing, the return is an error = ETIMEDOUT.

Example: socket.iowait(sock:fd(), `'r'`, 1.11)

Recommended size

For recv and recvfrom: use the optional size parameter to limit the number of bytes to receive. A fixed size such as 512 is often reasonable; a pre-calculated size that depends on context – such as the message format or the state of the network – is often better. For recvfrom, be aware that a size greater than the Maximum Transmission Unit can cause inefficient transport. For Mac OS X, be aware that the size can be tuned by changing sysctl net.inet.udp.maxdgram.

If size is not stated: Tarantool will make an extra call to calculate how many bytes are necessary. This extra call takes time, therefore not stating size may be inefficient.

If size is stated: on a UDP socket, excess bytes are discarded. On a TCP socket, excess bytes are not discarded and can be received by the next call.

Examples

Use of a TCP socket over the Internet

In this example a connection is made over the internet between a Tarantool instance and tarantool.org, then an HTTP “head” message is sent, and a response is received: “HTTP/1.1 200 OK” or something else if the site has moved. 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/1.0\r\nHost: tarantool.org\r\n\r\n")
---
- 40
...
tarantool> sock:read(17)
---
- HTTP/1.1 302 Move
...
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')
---
- 1
...
tarantool> message = sock_1:recvfrom(512)
---
...
tarantool> message
```

(continues on next page)

(continued from previous page)

```
---
- 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 a server instance. 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,
{
  handler = function(s)
    while true do
      local request
      request = s:read("\n");
      if request == "" or request == nil then
        break
      end
      print(request)
    end
  end,
  prepare = function()
    print('Initialized')
  end
}
)
```

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 instance'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.

4.2.23 Module 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
---
...
```

4.2.24 Module string

Overview

The string module has everything in the standard Lua string library, and some Tarantool extensions.

In this section we only discuss the additional functions that the Tarantool developers have added.

Below is a list of all additional string functions.

Name	Use
string.ljust()	Left-justify a string
string.rjust()	Right-justify a string
string.hex()	Given a string, return hexadecimal values
string.fromhex()	Given hexadecimal values, return a string
string.startswith()	Check if a string starts with a given substring
string.endswith()	Check if a string ends with a given substring
string.lstrip()	Remove characters from the left of a string
string.rstrip()	Remove characters from the right of a string
string.split()	Split a string into a table of strings
string.strip()	Remove spaces on the left and right of a string

string.ljust(input-string, width[, pad-character])

Return the string left-justified in a string of length width.

Parameters

- input-string – (string) the string to left-justify

- width – (integer) the width of the string after left-justifying
- pad-character – (string) a single character, default = 1 space

Return left-justified string (unchanged if width <= string length)

Rtype string

Example:

```
tarantool> string = require('string')
---
...
tarantool> string.ljust(' A', 5)
---
- ' A   '
...
```

string.rjust(input-string, width[, pad-character])

Return the string right-justified in a string of length width.

Parameters

- input-string – (string) the string to right-justify
- width – (integer) the width of the string after right-justifying
- pad-character – (string) a single character, default = 1 space

Return right-justified string (unchanged if width <= string length)

Rtype string

Example:

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

string.hex(input-string)

Return the hexadecimal value of the input string.

Parameters

- input-string – (string) the string to process

Return hexadecimal, 2 hex-digit characters for each input character

Rtype string

Example:

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

string.fromhex(hexadecimal-input-string)
Given a string containing pairs of hexadecimal digits, return a string with one byte for each pair. This is the reverse of string.hex(). The hexadecimal-input-string must contain an even number of hexadecimal digits.

Parameters

- hexadecimal-input-string – (string) string with pairs of hexadecimal digits

Return string with one byte for each pair of hexadecimal digits

Rtype string

Example:

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

string.startswith(input-string, start-string[, start-pos[, end-pos]])
Return True if input-string starts with start-string, otherwise return False.

Parameters

- input-string – (string) the string where start-string should be looked for
- start-string – (string) the string to look for
- start-pos – (integer) position: where to start looking within input-string
- end-pos – (integer) position: where to end looking within input-string

Return true or false

Rtype boolean

start-pos and end-pos may be negative, meaning the position should be calculated from the end of the string.

Example:

```
tarantool> string = require('string')
---
...
tarantool> string.startswith(' A', 'A', 2, 5)
---
- true
...
```

string.endswith(input-string, end-string[, start-pos[, end-pos]])
Return True if input-string ends with end-string, otherwise return False.

Parameters

- input-string – (string) the string where end-string should be looked for
- end-string – (string) the string to look for
- start-pos – (integer) position: where to start looking within input-string
- end-pos – (integer) position: where to end looking within input-string

Return true or false

Rtype boolean

start-pos and end-pos may be negative, meaning the position should be calculated from the end of the string.

Example:

```
tarantool> string = require('string')
---
...
tarantool> string.endswith('Baa', 'aa')
---
- true
...
```

string.lstrip(input-string[, list-of-characers])

Return the value of the input string, after removing characters on the left. The optional list-of-characters parameter is a set not a sequence, so string.lstrip(...,'ABC') does not mean strip 'ABC', it means strip 'A' or 'B' or 'C'.

Parameters

- input-string – (string) the string to process
- list-of-characters – (string) what characters can be stripped. Default = space.

Return result after stripping characters from input string

Rtype string

Example:

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

string.rstrip(input-string[, list-of-characters])

Return the value of the input string, after removing characters on the right. The optional list-of-characters parameter is a set not a sequence, so string.rstrip(...,'ABC') does not mean strip 'ABC', it means strip 'A' or 'B' or 'C'.

Parameters

- input-string – (string) the string to process
- list-of-characters – (string) what characters can be stripped. Default = space.

Return result after stripping characters from input string

Rtype string

Example:

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

(continues on next page)

(continued from previous page)

```
---
- ' ABC'
...
```

string.split(input-string[, split-string])
Split input-string into one or more output strings in a table. The places to split are the places where split-string occurs.

Parameters

- input-string – (string) the string to split
- split-string – (string) the string to find within input-string. Default = space.

Return table of strings that were split from input-string

Rtype table

Example:

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

string.strip(input-string[, list-of-characters])
Return the value of the input string, after removing characters on the left and the right. The optional list-of-characters parameter is a set not a sequence, so string.strip(...,'ABC') does not mean strip 'ABC', it means strip 'A' or 'B' or 'C'.

Parameters

- input-string – (string) the string to process
- list-of-characters – (string) what characters can be stripped. Default = space.

Return result after stripping characters from input string

Rtype string

Example:

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

4.2.25 Module swim

The swim module contains Tarantool's implementation of SWIM – Scalable Weakly-consistent Infection-style Process Group Membership Protocol. It is recommended for any type of Tarantool cluster where the number of nodes can be large. Its job is to discover and monitor the other members in the cluster and keep their

information in a "member table". It works by sending and receiving, in a background event loop, periodically, via UDP, messages.

Each message has several parts, including the ping such as "I am checking whether you are alive", the event such as "I am joining", the anti-entropy such as "I know that another member exists", and the payload such as "I or another member could have user-generated data". The maximum message size is about 1500 bytes. SWIM sends messages periodically to a random subset of the member table. SWIM processes replies from those members asynchronously.

Each entry in the member table has: a UUID, and a status. The status is "alive", "suspected", "dead", or "left". When a member fails to acknowledge a certain number of pings, its status is changed from "alive" to "suspected", that is, suspected of being dead. But SWIM tries to avoid false positives (misidentifying members as dead) which could happen when a member is overloaded and responds to pings too slowly, or when there is network trouble and packets can not go through some channels. When a member is suspected, SWIM randomly chooses other members and sends requests to them: "please ping this suspected member". This is called an indirect ping. Thus via different routes and additional hops the suspected member gets additional chances to reply, and thus "refute" the suspicion.

Because selection is random there is an even network load of about one message per member per protocol step, regardless of the cluster size. This is a major feature of SWIM. Because the protocol depends on members passing information on, also known as "gossiping", members do not need to broadcast messages to every member, which would cause a network load of N messages per member per protocol step, where N is the number of members in the cluster. However, selection is not entirely random, there is a preference for selecting least-recently-pinged members, like a round-robin.

Regarding the anti-entropy part of a message: this is necessary for maintaining the status in entries of the member table. Consider an example where two members, #1 and #2, are both alive. No events happen so only pings are being sent periodically. Then a third member, #3 appears. It knows about one of the existing members, #2. How can it discover the other member? Certainly #1 could notify #2 and #2 could notify #3, but messages go via UDP, so any notification event can be lost. However, regular messages containing "ping" and/or "event" also can contain an "anti-entropy" section, which is taken from a randomly-chosen part of the member table. So for this example, #2 will eventually randomly add to a regular message the anti-entropy note that #1 is alive, and thus #3 will discover #1 even though it did not receive a direct "I am alive" event message from #1.

Regarding the UUID part of an entry in the member table: this is necessary for stable identification, because UUID changes more rarely than URI (a combination of IP and port number). But if the UUID does change, SWIM will include both the new and old UUID in messages, so all other members will eventually learn about the new UUID and change the member table accordingly.

Regarding the payload part of a message: this is not always necessary, it is a feature which allows passing user-generated information via SWIM instead of via node-to-node communication. The swim module has methods for specifying a "payload", which is arbitrary user data with a maximum size of about 1.2 KB. The payload can be anything, and it will be eventually disseminated over the cluster and available at other members. Each member can have its own payload.

Messages can be encrypted. Encryption may not be necessary in a closed network but is necessary for safety if the cluster is on the public Internet. Users can specify an encryption algorithm, an encryption mode, and a private key. All parts of all messages (including ping, acknowledgment, event, payload, URI, and UUID) will be encrypted with that private key, as well as a random public key generated for each message to prevent pattern attacks.

In theory the event dissemination speed (the number of hops to pass information throughout the cluster) is O(log(cluster_size)). For that and other theoretical information see the Cornell University paper which originally described SWIM https://www.cs.cornell.edu/projects/Quicksilver/public_pdfs/SWIM.pdf

swim.new([cfg])
: Create a new SWIM instance. A SWIM instance maintains a member table and interacts with other

members. Multiple SWIM instances can be created in a single Tarantool process.

Parameters

- cfg (table) – an optional configuration parameter. If cfg is not specified or is nil, then the new SWIM instance is not bound to a socket and has nil attributes, so it cannot interact with other members and only a few methods are valid until swim_object:cfg() is called. If cfg is specified, then the effect is the same as calling s = swim.new() s:cfg(). For configuration description see swim_object:cfg().

Returns swim-object a swim object

Example: swim_object = swim.new({uri = 3333, uuid = '00000000-0000-1000-8000-000000000001', heartbeat_rate = 0.1})

object swim_object

A swim object is an object returned by swim.new(). It has methods cfg(), delete(), is_configured(), size(), quit(), add_member(), remove_member(), probe_member(), broadcast(), set_payload(), set_payload_raw(), set_codec(), self(), member_by_uuid(), pairs().

swim_object:cfg(cfg)

Configure or reconfigure a SWIM instance.

Parameters

- cfg (table) – the options to describe instance behavior

The cfg table may have these components:

heartbeat_rate (double) – rate of sending round messages, in seconds. Setting heartbeat_rate to X does not mean that every member will be checked every X seconds, instead X is the protocol speed. Protocol period depends on member count and heartbeat_rate. Default = 1.

ack_timeout (double) – time in seconds after which a ping is considered to be unacknowledged. Default = 30.

gc_mode (enum) – dead member collection mode. If gc_mode == 'off' then SWIM never removes dead members from the member table (though users may remove them with swim_object:remove_member()), and SWIM will continue to ping them as if they were alive. If gc_mode == 'on' then SWIM removes dead members from the member table after one round. Default = 'on'.

uri (string or number) – either an 'ip:port' address, or just a port number (if ip is omitted then 127.0.0.1 is assumed). If port == 0, then the kernel will select any free port for the IP address.

uuid (string or cdata struct tt_uuid) – a value which should be unique among SWIM instances. Users may choose any value but the recommendation is: use box.cfg.instance_uuid, the Tarantool instance's UUID.

All the cfg components are dynamic – swim_object:cfg() may be called more than once. If it is not being called for the first time and a component is not specified, then the component retains its previous value. If it is being called for the first time then uri and uuid are mandatory, since a SWIM instance cannot operate without URI and UUID.

swim_object:cfg() is atomic – if there is an error, then nothing changes.

Return true if configuration succeeds

Return nil, err if an error occurred. err is an error object

Example: swim_object:cfg({heartbeat_rate = 0.5})

After swim_object:cfg(), all other swim_object methods are callable.

cfg.

Expose all non-nil components of the read-only table which was set up or changed by swim_object:cfg().

Example:

```
tarantool> swim_object.cfg
---
- gc_mode: off
  uri: 3333
  uuid: 00000000-0000-1000-8000-000000000001
...
```

swim_object:delete()

Delete a SWIM instance immediately. Its memory is freed, its member table entry is deleted, and it can no longer be used. Other members will treat this member as 'dead'. After swim_object:delete() any attempt to use the deleted instance will cause an exception to be thrown.

Return none, this method does not fail

Example: swim_object:delete()

swim_object:is_configured()

Return false if a SWIM instance was created via swim.new() without an optional cfg argument, and was not configured with swim_object:cfg(). Otherwise return true.

Return boolean result, true if configured, otherwise false

Example: swim_object:is_configured()

swim_object:size()

Return the size of the member table. It will be at least 1 because the "self" member is included.

Return integer size

Example: swim_object:size()

swim_object:quit()

Leave the cluster. This is a graceful equivalent of swim_object:delete() – the instance is deleted, but before deletion it sends to each member in its member table a message, that this instance has left the cluster, and should not be considered dead. Other instances mark such member in their tables as 'left', and drop it after one round of dissemination. Consequences to the caller are the same as after swim_object:delete() – the instance is no longer usable, and an error will be thrown if there is an attempt to use it.

Return none, the method does not fail

Example: swim_object:quit()

swim_object:add_member(cfg)

Explicitly add a member into the member table. This method is useful when a new member is joining the cluster and does not yet know what members already exist. In that case it can start interaction explicitly by adding the details about an already-existing member into its member table. Subsequently SWIM will discover other members automatically via messages from the already-existing member.

Parameters

- cfg (table) – description of the member

The cfg table has two mandatory components, uuid and uri, which have the same format as uuid and uri in the table for swim_object:cfg().

Return true if member is added

Return nil, err if an error occurred. err is an error object

Example: swim_member_object = swim_object:add_member({uuid = ..., uri = ...})

swim_object:remove_member(uuid)
Explicitly and immediately remove a member from the member table.

Parameters

- uuid (string-or-cdata-struct-tt_uuid) – UUID

Return true if member is removed

Return nil, err if an error occurred. err is an error object.

Example: swim_object:delete('00000000-0000-1000-8000-000000000001')

swim_object:probe_member(uri)
Send a ping request to the specified uri address. If another member is listening at that address, it will receive the ping, and respond with an ACK (acknowledgment) message containing information such as UUID. That information will be added to the member table. swim_object:probe_member() is similar to swim_object:add_member(), but it does not require UUID, and it is not reliable because it uses UDP.

Parameters

- uri (string-or-number) – URI. Format is the same as for uri in swim_object:cfg().

Return true if member is pinged

Return nil, err if an error occurred. err is an error object.

Example: swim_object:probe_member(3333)

swim_object:broadcast([port])
Broadcast a ping request to all the network interfaces in the system. swim_object:broadcast() is like swim_object:probe_member() to many members at once.

Parameters

- port (number) – All the sent ping requests have this port as destination port in their UDP headers. By default a currently bound port is used.

Return true if broadcast is sent

Return nil, err if an error occurred. err is an error object.

Example:

```
tarantool> tarantool> fiber = require('fiber')
---
...
tarantool> swim = require('swim')
---
...
tarantool> s1 = swim.new({uri = 3333, uuid = '00000000-0000-1000-8000-000000000001', heartbeat_
↪rate = 0.1})
---
...
tarantool> s2 = swim.new({uri = 3334, uuid = '00000000-0000-1000-8000-000000000002', heartbeat_
↪rate = 0.1})
---
```

(continues on next page)

(continued from previous page)

```
...
tarantool> s1:size()
---
- 1
...
tarantool> s1:add_member({uri = s2:self():uri(), uuid = s2:self():uuid()})
---
- true
...
tarantool> s1:size()
---
- 1
...
tarantool> s2:size()
---
- 1
...

tarantool> fiber.sleep(0.2)
---
...
tarantool> s1:size()
---
- 2
...
tarantool> s2:size()
---
- 2
...
tarantool> s1:remove_member(s2:self():uuid()) s2:remove_member(s1:self():uuid())
---
...
tarantool> s1:size()
---
- 1
...
tarantool> s2:size()
---
- 1
...

tarantool> s1:probe_member(s2:self():uri())
---
- true
...
tarantool> fiber.sleep(0.1)
---
...
tarantool> s1:size()
---
- 2
...
tarantool> s2:size()
---
- 2
...
```

(continues on next page)

(continued from previous page)

```
tarantool> s1:remove_member(s2:self():uuid()) s2:remove_member(s1:self():uuid())
---
...
tarantool> s1:size()
---
- 1
...
tarantool> s2:size()
---
- 1
...
tarantool> s1:broadcast(3334)
---
- true
...
tarantool> fiber.sleep(0.1)
---
...
tarantool> s1:size()
---
- 2
...

tarantool> s2:size()
---
- 2
...
```

swim_object:set_payload(payload)

Set a payload, as formatted data. Payload is arbitrary user defined data up to 1200 bytes in size and disseminated over the cluster. So each cluster member will eventually learn what is the payload of other members in the cluster, because it is stored in the member table and can be queried with swim_member_object:payload(). Different members may have different payloads.

Parameters

- payload (object) – Arbitrary Lua object to disseminate. Set to nil to remove the payload, in which case it will be eventually removed on other instances. The object is serialized in MessagePack.

Return true if payload is set

Return nil, err if an error occurred. err is an error object

Example: swim_object:set_payload({field1 = 100, field2 = 200})

swim_object:set_payload_raw(payload[, size])

Set a payload, as raw data. Sometimes a payload does not need to be a Lua object. For example, a user may already have a well formatted MessagePack object and just wants to set it as a payload. Or cdata needs to to be exposed. set_payload_raw allows setting a payload as is, without MessagePack serialization.

Parameters

- payload (string-or-cdata) – any value
- size (number) – Payload size in bytes. If payload is string then size is optional, and if specified, then should not be larger than actual payload size. If size is less than

actual payload size, then only the first size bytes of payload are used. If payload is cdata then size is mandatory.

Return true if payload is set

Return nil, err if an error occurred. err is an error object

Example:

```
tarantool> tarantool> ffi = require('ffi')
---
...
tarantool> fiber = require('fiber')
---
...
tarantool> swim = require('swim')
---
...
tarantool> s1 = swim.new({uri = 0, uuid = '00000000-0000-1000-8000-000000000001', heartbeat_
↪rate = 0.1})
---
...
tarantool> s2 = swim.new({uri = 0, uuid = '00000000-0000-1000-8000-000000000002', heartbeat_
↪rate = 0.1})
---
...
tarantool> s1:add_member({uri = s2:self():uri(), uuid = s2:self():uuid()})
---
- true
...
tarantool> s1:set_payload({a = 100, b = 200})
---
- true
...
tarantool> s2:set_payload('any payload')
---
- true
...
tarantool> fiber.sleep(0.2)
---
...
tarantool> s1_view = s2:member_by_uuid(s1:self():uuid())
---
...
tarantool> s2_view = s1:member_by_uuid(s2:self():uuid())
---
...
tarantool> s1_view:payload()
---
- {'a': 100, 'b': 200}
...
tarantool> s2_view:payload()
---
- any payload
...
tarantool> cdata = ffi.new('char[?]', 2)
---
...
tarantool> cdata[0] = 1
```

(continues on next page)

(continued from previous page)

```
---
...
tarantool> cdata[1] = 2
---
...
tarantool> s1:set_payload_raw(cdata, 2)
---
- true
...
tarantool> fiber.sleep(0.2)
---
...
tarantool> cdata, size = s1_view:payload_cdata()
---
...
tarantool> cdata[0]
---
- 1
...
tarantool> cdata[1]
---
- 2
...
tarantool> size
---
- 2
...
```

swim_object:set_codec(codec_cfg)

Enable encryption for all following messages.

For a brief description of encryption algorithms see "enum_crypto_algo" and "enum crypto_mode" in the Tarantool source code file crypto.h.

When encryption is enabled, all the messages are encrypted with a chosen private key, and a randomly generated and updated public key.

Parameters

- codec_cfg (table) – description of the encryption

The components of the codec_cfg table may be:

algo (string) – encryption algorithm name. All the names in module crypto are supported: 'aes128', 'aes192', 'aes256', 'des'. Specify 'none' to disable encryption.

mode (string) – encryption algorithm mode. All the modes in module crypto are supported: 'ecb', 'cbc', 'cfb', 'ofb'. Default = 'cbc'.

key (cdata or string) – a private secret key which is kept secret and should never be stored hardcoded in source code.

key_size (integer) – size of the key in bytes. key_size is mandatory if key is cdata. key_size is optional if key is string, and if key_size is shorter than than actual key size then the key is truncated.

algo and mode and key and key_size should be the same for all SWIM instances, so that members can understand each others' messages.

Example;

```
tarantool> tarantool> swim = require('swim')
---
...
tarantool> s1 = swim.new({uri = 0, uuid = '00000000-0000-1000-8000-000000000001'})
---
...
tarantool> s1:set_codec({algo = 'aes128', mode = 'cbc', key = '1234567812345678'})
---
- true
...
```

swim_object:self()
: Return a swim member object (of self) from the member table, or from a cache containing earlier results of swim_object:self() or swim_object:member_by_uuid() or swim_object:pairs().

  Return swim member object, not nil because self() will not fail

  Example: swim_member_object = swim_object:self()

swim_object:member_by_uuid(uuid)
: Return a swim member object (given UUID) from the member table, or from a cache containing earlier results of swim_object:self() or swim_object:member_by_uuid() or swim_object:pairs().

  Parameters

  - uuid (string-or-cdata-struct-tt-uuid) – UUID

  Return swim member object, or nil if not found

  Example: swim_member_object = swim_object:member_by_uuid('00000000-0000-1000-8000-000000000001')

swim_object:pairs()
: Set up an iterator for returning swim member objects from the member table, or from a cache containing earlier results of swim_object:self() or swim_object:member_by_uuid() or swim_object:pairs().

  swim_object:pairs() should be in a 'for' loop, and there should only be one iterator in operation at one time. (The iterator is implemented in an extra light fashion so only one iterator object is available per SWIM instance.)

  Parameters

  - generator+object+key (varies) – as for any Lua pairs() iterators. generator function, iterator object (a swim member object), and initial key (a UUID).

  Example:

```
tarantool> tarantool> fiber = require('fiber')
---
...
tarantool> swim = require('swim')
---
...
tarantool> s1 = swim.new({uri = 0, uuid = '00000000-0000-1000-8000-000000000001', heartbeat_
↪rate = 0.1})
---
...
tarantool> s2 = swim.new({uri = 0, uuid = '00000000-0000-1000-8000-000000000002', heartbeat_
↪rate = 0.1})
---
```

(continues on next page)

(continued from previous page)

```
...
tarantool> s1:add_member({uri = s2:self():uri(), uuid = s2:self():uuid()})
---
- true
...
tarantool> fiber.sleep(0.2)
---
...
tarantool> s1:self()
---
- uri: 127.0.0.1:62341
  status: alive
  incarnation: 1
  uuid: 00000000-0000-1000-8000-000000000001
  payload_size: 0
...
tarantool> s1:member_by_uuid(s1:self():uuid())
---
- uri: 127.0.0.1:62341
  status: alive
  incarnation: 1
  uuid: 00000000-0000-1000-8000-000000000001
  payload_size: 0
...
tarantool> s1:member_by_uuid(s2:self():uuid())
---
- uri: 127.0.0.1:55435
  status: alive
  incarnation: 1
  uuid: 00000000-0000-1000-8000-000000000002
  payload_size: 0
...
tarantool> t = {}
---
...
tarantool> for k, v in s1:pairs() do table.insert(t, {k, v}) end
---
...
tarantool> t
---
- - - 00000000-0000-1000-8000-000000000002
    - uri: 127.0.0.1:55435
      status: alive
      incarnation: 1
      uuid: 00000000-0000-1000-8000-000000000002
      payload_size: 0
  - - 00000000-0000-1000-8000-000000000001
    - uri: 127.0.0.1:62341
      status: alive
      incarnation: 1
      uuid: 00000000-0000-1000-8000-000000000001
      payload_size: 0
...
```

object swim_member_object

Methods swim_object:member_by_uuid(), swim_object:self(), and swim_object:pairs()

return swim member objects. A swim member object has methods for reading its attributes: status(), uuid, uri(), incarnation(), payload_cdata, payload_str(), payload(), is_dropped().

swim_member_object:status()
Return the status, which may be 'alive', 'suspected', 'left', or 'dead'.

Return string 'alive' | 'suspected' | 'left' | dead'

swim_member_object:uuid()
Return the UUID as cdata struct tt_uuid.

Return cdata-struct-tt-uuid UUID

swim_member_object:uri()
Return the URI as a string 'ip:port'. Via this method a user can learn a real assigned port, if port = 0 was specified in swim_object:cfg().

Return string ip:port

swim_member_object:incarnation()
Return a number (unsigned integer) which is incremented each time a member is updated.

Return unsigned-integer incarnation

swim_member_object:payload_cdata()
Return member's payload.

Return pointer-to-cdata payload and size in bytes

swim_member_object:payload_str()
Return payload as a string object. Payload is not decoded. It is just returned as a string instead of cdata. If payload was not specified by swim_object:set_payload() or by swim_object:set_payload_raw(), then its size is 0 and nil is returned.

Return string-object payload, or nil if there is no payload

swim_member_object:payload()
Since the swim module is a Lua module, a user is likely to use Lua objects as a payload – tables, numbers, strings etc. And it is natural to expect that swim_member_object:payload() should return the same object which was passed into swim_object:set_payload() by another instance. swim_member_object:payload() tries to interpret payload as MessagePack, and if that fails then it returns the payload as a string.

swim_member_object:payload() caches its result. Therefore only the first call actually decodes cdata payload. All following calls return a pointer to the same result, unless payload is changed with a new incarnation. If payload was not specified (its size is 0), then nil is returned.

swim_member_object:is_dropped()
Returns true if this member object is a stray reference to a member which has already been dropped from the member table.

Return boolean true if member is dropped, otherwise false

Example:

```
tarantool> swim = require('swim')
---
...
tarantool> s = swim.new({uri = 0, uuid = '00000000-0000-1000-8000-000000000001'})
---
...
tarantool> self = s:self()
```

(continues on next page)

(continued from previous page)

```
---
...
tarantool> self:status()
---
- alive
...
tarantool> self:uuid()
---
- 00000000-0000-1000-8000-000000000001
...
tarantool> self:uri()
---
- 127.0.0.1:56367
...
tarantool> self:incarnation()
---
- 1
...
tarantool> self:is_dropped()
---
- false
...
tarantool> s:set_payload_raw('123')
---
- true
...
tarantool> self:payload_cdata()
---
- 'cdata<const char *>: 0x0103500050'
- 3
...
tarantool> self:payload_str()
---
- '123'
...
tarantool> s:set_payload({a = 100})
---
- true
...
tarantool> self:payload_cdata()
---
- 'cdata<const char *>: 0x0103500050'
- 4
...
tarantool> self:payload_str()
---
- !!binary gaFhZA==
...
tarantool> self:payload()
---
- {'a': 100}
...
```

swim_member_object:on_member_event(trigger-function[, ctx])
: Create an "on_member trigger". The trigger-function will be executed when a member in the member table is updated.

Parameters

- trigger-function (function) – this will become the trigger function
- ctx (cdata) – (optional) this will be passed to trigger-function

Return nil or function pointer.

The trigger-function should have three parameter declarations (Tarantool will pass values for them when it invokes the function): the member which is having the member event, the event object, and the ctx which will be the same value as what is passed to swim_object:on_member_event.

A member event is any of: appearance of a new member, drop of an existing member, or update of an existing member.

An event object is an object which the trigger-function can use for determining what type of member event has happened. The object's methods – such as is_new_status(), is_new_uri(), is_new_incarnation(), is_new_payload(), is_drop() – return boolean values.

A member event may have more than one associated trigger. Triggers are executed sequentially. Therefore if a trigger function causes yields or sleeps, other triggers may be forced to wait. However, since trigger execution is done in a separate fiber, SWIM itself is not forced to wait.

Example of an on-member trigger function:

```
tarantool> swim = require('swim')

local function on_event(member, event, ctx)
    if event:is_new() then
        ...
    elseif event:is_drop() then
        ...
    end

    if event:is_update() then
        -- All next conditions can be
        -- true simultaneously.
        if event:is_new_status() then
...
        end
        if event:is_new_uri() then
...
        end
        if event:is_new_incarnation() then
...
        end
        if event:is_new_payload() then
...
        end
    end
end
```

Notice in the above example that the function is ready for the possibility that multiple events can happen simultaneously for a single trigger activation. is_new() and is_drop() can not both be true, but is_new() and is_update() can both be true, or is_drop() and is_update() can both be true. Multiple simultaneous events are especially likely if there are many events and trigger functions are slow – in that case, for example, a member might be added and then updated after a while, and then after a while there will be a single trigger activation.

Also: is_new() and is_new_payload() can both be true. This case is not due to trigger functions that are slow. It occurs because "omitted payload" and "size-zero payload" are not the same thing.

For example: when a ping is received, a new member might be added, but ping messages do not include payload. The payload will appear later in a different message. If that is important for the application, then the function should not assume when is_new() is true that the member already has a payload, and should not assume that payload size says something about the payload's presence or absence.

Also: functions should not assume that is_new() and is_drop() will always be seen. If a new member appears but then is dropped before its appearance has caused a trigger activation, then there will be no trigger activation.

swim_member_object:on_member_event(nil, old-trigger)
Delete an on-member trigger.

Parameters

- old-trigger (function) – old-trigger

The old-trigger value should be the value returned by on_member_event(trigger-function[, ctx]).

swim_member_object:on_member_event(new-trigger, old-trigger[, ctx])
This is a variation of on_member_event(new-trigger, [, ctx]). The additional parameter is old-trigger. Instead of adding the new-trigger at the end of a list of triggers, this function will replace the entry in the list of triggers that matches old-trigger. The position within a list may be important because triggers are activated sequentially starting with the first trigger in the list.

The old-trigger value should be the value returned by on_member_event(trigger-function[, ctx]).

swim_member_object:on_member_event()
Return the list of on-member triggers.

SWIM internals

The SWIM internals section is not necessary for programmers who wish to use the SWIM module, it is for programmers who wish to change or replace the SWIM module.

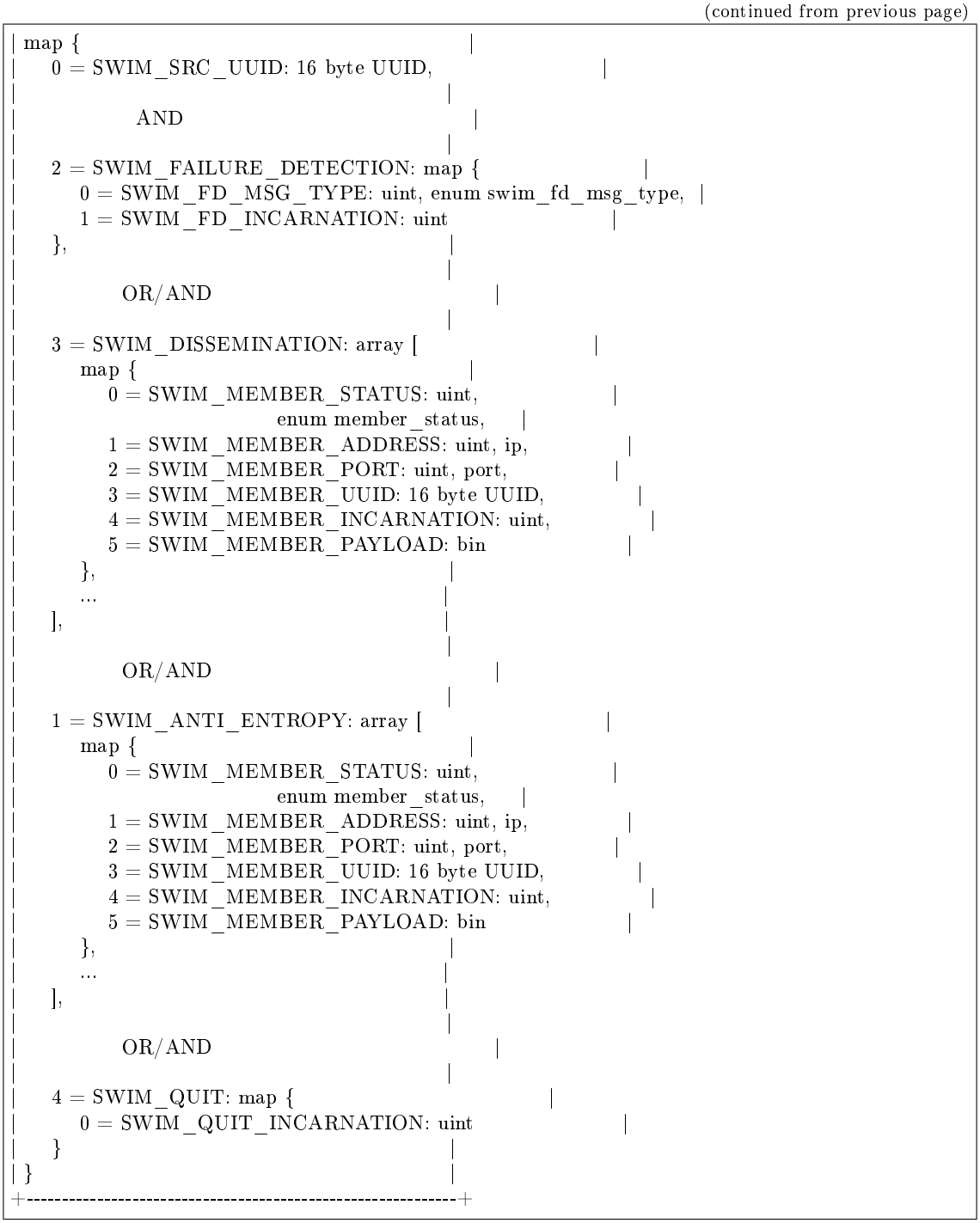
The SWIM wire protocol is open, will be backward compatible in case of any changes, and can be implemented by users who wish to simulate their own SWIM cluster members because they use another language than Lua, or another environment unrelated to Tarantool. The protocol is encoded as MsgPack.

```
SWIM packet structure:

+-----------------Public data, not encrypted------------------+
|                                                             |
|     Initial vector, size depends on chosen algorithm.      |
|                 Next data is encrypted.                     |
|                                                             |
+----------Meta section, handled by transport level-----------+
| map {                                                       |
|     0 = SWIM_META_TARANTOOL_VERSION: uint, Tarantool        |
|                                version ID,                  |
|     1 = SWIM_META_SRC_ADDRESS: uint, ip,                    |
|     2 = SWIM_META_SRC_PORT: uint, port,                     |
|     3 = SWIM_META_ROUTING: map {                            |
|         0 = SWIM_ROUTE_SRC_ADDRESS: uint, ip,               |
|         1 = SWIM_ROUTE_SRC_PORT: uint, port,                |
|         2 = SWIM_ROUTE_DST_ADDRESS: uint, ip,               |
|         3 = SWIM_ROUTE_DST_PORT: uint, port                 |
|     }                                                       |
| }                                                           |
+-------------------Protocol logic section--------------------+
```

(continues on next page)

(continued from previous page)



The Initial vector section appears only when encryption is enabled. This section contains a public key. For example, for AES algorithms it is a 16-byte initial vector stored as is. When no encryption is used, the section size is 0.

The later sections (Meta and Protocol Logic) are encrypted as one big data chunk if encryption is enabled.

The Meta section handles routing and protocol versions compatibility. It works at the 'transport' level of the SWIM protocol, and is always present. Keys in the meta section are:

SWIM_META_TARANTOOL_VERSION – mandatory field. Tarantool sets here its version as a 3 byte

integer: 1 byte for major, 1 byte for minor, 1 byte for patch. For example, Tarantool version 2.1.3 would be encoded like this: (((2 << 8) | 1) << 8) | 3;. This field will be used to support multiple versions of the protocol.

SWIM_META_SRC_ADDRESS and SWIM_META_SRC_PORT – mandatory. source IP address and port. IP is encoded as 4 bytes. "xxx.xxx.xxx.xxx" where each 'xxx' is encoding of one byte. Port is encoded as an integer. Example of how to encode "127.0.0.1:3313":

```
struct in_addr addr;
inet_aton("127.0.0.1", &addr);
pos = mp_encode_uint(pos, SWIM_META_SRC_ADDRESS);
pos = mp_encode_uint(pos, addr->s_addr);
pos = mp_encode_uint(pos, SWIM_META_SRC_PORT);
pos = mp_encode_uint(pos, 3313);
```

SWIM_META_ROUTING subsection – not mandatory. Responsible for packet forwarding. Used by SWIM suspicion mechanism. Read about suspicion in the SWIM paper. If this subsection is present then the following fields are mandatory: SWIM_ROUTE_SRC_ADDRESS and SWIM_ROUTE_SRC_PORT (source IP address and port) (should be an address of the message originator (can differ from SWIM_META_SRC_ADDRESS and from SWIM_META_SRC_ADDRESS_PORT); SWIM_ROUTE_DST_ADDRESS and SWIM_ROUTE_DST_PORT (destination IP address and port, for the the message's final destination). If a message was sent indirectly with help of SWIM_META_ROUTING, then the reply should be sent back by the same route.

For an example of how SWIM uses routing for indirect pings ... Assume there are 3 nodes: S1, S2, S3. S1 sends a message to S3 via S2. The following steps are executed in order to deliver the message:

```
S1 -> S2
{ src: S1, routing: {src: S1, dst: S3}, body: ... }
```

S2 receives the message and sees that routing.dst is not equal to S2, so it is a foreign packet. S2 forwards the packet to S3 preserving all the data including body and routing sections.

```
S2 -> S3
```

S3 receives the message and sees that routing.dst is equal to S3, so the message is delivered. If S3 wants to answer, it sends a response via the same proxy. It knows that the message was delivered from S2, so it sends an answer via S2.

The Protocol logic section handles SWIM logical protocol steps and actions.

SWIM_SRC_UUID – mandatory field. SWIM uses UUID as a unique identifier of a member, not IP/port. This field stores UUID of sender. Its type is MP_BIN. Size is always 16 bytes. UUID is encoded in host byte order, no bswaps are needed.

Next sections can all be present, or only some of them. A connector should be ready to handle any combinations.

In each section a member or the section as a whole has an incarnation number. This number is used so that old messages will be ignored, and false messages will be refuted. If the member incarnation is less than the locally stored incarnation, then the message is outdated. This can happen because UDP allows reordering and duplication.

Refutation usually happens when a false-positive failure detection has happened. In such a case the member thought to be dead receives that information from other members, increases its own incarnation, and spreads a message saying the member is alive (a "refutation").

When a member's incarnation number in a message is larger than the local number, all its attributes (IP, port, status) should be updated with the values received in the message. Payload is a bit different. Payload

can be updated only if it is present in the message. Because of its huge size (in comparison with UDP packet max size) payload is not always sent with every message.

SWIM_FAILURE_DETECTION subsection – describes a ping or ACK. In the SWIM_FAILURE_DETECTION subsection are: SWIM_FD_MSG_TYPE (0 is ping, 1 is ack); SWIM_FD_INCARNATION (incarnation number of the sender).

SWIM_DISSEMINATION subsection – a list of changed cluster members. It may include only a subset of changed cluster members if there are too many changes to fit into one UDP packet; In the SWIM_DISSEMINATION subsection are: SWIM_MEMBER_STATUS (mandatory) (0 = alive, 1 = suspected, 2 = dead, 3 = left); SWIM_MEMBER_ADDRESS and SWIM_MEMBER_PORT (mandatory) member IP and port; SWIM_MEMBER_UUID (mandatory) (member UUID); SWIM_MEMBER_INCARNATION (mandatory) (member incarnation number); SWIM_MEMBER_PAYLOAD (not mandatory) (member payload) (MessagePack type is MP_BIN). Note that absence of SWIM_MEMBER_PAYLOAD means nothing - it is not the same as a payload with zero size.

SWIM_ANTI_ENTROPY subsection – a helper for the dissemination. It contains all the same fields as the dissemination, but all of them are mandatory, including payload even when payload size is 0. Anti-entropy eventually spreads changes which for any reason are not spread by the dissemination.

SWIM_QUIT subsection – statement that the sender has left the cluster gracefully, for example via swim_object:quit(), and should not be considered dead. Sender status should be changed to 'left'. In the SWIM_QUIT subsection is: SWIM_QUIT_INCARNATION (sender incarnation number).

4.2.26 Module table

The table module has everything in the standard Lua table library, and some Tarantool extensions.

You can see this by saying "table": you will see this list of functions: clear (LuaJIT extension = erase all elements), concat (concatenate), copy (make a copy of an array), deepcopy (see description below), foreach, foreach1, getn (get the number of elements in an array), insert (insert an element into an array), maxn (get largest index) move (move elements between tables), new (LuaJIT extension = return a new table with pre-allocated elements), remove (remove an element from an array), sort (sort the elements of an array).

In this section we only discuss the additional function that the Tarantool developers have added: deepcopy.

table.deepcopy(input-table)

Return a "deep" copy of the table – a copy which follows nested structures to any depth and does not depend on pointers, it copies the contents.

Parameters

- input-table – (table) the table to copy

Return the copy of the table

Rtype table

Example:

```
tarantool> input_table = {1,{'a','b'}}
---
...

tarantool> output_table = table.deepcopy(input_table)
---
...
```

(continues on next page)

(continued from previous page)

```
tarantool> output_table
---
- - 1
  - - a
    - b
...
```

4.2.27 Module tap

Overview

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.

Index

Name	Use
tap.test()	Initialize
taptest:test()	Create a subtest and print the results
taptest:plan()	Indicate how many tests to perform
taptest:check()	Check the number of tests performed
taptest:diag()	Display a diagnostic message
taptest:ok()	Evaluate the condition and display the message
taptest:fail()	Evaluate the condition and display the message
taptest:skip()	Evaluate the condition and display the message
taptest:is()	Check if the two arguments are equal
taptest:isnt()	Check if the two arguments are different
taptest:is_deeply()	Recursively check if the two arguments are equal
taptest:like()	Check if the argument matches a pattern
taptest:unlike()	Check if the argument does not match a pattern
taptest:isnil() taptest:isstring() taptest:isnumber() taptest:istable() taptest:isboolean() taptest:isudata() taptest:iscdata()	Check if a value has a particular type
taptest.strict	Flag, true if comparisons with nil should be strict

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.

Parameters

- 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:test(test-name, func)

Create a subtest (if no func argument specified), or (if all arguments are specified) create a subtest, run the test function and print the result.

See the example.

Parameters

- name (string) – an arbitrary name to give for the test outputs.
- fun (function) – the test logic to run.

Return taptest

Rtype userdata or string

taptest:plan(count)

Indicate how many tests will be performed.

Parameters

- count (number) –

Return nil

taptest:check()

Checks the number of tests performed.

The result will be a display saying # bad plan: ... if the number of completed tests is not equal to the number of tests specified by taptest:plan(...). (This is a purely Tarantool feature: "bad plan" messages are out of the TAP13 standard.)

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. However, as a Tarantool extension, taptest:check() may appear at the end of any subtest. Therefore there are three ways to cause the check:

- by calling taptest:check() at the end of a script,
- by calling a function which ends with a call to taptest:check(),
- or by calling taptest:test('...', subtest-function-name) where subtest-function-name does not need to end with taptest:check() because it can be called after the subtest is complete.

Return true or false.

Rtype boolean

taptest:diag(message)
Display a diagnostic message.

Parameters

- 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.

Parameters

- condition (boolean) – an expression which is true or false
- test-name (string) – name of the 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.

Parameters

- test-name (string) – name of the test

Return true or false.

Rtype boolean

taptest:skip(message)
taptest:skip('x') is equivalent to taptest:ok(true, 'x' .. '# skip'). Displays the message.

Parameters

- test-name (string) – name of the 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.

  Parameters

  - got (number) – actual result
  - expected (number) – expected result
  - test-name (string) – name of the test

  Return true or false.

  Rtype boolean

taptest:isnt(got, expected, test-name)
: This is the negation of taptest:is().

  Parameters

  - got (number) – actual result
  - expected (number) – expected result
  - test-name (string) – name of the test

  Return true or false.

  Rtype boolean

taptest:is_deeply(got, expected, test-name)
: Recursive version of taptest:is(...), which can be be used to compare tables as well as scalar values.

  Return true or false.

  Rtype boolean

  Parameters

  - got (lua-value) – actual result
  - expected (lua-value) – expected result
  - test-name (string) – name of the test

taptest:like(got, expected, test-name)
: Verify a string against a pattern. Ok if match is found.

  Return true or false.

  Rtype boolean

  Parameters

  - got (lua-value) – actual result
  - expected (lua-value) – pattern
  - test-name (string) – name of the test

```
test:like(tarantool.version, '^[1-9]', "version")
```

taptest:unlike(got, expected, test-name)
: This is the negation of taptest:like().

  Parameters

  - got (number) – actual result
  - expected (number) – pattern
  - test-name (string) – name of the 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.

  Parameters

  - value (lua-value) –
  - test-name (string) – name of the test

  Return true or false.

  Rtype boolean

taptest.strict
: Set taptest.strict=true if taptest:is() and taptest:isnt() and taptest:is_deeply() must be compared strictly with nil. Set taptest.strict=false if nil and box.NULL both have the same effect. The default is false. For example, if and only if taptest.strict=true has happened, then taptest:is_deeply({a = box.NULL}, {}) will return false.

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
```

4.2.28 Module 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
...
```

4.2.29 Module uuid

Overview

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.

Index

Below is list of all uuid functions and members.

Name	Use
uuid.nil	A nil object
uuid() uuid.bin() uuid.str()	Get a UUID
uuid.fromstr() uuid.frombin() uuid_object:bin() uuid_object:str()	Get a converted UUID
uuid_object:isnil()	Check if a UUID is an all-zero value

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)

: Parameters

  - uuid_str – UUID in 36-byte hexadecimal string

: Return converted UUID

: Rtype cdata

uuid.frombin(uuid_bin)

: Parameters

  - 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

  Parameters

  - 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

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
...
```

4.2.30 Module utf8

Overview

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

The module is fully built-in so require('utf8') is not necessary.

Name	Use
casecmp and cmp	Comparisons
lower and upper	Case conversions
isalpha, isdigit, islower and isupper	Determine character types
sub	Substrings
len	Length in characters
next	Character-at-a-time iterations

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

Parameters

- string (UTF8-string) – a string encoded with UTF-8

Return -1 meaning "less", 0 meaning "equal", +1 meaning "greater"

Rtype number

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

The comparison is done with primary weights. Therefore the elements which affect secondary or later weights (such as "case" in Latin or Cyrillic alphabets, or "kana differentiation" in Japanese) are ignored. If asked "is this like a Microsoft case-insensitive accent-insensitive collation" we tend to answer "yes", though the Unicode Collation Algorithm is far more sophisticated than those terms imply.

Example:

```
tarantool> utf8.casecmp('é','e'),utf8.casecmp('E','e')
---
- 0
- 0
...
```

utf8.char(code-point[, code-point ...])

Parameters

- number (code-point) – a Unicode code point value, repeatable

Return a UTF-8 string

Rtype string

The code-point number is the value that corresponds to a character in the Unicode Character Database This is not the same as the byte values of the encoded character, because the UTF-8 encoding scheme is more complex than a simple copy of the code-point number.

Another way to construct a string with Unicode characters is with the \u{hex-digits} escape mechanism, for example '\u{41}\u{42}' and utf8.char(65,66) both produce the string 'AB'.

Example:

```
tarantool> utf8.char(229)
---
- å
...
```

utf8.cmp(UTF8-string, utf8-string)

Parameters

- string (UTF8-string) – a string encoded with UTF-8

Return -1 meaning "less", 0 meaning "equal", +1 meaning "greater"

Rtype number

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

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

Example:

```
tarantool> utf8.cmp('é','e'),utf8.cmp('E','e')
---
- 1
- 1
...
```

utf8.isalpha(UTF8-character)

Parameters

- string-or-number (UTF8-character) – a single UTF8 character, expressed as a one-byte string or a code point value

Return true or false

Rtype boolean

Return true if the input character is an "alphabetic-like" character, otherwise return false. Generally speaking a character will be considered alphabetic-like provided it is typically used within a word, as opposed to a digit or punctuation. It does not have to be a character in an alphabet.

Example:

```
tarantool> utf8.isalpha('Ж'),utf8.isalpha('å'),utf8.isalpha('9')
---
- true
- true
- false
...
```

utf8.isdigit(UTF8-character)

Parameters

- string-or-number (UTF8-character) – a single UTF8 character, expressed as a one-byte string or a code point value

Return true or false

Rtype boolean

Return true if the input character is a digit, otherwise return false.

Example:

```
tarantool> utf8.isdigit('Ж'),utf8.isdigit('å'),utf8.isdigit('9')
---
- false
- false
- true
...
```

utf8.islower(UTF8-character)

Parameters

- string-or-number (UTF8-character) – a single UTF8 character, expressed as a one-byte string or a code point value

Return true or false

Rtype boolean

Return true if the input character is lower case, otherwise return false.

Example:

```
tarantool> utf8.islower('Ж'),utf8.islower('å'),utf8.islower('9')
---
- false
- true
- false
...
```

utf8.isupper(UTF8-character)

Parameters

- string-or-number (UTF8-character) – a single UTF8 character, expressed as a one-byte string or a code point value

Return true or false

Rtype boolean

Return true if the input character is upper case, otherwise return false.

Example:

```
tarantool> utf8.isupper('Ж'),utf8.isupper('å'),utf8.isupper('9')
---
- true
- false
- false
...
```

utf8.len(UTF8-string[, start-byte[, end-byte]])

Parameters

- string (UTF8-string) – a string encoded with UTF-8
- integer (end-byte) – byte position of the first character
- integer – byte position where to stop

Return the number of characters in the string, or between start and end

Rtype number

Byte positions for start and end can be negative, which indicates "calculate from end of string" rather than "calculate from start of string".

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

UTF-8 is a variable-size encoding scheme. Typically a simple Latin letter takes one byte, a Cyrillic letter takes two bytes, a Chinese/Japanese character takes three bytes, and the maximum is four bytes.

Example:

```
tarantool> utf8.len('G'),utf8.len('ж')
---
- 1
- 1
...

tarantool> string.len('G'),string.len('ж')
---
- 1
- 2
...
```

utf8.lower(UTF8-string)

Parameters

- string (UTF8-string) – a string encoded with UTF-8

Return the same string, lower case

Rtype string

Example:

```
tarantool> utf8.lower('ÅΓЖABCDEFG')
---
- åγжabcdefg
...
```

utf8.next(UTF8-string[, start-byte])

Parameters

- string (UTF8-string) – a string encoded with UTF-8
- integer (start-byte) – byte position where to start within the string, default is 1

Return byte position of the next character and the code point value of the next character

Rtype table

The next function is often used in a loop to get one character at a time from a UTF-8 string.

Example:

In the string 'åa' the first character is 'å', it starts at position 1, it takes two bytes to store so the character after it will be at position 3, its Unicode code point value is (decimal) 229.

```
tarantool> -- show next-character position + first-character codepoint
tarantool> utf8.next('åa', 1)
---
- 3
```

(continues on next page)

(continued from previous page)

```
- 229
...
tarantool> -- (loop) show codepoint of every character
tarantool> for position,codepoint in utf8.next,'åa' do print(codepoint) end
229
97
...
```

utf8.sub(UTF8-string, start-character[, end-character])

Parameters

- string (UTF8-string) – a string encoded as UTF-8
- number (end-character) – the position of the first character
- number – the position of the last character

Return a UTF-8 string, the "substring" of the input value

Rtype string

Character positions for start and end can be negative, which indicates "calculate from end of string" rather than "calculate from start of string".

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

Example:

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

utf8.upper(UTF8-string)

Parameters

- string (UTF8-string) – a string encoded with UTF-8

Return the same string, upper case

Rtype string

Note: In rare cases the upper-case result may be longer than the lower-case input, for example utf8.upper('ß') is 'SS'.

Example:

```
tarantool> utf8.upper('åγжabcdefg')
---
- ÅΓЖABCDEFG
...
```

4.2.31 Module uri

Overview

A "URI" is a "Uniform Resource Identifier". The IETF standard says a URI string looks like this:

```
[scheme:]scheme-specific-part[#fragment]
```

A common type, a hierarchical URI, looks like this:

```
[scheme:][//authority][path][?query][#fragment]
```

For example the string `'https://tarantool.org/x.html#y'` has three components:

- https is the scheme,
- tarantool.org/x.html is the path,
- y is the fragment.

Tarantool's URI module provides routines which convert URI strings into their components, or turn components into URI strings.

Index

Below is a list of all uri functions.

Name	Use
uri.parse()	Get a table of URI components
uri.format()	Construct a URI from components

uri.parse(URI-string)

Parameters

- URI-string – a Uniform Resource Identifier

Return URI-components-table. Possible components are fragment, host, login, password, path, query, scheme, service.

Rtype Table

Example:

```
tarantool> uri = require('uri')
---
...

tarantool> uri.parse('http://x.html#y')
---
- host: x.html
  scheme: http
  fragment: y
...
```

uri.format(URI-components-table[, include-password])

Parameters

- URI-components-table – a series of name:value pairs, one for each component

- include-password – boolean. If this is supplied and is true, then the password component is rendered in clear text, otherwise it is omitted.

Return URI-string. Thus uri.format() is the reverse of uri.parse().

Rtype string

Example:

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

4.2.32 Module xlog

The xlog module contains one function: pairs(). It can be used to read Tarantool's snapshot files or write-ahead-log (WAL) files. A description of the file format is in section Data persistence and the WAL file format.

xlog.pairs([file-name])

Open a file, and allow iterating over one file entry at a time.

Returns iterator which can be used in a for/end loop.

Rtype iterator

Possible errors: File does not contain properly formatted snapshot or write-ahead-log information.

Example:

This will read the first write-ahead-log (WAL) file that was created in the wal_dir directory in our "Getting started" exercises.

Each result from pairs() is formatted with MsgPack so its structure can be specified with __serialize.

```
xlog = require('xlog')
t = {}
for k, v in xlog.pairs('00000000000000000000.xlog') do
  table.insert(t, setmetatable(v, { __serialize = "map"}))
end
return t
```

The first lines of the result will look like:

```
(...)
---
- - {'BODY':   {'space_id': 272, 'index_base': 1, 'key': ['max_id'],
              'tuple': [['+', 2, 1]]},
    'HEADER': {'type': 'UPDATE', 'timestamp': 1477846870.8541,
              'lsn': 1, 'server_id': 1}}
  - {'BODY':   {'space_id': 280,
               'tuple': [512, 1, 'tester', 'memtx', 0, {}, []]},
    'HEADER': {'type': 'INSERT', 'timestamp': 1477846870.8597,
              'lsn': 2, 'server_id': 1}}
```

4.2.33 Module yaml

Overview

The yaml module takes strings in YAML format and decodes them, or takes a series of non-YAML values and encodes them.

Index

Below is a list of all yaml functions and members.

Name	Use
yaml.encode()	Convert a Lua object to a YAML string
yaml.decode()	Convert a YAML string to a Lua object
yaml.NULL	Analog of Lua's "nil"

yaml.encode(lua_value)
: Convert a Lua object to a YAML string.

  Parameters

  - 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.

  Parameters

  - 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.

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
...
```

(continues on next page)

(continued from previous page)

```
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.

4.2.34 Other package components

All the Tarantool modules are, at some level, inside a package which, appropriately, is named package. There are also miscellaneous functions and variables which are outside all modules.

Name	Use
tonumber64()	Convert a string or a Lua number to a 64-bit integer
dostring()	Parse and execute an arbitrary chunk of Lua code
package.path	Where Tarantool looks for Lua additions
package.cpath	Where Tarantool looks for C additions
package.loaded	What Tarantool has already looked for and found
package.setsearchroot	Set the root path for a directory search
package.searchroot	Get the root path for a directory search

tonumber64(value)

Convert a string or a Lua number to a 64-bit integer. The input value can be expressed in decimal, binary (for example 0b1010), or hexadecimal (for example -0xffff). 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.

Parameters

- 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>'''
...
```

(continues on next page)

(continued from previous page)

```
tarantool> dostring('return 1')
---
- 1
...
tarantool> dostring('return ...', 'hello', 'world')
---
- hello
- world
...
tarantool> dostring([[
         >   local f = function(key)
         >     local t = box.space.tester:select{key}
         >     if t ~= nil then
         >       return t[1]
         >     else
         >       return nil
         >     end
         >   end
         >   return f(...)]], 1)
---
- null
...
```

package.path
: This is a string that Tarantool uses to search for Lua modules, especially imporant for require(). See Modules, rocks and applications.

package.cpath
: This is a string that Tarantool uses to search for C modules, especially imporant for require(). See Modules, rocks and applications.

package.loaded
: This is a string that shows what Lua or C modules Tarantool has loaded, so that their functions and members are available. Initially it has all the pre-loaded modules, which don't need require().

package.setsearchroot([search-root])
: Set the search root. The search root is the root directory from which dependencies are loaded.

  Parameters

  - search-root (string) – the path. Default = current directory.

  The search-root string must contain a relative or absolute path. If it is a relative path, then it will be expanded to an absolute path. If search-root is omitted, or is box.NULL, then the search root is reset to the current directory, which is found with debug.sourcedir().

  Example:

  Suppose that a Lua file myapp/init.lua is the project root. Suppose the current path is /home/tara. Add this as the first line of myapp/init.lua: package.setsearchroot() Start the project with $ tarantool myapp/init.lua The search root will be the default, made absolute: /home/tara/myapp. Within the Lua application all dependencies will be searched relative to /home/tara/myapp.

package.searchroot()
: Return a string with the current search root. After package.setsearchroot('/home') the returned string will be /home'.

4.2.35 Database error codes

In the current version of the binary protocol, error messages, which are normally more descriptive than error codes, are not present in server responses. 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 are 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	(In replication) A server instance cannot modify data unless it is a master.
ER_ILLEGAL_PARAMS	Illegal parameters. Malformed protocol message.
ER_MEMORY_ISSUE	Out of memory: memtx_memory 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	The 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	An error occurred during update of a field.
ER_TUPLE_FOUND	A duplicate key exists in a unique index.

4.2.36 Handling errors

Here are some procedures that can make Lua functions more robust when there are errors, particularly database errors.

1. Invoke with pcall.

   Take advantage of Lua's mechanisms for "Error handling and exceptions", particularly pcall. That is, instead of simply invoking with
   box.space.space-name:function-name()
   say
   if pcall(box.space.space-name.function-name, box.space.space-name) ...

   For some Tarantool box functions, pcall also returns error details including a file-name and line-number within Tarantool's source code. This can be seen by unpacking. For example:
   x, y = pcall(function() box.schema.space.create('') end)
   y:unpack()

   See the tutorial Sum a JSON field for all tuples to see how pcall can fit in an application.

2. Examine and raise with box.error.

   To make a new error and pass it on, the box.error module provides box.error(code, errtext [, errtext ...]).

   To find the last error, the box.error module provides box.error.last(). (There is also a way to find the text of the last operating-system error for certain functions – errno.strerror([code]).)

3. Log.

   Put messages in a log using the log module.

   And filter messages that are automatically generated, with the log configuration parameter.

Generally, for Tarantool built-in functions which are designed to return objects: the result will be an object, or nil, or a Lua error. For example consider the fio_read.lua program in our cookbook:

```
#!/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)
```

After a function call that might fail, like fio.open() above, it is common to see syntax like if not f then ... or if f == nil then ..., which check for common failures. But if there had been a syntax error, for example fio.opex instead of fio.open, then there would have been a Lua error and f would not have been changed. If checking for such an obvious error had been a concern, the programmer would probably have used pcall().

All functions in Tarantool modules should work this way, unless the manual explicitly says otherwise.

4.2.37 Debug facilities

Overview

Tarantool users can benefit from built-in debug facilities that are part of:

- Lua (debug library, see details below) and
- LuaJit (debug.* functions).

The debug library provides an interface for debugging Lua programs. All functions in this library reside in the debug table. Those functions that operate on a thread have an optional first parameter that specifies the thread to operate on. The default is always the current thread.

Note: This library should be used only for debugging and profiling and not as a regular programming tool, as the functions provided here can take too long to run. Besides, several of these functions can compromise otherwise secure code.

Index

Below is a list of all debug functions.

Name	Use
debug.debug()	Enter an interactive mode
debug.getfenv()	Get an object's environment
debug.gethook()	Get a thread's current hook settings
debug.getinfo()	Get information about a function
debug.getlocal()	Get a local variable's name and value
debug.getmetatable()	Get an object's metatable
debug.getregistry()	Get the registry table
debug.getupvalue()	Get an upvalue's name and value
debug.setfenv()	Set an object's environment
debug.sethook()	Set a given function as a hook
debug.setlocal()	Assign a value to a local variable
debug.setmetatable()	Set an object's metatable
debug.setupvalue()	Assign a value to an upvalue
debug.sourcedir()	Get the source directory name
debug.sourcefile()	Get the source file name
debug.traceback()	Get a traceback of the call stack

debug.debug()

Enters an interactive mode and runs each string that the user types in. The user can, among other things, inspect global and local variables, change their values and evaluate expressions.

Enter cont to exit this function, so that the caller can continue its execution.

Note: Commands for debug.debug() are not lexically nested within any function and so have no direct access to local variables.

---

debug.getfenv(object)

Parameters

- object – object to get the environment of

Return the environment of the object

debug.gethook([thread])

Return the current hook settings of the thread as three values:

- the current hook function
- the current hook mask
- the current hook count as set by the debug.sethook() function

debug.getinfo([thread], function[, what])

Parameters

- function – function to get information on
- what (string) – what information on the function to return

Return a table with information about the function

You can pass in a function directly, or you can give a number that specifies a function running at level function of the call stack of the given thread: level 0 is the current function (getinfo() itself), level 1 is the function that called getinfo(), and so on. If function is a number larger than the number of active functions, getinfo() returns nil.

The default for what is to get all information available, except the table of valid lines. If present, the option f adds a field named func with the function itself. If present, the option L adds a field named activelines with the table of valid lines.

debug.getlocal([thread], level, local)

Parameters

- level (number) – level of the stack
- local (number) – index of the local variable

Return the name and the value of the local variable with the index local of the function at level level of the stack or nil if there is no local variable with the given index; raises an error if level is out of range

---

Note: You can call debug.getinfo() to check whether the level is valid.

---

debug.getmetatable(object)

Parameters

- object – object to get the metatable of

Return a metatable of the object or nil if it does not have a metatable

debug.getregistry()

Return the registry table

debug.getupvalue(func, up)

Parameters

- func (function) – function to get the upvalue of
- up (number) – index of the function upvalue

Return the name and the value of the upvalue with the index up of the function func or nil if there is no upvalue with the given index

debug.setfenv(object, table)

Sets the environment of the object to the table.

Parameters

- object – object to change the environment of
- table (table) – table to set the object environment to

Return the object

debug.sethook([thread], hook, mask[, count])

Sets the given function as a hook. When called without arguments, turns the hook off.

Parameters

- hook (function) – function to set as a hook
- mask (string) – describes when the hook will be called; may have the following values:
  - c - the hook is called every time Lua calls a function
  - r - the hook is called every time Lua returns from a function
  - l - the hook is called every time Lua enters a new line of code

- count (number) – describes when the hook will be called; when different from zero, the hook is called after every count instructions.

debug.setlocal([thread], level, local, value)

Assigns the value value to the local variable with the index local of the function at level level of the stack.

Parameters

- level (number) – level of the stack
- local (number) – index of the local variable
- value – value to assign to the local variable

Return the name of the local variable or nil if there is no local variable with the given index; raises an error if level is out of range

---

Note: You can call debug.getinfo() to check whether the level is valid.

---

debug.setmetatable(object, table)

Sets the metatable of the object to the table.

Parameters

- object – object to change the metatable of
- table (table) – table to set the object metatable to

debug.setupvalue(func, up, value)

Assigns the value value to the upvalue with the index up of the function func.

Parameters

- func (function) – function to set the upvalue of
- up (number) – index of the function upvalue
- value – value to assign to the function upvalue

Return the name of the upvalue or nil if there is no upvalue with the given index

debug.sourcedir([level])

Parameters

- level (number) – the level of the call stack which should contain the path (default is 2)

Return a string with the relative path to the source file directory

Instead of debug.sourcedir() one can say debug.__dir__ which means the same thing.

Determining the real path to a directory is only possible if the function was defined in a Lua file (this restriction may not apply for loadstring() since Lua will store the entire string in debug info).

If debug.sourcedir() is part of a return argument, then it should be inside parentheses: return (debug.sourcedir()).

debug.sourcefile([level])

Parameters

- level (number) – the level of the call stack which should contain the path (default is 2)

Return a string with the relative path to the source file

Instead of debug.sourcefile() one can say debug.__file__ which means the same thing.

Determining the real path to a file is only possible if the function was defined in a Lua file (this restriction may not apply to loadstring() since Lua will store the entire string in debug info).

If debug.sourcefile() is part of a return argument, then it should be inside parentheses: return (debug.sourcefile()).

debug.traceback([thread], [message][, level])

Parameters

- message (string) – an optional message prepended to the traceback
- level (number) – specifies at which level to start the traceback (default is 1)

Return a string with a traceback of the call stack

Debug example:

Make a file in the /tmp directory named example.lua, containing:

```
function w()
  print(debug.sourcedir())
  print(debug.sourcefile())
  print(debug.traceback())
  print(debug.getinfo(1)['currentline'])
end
w()
```

Execute tarantool /tmp/example.lua. Expect to see this:

```
/tmp
/tmp/example.lua
stack traceback:
    /tmp/example.lua:4: in function 'w'
    /tmp/example.lua:7: in main chunk
5
```

4.3 Rocks reference

This reference covers third-party Lua modules for Tarantool.

4.3.1 SQL DBMS Modules

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 SQL servers 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.

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 instance. After that, one can pass any MySQL statement to the server instance 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
```

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
---
```

(continues on next page)

(continued from previous page)

```
...
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 instance 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/2.1/ubuntu/ precise main
deb-src http://tarantool.org/dist/2.1/ubuntu/ precise main

$ # Install tarantool-dev. The displayed line should show version = 2.1
$ sudo apt-get -y install tarantool-dev | grep -E "Setting up|already"
```

(continues on next page)

(continued from previous page)

```
Setting up tarantool-dev (2.1.0.222.g48b98bb~precise-1) ...
$

$ # Use luarocks to install locally, that is, relative to $HOME
$ luarocks install mysql MYSQL_LIBDIR=/usr/local/mysql/lib --local
Installing http://rocks.tarantool.org/mysql-scm-1.rockspec...
... (more info about building the Tarantool/MySQL driver appears here)
mysql scm-1 is now built and installed in ~/.luarocks/

$ # Ensure driver.so now has been created in a place
$ # tarantool will look at
$ find ~/.luarocks -name "driver.so"
~/.luarocks/lib/lua/5.1/mysql/driver.so

$ # Change directory to a directory which can be used for
$ # temporary tests. For this example we assume that the name
$ # of this directory is /home/pgulutzan/tarantool_sandbox.
$ # (Change "/home/pgulutzan" to whatever is the user's actual
$ # home directory for the machine that's used for this test.)
$ cd /home/pgulutzan/tarantool_sandbox

$ # Start the Tarantool server instance. Do not use a Lua initialization file.

$ tarantool
tarantool: version 2.1.0-222-g48b98bb
type 'help' for interactive help
tarantool>
```

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 instance, (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
```

(continues on next page)

(continued from previous page)

```
---
...
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.

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 instance. After that, one can pass any PostgreSQL statement to the server instance 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
```

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 = 5432
- 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
---
...
```

(continues on next page)

(continued from previous page)

```
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 parameter markers ($1 $2 $3 etc.) 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 instance 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/2.0/ubuntu/ precise main
deb-src http://tarantool.org/dist/2.0/ubuntu/ precise main

$ # Install tarantool-dev. The displayed line should show version = 2.0
$ sudo apt-get -y install tarantool-dev | grep -E "Setting up|already"
Setting up tarantool-dev (2.0.4.222.g48b98bb~precise-1) ...
$

$ # Use luarocks to install locally, that is, relative to $HOME
$ luarocks install pg POSTGRESQL_LIBDIR=/usr/lib/x86_64-linux-gnu --local
Installing http://rocks.tarantool.org/pg-scm-1.rockspec...
... (more info about building the Tarantool/PostgreSQL driver appears here)
```

(continues on next page)

(continued from previous page)

```
pg scm-1 is now built and installed in ~/.luarocks/

$ # Ensure driver.so now has been created in a place
$ # tarantool will look at
$ find ~/.luarocks -name "driver.so"
~/.luarocks/lib/lua/5.1/pg/driver.so

$ # Change directory to a directory which can be used for
$ # temporary tests. For this example we assume that the
$ # name of this directory is $HOME/tarantool_sandbox.
$ # (Change "$HOME" to whatever is the user's actual
$ # home directory for the machine that's used for this test.)
cd $HOME/tarantool_sandbox

$ # Start the Tarantool server instance. Do not use a Lua initialization file.

$ tarantool
tarantool: version 2.0.4-412-g803b15c
type 'help' for interactive help
tarantool>
```

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 a 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.

4.3.2 Module expirationd

For a commercial-grade example of a Lua rock that works with Tarantool, let us look at the source code of 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
...
    expiration_process(task, tuple)
...
    /* expiration_process() contains:
    if task.is_tuple_expired(task.args, tuple) then
    task.expired_tuples_count = task.expired_tuples_count + 1
    task.process_expired_tuple(task.space_id, task.args, tuple) */
```

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)
    box.space[space_id]:delete(construct_key(space_id, tuple))
end
    /* construct_key() contains:
    local function construct_key(space_id, tuple)
        return fun.map(
            function(x) return tuple[x.fieldno] end,
           box.space[space_id].index[0].parts
        ):totable()
    end */
```

Ultimately the tuple-expiry process leads to default_tuple_drop() which does a "delete" of a tuple from its original space. First the fun 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 is 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 directory on the Lua path (type print(package.path) to see the Lua path).
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, but is on the Lua path (package.path) or the C path (package.cpath). 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. Particularly important are {options} which can be added as a final parameter in expirationd.run_task:

- force (boolean) – run task even on replica. Default: force=false so ordinarily expirationd ignores replicas.

- tuples_per_iteration (integer) – number of tuples that will be checked by one iteration Default: tuples_per_iteration=1024.
- full_scan_time (number) – number of seconds required for full index scan Default: full_scan_time=3600.
- vinyl_assumed_space_len (integer) – assumed size of vinyl space, for the first iteration only. Default: vinyl_assumed_space_len=10000000.
- vinyl_assumed_space_len_factor (integer) – factor for recalculation of size of vinyl space. Default: vinyl_assumed_space_len_factor=2. (The size of a vinyl space cannot be easily calculated, so on the first iteration it will be the "assumed" size, on the second iteration it will be "assumed" times "factor", on the third iteration it will be "assumed" times "factor" times factor", and so on.)

4.3.3 Module shard

With sharding, the tuples of a tuple set are distributed to multiple nodes, with a Tarantool database server instance on each node. With this arrangement, each instance 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.

Instance A currently-running in-memory copy of the Tarantool server, sometimes called a "server instance". Usually each shard is associated with one instance, or, if both sharding and replicating are going on, each shard is associated with one replica set.

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 replicated data copies in each shard.

Replica An instance which is part of a replica set.

Replica set Often a single shard is associated with a single instance; however, often the shard is replicated. When a shard is replicated, the multiple instances ("replicas"), which handle the shard's replicated data, are a "replica set".

Replicated data A complete copy of the data. The shard module handles both sharding and replication. One shard can contain one or more replicated data copies. When a write occurs, the write is attempted on every replicated data copy 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 replicated data copies should be in different zones.

The shard package is distributed separately from the main tarantool package. To acquire it, do a separate installation:

- with Tarantool 1.7.4+, say:

```
$ tarantoolctl rocks install shard
```

- install with yum or apt, for example on Ubuntu say:

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

- or download from GitHub tarantool/shard and use the Lua files 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 replicated data copies of the same shard should not be in the same zone.

Example: shard.init syntax for one shard

- The number of replicated data copies per shard (redundancy) is 3.
- The number of instances 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 = 'test_user',
         >   password = 'pass',
         >   redundancy = '3',
         >   binary = 33131,
         > }
---
...
tarantool> shard.init(cfg)
---
...
```

Example: shard.init syntax for three shards

This describes three shards. Each shard has two replicated data copies. Since the number of servers is 7, and the number of replicated data copies 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 = 'test_user',
         >   password = 'pass',
         >   redundancy = '2',
         >   binary = 33131,
         > }
---
...
tarantool> shard.init(cfg)
---
...
```

Every data-access function in the box module has an analogue in the shard module:

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{...}

For example, to insert in table T in a sharded database you simply say shard.T:insert{...} instead of box.space.T:insert{...}.

A shard.T:select{} request without a primary key will cause an error.

Every queued data-access function has an analogue in the shard module:

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{...}

The user must add an operation_id. For details of queued data-access functions, and of maintenance-related functions, see the README.

Example: shard, minimal configuration

There is only one shard, and that shard contains only one replicated data copy. 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.create('test_user', {password = 'pass'})
tarantool> box.schema.user.grant('test_user', 'read,write,execute', 'universe')
tarantool> cfg = {
         >   servers = {
         >     { uri = 'localhost:3301', zone = '1' },
         >   },
         >   login = 'test_user';
         >   password = 'pass';
         >   redundancy = 1;
         >   binary = 3301;
         > }
tarantool> shard = require('shard')
tarantool> shard.init(cfg)
tarantool> -- Now put something in ...
tarantool> shard.tester:insert{1,'Tuple #1'}
```

If you cut and paste 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)
2017-09-06 ... I> Sharding initialization started...
2017-09-06 ... I> establishing connection to cluster servers...
2017-09-06 ... I> connected to all servers
2017-09-06 ... I> started
2017-09-06 ... I> redundancy = 1
2017-09-06 ... I> Adding localhost:3301 to shard 1
2017-09-06 ... I> shards = 1
2017-09-06 ... I> Done
---
- true
...
tarantool> -- Now put something in ...
---
...
tarantool> shard.tester:insert{1,'Tuple #1'}
---
- - [1, 'Tuple #1']
...
```

Example: shard, scaling out

There are two shards, and each shard contains one replicated data copy. 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.create('test_user', {password = 'pass'})
tarantool> box.schema.user.grant('test_user', 'read,write,execute', 'universe')
tarantool> console = require('console')
tarantool> cfg = {
         >   servers = {
         >     { uri = 'localhost:3301', zone = '1' },
         >     { uri = 'localhost:3302', zone = '2' },
         >   },
         >   login = 'test_user',
         >   password = 'pass',
         >   redundancy = 1,
         >   binary = 3301,
         > }
tarantool> shard = require('shard')
tarantool> shard.init(cfg)
tarantool> -- 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.create('test_user', {password = 'pass'})
tarantool> box.schema.user.grant('test_user', 'read,write,execute', 'universe')
tarantool> console = require('console')
tarantool> cfg = {
         >   servers = {
         >     { uri = 'localhost:3301', zone = '1' };
         >     { uri = 'localhost:3302', zone = '2' };
         >   };
         >   login = 'test_user';
         >   password = 'pass';
         >   redundancy = 1;
         >   binary = 3302;
         > }
tarantool> shard = require('shard')
tarantool> shard.init(cfg)
tarantool> -- 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.

For details, see the README.

4.3.4 Module vshard

Summary

The vshard module introduces the sharding feature, which enables horizontal scaling in Tarantool.

While a project is growing, scaling the databases may become the most challenging issue. Once a single server cannot withstand the load, scaling methods should be applied.

There are two different approaches for scaling data, vertical and horizontal scaling:

- Vertical scaling implies that the hardware capacities of a single server would be increased.
- Horizontal scaling implies that a dataset is partitioned and distributed over multiple servers. In case new servers are added, the dataset is re-distributed evenly across all servers, both the original and new ones.

Sharding is a database architecture that allows for horizontal scaling.

With vshard, the tuples of a dataset are distributed across multiple nodes, with a Tarantool database server instance on each node. Each instance handles only a subset of the total data, so larger loads can be handled by simply adding more servers. The initial dataset is partitioned into multiple parts, so each part is stored on a separate server. The dataset is partitioned using sharding keys.

The vshard module is based on the concept of virtual buckets, where a tuple set is partitioned into a large number of abstract virtual nodes (virtual buckets, or buckets) rather than into a smaller number of physical nodes.

Hashing a sharding key into a large number of virtual buckets allows seamlessly changing the number of servers in the cluster. The rebalancing mechanism distributes buckets evenly among all shards in case some servers were added or removed.

The buckets have states, so it is easy to monitor the server states. For example, a server instance is active and available for all types of requests, or a failover occurred and the instance accepts only read requests.

The vshard module provides analogs for the data-manipulation functions of the Tarantool box library (select, insert, replace, update, delete).

Installation

The vshard module is distributed separately from the main Tarantool package. To acquire it, do a separate installation:

```
$ tarantoolctl rocks install vshard
```

Note: The vshard module requires Tarantool version 1.9+.

Quick start

The vshard/example/ directory includes a pre-configured development cluster of 1 router and 2 replica sets of 2 nodes (2 storages) each, making 5 Tarantool instances in total:

- router_1 – a router instance
- storage_1_a – a storage instance, the master of the first replica set
- storage_1_b – a storage instance, the replica of the first replica set
- storage_2_a – a storage instance, the master of the second replica set
- storage_2_b – a storage instance, the replica of the second replica set

All instances are managed using the tarantoolctl utility which comes with Tarantool.

Change the directory to example/ and use make to run the development cluster:

```
$ cd example/
$ make
tarantoolctl stop storage_1_a  # stop the first storage instance
Stopping instance storage_1_a...
tarantoolctl stop storage_1_b
<...>
rm -rf data/
tarantoolctl start storage_1_a # start the first storage instance
Starting instance storage_1_a...
Starting configuration of replica 8a274925-a26d-47fc-9e1b-af88ce939412
I am master
Taking on replicaset master role...
Run console at unix/:./data/storage_1_a.control
started
mkdir ./data/storage_1_a
<...>
tarantoolctl start router_1 # start the router
Starting instance router_1...
Starting router configuration
Calling box.cfg()...
<...>
Run console at unix/:./data/router_1.control
started
mkdir ./data/router_1
Waiting cluster to start
echo "vshard.router.bootstrap()" | tarantoolctl enter router_1
connected to unix/:./data/router_1.control
unix/:./data/router_1.control> vshard.router.bootstrap()
---
- true
...
unix/:./data/router_1.control>
tarantoolctl enter router_1 # enter the admin console
connected to unix/:./data/router_1.control
unix/:./data/router_1.control>
```

Some tarantoolctl commands:

- tarantoolctl start router_1 – start the router instance
- tarantoolctl enter router_1 – enter the admin console

The full list of tarantoolctl commands for managing Tarantool instances is available in the tarantoolctl reference.

Essential make commands you need to know:

- make start – start all Tarantool instances
- make stop – stop all Tarantool instances
- make logcat – show logs from all instances
- make enter – enter the admin console on router_1
- make clean – clean up all persistent data
- make test – run the test suite (you can also run test-run.py in the test directory)
- make – execute make stop, make clean, make start and make enter

For example, to start all instances, use make start:

```
$ make start
$ ps x|grep tarantool
46564   ??  Ss     0:00.34 tarantool storage_1_a.lua <running>
46566   ??  Ss     0:00.19 tarantool storage_1_b.lua <running>
46568   ??  Ss     0:00.35 tarantool storage_2_a.lua <running>
46570   ??  Ss     0:00.20 tarantool storage_2_b.lua <running>
46572   ??  Ss     0:00.25 tarantool router_1.lua <running>
```

To perform commands in the admin console, use the router API:

```
unix/:./data/router_1.control> vshard.router.info()
---
- replicasets:
    ac522f65-aa94-4134-9f64-51ee384f1a54:
      replica: &0
        network_timeout: 0.5
        status: available
        uri: storage@127.0.0.1:3303
        uuid: 1e02ae8a-afc0-4e91-ba34-843a356b8ed7
      uuid: ac522f65-aa94-4134-9f64-51ee384f1a54
      master: *0
    cbf06940-0790-498b-948d-042b62cf3d29:
      replica: &1
        network_timeout: 0.5
        status: available
        uri: storage@127.0.0.1:3301
        uuid: 8a274925-a26d-47fc-9e1b-af88ce939412
      uuid: cbf06940-0790-498b-948d-042b62cf3d29
      master: *1
  bucket:
    unreachable: 0
    available_ro: 0
    unknown: 0
    available_rw: 3000
  status: 0
  alerts: []
...
```

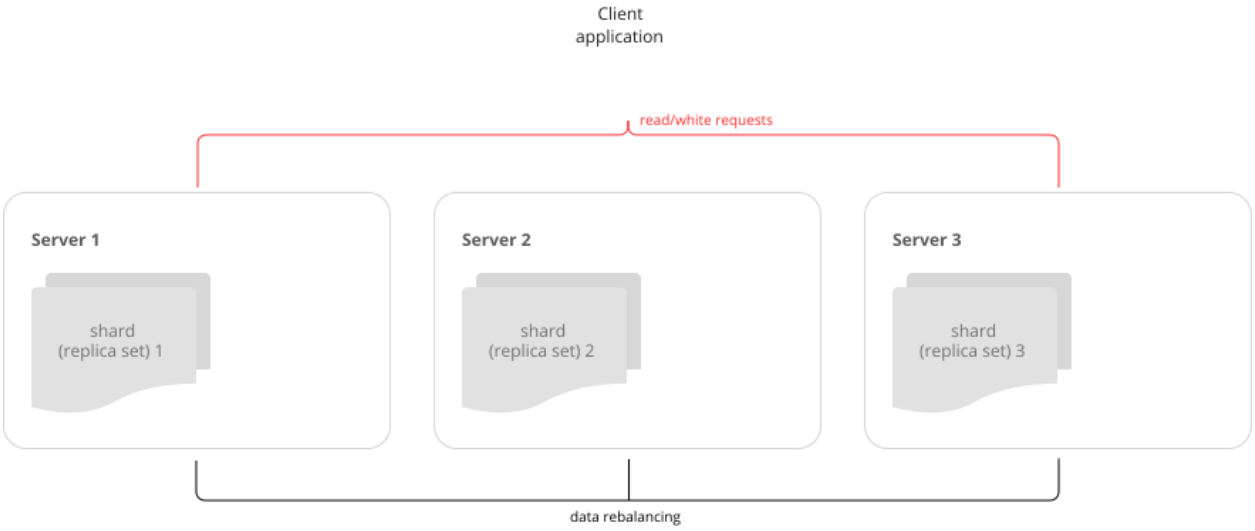
Architecture

A sharded cluster in Tarantool consists of storages, routers, and a rebalancer.

A storage is a node storing a subset of a dataset. Multiple replicated storages are deployed as replica sets to provide redundancy (a replica set can also be called a shard).

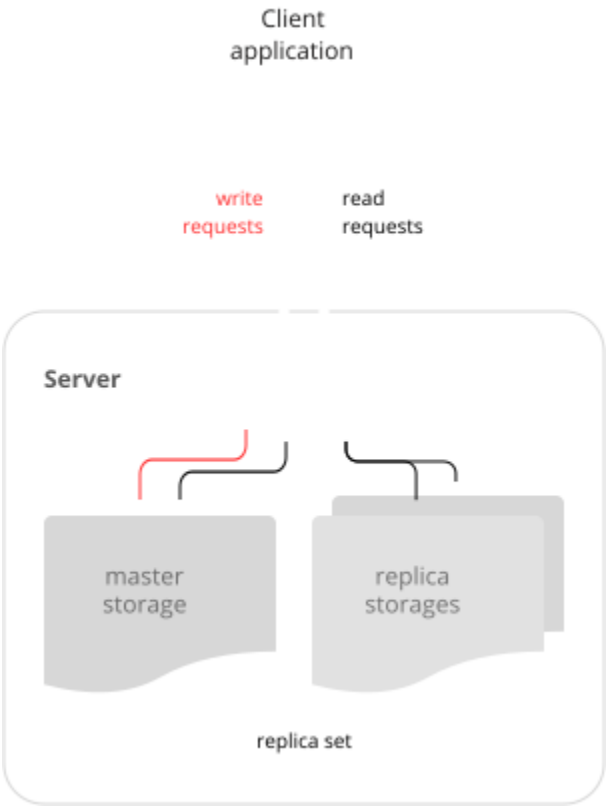
A router is a standalone software component that routes read and write requests from the client application to shards.

A rebalancer is an internal component that distributes the dataset among all shards evenly in case some servers are added or removed. It also balances the load considering the capacities of existing replica sets.



Storage

Storage is a node storing a subset of a dataset. Multiple replicated storages comprise a replica set. Each storage in a replica set has a role, master or replica. A master processes read and write requests. A replica processes read requests but cannot process write requests.



Virtual buckets

The sharded dataset is partitioned into a large number of abstract nodes called virtual buckets (further referred to as buckets).

The dataset is partitioned using the sharding key (or bucket id, in Tarantool terminology). Bucket id is a number from 1 to N, where N is the total number of buckets.

Each replica set stores a unique subset of buckets. One bucket cannot belong to multiple replica sets at a time.

The total number of buckets is determined by the administrator who sets up the initial cluster configuration.

Every Tarantool space you plan to shard must have a bucket id field indexed by the bucket id index. Spaces without the bucket id indexes don't participate in sharding but can be used as regular spaces. By default, the name of the index coincides with the bucket id.

Migration of buckets

A rebalancer is a background rebalancing process that ensures an even distribution of buckets across the shards. During rebalancing, buckets are being migrated among replica sets.

A replica set from which the bucket is being migrated is called a source; a target replica set to which the bucket is being migrated is called a destination.

A replica set lock makes a replica set invisible to the rebalancer. A locked replica set can neither receive new buckets nor migrate its own buckets.

While a bucket is being migrated, it can have different states:

- ACTIVE – the bucket is available for read and write requests.
- PINNED – the bucket is locked for migrating to another replica set. Otherwise pinned buckets are similar to buckets in the ACTIVE state.
- SENDING – the bucket is currently being copied to the destination replica set; read requests to the source replica set are still processed.
- RECEIVING – the bucket is currently being filled; all requests to it are rejected.
- SENT – the bucket was migrated to the destination replica set. The router uses the SENT state to calculate the new location of the bucket. A bucket in the SENT state goes to the GARBAGE state automatically after BUCKET_SENT_GARBAGE_DELAY seconds, which by default is 0.5 seconds.
- GARBAGE – the bucket was already migrated to the destination replica set during rebalancing; or the bucket was initially in the RECEIVING state, but some error occurred during the migration.

Buckets in the GARBAGE state are deleted by the garbage collector.

Migration is performed as follows:

1. At the destination replica set, a new bucket is created and assigned the RECEIVING state, the data copying starts, and the bucket rejects all requests.
2. The source bucket in the source replica set is assigned the SENDING state, and the bucket continues to process read requests.
3. Once the data is copied, the bucket on the source replica set is assigned the SENT and it starts rejecting all requests.
4. The bucket on the destination replica set is assigned the ACTIVE state and starts accepting all requests.

The _bucket system space

The _bucket system space of each replica set stores the ids of buckets present in the replica set. The space contains the following fields:

- bucket – bucket id
- status – state of the bucket
- destination – UUID of the destination replica set

An example of _bucket.select{}:

```
---
- - [1, ACTIVE, abfe2ef6-9d11-4756-b668-7f5bc5108e2a]
  - [2, SENT, 19f83dcb-9a01-45bc-a0cf-b0c5060ff82c]
...
```

Once the bucket is migrated, the destination replica set identified by UUID is filled in the table. While the bucket is still located on the source replica set, the value of the destination replica set UUID is equal to NULL.

Router

All requests from the application come to the sharded cluster through a router. The router keeps the topology of a sharded cluster transparent for the application, thus keeping the application unaware of:

- the number and location of shards,
- data rebalancing process,
- the fact and the process of a failover that occurred after a replica's failure.

The router does not have a persistent state, nor does it store the cluster topology or balance the data. The router is a standalone software component that can run in the storage layer or application layer depending on the application features.

The routing table

A routing table on the router stores the map of all bucket ids to replica sets. It ensures the consistency of sharding in case of failover.

The router keeps a persistent pool of connections to all the storages that are created at startup. This helps prevent configuration errors. Once the connection pool is created, the router caches the current state of the routing table in order to speed up routing. If a bucket migrated to another storage after rebalancing, or a failover occurred and caused one of the shards switching to another replica, the discovery fiber on the router updates the routing table automatically.

As the bucket id is explicitly indicated both in the data and in the mapping table on the router, the data is consistent regardless of the application logic. It also makes rebalancing transparent for the application.

Processing requests

Requests to the database can be performed by the application or using stored procedures. Either way, the bucket id should be explicitly specified in the request.

All requests are forwarded to the router first. The only operation supported by the router is call. The operation is performed via the vshard.router.call() function:

```
result = vshard.router.call(<bucket_id>, <mode>, <function_name>, {<argument_list>}, {<opts>})
```

Requests are processed as follows:

1. The router uses the bucket id to search for a replica set with the corresponding bucket in the routing table.

   If the map of the bucket id to the replica set is not known to the router (the discovery fiber hasn't filled the table yet), the router makes requests to all storages to find out where the bucket is located.

2. Once the bucket is located, the shard checks:

   - whether the bucket is stored in the _bucket system space of the replica set;
   - whether the bucket is ACTIVE or PINNED (for a read request, it can also be SENDING).

3. If all the checks succeed, the request is executed. Otherwise, it is terminated with the error: "wrong bucket".

Administration

Configuring a sharded cluster

A minimal viable sharded cluster should consist of:

- one or more replica sets, each containing two or more storage instances
- one or more router instances

The number of storage instances in a replica set defines the redundancy factor of the data. The recommended value is 3 or more. The number of router instances is not limited, because routers are completely stateless. We recommend increasing the number of routers when an existing router instance becomes CPU or I/O bound.

vshard supports multiple router instances on a single Tarantool instance. Each router can be connected to any vshard cluster. Multiple router instances can be connected to the same cluster.

As the router and storage applications perform completely different sets of functions, they should be deployed to different Tarantool instances. Although it is technically possible to place the router application on every storage node, this approach is highly discouraged and should be avoided on production deployments.

All storage instances can be deployed using identical instance (configuration) files.

Self-identification is currently performed using tarantoolctl:

```
$ tarantoolctl instance_name
```

All router instances can also be deployed using identical instance (configuration) files.

All cluster nodes must share a common topology. An administrator must ensure that the configurations are identical. We suggest using a configuration management tool like Ansible or Puppet to deploy the cluster.

Sharding is not integrated into any system for centralized configuration management. It is expected that the application itself is responsible for interacting with such a system and passing the sharding parameters.

Sample configuration

The configuration of a simple sharded cluster can look like this:

```
local cfg = {
    memtx_memory = 100 * 1024 * 1024,
    replication_connect_quorum = 0,
    bucket_count = 10000,
    rebalancer_disbalance_threshold = 10,
    rebalancer_max_receiving = 100,
    sharding = {
        ['cbf06940-0790-498b-948d-042b62cf3d29'] = {
            replicas = {
                ['8a274925-a26d-47fc-9e1b-af88ce939412'] = {
                    uri = 'storage:storage@127.0.0.1:3301',
                    name = 'storage_1_a',
                    master = true
                },
                ['3de2e3e1-9ebe-4d0d-abb1-26d301b84633'] = {
                    uri = 'storage:storage@127.0.0.1:3302',
                    name = 'storage_1_b'
                }
            },
        },
        ['ac522f65-aa94-4134-9f64-51ee384f1a54'] = {
            replicas = {
                ['1e02ae8a-afc0-4e91-ba34-843a356b8ed7'] = {
                    uri = 'storage:storage@127.0.0.1:3303',
                    name = 'storage_2_a',
                    master = true
                },
                ['001688c3-66f8-4a31-8e19-036c17d489c2'] = {
                    uri = 'storage:storage@127.0.0.1:3304',
                    name = 'storage_2_b'
                }
            },
        },
    },
}
```

This cluster includes one router instance and two storage instances. Each storage instance includes one master and one replica.

The sharding field defines the logical topology of a sharded Tarantool cluster. All the other fields are passed to box.cfg() as they are, without modifications. See the Configuration reference section for details.

On routers call vshard.router.cfg(cfg):

```
cfg.listen = 3300

-- Start the database with sharding
vshard = require('vshard')
vshard.router.cfg(cfg)
```

On storages call vshard.storage.cfg(cfg, instance_uuid):

```
-- Get instance name
local MY_UUID = "de0ea826-e71d-4a82-bbf3-b04a6413e417"
```

(continues on next page)

(continued from previous page)

```
-- Call a configuration provider
local cfg = require('localcfg')

-- Start the database with sharding
vshard = require('vshard')
vshard.storage.cfg(cfg, MY_UUID)
```

vshard.storage.cfg() automatically calls box.cfg() and configures the listen port and replication parameters.

See router.lua and storage.lua in the vshard/example directory for a sample configuration.

Replica weights

The router sends all requests to the master instance only. Setting replica weights allows sending read requests not only to the master instance, but to any available replica that is the 'nearest' to the router. Weights are used to define distances between replicas within a replica set.

Weights can be used, for example, to define the physical distance between the router and each replica in each replica set. In such a case read requests are sent to the nearest replica.

Setting weights can also help to define the most powerful replicas: the ones that can process the largest number of requests per second.

The idea is to specify the zone for every router and every replica, therefore filling a matrix of relative zone weights. This approach allows setting different weights in different zones for the same replica set.

To set weights, use the zone attribute for each replica during configuration:

```
local cfg = {
  sharding = {
    ['...replicaset_uuid...'] = {
      replicas = {
        ['...replica_uuid...'] = {
            ...,
            zone = <number or string>
        }
      }
    }
  }
}
```

Then, specify relative weights for each zone pair in the weights parameter of vshard.router.cfg. For example:

```
weights = {
    [1] = {
        [2] = 1, -- routers of the 1st zone see the weight of the 2nd zone as 1
        [3] = 2, -- routers of the 1st zone see the weight of the 3rd zone as 2


   [4] = 3, -- ...
    },
    [2] = {
        [1] = 10,
        [2] = 0,
        [3] = 10,
```

(continues on next page)

(continued from previous page)

```
        [4] = 20,
    },
    [3] = {
        [1] = 100,
        [2] = 200, -- routers of the 3rd zone see the weight of the 2nd zone as 200. Mind that it is not equal to the
↪weight of the 2nd zone = 2 visible from the 1st zone
        [4] = 1000,
    }
}

local cfg = vshard.router.cfg({weights = weights, sharding = ...})
```

Replica set weights

A replica set weight is not the same as the replica weight. The weight of a replica set defines the capacity of the replica set: the larger the weight, the more buckets the replica set can store. The total size of all sharded spaces in the replica set is also its capacity metric.

You can consider replica set weights as the relative amount of data within a replica set. For example, if replicaset_1 = 100, and replicaset_2 = 200, the second replica set stores twice as many buckets as the first one. By default, all weights of all replica sets are equal.

You can use weights, for example, to store the prevailing amount of data on a replica set with more memory space.

Rebalancing process

There is an etalon number of buckets for a replica set. (Etalon in this context means "ideal".) If there is no deviation from this number in the whole replica set, then the buckets are distributed evenly.

The etalon number is calculated automatically considering the number of buckets in the cluster and weights of the replica sets.

For example: The user specified the number of buckets is 3000, and weights of 3 replica sets are 1, 0.5, and 1.5. The resulting etalon numbers of buckets for the replica sets are: 1st replica set – 1000, 2nd replica set – 500, 3rd replica set – 1500.

This approach allows assigning a zero weight to a replica set, which initiates migration of its buckets to the remaining cluster nodes. It also allows adding a new zero-load replica set, which initiates migration of the buckets from the loaded replica sets to the zero-load replica set.

Note: A new zero-load replica set should be assigned a weight for rebalancing to start.

The rebalancer wakes up periodically and redistributes data from the most loaded nodes to less loaded nodes. Rebalancing starts if the disbalance threshold of a replica set exceeds a disbalance threshold specified in the configuration.

The disbalance threshold is calculated as follows:

```
|etalon_bucket_number - real_bucket_number| / etalon_bucket_number * 100
```

When a new shard is added, the configuration can be updated dynamically:

1. The configuration should be updated on all the routers first, and then on all the storages.

2. The new shard becomes available for rebalancing in the storage layer.
3. As a result of rebalancing, buckets are migrated to the new shard.
4. If a migrated bucket is requested, router receives an error code containing information about the new location of the bucket.

At this time, the new shard is already present in the router's pool of connections, so redirection is transparent for the application.

Replica set lock and bucket pin

A replica set lock makes a replica set invisible to the rebalancer: a locked replica set can neither receive new buckets nor migrate its own buckets.

A bucket pin blocks a specific bucket from migrating: a pinned bucket stays on the replica set to which it is pinned, until it is unpinned.

Pinning all replica set buckets is not equivalent to locking a replica set. Even if you pin all buckets, a non-locked replica set can still receive new buckets.

Replica set lock is helpful, for example, to separate a replica set from production replica sets for testing, or to preserve some application metadata that must not be sharded for a while. A bucket pin is used for similar cases but in a smaller scope.

By both locking a replica set and pinning all buckets, one can isolate an entire replica set.

Locked replica sets and pinned buckets affect the rebalancing algorithm as the rebalancer must ignore locked replica sets and consider pinned buckets when attempting to reach the best possible balance.

The issue is not trivial as a user can pin too many buckets to a replica set, so a perfect balance becomes unreachable. For example, consider the following cluster (assume all replica set weights are equal to 1).

The initial configuration:

```
rs1: bucket_count = 150
rs2: bucket_count = 150, pinned_count = 120
```

Adding a new replica set:

```
rs1: bucket_count = 150
rs2: bucket_count = 150, pinned_count = 120
rs3: bucket_count = 0
```

The perfect balance would be 100 - 100 - 100, which is impossible since the rs2 replica set has 120 pinned buckets. The best possible balance here is the following:

```
rs1: bucket_count = 90
rs2: bucket_count = 120, pinned_count 120
rs3: bucket_count = 90
```

The rebalancer moved as many buckets as possible from rs2 to decrease the disbalance. At the same time it respected equal weights of rs1 and rs3.

The algorithms for implementing locks and pins are completely different, although they look similar in terms of functionality.

Replica set lock and rebalancing

Locked replica sets simply do not participate in rebalancing. This means that even if the actual total number of buckets is not equal to the etalon number, the disbalance cannot be fixed due to the lock. When the rebalancer detects that one of the replica sets is locked, it recalculates the etalon number of buckets of the non-locked replica sets as if the locked replica set and its buckets did not exist at all.

Bucket pin and rebalancing

Rebalancing replica sets with pinned buckets requires a more complex algorithm. Here pinned_count[o] is the number of pinned buckets, and etalon_count is the etalon number of buckets for a replica set:

1. The rebalancer calculates the etalon number of buckets as if all buckets were not pinned. Then the rebalancer checks each replica set and compares the etalon number of buckets with the number of pinned buckets in a replica set. If pinned_count < etalon_count, non-locked replica sets (at this point all locked replica sets already are filtered out) with pinned buckets can receive new buckets.
2. If pinned_count > etalon_count, the disbalance cannot be fixed, as the rebalancer cannot move pinned buckets out of this replica set. In such a case the etalon number is updated and set equal to the number of pinned buckets. The replica sets with pinned_count > etalon_count are not processed by the rebalancer, and the number of pinned buckets is subtracted from the total number of buckets. The rebalancer tries to move out as many buckets as possible from such replica sets.
3. This procedure is restarted from step 1 for replica sets with pinned_count >= etalon_count until pinned_count <= etalon_count on all replica sets. The procedure is also restarted when the total number of buckets is changed.

Here is the pseudocode for the algorithm:

```
function cluster_calculate_perfect_balance(replicasets, bucket_count)
        -- rebalance the buckets using weights of the still viable replica sets --
end;

cluster = <all of the non-locked replica sets>;
bucket_count = <the total number of buckets in the cluster>;
can_reach_balance = false
while not can_reach_balance do
        can_reach_balance = true
        cluster_calculate_perfect_balance(cluster, bucket_count);
        foreach replicaset in cluster do
                if replicaset.perfect_bucket_count <
                  replicaset.pinned_bucket_count then
                        can_reach_balance = false
                        bucket_count -= replicaset.pinned_bucket_count;
                        replicaset.perfect_bucket_count =
                                replicaset.pinned_bucket_count;
                end;
        end;
end;
cluster_calculate_perfect_balance(cluster, bucket_count);
```

The complexity of the algorithm is O(N^2), where N is the number of replica sets. On each step, the algorithm either finishes the calculation, or ignores at least one new replica set overloaded with the pinned buckets, and updates the etalon number of buckets on other replica sets.

Bucket ref

Bucket ref is an in-memory counter that is similar to the bucket pin, but has the following differences:

1. Bucket ref is not persistent. Refs are intended for forbidding bucket transfer during request execution, but on restart all requests are dropped.
2. There are two types of bucket refs: read-only (RO) and read-write (RW).

   If a bucket has RW refs, it can not be moved. However, when the rebalancer needs it to be sent, it locks the bucket for new write requests, waits until all current requests are finished, and then sends the bucket.

   If a bucket has RO refs, it can be sent, but cannot be dropped. Such a bucket can even enter GARBAGE or SENT state, but its data is kept until the last reader is gone.

   A single bucket can have both RO and RW refs.
3. Bucket ref is countable.

The vshard.storage.bucket_ref/unref() methods are called automatically when vshard.router.call() or vshard.storage.call() is used. For raw API like r = vshard.router.route() r:callro/callrw you should explicitly call the bucket_ref() method inside the function. Also, make sure that you call bucket_unref() after bucket_ref(), otherwise the bucket cannot be moved from the storage until the instance restart.

To see how many refs there are for a bucket, use vshard.storage.buckets_info([bucket_id]) (the bucket_id parameter is optional).

For example:

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

Defining spaces

Spaces should be defined within a storage application using box.once(). For example:

```
box.once("testapp:schema:1", function()
    local customer = box.schema.space.create('customer')
    customer:format({
        {'customer_id', 'unsigned'},
        {'bucket_id', 'unsigned'},
        {'name', 'string'},
    })
    customer:create_index('customer_id', {parts = {'customer_id'}})
    customer:create_index('bucket_id', {parts = {'bucket_id'}, unique = false})

    local account = box.schema.space.create('account')
    account:format({
        {'account_id', 'unsigned'},
        {'customer_id', 'unsigned'},
```

(continues on next page)

(continued from previous page)

```
        {'bucket_id', 'unsigned'},
        {'balance', 'unsigned'},
        {'name', 'string'},
    })
    account:create_index('account_id', {parts = {'account_id'}})
    account:create_index('customer_id', {parts = {'customer_id'}, unique = false})
    account:create_index('bucket_id', {parts = {'bucket_id'}, unique = false})
    box.snapshot()

    box.schema.func.create('customer_lookup')
    box.schema.role.grant('public', 'execute', 'function', 'customer_lookup')
    box.schema.func.create('customer_add')
end)
```

Bootstrapping and restarting a storage

If a replica set master fails, it is recommended to:

1. Switch one of the replicas into the master mode. This allows the new master to process all the incoming requests.
2. Update the configuration of all the cluster members. This forwards all the requests to the new master.

Monitoring the master and switching the instance modes can be handled by any external utility.

To perform a scheduled downtime of a replica set master, it is recommended to:

1. Update the configuration of the master and wait for the replicas to get into sync. All the requests then are forwarded to a new master.
2. Switch another instance into the master mode.
3. Update the configuration of all the nodes.
4. Shut down the old master.

To perform a scheduled downtime of a replica set, it is recommended to:

1. Migrate all the buckets to the other cluster storages.
2. Update the configuration of all the nodes.
3. Shut down the replica set.

In case a whole replica set fails, some part of the dataset becomes inaccessible. Meanwhile, the router tries to reconnect to the master of the failed replica set. This way, once the replica set is up and running again, the cluster is automatically restored.

Fibers

Searches for buckets, buckets recovery, and buckets rebalancing are performed automatically and do not require human intervention.

Technically, there are multiple fibers responsible for different types of operations:

- a discovery fiber on the router searches for buckets in the background
- a failover fiber on the router maintains replica connections
- a garbage collector fiber on each master storage removes the contents of buckets that were moved

- a bucket recovery fiber on each master storage recovers buckets in the SENDING and RECEIVING states in case of reboot
- a rebalancer on a single master storage among all replica sets executes the rebalancing process.

  See the Rebalancing process section for details.

Garbage collector

A garbage collector fiber runs in the background on the master storages of each replica set. It starts deleting the contents of the bucket in the GARBAGE state part by part. Once the bucket is empty, its record is deleted from the _bucket system space.

Bucket recovery

A bucket recovery fiber runs on the master storages. It helps to recover buckets in the SENDING and RECEIVING states in case of reboot.

Buckets in the SENDING state are recovered as follows:

1. The system first searches for buckets in the SENDING state.
2. If such a bucket is found, the system sends a request to the destination replica set.
3. If the bucket on the destination replica set is ACTIVE, the original bucket is deleted from the source node.

Buckets in the RECEIVING state are deleted without extra checks.

Failover

A failover fiber runs on every router. If a master of a replica set becomes unavailable, the failover fiber redirects read requests to the replicas. Write requests are rejected with an error until the master becomes available.

Configuration reference

Basic parameters

- sharding
- weights
- shard_index
- bucket_count
- collect_bucket_garbage_interval
- collect_lua_garbage
- sync_timeout
- rebalancer_disbalance_threshold
- rebalancer_max_receiving

sharding
: A field defining the logical topology of the sharded Tarantool cluster.

  Type: table
  Default: false
  Dynamic: yes

weights
: A field defining the configuration of relative weights for each zone pair in a replica set. See the Replica weights section.

  Type: table
  Default: false
  Dynamic: yes

shard_index
: An index over the bucket id.

  Type: non-empty string or non-negative integer
  Default: coincides with the bucket id number
  Dynamic: no

bucket_count
: The total number of buckets in a cluster.

  This number should be several orders of magnitude larger than the potential number of cluster nodes, considering potential scaling out in the foreseeable future.

  Example:

  If the estimated number of nodes is M, then the data set should be divided into 100M or even 1000M buckets, depending on the planned scaling out. This number is certainly greater than the potential number of cluster nodes in the system being designed.

  Keep in mind that too many buckets can cause a need to allocate more memory to store routing information. On the other hand, an insufficient number of buckets can lead to decreased granularity when rebalancing.

  Type: number
  Default: 3000
  Dynamic: no

collect_bucket_garbage_interval
: The interval between garbage collector actions, in seconds.

  Type: number

Default: 0.5
Dynamic: yes

collect_lua_garbage
: If set to true, the Lua collectgarbage() function is called periodically.

Type: boolean
Default: no
Dynamic: yes

sync_timeout
: Timeout to wait for synchronization of the old master with replicas before demotion. Used when switching a master or when manually calling the sync() function.

Type: number
Default: 1
Dynamic: yes

rebalancer_disbalance_threshold
: A maximum bucket disbalance threshold, in percent. The threshold is calculated for each replica set using the following formula:

```
|etalon_bucket_count - real_bucket_count| / etalon_bucket_count * 100
```

Type: number
Default: 1
Dynamic: yes

rebalancer_max_receiving
: The maximum number of buckets that can be received in parallel by a single replica set. This number must be limited, because when a new replica set is added to a cluster, the rebalancer sends a very large amount of buckets from the existing replica sets to the new replica set. This produces a heavy load on a new replica set.

Example:

Suppose rebalancer_max_receiving is equal to 100, bucket_count is equal to 1000. There are 3 replica sets with 333, 333 and 334 buckets on each respectively. When a new replica set is added, each replica set's etalon_bucket_count becomes equal to 250. Rather than receiving all 250 buckets at once, the new replica set receives 100, 100 and 50 buckets sequentially.

Type: number
Default: 100
Dynamic: yes

Replica set functions

- uuid
- weight

uuid
: A unique identifier of a replica set.

  Type:
  Default:
  Dynamic:

weight
: A weight of a replica set. See the Replica set weights section for details.

  Type:
  Default: 1
  Dynamic:

API reference

Router public API

- vshard.router.bootstrap()
- vshard.router.cfg(cfg)
- vshard.router.new(name, cfg)
- vshard.router.call(bucket_id, mode, function_name, {argument_list}, {options})
- vshard.router.callro(bucket_id, function_name, {argument_list}, {options})
- vshard.router.callrw(bucket_id, function_name, {argument_list}, {options})
- vshard.router.callre(bucket_id, function_name, {argument_list}, {options})
- vshard.router.callbro(bucket_id, function_name, {argument_list}, {options})
- vshard.router.callbre(bucket_id, function_name, {argument_list}, {options})
- vshard.router.route(bucket_id)
- vshard.router.routeall()
- vshard.router.bucket_id(key)
- vshard.router.bucket_count()
- vshard.router.sync(timeout)
- vshard.router.discovery_wakeup()
- vshard.router.info()
- vshard.router.buckets_info()

- replicaset.call()
- replicaset.callro()
- replicaset.callrw()
- replicaset.callre()

vshard.router.bootstrap()
Perform the initial cluster bootstrap and distribute all buckets across the replica sets.

vshard.router.cfg(cfg)
Configure the database and start sharding for the specified router instance. See the sample configuration above.

Parameters

- cfg – a configuration table

vshard.router.new(name, cfg)
Create a new router instance. vshard supports multiple routers in a single Tarantool instance. Each router can be connected to any vshard cluster, and multiple routers can be connected to the same cluster.

A router created via vshard.router.new() works in the same way as a static router, but the method name is preceded by a colon (vshard.router:method_name(...)), while for a static router the method name is preceded by a period (vshard.router.method_name(...)).

A static router can be obtained via the vshard.router.static() method and then used like a router created via the vshard.router.new() method.

Note: box.cfg is shared among all the routers of a single instance.

Parameters

- name – a router instance name. This name is used as a prefix in logs of the router and must be unique within the instance
- cfg – a configuration table. The sample configuration is described above.

Return a router instance, if created successfully; otherwise, nil and an error object

vshard.router.call(bucket_id, mode, function_name, {argument_list}, {options})
Call the function identified by function-name on the shard storing the bucket identified by bucket_id. See the Processing requests section for details on function operation.

Parameters

- bucket_id – a bucket identifier
- mode – either a string = 'read'|'write', or a map with mode='read'|'write' and/or prefer_replica=true|false and/or balance=true|false.
- function_name – a function to execute
- argument_list – an array of the function's arguments
- options –
  - timeout – a request timeout, in seconds. If the router cannot identify a shard with the specified bucket_id, the operation will be repeated until the timeout is reached.

The mode parameter has two possible forms: a string or a map. Examples of the string form are: 'read', 'write'. Examples of the map form are: {mode='read'}, {mode='write'}, {mode='read', prefer_replica=true}, {mode='read', balance=true}, {mode='read', prefer_replica=true, balance=true}. If 'write' is specified then the target is the master. If prefer_replica=true is specified then the preferred target is one of the replicas, but the target is the master if there is no conveniently available replica. It may be good to specify prefer_replica=true for functions which are expensive in terms of resource use, to avoid slowing down the master. If balance=true then there is load balancing – reads are distributed over all the nodes in the replica set in round-robin fashion, with a preference for replicas if prefer_replica=true is also set.

Return

The original return value of the executed function, or nil and error object. The error object has a type attribute equal to ShardingError or one of the regular Tarantool errors (ClientError, OutOfMemory, SocketError, etc.).

ShardingError is returned on errors specific for sharding: the replica set is not available, the master is missing, wrong bucket id, etc. It has an attribute code containing one of the values from the vshard.error.code.* LUA table, an optional attribute containing a message with the human-readable error description, and other attributes specific for the error code.

Examples:

To call customer_add function from vshard/example, say:

```
vshard.router.call(100, 'write', 'customer_add', {{customer_id = 2, bucket_id = 100, name = 'name2',
↪accounts = {}}}, {timeout = 100})
-- or, the same thing but with a map for the second argument
vshard.router.call(100, {mode='write'}, 'customer_add', {{customer_id = 2, bucket_id = 100, name =
↪'name2', accounts = {}}}, {timeout = 100})
```

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

Call the function identified by function-name on the shard storing the bucket identified by bucket_id, in read-only mode (similar to calling vshard.router.call with mode='read'). See the Processing requests section for details on function operation.

Parameters

- bucket_id – a bucket identifier
- function_name – a function to execute
- argument_list – an array of the function's arguments
- options –
  - timeout – a request timeout, in seconds. In case the router cannot identify a shard with the bucket id, the operation will be repeated until the timeout is reached.

Return

The original return value of the executed function, or nil and error object. The error object has a type attribute equal to ShardingError or one of the regular Tarantool errors (ClientError, OutOfMemory, SocketError, etc.).

ShardingError is returned on errors specific for sharding: the replica set is not available, the master is missing, wrong bucket id, etc. It has an attribute code containing one of the values from the vshard.error.code.* LUA table, an optional attribute containing a message with the human-readable error description, and other attributes specific for this error code.

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

Call the function identified by function-name on the shard storing the bucket identified by bucket_id, in

read-write mode (similar to calling vshard.router.call with mode='write'). See the Processing requests section for details on function operation.

Parameters

- bucket_id – a bucket identifier
- function_name – a function to execute
- argument_list – an array of the function's arguments
- options –
  - timeout – a request timeout, in seconds. In case the router cannot identify a shard with the bucket id, the operation will be repeated until the timeout is reached.

Return

The original return value of the executed function, or nil and error object. The error object has a type attribute equal to ShardingError or one of the regular Tarantool errors (ClientError, OutOfMemory, SocketError, etc.).

ShardingError is returned on errors specific for sharding: the replica set is not available, the master is missing, wrong bucket id, etc. It has an attribute code containing one of the values from the vshard.error.code.* LUA table, an optional attribute containing a message with the human-readable error description, and other attributes specific for this error code.

vshard.router.callre(bucket_id, function_name, {argument_list}, {options})

Call the function identified by function-name on the shard storing the bucket identified by bucket_id, in read-only mode (similar to calling vshard.router.call with mode='read'), with preference for a replica rather than a master (similar to calling vshard.router.call with prefer_replica = true). See the Processing requests section for details on function operation.

Parameters

- bucket_id – a bucket identifier
- function_name – a function to execute
- argument_list – an array of the function's arguments
- options –
  - timeout – a request timeout, in seconds. In case the router cannot identify a shard with the bucket id, the operation will be repeated until the timeout is reached.

Return

The original return value of the executed function, or nil and error object. The error object has a type attribute equal to ShardingError or one of the regular Tarantool errors (ClientError, OutOfMemory, SocketError, etc.).

ShardingError is returned on errors specific for sharding: the replica set is not available, the master is missing, wrong bucket id, etc. It has an attribute code containing one of the values from the vshard.error.code.* LUA table, an optional attribute containing a message with the human-readable error description, and other attributes specific for this error code.

vshard.router.callbro(bucket_id, function_name, {argument_list}, {options})

This has the same effect as vshard.router.call() with mode parameter = {mode='read', balance=true}.

vshard.router.callbre(bucket_id, function_name, {argument_list}, {options})

This has the same effect as vshard.router.call() with mode parameter = {mode='read', balance=true, prefer_replica=true}.

vshard.router.route(bucket_id)
Return the replica set object for the bucket with the specified bucket id value.

Parameters

- bucket_id – a bucket identifier

Return a replica set object

Example:

```
replicaset = vshard.router.route(123)
```

vshard.router.routeall()
Return all available replica set objects.

Return a map of the following type: {UUID = replicaset}

Rtype a replica set object

Example:

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

vshard.router.bucket_id(key)
Calculate the bucket id using a simple built-in hash function.

Parameters

- key – a hash key. This can be any Lua object (number, table, string).

Return a bucket identifier

Rtype number

Example:

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

vshard.router.bucket_count()
Return the total number of buckets specified in vshard.router.cfg().

Return the total number of buckets

Rtype number

vshard.router.sync(timeout)
Wait until the dataset is synchronized on replicas.

Parameters

- timeout – a timeout, in seconds

vshard.router.discovery_wakeup()
Force wakeup of the bucket discovery fiber.

vshard.router.info()
Return information about each instance.

Return

Replica set parameters:

- replica set uuid
- master instance parameters

- replica instance parameters

Instance parameters:

- uri — URI of the instance
- uuid — UUID of the instance
- status – status of the instance (available, unreachable, missing)
- network_timeout – a timeout for the request. The value is updated automatically on each 10th successful request and each 2nd failed request.

Bucket parameters:

- available_ro – the number of buckets known to the router and available for read requests
- available_rw – the number of buckets known to the router and available for read and write requests
- unavailable – the number of buckets known to the router but unavailable for any requests
- unreachable – the number of buckets whose replica sets are not known to the router

Example:

```
tarantool> vshard.router.info()
---
- replicasets:
    ac522f65-aa94-4134-9f64-51ee384f1a54:
      replica: &0
        network_timeout: 0.5
        status: available
        uri: storage@127.0.0.1:3303
        uuid: 1e02ae8a-afc0-4e91-ba34-843a356b8ed7
      uuid: ac522f65-aa94-4134-9f64-51ee384f1a54
      master: *0
    cbf06940-0790-498b-948d-042b62cf3d29:
      replica: &1
        network_timeout: 0.5
        status: available
        uri: storage@127.0.0.1:3301
        uuid: 8a274925-a26d-47fc-9e1b-af88ce939412
      uuid: cbf06940-0790-498b-948d-042b62cf3d29
      master: *1
  bucket:
    unreachable: 0
    available_ro: 0
    unknown: 0
    available_rw: 3000
  status: 0
  alerts: []
...
```

vshard.router.buckets_info()

Return information about each bucket. Since a bucket map can be huge, only the required range of buckets can be specified.

Parameters

- offset – the offset in a bucket map of the first bucket to show
- limit – the maximum number of buckets to show

Return a map of the following type: {bucket_id = 'unknown'/replicaset_uuid}

replicaset.call(replicaset, function_name, {argument_list}, {options})

Call a function on a nearest available master (distances are defined using replica.zone and cfg.weights matrix) with specified arguments.

Note: The replicaset.call method is similar to replicaset.callrw.

Parameters

- replicaset – UUID of a replica set
- function_name – function to execute
- argument_list – array of the function's arguments
- options –
  - timeout – a request timeout, in seconds. In case the router cannot identify a shard with the bucket id, the operation will be repeated until the timeout is reached.

replicaset.callrw(replicaset, function_name, {argument_list}, {options})

Call a function on a nearest available master (distances are defined using replica.zone and cfg.weights matrix) with a specified arguments.

Note: The replicaset.callrw method is similar to replicaset.call.

Parameters

- replicaset – UUID of a replica set
- function_name – function to execute
- argument_list – array of the function's arguments
- options –
  - timeout – a request timeout, in seconds. In case the router cannot identify a shard with the bucket id, the operation will be repeated until the timeout is reached.

replicaset.callro(function_name, {argument_list}, {options})

Call a function on the nearest available replica (distances are defined using replica.zone and cfg.weights matrix) with specified arguments. It is recommended to call only read-only functions using replicaset.callro(), as the function can be executed not only on a master, but also on replicas.

Parameters

- replicaset – UUID of a replica set
- function_name – function to execute
- argument_list – array of the function's arguments
- options –
  - timeout – a request timeout, in seconds. In case the router cannot identify a shard with the bucket id, the operation will be repeated until the timeout is reached.

replicaset.callre(function_name, {argument_list}, {options})
Call a function on the nearest available replica (distances are defined using replica.zone and cfg.weights matrix) with specified arguments, with preference for a replica rather than a master (similar to calling vshard.router.call with prefer_replica = true). It is recommended to call only read-only functions using replicaset.callre(), as the function can be executed not only on a master, but also on replicas.

Parameters

- replicaset – UUID of a replica set
- function_name – function to execute
- argument_list – array of the function's arguments
- options –
  - timeout – a request timeout, in seconds. In case the router cannot identify a shard with the bucket id, the operation will be repeated until the timeout is reached.

Router internal API

- vshard.router.bucket_discovery(bucket_id)

vshard.router.bucket_discovery(bucket_id)
Search for the bucket in the whole cluster. If the bucket is not found, it is likely that it does not exist. The bucket might also be moved during rebalancing and currently is in the RECEIVING state.

Parameters

- bucket_id – a bucket identifier

Storage public API

- vshard.storage.cfg(cfg, name)
- vshard.storage.info()
- vshard.storage.call(bucket_id, mode, function_name, {argument_list})
- vshard.storage.sync(timeout)
- vshard.storage.bucket_pin(bucket_id)
- vshard.storage.bucket_unpin(bucket_id)
- vshard.storage.bucket_ref(bucket_id, mode)
- vshard.storage.bucket_refro()
- vshard.storage.bucket_refrw()
- vshard.storage.bucket_unref(bucket_id, mode)
- vshard.storage.bucket_unrefro()
- vshard.storage.bucket_unrefrw()
- vshard.storage.find_garbage_bucket(bucket_index, control)
- vshard.storage.rebalancer_disable()
- vshard.storage.rebalancer_enable()
- vshard.storage.is_locked()

- vshard.storage.rebalancing_is_in_progress()
- vshard.storage.buckets_info()
- vshard.storage.buckets_count()
- vshard.storage.sharded_spaces()

vshard.storage.cfg(cfg, name)
Configure the database and start sharding for the specified storage instance.

Parameters

- cfg – a storage configuration
- instance_uuid – UUID of the instance

vshard.storage.info()
Return information about the storage instance in the following format:

```
tarantool> vshard.storage.info()
---
- buckets:
    2995:
      status: active
      id: 2995
    2997:
      status: active
      id: 2997
    2999:
      status: active
      id: 2999
  replicasets:
    2dd0a343-624e-4d3a-861d-f45efc571cd3:
      uuid: 2dd0a343-624e-4d3a-861d-f45efc571cd3
      master:
        state: active
        uri: storage:storage@127.0.0.1:3301
        uuid: 2ec29309-17b6-43df-ab07-b528e1243a79
    c7ad642f-2cd8-4a8c-bb4e-4999ac70bba1:
      uuid: c7ad642f-2cd8-4a8c-bb4e-4999ac70bba1
      master:
        state: active
        uri: storage:storage@127.0.0.1:3303
        uuid: 810d85ef-4ce4-4066-9896-3c352fec9e64
...
```

vshard.storage.call(bucket_id, mode, function_name, {argument_list})
Call the specified function on the current storage instance.

Parameters

- bucket_id – a bucket identifier
- mode – a type of the function: ‘read’ or ‘write’
- function_name – function to execute
- argument_list – array of the function’s arguments

Return

The original return value of the executed function, or nil and error object.

vshard.storage.sync(timeout)
Wait until the dataset is synchronized on replicas.

Parameters

- timeout – a timeout, in seconds

vshard.storage.bucket_pin(bucket_id)
Pin a bucket to a replica set. A pinned bucket cannot be moved even if it breaks the cluster balance.

Parameters

- bucket_id – a bucket identifier

Return true if the bucket is pinned successfully; or nil and err explaining why the bucket cannot be pinned

vshard.storage.bucket_unpin(bucket_id)
Return a pinned bucket back into the active state.

Parameters

- bucket_id – a bucket identifier

Return true if the bucket is unpinned successfully; or nil and err explaining why the bucket cannot be unpinned

vshard.storage.bucket_ref(bucket_id, mode)
Create an RO or RW ref.

Parameters

- bucket_id – a bucket identifier
- mode – 'read' or 'write'

Return true if the bucket ref is created successfully; or nil and err explaining why the ref cannot be created

vshard.storage.bucket_refro()
An alias for vshard.storage.bucket_ref in the RO mode.

vshard.storage.bucket_refrw()
An alias for vshard.storage.bucket_ref in the RW mode.

vshard.storage.bucket_unref(bucket_id, mode)
Remove a RO/RW ref.

Parameters

- bucket_id – a bucket identifier
- mode – 'read' or 'write'

Return true if the bucket ref is removed successfully; or nil and err explaining why the ref cannot be removed

vshard.storage.bucket_unrefro()
An alias for vshard.storage.bucket_unref in the RO mode.

vshard.storage.bucket_unrefrw()
An alias for vshard.storage.bucket_unref in the RW mode.

vshard.storage.find_garbage_bucket(bucket_index, control)
Find a bucket which has data in a space but is not stored in a _bucket space; or is in a GARBAGE state.

Parameters

- bucket_index – index of a space with the part of a bucket id
- control – a garbage collector controller. If there is an increased buckets generation, then the search should be interrupted.

Return an identifier of the bucket in the garbage state, if found; otherwise, nil

vshard.storage.buckets_info()
Return information about each bucket located in storage. For example:

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

vshard.storage.buckets_count()
Return the number of buckets located in storage.

vshard.storage.recovery_wakeup()
Immediately wake up a recovery fiber, if it exists.

vshard.storage.rebalancing_is_in_progress()
Return a flag indicating whether rebalancing is in progress. The result is true if the node is currently applying routes received from a rebalancer node in the special fiber.

vshard.storage.is_locked()
Return a flag indicating whether storage is invisible to the rebalancer.

vshard.storage.rebalancer_disable()
Disable rebalancing. A disabled rebalancer sleeps until it is enabled again with vshard.storage.rebalancer_enable().

vshard.storage.rebalancer_enable()
Enable rebalancing.

vshard.storage.sharded_spaces()
Show the spaces that are visible to rebalancer and garbage collector fibers.

Storage internal API

- vshard.storage.bucket_stat(bucket_id)
- vshard.storage.bucket_recv(bucket_id, from, data)
- vshard.storage.bucket_delete_garbage(bucket_id)
- vshard.storage.bucket_collect(bucket_id)
- vshard.storage.bucket_force_create(first_bucket_id, count)
- vshard.storage.bucket_force_drop(bucket_id, to)
- vshard.storage.bucket_send(bucket_id, to)
- vshard.storage.buckets_discovery()

- vshard.storage.rebalancer_request_state()

vshard.storage.bucket_recv(bucket_id, from, data)
Receive a bucket identified by bucket id from a remote replica set.

Parameters

- bucket_id – a bucket identifier
- from – UUID of source replica set
- data – data logically stored in a bucket identified by bucket_id, in the same format as the return value from bucket_collect() <storage_api-bucket_collect>

vshard.storage.bucket_stat(bucket_id)
Return information about the bucket id:

```
tarantool> vshard.storage.bucket_stat(1)
---
- 0
- status: active
  id: 1
...
```

Parameters

- bucket_id – a bucket identifier

vshard.storage.bucket_delete_garbage(bucket_id)
Force garbage collection for the bucket identified by bucket_id in case the bucket was transferred to a different replica set.

Parameters

- bucket_id – a bucket identifier

vshard.storage.bucket_collect(bucket_id)
Collect all the data that is logically stored in the bucket identified by bucket_id:

```
tarantool> vshard.storage.bucket_collect(1)
---
- 0
- - - 514
    - - [10, 1, 1, 100, 'Account 10']
      - [11, 1, 1, 100, 'Account 11']
      - [12, 1, 1, 100, 'Account 12']
      - [50, 5, 1, 100, 'Account 50']
      - [51, 5, 1, 100, 'Account 51']
      - [52, 5, 1, 100, 'Account 52']
  - - 513
    - - [1, 1, 'Customer 1']
      - [5, 1, 'Customer 5']
...
```

Parameters

- bucket_id – a bucket identifier

vshard.storage.bucket_force_create(first_bucket_id, count)
Force creation of the buckets (single or multiple) on the current replica set. Use only for manual emergency recovery or for initial bootstrap.

Parameters

- first_bucket_id – an identifier of the first bucket in a range
- count – the number of buckets to insert (default = 1)

vshard.storage.bucket_force_drop(bucket_id)
Drop a bucket manually for tests or emergency cases.

Parameters

- bucket_id – a bucket identifier

vshard.storage.bucket_send(bucket_id, to)
Send a specified bucket from the current replica set to a remote replica set.

Parameters

- bucket_id – bucket identifier
- to – UUID of a remote replica set

vshard.storage.rebalancer_request_state()
Check all buckets of the host storage that have the SENT or ACTIVE state, return the number of active buckets.

Return the number of buckets in the active state, if found; otherwise, nil

vshard.storage.buckets_discovery()
Collect an array of active bucket identifiers for discovery.

Glossary

Vertical scaling Adding more power to a single server: using a more powerful CPU, adding more capacity to RAM, adding more storage space, etc.

Horizontal scaling Adding more servers to the pool of resources, then partitioning and distributing a dataset across the servers.

Sharding A database architecture that allows partitioning a dataset using a sharding key and distributing a dataset across multiple servers. Sharding is a special case of horizontal scaling.

Node A virtual or physical server instance.

Cluster A set of nodes that make up a single group.

Storage A node storing a subset of a dataset.

Replica set A set of storage nodes storing copies of a dataset. Each storage in a replica set has a role, master or replica.

Master A storage in a replica set processing read and write requests.

Replica A storage in a replica set processing only read requests.

Read requests Read-only requests, that is, select requests.

Write requests Data-change operations, that is create, replace, update, delete requests.

Buckets (virtual buckets) The abstract virtual nodes into which the dataset is partitioned by the sharding key (bucket id).

Bucket id A sharding key defining which bucket belongs to which replica set. A bucket id may be calculated from a hash key.

Router A proxy server responsible for routing requests from an application to nodes in a cluster.

4.3.5 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.

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.

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.

4.4 Configuration reference

This reference covers all options and parameters which can be set for Tarantool on the command line or in an initialization file.

Tarantool is started by entering either of the following command:

```
$ tarantool
```

```
$ tarantool options
```

```
$ tarantool lua-initialization-file [ arguments ]
```

4.4.1 Command options

-h, --help
: Print an annotated list of all available options and exit.

-V, --version
: Print product name and version, for example:

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

In this example:

"Tarantool" is the name of the reusable asynchronous networking programming framework.

The 3-number version follows the standard <major>-<minor>-<patch> scheme, in which <major> number is changed only rarely, <minor> is incremented for each new milestone and indicates possible incompatible changes, and <patch> stands for the number of bug fix releases made after the start of the milestone. For non-released versions only, there may be a commit number and commit SHA1 to indicate how much this particular build has diverged from the last release.

"Target" is the platform tarantool was built on. Some platform-specific details may follow this line.

Note: Tarantool uses git describe to produce its version id, and this id can be used at any time to check out the corresponding source from our git repository.

4.4.2 URI

Some configuration parameters and some functions depend on a URI, or "Universal Resource Identifier". The URI string format is similar to the generic syntax for a URI schema. So it may contain (in order) a user name for login, a password, a host name or host IP address, and a port number. Only the port number is always mandatory. The password is mandatory if the user name is specified, unless the user name is 'guest'. So, formally, the URI syntax is [host:]port or [username:password@]host:port. If host is omitted, then '0.0.0.0' or '[::]' is assumed, meaning respectively any IPv4 address or any IPv6 address, on the local machine. If username:password is omitted, then 'guest' is assumed. Some examples:

URI fragment	Example
port	3301
host:port	127.0.0.1:3301
username:password@host:port	notguest:sesame@mail.ru:3301

In certain circumstances a Unix domain socket may be used where a URI is expected, for example "unix/:/tmp/unix_domain_socket.sock" or simply "/tmp/unix_domain_socket.sock".

A method for parsing URIs is illustrated in Module uri.

4.4.3 Initialization file

If the command to start Tarantool includes lua-initialization-file, then Tarantool begins by invoking the Lua program in the file, which by convention may have the name "script.lua". The Lua program may get further arguments from the command line or may use operating-system functions, such as getenv(). The

Lua program almost always begins by invoking box.cfg(), if the database server will be used or if ports need to be opened. For example, suppose script.lua contains the lines

```
#!/usr/bin/env tarantool
box.cfg{
    listen              = os.getenv("LISTEN_URI"),
    memtx_memory        = 100000,
    pid_file            = "tarantool.pid",
    rows_per_wal        = 50
}
print('Starting ', arg[1])
```

and suppose the environment variable LISTEN_URI contains 3301, and suppose the command line is ~/tarantool/src/tarantool script.lua ARG. Then the screen might look like this:

```
$ 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
```

If you wish to start an interactive session on the same terminal after initialization is complete, you can use console.start().

4.4.4 Configuration parameters

Configuration parameters have the form:

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

Since box.cfg may contain many configuration parameters and since some of the parameters (such as directory addresses) are semi-permanent, it's best to keep box.cfg in a Lua file. Typically this Lua file is the initialization file which is specified on the tarantool command line.

Most configuration parameters are for allocating resources, opening ports, and specifying database behavior. All parameters are optional. A few parameters are dynamic, that is, they can be changed at runtime by calling box.cfg{} a second time.

To see all the non-null parameters, say box.cfg (no parentheses). To see a particular parameter, for example the listen address, say box.cfg.listen.

The following sections describe all parameters for basic operation, for storage, for binary logging and snapshots, for replication, for networking, for logging, and for feedback.

Basic parameters

- background
- custom_proc_title
- listen

- memtx_dir
- pid_file
- read_only
- vinyl_dir
- vinyl_timeout
- username
- wal_dir
- work_dir
- worker_pool_threads
- strip_core

background
: Run the server as a background task. The log and pid_file parameters must be non-null for this to work.

  Type: boolean

  Default: false

  Dynamic: no

custom_proc_title
: Add the given string to the server's process title (what's shown in the COMMAND column for ps -ef and top -c commands).

  For example, ordinarily ps -ef shows the Tarantool server process thus:

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

  But if the configuration parameters include custom_proc_title='sessions' then the output looks like:

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

  Type: string

  Default: null

  Dynamic: yes

listen
: The read/write data port number or URI (Universal Resource Identifier) string. Has no default value, so must be specified if connections will occur from remote clients that do not use the "admin port". Connections made with listen = URI are called "binary port" or "binary protocol" connections.

  A typical value is 3301.

Note: A replica also binds to this port, and accepts connections, but these connections can only serve reads until the replica becomes a master.

Type: integer or string
Default: null
Dynamic: yes

memtx_dir
: A directory where memtx stores snapshot (.snap) files. Can be relative to work_dir. If not specified, defaults to work_dir. See also wal_dir.

Type: string
Default: "."
Dynamic: no

pid_file
: Store the process id in this file. Can be relative to work_dir. A typical value is "tarantool.pid".

Type: string
Default: null
Dynamic: no

read_only
: Say box.cfg{read_only=true...} to put the server instance in read-only mode. After this, any requests that try to change persistent data will fail with error ER_READONLY. Read-only mode should be used for master-replica replication. Read-only mode does not affect data-change requests for spaces defined as temporary. Although read-only mode prevents the server from writing to the WAL, it does not prevent writing diagnostics with the log module.

Type: boolean
Default: false
Dynamic: yes

Setting read_only == true affects spaces differently depending on the options that were used during box.schema.space.create, as summarized by this chart:

Option	Can be created?	Can be written to?	Is replicated?	Is persistent?
(default)	no	no	yes	yes
temporary	no	yes	no	no
is_local	no	yes	no	yes

vinyl_dir
: A directory where vinyl files or subdirectories will be stored. Can be relative to work_dir. If not specified, defaults to work_dir.

  Type: string
  Default: "."
  Dynamic: no

vinyl_timeout
: The vinyl storage engine has a scheduler which does compaction. When vinyl is low on available memory, the compaction scheduler may be unable to keep up with incoming update requests. In that situation, queries may time out after vinyl_timeout seconds. This should rarely occur, since normally vinyl would throttle inserts when it is running low on compaction bandwidth. Compaction can also be ordered manually with index_object:compact().

  Type: float
  Default: 60
  Dynamic: yes

username
: UNIX user name to switch to after start.

  Type: string
  Default: null
  Dynamic: no

wal_dir
: A directory where write-ahead log (.xlog) files are stored. Can be relative to work_dir. Sometimes wal_dir and memtx_dir are specified with different values, so that write-ahead log files and snapshot files can be stored on different disks. If not specified, defaults to work_dir.

  Type: string
  Default: "."
  Dynamic: no

work_dir
: A directory where database working files will be stored. The server instance switches to work_dir with chdir(2) after start. Can be relative to the current directory. If not specified, defaults to the current directory. Other directory parameters may be relative to work_dir, for example:

```
box.cfg{
    work_dir = '/home/user/A',
    wal_dir = 'B',
    memtx_dir = 'C'
}
```

will put xlog files in /home/user/A/B, snapshot files in /home/user/A/C, and all other files or subdirectories in /home/user/A.

Type: string
Default: null
Dynamic: no

worker_pool_threads
: The maximum number of threads to use during execution of certain internal processes (currently socket.getaddrinfo() and coio_call()).

Type: integer
Default: 4
Dynamic: yes

strip_core
: Whether coredump files should include memory allocated for tuples. (This can be large if Tarantool runs under heavy load.) Setting to true means "do not include". In an older version of Tarantool the default value of this parameter was false.

Type: boolean
Default: true
Dynamic: no

Configuring the storage

- memtx_memory
- memtx_max_tuple_size
- memtx_min_tuple_size
- vinyl_bloom_fpr
- vinyl_cache
- vinyl_max_tuple_size
- vinyl_memory
- vinyl_page_size
- vinyl_range_size
- vinyl_run_count_per_level
- vinyl_run_size_ratio
- vinyl_read_threads
- vinyl_write_threads

memtx_memory
: How much memory Tarantool allocates to actually store tuples, in bytes. When the limit is reached, INSERT or UPDATE requests begin failing with error ER_MEMORY_ISSUE. The server does not go beyond the memtx_memory limit to allocate tuples, but there is additional memory used to store indexes and connection information. Depending on actual configuration and workload, Tarantool can consume up to 20% more than the memtx_memory limit.

  Type: float
  Default: 256 * 1024 * 1024 = 268435456
  Dynamic: yes but it cannot be decreased

memtx_max_tuple_size
: Size of the largest allocation unit, in bytes, for the memtx storage engine. It can be increased if it is necessary to store large tuples. See also: vinyl_max_tuple_size.

  Type: integer
  Default: 1024 * 1024 = 1048576
  Dynamic: no

memtx_min_tuple_size
: Size of the smallest allocation unit, in bytes. It can be decreased if most of the tuples are very small. The value must be between 8 and 1048280 inclusive.

  Type: integer
  Default: 16
  Dynamic: no

vinyl_bloom_fpr
: Bloom filter false positive rate – the suitable probability of the bloom filter to give a wrong result. The vinyl_bloom_fpr setting is a default value for one of the options in the Options for space_object:create_index() chart.

  Type: float
  Default = 0.05
  Dynamic: no

vinyl_cache
: The cache size for the vinyl storage engine, in bytes. The cache can be resized dynamically.

  Type: integer
  Default = 128 * 1024 * 1024 = 134217728
  Dynamic: yes

vinyl_max_tuple_size
: Size of the largest allocation unit, in bytes, for the vinyl storage engine. It can be increased if it is necessary to store large tuples. See also: memtx_max_tuple_size.

  Type: integer
  Default: 1024 * 1024 = 1048576
  Dynamic: no

vinyl_memory
: The maximum number of in-memory bytes that vinyl uses.

  Type: integer
  Default = 128 * 1024 * 1024 = 134217728
  Dynamic: yes but it cannot be decreased

vinyl_page_size
: Page size, in bytes. Page is a read/write unit for vinyl disk operations. The vinyl_page_size setting is a default value for one of the options in the Options for space_object:create_index() chart.

  Type: integer
  Default = 8 * 1024 = 8192
  Dynamic: no

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

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

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

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

  Type: integer
  Default = nil
  Dynamic: no

vinyl_run_count_per_level
: The maximal number of runs per level in vinyl LSM tree. If this number is exceeded, a new level is created. The vinyl_run_count_per_level setting is a default value for one of the options in the Options for space_object:create_index() chart.

Type: integer
Default = 2
Dynamic: no

vinyl_run_size_ratio
: Ratio between the sizes of different levels in the LSM tree. The vinyl_run_size_ratio setting is a default value for one of the options in the Options for space_object:create_index() chart.

Type: float
Default = 3.5
Dynamic: no

vinyl_read_threads
: The maximum number of read threads that vinyl can use for some concurrent operations, such as I/O and compression.

Type: integer
Default = 1
Dynamic: no

vinyl_write_threads
: The maximum number of write threads that vinyl can use for some concurrent operations, such as I/O and compression.

Type: integer
Default = 2
Dynamic: no

Checkpoint daemon

- checkpoint_count
- checkpoint_interval
- checkpoint_wal_threshold

The checkpoint daemon is a fiber which is constantly running. At intervals, it may make new snapshot (.snap) files and then may delete old snapshot files.

The checkpoint_interval and checkpoint_count configuration settings determine how long the intervals are, and how many snapshots should exist before deletions occur.

Tarantool garbage collector

The checkpoint daemon may activate the Tarantool garbage collector which deletes old files. This garbage collector is distinct from the Lua garbage collector which is for Lua objects, and distinct from a Tarantool garbage collector which specializes in handling shard buckets.

If the checkpoint daemon deletes an old snapshot file, then the Tarantool garbage collector will also delete any write-ahead log (.xlog) files which are older than the snapshot file and which contain information that is present in the snapshot file. It will also delete obsolete vinyl .run files.

The checkpoint daemon and the Tarantool garbage collector will not delete a file if:

- a backup is ongoing and the file has not been backed up (see "Hot backup"), or
- replication is ongoing and the file has not been relayed to a replica (see "Replication architecture"),
- a replica is connecting, or
- a replica has fallen behind. The progress of each replica is tracked; if a replica's position is far from being up to date, then the server stops to give it a chance to catch up. If an administrator concludes that a replica is permanently down, then the correct procedure is to restart the server, or (preferably) remove the replica from the cluster.

checkpoint_interval
: The interval between actions by the checkpoint daemon, in seconds. If checkpoint_interval is set to a value greater than zero, and there is activity which causes change to a database, then the checkpoint daemon will call box.snapshot every checkpoint_interval seconds, creating a new snapshot file each time. If checkpoint_interval is set to zero, then the checkpoint daemon is disabled.

  For example:

```
box.cfg{checkpoint_interval=60}
```

  will cause the checkpoint daemon to create a new database snapshot once per minute, if there is activity.

  Type: integer
  Default: 3600 (one hour)
  Dynamic: yes

checkpoint_count
: The maximum number of snapshots that may exist on the memtx_dir directory before the checkpoint daemon will delete old snapshots. If checkpoint_count equals zero, then the checkpoint daemon does not delete old snapshots. For example:

```
box.cfg{
    checkpoint_interval = 3600,
    checkpoint_count  = 10
}
```

  will cause the checkpoint daemon to create a new snapshot each hour until it has created ten snapshots. After that, it will delete the oldest snapshot (and any associated write-ahead-log files) after creating a new one.

  Remember that, as noted earlier, snapshots will not be deleted if replication is ongoing and the file has not been relayed to a replica. Therefore checkpoint_count has no effect unless all replicas are alive.

  Type: integer
  Default: 2
  Dynamic: yes

checkpoint_wal_threshold
: The threshold for the total size in bytes of all WAL files created since the last checkpoint. Once the configured threshold is exceeded, the WAL thread notifies the checkpoint daemon that it must make a new checkpoint and delete old WAL files.

  Type: integer
  Default: 10^18 (a large number so in effect there is no limit by default)
  Dynamic: yes

  This parameter was added in version 2.1. It enables administrators to handle a problem that could occur with calculating how much disk space to allocate for a partition containing WAL files. For example, suppose checkpoint_interval = 2 and checkpoint_count = 5 and the average amount that Tarantool writes between each checkpoint interval = 1 GB. Then one could calculate that the necessary amount is (2*5*1) 10GB. But this calculation would be wrong if, instead of writing 1 GB during one checkpoint interval, Tarantool encounters an unusual spike and tries to write 11 GB, causing an operating-system ENOSPC ("no space") error. By setting checkpoint_wal_threshold to a lower value, say 9 GB, an administrator could prevent the error.

Binary logging and snapshots

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

force_recovery
: If force_recovery equals true, Tarantool tries to continue if there is an error while reading a snapshot file (at server instance start) or a write-ahead log file (at server instance start or when applying an update at a replica): skips invalid records, reads as much data as possible and lets the process finish with a warning. Users can prevent the error from recurring by writing to the database and executing box.snapshot().

  Otherwise, Tarantool aborts recovery if there is an error while reading.

  Type: boolean
  Default: false
  Dynamic: no

rows_per_wal
: How many log records to store in a single write-ahead log file. When this limit is reached, Tarantool creates another WAL file named <first-lsn-in-wal>.xlog. This can be useful for simple rsync-based backups.

  Type: integer

Default: 500000
Dynamic: no

wal_max_size
: The maximum number of bytes in a single write-ahead log file. When a request would cause an .xlog file to become larger than wal_max_size, Tarantool creates another WAL file – the same effect that happens when the rows_per_wal limit is reached.

Type: integer
Default: 268435456 (256 * 1024 * 1024)
Dynamic: no

snap_io_rate_limit
: Reduce the throttling effect of box.snapshot on INSERT/UPDATE/DELETE performance by setting a limit on how many megabytes per second it can write to disk. The same can be achieved by splitting wal_dir and memtx_dir locations and moving snapshots to a separate disk. The limit also affects what box.stat.vinyl().regulator may show for the write rate of dumps to .run and .index files.

Type: float
Default: null
Dynamic: yes

wal_mode
: Specify fiber-WAL-disk synchronization mode as:

  - none: write-ahead log is not maintained;
  - write: fibers wait for their data to be written to the write-ahead log (no fsync(2));
  - fsync: fibers wait for their data, fsync(2) follows each write(2);

Type: string
Default: “write”
Dynamic: no

wal_dir_rescan_delay
: Number of seconds between periodic scans of the write-ahead-log file directory, when checking for changes to write-ahead-log files for the sake of replication or hot standby.

Type: float
Default: 2
Dynamic: no

Hot standby

hot_standby
: Whether to start the server in hot standby mode.

  Hot standby is a feature which provides a simple form of failover without replication.

  The expectation is that there will be two instances of the server using the same configuration. The first one to start will be the "primary" instance. The second one to start will be the "standby" instance.

  To initiate the standby instance, start a second instance of the Tarantool server on the same computer with the same box.cfg configuration settings – including the same directories and same non-null URIs – and with the additional configuration setting hot_standby = true. Expect to see a notification ending with the words I> Entering hot standby mode. This is fine. It means that the standby instance is ready to take over if the primary instance goes down.

  The standby instance will initialize and will try to take a lock on wal_dir, but will fail because the primary instance has made a lock on wal_dir. So the standby instance goes into a loop, reading the write ahead log which the primary instance is writing (so the two instances are always in sync), and trying to take the lock. If the primary instance goes down for any reason, the lock will be released. In this case, the standby instance will succeed in taking the lock, will connect on the listen address and will become the primary instance. Expect to see a notification ending with the words I> ready to accept requests.

  Thus there is no noticeable downtime if the primary instance goes down.

  Hot standby feature has no effect:

  - if wal_dir_rescan_delay = a large number (on Mac OS and FreeBSD); on these platforms, it is designed so that the loop repeats every wal_dir_rescan_delay seconds.
  - if wal_mode = 'none'; it is designed to work with wal_mode = `'write'` or wal_mode = `'fsync'`.
  - for spaces created with engine = 'vinyl'; it is designed to work for spaces created with engine = `'memtx'`.

  Type: boolean
  Default: false
  Dynamic: no

Replication

- replication
- replication_timeout
- replication_connect_timeout
- replication_connect_quorum
- replication_skip_conflict
- replication_sync_lag
- replication_sync_timeout
- replicaset_uuid
- instance_uuid

replication

If replication is not an empty string, the instance is considered to be a Tarantool replica. The replica will try to connect to the master specified in replication with a URI (Universal Resource Identifier), for example:

konstantin:secret_password@tarantool.org:3301

If there is more than one replication source in a replica set, specify an array of URIs, for example (replace 'uri' and 'uri2' in this example with valid URIs):

box.cfg{ replication = { 'uri1', 'uri2' } }

If one of the URIs is "self" – that is, if one of the URIs is for the instance where box.cfg{} is being executed on – then it is ignored. Thus it is possible to use the same replication specification on multiple server instances, as shown in these examples.

The default user name is 'guest'.

A read-only replica does not accept data-change requests on the listen port.

The replication parameter is dynamic, that is, to enter master mode, simply set replication to an empty string and issue:

box.cfg{ replication = new-value }

Type: string
Default: null
Dynamic: yes

replication_timeout

A replica sends heartbeat messages to the master every second, and the master is programmed to reconnect automatically if it doesn't see heartbeat messages more often than replication_timeout seconds.

See more in Monitoring a replica set.

Type: integer
Default: 1
Dynamic: yes

replication_connect_timeout

The number of seconds that a replica will wait when trying to connect to a master in a cluster. See orphan status for details.

This parameter is different from replication_timeout, which is only used to automatically reconnect replication when it gets no heartbeats.

Type: float
Default: 4
Dynamic: yes

replication_connect_quorum
: By default a replica will try to connect to all the masters, or it will not start. (The default is recommended so that all replicas will receive the same replica set UUID.)

  However, by specifying replication_connect_quorum = N, where N is a number greater than or equal to zero, users can state that the replica only needs to connect to N masters.

  This parameter has effect during bootstrap and during configuration update. Setting replication_connect_quorum = 0 makes Tarantool require no immediate reconnect only in case of recovery. See orphan status for details.

  Example:

```
box.cfg{replication_connect_quorum=2}
```

  Type: integer

  Default: null

  Dynamic: yes

replication_skip_conflict
: By default, if a replica adds a unique key that another replica has added, replication stops with error = ER_TUPLE_FOUND.

  However, by specifying replication_skip_conflict = true, users can state that such errors may be ignored.

  Example:

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

  Type: boolean

  Default: false

  Dynamic: yes

replication_sync_lag
: The maximum lag allowed for a replica. When a replica syncs (gets updates from a master), it may not catch up completely. The number of seconds that the replica is behind the master is called the "lag". Syncing is considered to be complete when the replica's lag is less than or equal to replication_sync_lag.

  If a user sets replication_sync_lag to nil or to 365 * 100 * 86400 (TIMEOUT_INFINITY), then lag does not matter – the replica is always considered to be "synced". Also, the lag is ignored (assumed to be infinite) in case the master is running Tarantool older than 1.7.7, which does not send heartbeat messages.

  This parameter is ignored during bootstrap. See orphan status for details.

  Type: float

  Default: 10

  Dynamic: yes

replication_sync_timeout
: The number of seconds that a replica will wait when trying to sync with a master in a cluster, or a quorum of masters, after connecting or during configuration update. This could fail indefinitely if replication_sync_lag is smaller than network latency, or if the replica cannot keep pace with master updates. If replication_sync_timeout expires, the replica enters orphan status.

  Type: float
  Default: 300
  Dynamic: yes

replicaset_uuid
: As described in section "Replication architecture", each replica set is identified by a universally unique identifier called replica set UUID, and each instance is identified by an instance UUID.

  Ordinarily it is sufficient to let the system generate and format the UUID strings which will be permanently stored.

  However, some administrators may prefer to store Tarantool configuration information in a central repository, for example Apache ZooKeeper. Such administrators can assign their own UUID values for either – or both – instances (instance_uuid) and replica set (replicaset_uuid), when starting up for the first time.

  General rules:

  - The values must be true unique identifiers, not shared by other instances or replica sets within the common infrastructure.
  - The values must be used consistently, not changed after initial setup (the initial values are stored in snapshot files and are checked whenever the system is restarted).
  - The values must comply with RFC 4122. The nil UUID is not allowed.

  The UUID format includes sixteen octets represented as 32 hexadecimal (base 16) digits, displayed in five groups separated by hyphens, in the form 8-4-4-4-12 for a total of 36 characters (32 alphanumeric characters and four hyphens).

  Example:

```
box.cfg{replicaset_uuid='7b853d13-508b-4b8e-82e6-806f088ea6e9'}
```

  Type: string
  Default: null
  Dynamic: no

instance_uuid
: For replication administration purposes, it is possible to set the universally unique identifiers of the instance (instance_uuid) and the replica set (replicaset_uuid), instead of having the system generate the values.

  See the description of replicaset_uuid parameter for details.

  Example:

```
box.cfg{instance_uuid='037fec43-18a9-4e12-a684-a42b716fcd02'}
```

Type: string
Default: null
Dynamic: no

Networking

- io_collect_interval,
- net_msg_max
- readahead,

io_collect_interval
: The instance will sleep for io_collect_interval seconds between iterations of the event loop. Can be used to reduce CPU load in deployments in which the number of client connections is large, but requests are not so frequent (for example, each connection issues just a handful of requests per second).

  Type: float
  Default: null
  Dynamic: yes

net_msg_max
: To handle messages, Tarantool allocates fibers. To prevent fiber overhead from affecting the whole system, Tarantool restricts how many messages the fibers handle, so that some pending requests are blocked.

  On powerful systems, increase net_msg_max and the scheduler will immediately start processing pending requests.

  On weaker systems, decrease net_msg_max and the overhead may decrease although this may take some time because the scheduler must wait until already-running requests finish.

  When net_msg_max is reached, Tarantool suspends processing of incoming packages until it has processed earlier messages. This is not a direct restriction of the number of fibers that handle network messages, rather it is a system-wide restriction of channel bandwidth. This in turn causes restriction of the number of incoming network messages that the transaction processor thread handles, and therefore indirectly affects the fibers that handle network messages. (The number of fibers is smaller than the number of messages because messages can be released as soon as they are delivered, while incoming requests might not be processed until some time after delivery.)

  On typical systems, the default value (768) is correct.

  Type: integer
  Default: 768
  Dynamic: yes

readahead
: The size of the read-ahead buffer associated with a client connection. The larger the buffer, the more memory an active connection consumes and the more requests can be read from the operating system buffer in a single system call. The rule of thumb is to make sure the buffer can contain at least a few dozen requests. Therefore, if a typical tuple in a request is large, e.g. a few kilobytes or even megabytes, the read-ahead buffer size should be increased. If batched request processing is not used, it's prudent to leave this setting at its default.

  Type: integer
  Default: 16320
  Dynamic: yes

Logging

- log_level
- log
- log_nonblock
- too_long_threshold
- log_format

log_level
: What level of detail the log will have. There are seven levels:

  - 1 – SYSERROR
  - 2 – ERROR
  - 3 – CRITICAL
  - 4 – WARNING
  - 5 – INFO
  - 6 – VERBOSE
  - 7 – DEBUG

  By setting log_level, one can enable logging of all classes below or equal to the given level. Tarantool prints its logs to the standard error stream by default, but this can be changed with the log configuration parameter.

  Type: integer
  Default: 5
  Dynamic: yes

  Warning: prior to Tarantool 1.7.5 there were only six levels and DEBUG was level 6. Starting with Tarantool 1.7.5 VERBOSE is level 6 and DEBUG is level 7. VERBOSE is a new level for monitoring repetitive events which would cause too much log writing if INFO were used instead.

log
: By default, Tarantool sends the log to the standard error stream (stderr). If log is specified, Tarantool sends the log to a file, or to a pipe, or to the system logger.

Example setting for sending the log to a file:

```
box.cfg{log = 'tarantool.log'}
-- or
box.cfg{log = 'file:tarantool.log'}
```

This will open the file tarantool.log for output on the server's default directory. If the log string has no prefix or has the prefix "file:", then the string is interpreted as a file path.

Example setting for sending the log to a pipe:

```
box.cfg{log = '| cronolog tarantool.log'}
-- or
box.cfg{log = 'pipe: cronolog tarantool.log'}'
```

This will start the program cronolog when the server starts, and will send all log messages to the standard input (stdin) of cronolog. If the log string begins with '|' or has the prefix "pipe:", then the string is interpreted as a Unix pipeline.

Example setting for sending the log to syslog:

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

If the log string begins with "syslog:", then it is interpreted as a message for the syslogd program which normally is running in the background of any Unix-like platform. The setting can be 'syslog:', 'syslog:facility=...', 'syslog:identity=...', 'syslog:server=...', or a combination.

The syslog:identity setting is an arbitrary string which will be placed at the beginning of all messages. The default value is: tarantool.

The syslog:facility setting is currently ignored but will be used in the future. The value must be one of the syslog keywords, which tell syslogd where the message should go. The possible values are: auth, authpriv, cron, daemon, ftp, kern, lpr, mail, news, security, syslog, user, uucp, local0, local1, local2, local3, local4, local5, local6, local7. The default value is: user.

The syslog:server setting is the locator for the syslog server. It can be a Unix socket path beginning with "unix:", or an ipv4 port number. The default socket value is: dev/log (on Linux) or /var/run/syslog (on Mac OS). The default port value is: 514, the UDP port.

When logging to a file, Tarantool reopens the log on SIGHUP. When log is a program, its pid is saved in the log.logger_pid variable. You need to send it a signal to rotate logs.

Type: string
Default: null
Dynamic: no

log_nonblock

If log_nonblock equals true, Tarantool does not block during logging when the system is not ready for writing, and drops the message instead. If log_level is high, and many messages go to the log, setting log_nonblock to true may improve logging performance at the cost of some log messages getting lost.

This parameter has effect only if the output is going to "syslog:" or "pipe:". Setting log_nonblock to true is illegal if the output is going to a file.

The default log_nonblock value is nil, which means that blocking behavior corresponds to the type of logger. This is a behavior change: in earlier versions of the Tarantool server, the default value was true.

Type: boolean
Default: nil
Dynamic: no

too_long_threshold
: If processing a request takes longer than the given value (in seconds), warn about it in the log. Has effect only if log_level is more than or equal to 4 (WARNING).

Type: float
Default: 0.5
Dynamic: yes

log_format
: Log entries have two possible formats:

- 'plain' (the default), or
- 'json' (with more detail and with JSON labels).

Here is what a log entry looks like after box.cfg{log_format='plain'}:

```
2017-10-16 11:36:01.508 [18081] main/101/interactive I> set 'log_format' configuration option to "plain"
```

Here is what a log entry looks like after box.cfg{log_format='json'}:

```
{"time": "2017-10-16T11:36:17.996-0600",
"level": "INFO",
"message": "set 'log_format' configuration option to \"json\"",
"pid": 18081,|
"cord_name": "main",
"fiber_id": 101,
"fiber_name": "interactive",
"file": "builtin\/box\/load_cfg.lua",
"line": 317}
```

The log_format='plain' entry has time, process id, cord name, fiber_id, fiber_name, log level, and message.

The log_format='json' entry has the same things along with their labels, and in addition has the file name and line number of the Tarantool source.

Setting log_format to 'json' is illegal if the output is going to "syslog:".

Type: string
Default: 'plain'

Dynamic: yes

Logging example

This will illustrate how "rotation" works, that is, what happens when the server instance is writing to a log and signals are used when archiving it.

Start with two terminal shells, Terminal #1 and Terminal #2.

On Terminal #1: start an interactive Tarantool session, then say the logging will go to Log_file, then put a message "Log Line #1" in the log file:

```
box.cfg{log='Log_file'}
log = require('log')
log.info('Log Line #1')
```

On Terminal #2: use mv so the log file is now named Log_file.bak. The result of this is: the next log message will go to Log_file.bak.

```
mv Log_file Log_file.bak
```

On Terminal #1: put a message "Log Line #2" in the log file.

```
log.info('Log Line #2')
```

On Terminal #2: use ps to find the process ID of the Tarantool instance.

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

On Terminal #2: use kill -HUP to send a SIGHUP signal to the Tarantool instance. The result of this is: Tarantool will open Log_file again, and the next log message will go to Log_file. (The same effect could be accomplished by executing log.rotate() on the instance.)

kill -HUP process_id

On Terminal #1: put a message "Log Line #3" in the log file.

```
log.info('Log Line #3')
```

On Terminal #2: use less to examine files. Log_file.bak will have these lines, except that the date and time will depend on when the example is done:

```
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`
```

and Log_file will have

```
log file has been reopened
2015-11-30 15:15:32.629 [27469] main/101/interactive I> Log Line #3
```

Feedback

- feedback_enabled
- feedback_host
- feedback_interval

By default a Tarantool daemon sends a small packet once per hour, to https://feedback.tarantool.io. The packet contains three values from box.info: box.info.version, box.info.uuid, and box.info.cluster_uuid. By changing the feedback configuration parameters, users can adjust or turn off this feature.

feedback_enabled
: Whether to send feedback.

  If this is set to true, feedback will be sent as described above. If this is set to false, no feedback will be sent.

  Type: boolean
  Default: true
  Dynamic: yes

feedback_host
: The address to which the packet is sent. Usually the recipient is Tarantool, but it can be any URL.

  Type: string
  Default: 'https://feedback.tarantool.io'
  Dynamic: yes

feedback_interval
: The number of seconds between sendings, usually 3600 (1 hour).

  Type: float
  Default: 3600
  Dynamic: yes

Deprecated parameters

These parameters are deprecated since Tarantool version 1.7.4:

- coredump
- logger
- logger_nonblock
- panic_on_snap_error,
- panic_on_wal_error
- replication_source
- slab_alloc_arena
- slab_alloc_factor
- slab_alloc_maximal
- slab_alloc_minimal
- snap_dir

- snapshot_count
- snapshot_period

coredump
: Deprecated, do not use.

  Type: boolean
  Default: false
  Dynamic: no

logger
: Deprecated in favor of log. The parameter was only renamed, while the type, values and semantics remained intact.

logger_nonblock
: Deprecated in favor of log_nonblock. The parameter was only renamed, while the type, values and semantics remained intact.

panic_on_snap_error
: Deprecated in favor of force_recovery.

  If there is an error while reading a snapshot file (at server instance start), abort.

  Type: boolean
  Default: true
  Dynamic: no

panic_on_wal_error
: Deprecated in favor of force_recovery.

  Type: boolean
  Default: true
  Dynamic: yes

replication_source
: Deprecated in favor of replication. The parameter was only renamed, while the type, values and semantics remained intact.

slab_alloc_arena
: Deprecated in favor of memtx_memory.

  How much memory Tarantool allocates to actually store tuples, in gigabytes. When the limit is reached, INSERT or UPDATE requests begin failing with error ER_MEMORY_ISSUE. While the server does not go beyond the defined limit to allocate tuples, there is additional memory used to store indexes and connection information. Depending on actual configuration and workload, Tarantool can consume up to 20% more than the limit set here.

  Type: float

Default: 1.0
Dynamic: no

slab_alloc_factor
: Deprecated, do not use.

  The multiplier for computing the sizes of memory chunks that tuples are stored in. A lower value may result in less wasted memory depending on the total amount of memory available and the distribution of item sizes.

  Type: float
  Default: 1.1
  Dynamic: no

slab_alloc_maximal
: Deprecated in favor of memtx_max_tuple_size. The parameter was only renamed, while the type, values and semantics remained intact.

slab_alloc_minimal
: Deprecated in favor of memtx_min_tuple_size. The parameter was only renamed, while the type, values and semantics remained intact.

snap_dir
: Deprecated in favor of memtx_dir. The parameter was only renamed, while the type, values and semantics remained intact.

snapshot_period
: Deprecated in favor of checkpoint_interval. The parameter was only renamed, while the type, values and semantics remained intact.

snapshot_count
: Deprecated in favor of checkpoint_count. The parameter was only renamed, while the type, values and semantics remained intact.

4.5 Utility tarantoolctl

tarantoolctl is a utility for administering Tarantool instances, checkpoint files and modules. It is shipped and installed as part of Tarantool distribution.

See also tarantoolctl usage examples in Server administration section.

4.5.1 Command format

tarantoolctl COMMAND NAME [URI] [FILE] [OPTIONS..]

where:

- COMMAND is one of the following: start, stop, status, restart, logrotate, check, enter, eval, connect, cat, play, rocks.
- NAME is the name of an instance file or a module.
- FILE is the path to some file (.lua, .xlog or .snap).

- URI is the URI of some Tarantool instance.
- OPTIONS are options taken by some tarantoolctl commands.

4.5.2 Commands for managing Tarantool instances

tarantoolctl start NAME Start a Tarantool instance.

Additionally, this command sets the TARANTOOLCTL environment variable to 'true', to mark that the instance was started by tarantoolctl.

tarantoolctl stop NAME Stop a Tarantool instance.

tarantoolctl status NAME Show an instance's status (started/stopped). If pid file exists and an alive control socket exists, the return code is 0. Otherwise, the return code is not 0.

Reports typical problems to stderr (e.g. pid file exists and control socket doesn't).

tarantoolctl restart NAME Stop and start a Tarantool instance.

Additionally, this command sets the TARANTOOL_RESTARTED environment variable to 'true', to mark that the instance was restarted by tarantoolctl.

tarantoolctl logrotate NAME Rotate logs of a started Tarantool instance. Works only if logging-into-file is enabled in the instance file. Pipe/syslog make no effect.

tarantoolctl check NAME Check an instance file for syntax errors.

tarantoolctl enter NAME [--language=language] Enter an instance's interactive Lua or SQL console.

Supported option:

- --language=language to set interactive console language. Can be either Lua or SQL.

tarantoolctl eval NAME FILE Evaluate a local Lua file on a running Tarantool instance.

tarantoolctl connect URI Connect to a Tarantool instance on an admin-console port. Supports both TCP/Unix sockets.

4.5.3 Commands for managing checkpoint files

tarantoolctl cat FILE.. [--space=space_no ..] [--show-system] [--from=from_lsn] [--to=to_lsn] [--replica=replica_id ..]
Print into stdout the contents of .snap/.xlog files.

tarantoolctl play URI FILE.. [--space=space_no ..] [--show-system] [--from=from_lsn] [--to=to_lsn] [--replica=replica_id ..
Play the contents of .snap/.xlog files to another Tarantool instance.

Supported options:

- --space=space_no to filter the output by space number. May be passed more than once.
- --show-system to show the contents of system spaces.
- --from=from_lsn to show operations starting from the given lsn.
- --to=to_lsn to show operations ending with the given lsn.
- --replica=replica_id to filter the output by replica id. May be passed more than once.

4.5.4 Commands for managing Tarantool modules

tarantoolctl rocks install NAME Install a module in the current directory.

tarantoolctl rocks remove NAME Remove a module.

tarantoolctl rocks show NAME Show information about an installed module.

tarantoolctl rocks search NAME Search the repository for modules.

tarantoolctl rocks list List all installed modules.

tarantoolctl rocks pack {<rockspec> | <name> [<version>]} Create a rock by packing sources or binaries.

As an argument, you can specify:

- a .rockspec file to create a source rock containing the module's sources, or
- the name of an installed module (and its version if there are more than one) to create a binary rock containing the compiled module.

tarantoolctl rocks unpack {<rock_file> | <rockspec> | <name> [version]} Unpack the contents of a rock into a new directory under the current one.

As an argument, you can specify:

- source or binary rock files,
- .rockspec files, or
- names of rocks or .rockspec files in remote repositories (and the rock version if there are more than one).

Supported options:

- --server=server_name check this server first, then the usual list.
- --only-server=server_name check this server only, ignore the usual list.

4.6 Tips on Lua syntax

The Lua syntax for data-manipulation functions can vary. Here are examples of the variations with select() requests. The same rules exist for the other data-manipulation functions.

Every one of the examples does the same thing: select a tuple set from a space named 'tester' where the primary-key field value equals 1. For these examples, we assume that the numeric id of 'tester' is 512, which happens to be the case in our sandbox example only.

4.6.1 Object reference variations

First, there are three object reference variations:

```
-- #1 module . submodule . name
tarantool> box.space.tester:select{1}
-- #2 replace name with a literal in square brackets
tarantool> box.space['tester']:select{1}
-- #3 use a variable for the entire object reference
tarantool> s = box.space.tester
tarantool> s:select{1}
```

Examples in this manual usually have the "box.space.tester:" form (#1). However, this is a matter of user preference and all the variations exist in the wild.

Also, descriptions in this manual use the syntax "space_object:" for references to objects which are spaces, and "index_object:" for references to objects which are indexes (for example box.space.tester.index.primary:).

4.6.2 Parameter variations

Then, there are seven parameter variations:

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

Lua allows to omit parentheses () when invoking a function if its only argument is a Lua table, and we use it sometimes in our examples. This is why select{1} is equivalent to select({1}). Literal values such as 1 (a scalar value) or {1} (a Lua table value) may be replaced by variable names, as in examples #6 and #7.

Although there are special cases where braces can be omitted, they are preferable because they signal "Lua table". Examples and descriptions in this manual have the {1} form. However, this too is a matter of user preference and all the variations exist in the wild.

4.6.3 Rules for object names

Database objects have loose rules for names: the maximum length is 65000 bytes (not characters), and almost any legal Unicode character is allowed, including spaces, ideograms and punctuation.

In those cases, to prevent confusion with Lua operators and separators, object references should have the literal-in-square-brackets form (#2), or the variable form (#3). For example:

```
tarantool> box.space['1*A']:select{1}
tarantool> s = box.space['1*A !@$%^&*()_+12345678901234567890']
tarantool> s:select{1}
```

Disallowed:

- characters which are unassigned code points,
- line and paragraph separators,
- control characters,
- the replacement character (U+FFFD).

Not recommended: characters which cannot be displayed.

Names are “case sensitive”, so ‘A’ and ‘a’ are not the same.

CHAPTER 5

Tutorials

5.1 Lua tutorials

Here are three tutorials on using Lua stored procedures with Tarantool:

- Insert one million tuples with a Lua stored procedure,
- Sum a JSON field for all tuples,
- Indexed pattern search.

5.1.1 Insert one million tuples with a Lua stored procedure

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 for our "Getting started" exercises. So there is a single space, and a numeric primary key, and a running Tarantool server instance 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”.

The screen now looks like this:

```
tarantool> function string_funciton()
         >   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 instance! 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" .

The screen now looks like this:

```
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().

The screen now looks like this:

```
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()
```

(continues on next page)

(continued from previous page)

```
---
- 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".

The screen now looks like this:

```
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.

The screen now looks like this:

```
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, space_object:insert(), and space_object:replace().

The screen now looks like this:

```
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.tester:replace(t)
  end
end
start_time = os.clock()
main_function()
end_time = os.clock()
'insert done in ' .. end_time - start_time .. ' seconds'
```

The standard Lua function os.clock() 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.tester:replace(t)
  end
end
start_time = os.clock()
main_function()
end_time = os.clock()
'insert done in ' .. end_time - start_time .. ' seconds'
```

The screen now looks like this:

```
tarantool> function string_function()
         >   local random_number
         >   local random_string
         >   random_string = ""
         >   for x = 1,10,1 do
         >     random_number = math.random(65, 90)
         >     random_string = random_string .. string.char(random_number)
         >   end
         >   return random_string
         > end
---
...
tarantool> function main_function()
         >   local string_value, t
         >   for i = 1,1000000,1 do
         >     string_value = string_function()
         >     t = box.tuple.new({i,string_value})
         >     box.space.tester:replace(t)
         >   end
         > end
---
...
tarantool> start_time = os.clock()
---
...
tarantool> main_function()
---
...
tarantool> end_time = os.clock()
---
```

(continues on next page)

(continued from previous page)

```
...
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.

5.1.2 Sum a JSON field for all tuples

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.tester:pairs() do
6      is_valid_json, lua_table = pcall(json.decode, t[2])
7      if is_valid_json then
8        field_value = lua_table[field_name]
9        if type(field_value) == "number" then sum = sum + field_value end
10     end
11   end
12   return sum
13 end
```

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 in our "Getting started" exercises,

```
-- 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.

5.1.3 Indexed pattern search

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")
```

(continues on next page)

(continued from previous page)

```
  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
  -- 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 <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.

EXAMPLE:

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']
...
```

5.2 C tutorial

Here is one C tutorial: C stored procedures.

5.2.1 C stored procedures

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 five tasks:

(1) easy.c – prints "hello world";

(2) harder.c – decodes a passed parameter value;

(3) hardest.c – uses the C API to do a DBMS insert;

(4) read.c – uses the C API to do a DBMS select;

(5) write.c – uses the C API to do a DBMS replace.

After following the instructions, and seeing that the results are what is described here, users should feel confident about writing their own stored procedures.

Preparation

Check that these items exist on the computer:

- Tarantool 2.1
- A gcc compiler, any modern version should work
- module.h and files #included in it
- msgpuck.h
- libmsgpuck.a (only for some recent msgpuck versions)

The module.h file will exist if Tarantool 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 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
```

The libmsgpuck.a static library is necessary with msgpuck versions produced after February 2017. If and only if you encounter linking problems when using the gcc statements in the examples for this tutorial, you should put libmsgpuck.a on the path (libmsgpuck.a is produced from both msgpuck and Tarantool source downloads so it should be easy to find). For example, instead of "gcc -shared -o harder.so -fPIC harder.c" for the second example below, you will need to say "gcc -shared -o harder.so -fPIC harder.c libmsgpuck.a".

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;
}
int easy2(box_function_ctx_t *ctx, const char *args, const char *args_end)
{
  printf("hello world -- easy2\n");
  return 0;
}
```

Compile the program, producing a library file named easy.so:

```
$ gcc -shared -o easy.so -fPIC easy.c
```

Now go back to the client and execute these requests:

```
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(), 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.

The result should look like this:

```
tarantool> capi_connection:call('easy')
hello world
---
- []
...
```

Now let's call the other function in easy.c – easy2(). This is almost the same as the easy() function, but there's a detail: when the file name is not the same as the function name, then we have to specify file-name. function-name.

```
box.schema.func.create('easy.easy2', {language = 'C'})
box.schema.user.grant('guest', 'execute', 'function', 'easy.easy2')
capi_connection:call('easy.easy2')
```

. . . and this time the result will be "hello world – easy2".

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
```

Now go back to the client and execute these requests:

```
box.schema.func.create('harder', {language = 'C'})
box.schema.user.grant('guest', 'execute', 'function', 'harder')
passable_table = {}
table.insert(passable_table, 1)
```

(continues on next page)

(continued from previous page)

```
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. 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.

And now the screen looks like this:

```
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]; /* Must be big enough for mp_encode results */
  char *tuple_pointer = tuple;
  tuple_pointer = mp_encode_array(tuple_pointer, 2);
  tuple_pointer = mp_encode_uint(tuple_pointer, 10000);
  tuple_pointer = mp_encode_str(tuple_pointer, "String 2", 8);
  int n = box_insert(space_id, tuple, tuple_pointer, NULL);
  return n;
}
```

Compile the program, producing a library file named hardest.so:

```
$ gcc -shared -o hardest.so -fPIC hardest.c
```

Now go back to the client and execute these requests:

```
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 msgpuck.h functions;

(3) inserting a tuple using box_insert().

Warning: char tuple[1024]; is used here as just a quick way of saying "allocate more than enough bytes". For serious programs the developer must be careful to allow enough space for all the bytes that the mp_encode routines will use up.

Now, still on the client, execute this request:

```
box.space.capi_test:select()
```

The result should look like this:

```
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.

read.c

Go back to the shell where the easy.c and the harder.c and the hardest.c programs were created.

Create a file. Name it read.c. Put these 43 lines in it:

```
#include "module.h"
#include <msgpuck.h>
int read(box_function_ctx_t *ctx, const char *args, const char *args_end)
{
  char tuple_buf[1024];      /* where the raw MsgPack tuple will be stored */
  uint32_t space_id = box_space_id_by_name("capi_test", strlen("capi_test"));
  uint32_t index_id = 0;     /* The number of the space's first index */
  uint32_t key = 10000;      /* The key value that box_insert() used */
  mp_encode_array(tuple_buf, 0); /* clear */
  box_tuple_format_t *fmt = box_tuple_format_default();
  box_tuple_t *tuple = box_tuple_new(fmt, tuple_buf, tuple_buf+512);
  assert(tuple != NULL);
  char key_buf[16];          /* Pass key_buf = encoded key = 1000 */
  char *key_end = key_buf;
  key_end = mp_encode_array(key_end, 1);
  key_end = mp_encode_uint(key_end, key);
  assert(key_end < key_buf + sizeof(key_buf));
  /* Get the tuple. There's no box_select() but there's this. */
  int r = box_index_get(space_id, index_id, key_buf, key_end, &tuple);
```

(continues on next page)

(continued from previous page)

```
  assert(r == 0);
  assert(tuple != NULL);
  /* Get each field of the tuple + display what you get. */
  int field_no;             /* The first field number is 0. */
  for (field_no = 0; field_no < 2; ++field_no)
  {
    const char *field = box_tuple_field(tuple, field_no);
    assert(field != NULL);
    assert(mp_typeof(*field) == MP_STR || mp_typeof(*field) == MP_UINT);
    if (mp_typeof(*field) == MP_UINT)
    {
      uint32_t uint_value = mp_decode_uint(&field);
      printf("uint value=%u.\n", uint_value);
    }
    else /* if (mp_typeof(*field) == MP_STR) */
    {
      const char *str_value;
      uint32_t str_value_length;
      str_value = mp_decode_str(&field, &str_value_length);
      printf("string value=%.*s.\n", str_value_length, str_value);
    }
  }
  return 0;
}
```

Compile the program, producing a library file named read.so:

```
$ gcc -shared -o read.so -fPIC read.c
```

Now go back to the client and execute these requests:

```
box.schema.func.create('read', {language = "C"})
box.schema.user.grant('guest', 'execute', 'function', 'read')
box.schema.user.grant('guest', 'read,write', 'space', 'capi_test')
capi_connection:call('read')
```

This time the C function is doing four things:

(1) once again, finding the numeric identifier of the capi_test space by calling box_space_id_by_name();

(2) formatting a search key = 10000 using more msgpuck.h functions;

(3) getting a tuple using box_index_get();

(4) going through the tuple's fields with box_tuple_get() and then decoding each field depending on its type. In this case, since what we are getting is the tuple that we inserted with hardest.c, we know in advance that the type is either MP_UINT or MP_STR; however, it's very common to have a case statement here with one option for each possible type.

The result of capi_connection:call('read') should look like this:

```
tarantool> capi_connection:call('read')
uint value=10000.
string value=String 2.
---
- []
...
```

This proves that the read() function succeeded. Once again the important functions that start with box – box_index_get() and box_tuple_field() – came from the C API.

write.c

Go back to the shell where the programs easy.c, harder.c, hardest.c and read.c were created.

Create a file. Name it write.c. Put these 24 lines in it:

```
#include "module.h"
#include <msgpuck.h>
int write(box_function_ctx_t *ctx, const char *args, const char *args_end)
{
  static const char *space = "capi_test";
  char tuple_buf[1024]; /* Must be big enough for mp_encode results */
  uint32_t space_id = box_space_id_by_name(space, strlen(space));
  if (space_id == BOX_ID_NIL) {
    return box_error_set(__FILE__, __LINE__, ER_PROC_C,
    "Can't find space %s", "capi_test");
  }
  char *tuple_end = tuple_buf;
  tuple_end = mp_encode_array(tuple_end, 2);
  tuple_end = mp_encode_uint(tuple_end, 1);
  tuple_end = mp_encode_uint(tuple_end, 22);
  box_txn_begin();
  if (box_replace(space_id, tuple_buf, tuple_end, NULL) != 0)
    return -1;
  box_txn_commit();
  fiber_sleep(0.001);
  struct tuple *tuple = box_tuple_new(box_tuple_format_default(),
                                      tuple_buf, tuple_end);
  return box_return_tuple(ctx, tuple);
}
```

Compile the program, producing a library file named write.so:

```
$ gcc -shared -o write.so -fPIC write.c
```

Now go back to the client and execute these requests:

```
box.schema.func.create('write', {language = "C"})
box.schema.user.grant('guest', 'execute', 'function', 'write')
box.schema.user.grant('guest', 'read,write', 'space', 'capi_test')
capi_connection:call('write')
```

This time the C function is doing six things:

(1) once again, finding the numeric identifier of the capi_test space by calling box_space_id_by_name();

(2) making a new tuple;

(3) starting a transaction;

(4) replacing a tuple in box.space.capi_test

(5) ending a transaction;

(6) the final line is a replacement for the loop in read.c – instead of getting each field and printing it, use the box_return_tuple(...) function to return the entire tuple to the caller and let the caller display it.

The result of capi_connection:call('write') should look like this:

```
tarantool> capi_connection:call('write')
---
- [[1, 22]]
...
```

This proves that the write() function succeeded. Once again the important functions that start with box – box_txn_begin(), box_txn_commit() and box_return_tuple() – came from the C API.

Conclusion: the long description of the whole 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.
- Get rid of the capi_test space with box.schema.capi_test:drop().
- 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.

5.3 SQL tutorial

This tutorial is a demonstration of the SQL feature introduced in Tarantool 2.x series. There are two ways to go through this tutorial:

- read what we say the results are and take our word for it, or
- copy and paste each section and see everything work with Tarantool 2.1.

You will encounter all the functionality that you'd encounter in an "SQL-101" course.

5.3.1 Starting up with a first table and SELECTs

Initialize

Requests will be done using Tarantool as a client. Start Tarantool and, optionally, enter the Tarantool configuration request, for example:

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

Before Tarantool 2.0 you needed to say box.cfg{...} prior to performing any database operations. Now you can start working with the database outright. Tarantool initiates the database module and applies default settings.

set

A feature of the client console program is that you can switch languages and specify the end-of-statement delimiter.

Here we say: default language is SQL and statements end with semicolons.

```
tarantool> \set language sql
tarantool> \set delimiter ;
```

CREATE, INSERT, UPDATE, SELECT

Start with simple SQL statements just to be sure they're there.

```
CREATE TABLE table1 (column1 INTEGER PRIMARY KEY, column2 VARCHAR(100));
INSERT INTO table1 VALUES (1, 'A');
UPDATE table1 SET column2 = 'B';
SELECT * FROM table1 WHERE column1 = 1;
```

The result of the SELECT statement will look like this:

```
tarantool> SELECT * FROM table1 WHERE column1 = 1;
---
- - [1, 'B']
...
```

Reality check: actually the result will include include initial fields called "metadata", the names and data types of each column. For all SELECT examples we show only the result rows without showing the metadata.

CREATE TABLE

Here is CREATE TABLE with more details:

- There are multiple columns, with different data types.
- There is a PRIMARY KEY (unique and not-null) for two of the columns.

```
CREATE TABLE table2 (column1 INTEGER,
                     column2 VARCHAR(100),
                     column3 SCALAR,
                     column4 FLOAT,
                     PRIMARY KEY (column1, column2));
```

The result will be: "rowcount: 1" (no error).

INSERT

Try to put 5 rows in the table:

- The INTEGER and FLOAT columns get numbers.
- The VARCHAR and SCALAR columns get strings (the SCALAR strings are expressed as hexadecimals).

```
INSERT INTO table2 VALUES (1, 'AB', X'4142', 5.5);
INSERT INTO table2 VALUES (1, 'CD', X'2020', 1E4);
INSERT INTO table2 VALUES (1, 'AB', X'A5', -5.5);
INSERT INTO table2 VALUES (2, 'AB', X'2020', 12.34567);
INSERT INTO table2 VALUES (-1000, '', X'', 0.0);
```

The result will be:

- The third INSERT will fail because of a primary-key violation (1, 'AB' is a duplication).
- The other four INSERT statements will succeed.

SELECT with ORDER BY clause

Retrieve the 4 rows in the table, in descending order by column2, then (where the column2 values are the same) in ascending order by column4.

"*" is short for "all columns".

```
SELECT * FROM table2 ORDER BY column2 DESC, column4 ASC;
```

The result will be:

```
- - [1, 'CD', '  ', 10000]
  - [1, 'AB', 'AB', 5.5]
  - [2, 'AB', '  ', 12.34567]
  - [-1000, '', '', 0]
```

SELECT with WHERE clauses

Retrieve some of what you inserted:

- The first statement uses the LIKE comparison operator which is asking for "first character must be 'A', the next characters can be anything."
- The second statement uses logical operators and parentheses, so the ANDed expressions must be true, or the ORed expression must be true. Notice the columns don't have to be indexed.

```
SELECT column1, column2, column1 * column4 FROM table2 WHERE column2
LIKE 'A%';
SELECT column1, column2, column3, column4 FROM table2
    WHERE (column1 < 2 AND column4 < 10)
    OR column3 = X'2020';
```

The results will be:

```
- - [1, 'AB', 5.5]
  - [2, 'AB', 24.69134]
```

and

```
- - [-1000, '', '', 0]
  - [1, 'AB', 'AB', 5.5]
  - [1, 'CD', '  ', 10000]
  - [2, 'AB', '  ', 12.34567]
```

SELECT with GROUP BY and aggregating

Retrieve with grouping.

The rows which have the same values for column2 are grouped and are aggregated – summed, counted, averaged – for column4.

```
SELECT column2, SUM(column4), COUNT(column4), AVG(column4)
FROM table2
GROUP BY column2;
```

The result will be:

```
- - ['', 0, 1, 0]
  - ['AB', 17.84567, 2, 8.922835]
  - ['CD', 10000, 1, 10000]
```

5.3.2 Complications and complex SELECTs

NULLs

Insert more rows, containing NULL values.

NULL is not the same as Lua nil; it commonly is used in SQL for unknown or not-applicable.

```
INSERT INTO table2 VALUES (1, NULL, X'4142', 5.5);
INSERT INTO table2 VALUES (0, '!!@', NULL, NULL);
INSERT INTO table2 VALUES (0, '!!!', X'00', NULL);
```

The result will be:

- The first INSERT will fail because NULL is not permitted for a column that was defined with a PRIMARY KEY clause.
- The other INSERT statements will succeed.

Indexes

Make a new index on column4.

There already is an index for the primary key. Indexes are useful for making queries faster. In this case, the index also acts as a constraint, because it prevents two rows from having the same values in column4. However, it is not an error that column4 has multiple occurrences of NULLs.

```
CREATE UNIQUE INDEX i ON table2 (column4);
```

The result will be: “rowcount: 1” (no error).

Create a subset table

Make a table which will have some of the columns of table2, and some of the rows of table2.

You can do this by combining INSERT with SELECT. Then select everything in the resultant subset table.

```
CREATE TABLE table3 (column1 INTEGER, column2 VARCHAR(100), PRIMARY KEY
(column2));
INSERT INTO table3 SELECT column1, column2 FROM table2 WHERE column1 <> 2;
SELECT * FROM table3;
```

The result will be:

```
- - [-1000, '']
  - [0, '!!!']
  - [0, '!!@']
  - [1, 'AB']
  - [1, 'CD']
```

SELECT with a subquery

A subquery is a query within a query.

Here we find all the rows in table2 whose (column1, column2) values are not in table3.

```
SELECT * FROM table2 WHERE (column1, column2) NOT IN (SELECT column1,
column2 FROM table3);
```

The result is, unsurprisingly, the single row which we deliberately excluded when we inserted the rows in the INSERT ... SELECT statement:

```
- - [2, 'AB', '  ', 12.34567]
```

SELECT with a join

A join is a combination of two tables. There is more than one way to do them in Tarantool: "Cartesian joins", "left outer joins", etc.

Here we're just showing the most typical case, where column values from one table match column values from another table.

```
SELECT * FROM table2, table3
   WHERE table2.column1 = table3.column1 AND table2.column2 = table3.column2
   ORDER BY table2.column4;
```

The result will be:

```
- - [0, '!!!', "\0", null, 0, '!!!']
  - [0, '!!@', null, null, 0, '!!@']
  - [-1000, '', '', 0, -1000, '']
  - [1, 'AB', 'AB', 5.5, 1, 'AB']
  - [1, 'CD', ' ', 10000, 1, 'CD']
```

5.3.3 Constraints affecting updates

CREATE TABLE, with a CHECK clause

First we make a table which includes a "constraint" that there must not be any rows containing 13 in column2. Then we try to insert such a row.

```
CREATE TABLE table4 (column1 INTEGER PRIMARY KEY, column2 INTEGER, CHECK
(column2 <> 13));
INSERT INTO table4 VALUES (12, 13);
```

Result: the insert fails, as it should, with the message "error: 'CHECK constraint failed: TABLE4'".

CREATE TABLE, with a FOREIGN KEY clause

First we make a table which includes a "constraint" that there must not be any rows containing values that do not appear in table2.

When we made table2, we specified that its "primary key" columns were (column1, column2).

```
CREATE TABLE table5 (column1 INTEGER, column2 VARCHAR(100),
   PRIMARY KEY (column1),
   FOREIGN KEY (column1, column2) REFERENCES table2 (column1, column2));
INSERT INTO table5 VALUES (2,'AB');
INSERT INTO table5 VALUES (3,'AB');
```

Result:

- The first INSERT statement succeeds because table3 contains a row with [2, 'AB', ' ', 12.34567].
- The second INSERT statement, correctly, fails with the message "error: FOREIGN KEY constraint failed".

UPDATE

Due to earlier INSERT statements, these values are in table2 column4: {0, NULL, NULL, 5.5, 10000, 12.34567}. We will add 5 to every one of them except the one with 0. (Adding 5 to NULL will result in NULL, as SQL arithmetic requires.) Then we'll use SELECT to see what happened to column4.

```
UPDATE table2 SET column4 = column4 + 5 WHERE column4 <> 0;
SELECT column4 FROM table2 ORDER BY column4;
```

The result is: {NULL, NULL, 0, 10.5, 17.34567, 10005}.

DELETE

Due to earlier INSERT statements, there are now 6 rows in table2:

```
- - [-1000, '', '', 0]
  - [0, '!!!', "\0", null]
  - [0, '!!@', null, null]
  - [1, 'AB', 'AB', 10.5]
  - [1, 'CD', '  ', 10005]
  - [2, 'AB', '  ', 17.34567]
```

We will try to delete the last and first of these rows.

```
DELETE FROM table2 WHERE column1 = 2;
DELETE FROM table2 WHERE column1 = -1000;
SELECT COUNT(column1) FROM table2;
```

The result will be:

- The first DELETE statement causes an error message because (remember?) there's a foreign-key constraint.
- The second DELETE statement succeeds.
- The SELECT statement shows that there are now only 5 rows remaining.

ALTER TABLE, with a FOREIGN KEY clause

Now we want to make another "constraint", that there must not be any rows in table1 containing values that do not appear in table5. We couldn't do this when we created table1 because at that time table5 did not exist. But we can add constraints to existing tables with the ALTER TABLE statement.

```
ALTER TABLE table1 ADD CONSTRAINT c
   FOREIGN KEY (column1) REFERENCES table5 (column1);
DELETE FROM table1;
ALTER TABLE table1 ADD CONSTRAINT c
   FOREIGN KEY (column1) REFERENCES table5 (column1);
```

Result: the ALTER TABLE statement fails the first time because there is a row in table1, and ADD CONSTRAINT requires that the table be empty. But after we delete that row, the ALTER TABLE statement succeeds the second time. Thus we have set up a chain of references, from table1 to table5 and from table5 to table2.

Triggers

The idea of a trigger is: if a change (INSERT or UPDATE or DELETE) happens, then a further action – perhaps another INSERT or UPDATE or DELETE – will happen.

There are many variants, the one we'll illustrate here is: just after doing an update in table3, do an update in table2. We will specify this as FOR EACH ROW, so (since there are 5 rows in table3) the trigger will be activated 5 times.

```
SELECT column4 FROM table2 WHERE column1 = 2;
CREATE TRIGGER tr AFTER UPDATE ON table3 FOR EACH ROW
BEGIN UPDATE table2 SET column4 = column4 + 1 WHERE column1 = 2; END;
UPDATE table3 SET column2 = column2;
SELECT column4 FROM table2 WHERE column1 = 2;
```

Result:

- The first SELECT shows that the original value of column4 in table2 where column1 = 2 was: 17.34567.
- The second SELECT returns:

```
- - [22.34567]
```

5.3.4 Operators and functions

String operations

You can manipulate string data (usually defined with CHAR or VARCHAR data types) in many ways.

We'll illustrate here:

- the || operator for concatenation and

- the SUBSTR function for extraction.

```
SELECT column2, column2 || column2, SUBSTR(column2, 2, 1) FROM table2;
```

The result will be:

```
- - ['!!!', '!!!!!!', '!']
  - ['!!@', '!!@!!@', '!']
  - ['AB', 'ABAB', 'B']
  - ['CD', 'CDCD', 'D']
  - ['AB', 'ABAB', 'B']
```

Number operations

You can also manipulate number data (usually defined with INTEGER or FLOAT data types) in many ways.

We'll illustrate here:

- the << operator for shift left and
- the % operator for modulo.

```
SELECT column1, column1 << 1, column1 << 2, column1 % 2 FROM table2;
```

The result will be:

```
- - [0, 0, 0, 0]
  - [0, 0, 0, 0]
  - [1, 2, 4, 1]
  - [1, 2, 4, 1]
  - [2, 4, 8, 0]
```

Ranges and limits

Tarantool can handle:

- integers anywhere in the 4-byte integer range,
- approximate-numerics anywhere in the 8-byte IEEE floating point range,
- any Unicode characters, with UTF-8 encoding and a choice of collations.

Here we will insert some such values in a new table, and see what happens when we select them, with arithmetic on a number column and ordering by a string column.

```
CREATE TABLE t6 (column1 INTEGER, column2 VARCHAR(10), column4 FLOAT,
PRIMARY KEY (column1));
INSERT INTO t6 VALUES (-1234567890, 'АБВГД', 123456.123456);
INSERT INTO t6 VALUES (+1234567890, 'GD', 1e30);
INSERT INTO t6 VALUES (10, 'FADEW?', 0.000001);
INSERT INTO t6 VALUES (5, 'ABCDEFG', NULL);
SELECT column1 + 1, column2, column4 * 2 FROM t6 ORDER BY column2;
```

The result is:

```
- - [6, 'ABCDEFG', null]
  - [11, 'FADEW?', 2e-06]
  - [1234567891, 'GD', 2e+30]
  - [-1234567889, 'АБВГД', 246912.246912]
```

Views

A view, or "viewed table", is virtual, that is, its rows aren't physically in the database, their values are calculated from other tables.

Here we'll create a view v3 based on table3, then we select from it.

```
CREATE VIEW v3 AS SELECT SUBSTR(column2,1,2), column4 FROM t6 WHERE
column4 >= 0;
SELECT * FROM v3;
```

The result is:

```
- - ['АБ', 123456.123456]
  - ['FA', 1e-06]
  - ['GD', 1e+30]
```

Common table expressions

By putting WITH + SELECT in front of a SELECT, we can make a sort of temporary view that lasts for the duration of the statement.

Here we'll select from the sort of temporary view.

```
WITH cte AS (
          SELECT SUBSTR(column2,1,2), column4 FROM t6 WHERE column4
          >= 0)
SELECT * FROM cte;
```

Result: the same as the result we got with CREATE VIEW earlier:

```
- - ['АБ', 123456.123456]
  - ['FA', 1e-06]
  - ['GD', 1e+30]
```

VALUES

Tarantool can handle statements like SELECT 55; (select without FROM) like some other popular DBMSs. But it also handles the more standard statement VALUES (expression [, expression ...]);.

Here we'll use both styles.

```
SELECT 55 * 55, 'The rain in Spain';
VALUES (55 * 55, 'The rain in Spain');
```

The result of either statement will be:

```
- - [3025, 'The rain in Spain']
```

Metadata

What database objects have we created? We can find out about:

- tables with SELECT * FROM "_space";
- indexes with SELECT * FROM "_index";
- triggers with SELECT * FROM "_trigger"; (These names will be familiar to old Tarantool users because we're actually selecting from NoSQL "system spaces".)

Here we will select from _space.

```
SELECT "id", "name", "owner", "engine" FROM "_space" WHERE "name"='TABLE3';
```

The result is (we know we will get a row because we created table3 earlier):

```
- - [517, 'table3', 1, 'memtx']
```

5.3.5 Calling from a host language to make a big table

box.execute()

Now we will change the settings so that the console accepts statements written in Lua instead of statements written in SQL. (More ways to switch languages will exist in Tarantool clients in our next version.)

This doesn't mean we have left the SQL world though, because we can invoke SQL statements using a Lua function: box.execute(string).

Here we'll switch languages, and ask to select again what's in table3. These statements must be entered separately.

```
tarantool> \set language lua
tarantool> box.execute([[SELECT * FROM table3;]]);
```

Showing both the statements and the results:

```
tarantool> \set language lua
---
...
tarantool> box.execute([[SELECT * FROM table3;]]);
---
- - [-1000, '']
  - [0, '!!!']
  - [0, '!!@']
  - [1, 'AB']
  - [1, 'CD']
...
```

Create a million-row table

We've illustrated a lot of SQL, but does it scale? To answer that, let's make a bigger table.

For this we are going to use Lua. We will not explain the Lua, because that's in the Lua section of the Tarantool manual. Just copy-and-paste these instructions and wait for about a minute.

```
box.execute("CREATE TABLE tester (s1 INT PRIMARY KEY, s2 VARCHAR(10))");

function string_function()
  local random_number
  local random_string
  random_string = ""
  for x = 1,10,1 do
    random_number = math.random(65, 90)
    random_string = random_string .. string.char(random_number)
  end
  return random_string
end;

function main_function()
  local string_value, t, sql_statement
  for i = 1,1000000,1 do
    string_value = string_function()
    sql_statement = "INSERT INTO tester VALUES (" .. i .. ",'" .. string_value .. "')"
    box.execute(sql_statement)
  end
end;
start_time = os.clock();
main_function();
end_time = os.clock();
'insert done in ' .. end_time - start_time .. ' seconds';
```

The result is: you now have a table with a million rows, with a message saying "insert done in 88.570578 seconds".

Select from a million-row table

Now that we have something a bit larger to play with, let's see how long it takes to SELECT.

The first query we'll do will automatically go via an index, because s1 is the primary key.

The second query we'll do will not go via an index, because for s2 we didn't say CREATE INDEX xxxx ON tester (s2);.

```
box.execute([[SELECT * FROM tester WHERE s1 = 73446;]]);
box.execute([[SELECT * FROM tester WHERE s2 LIKE 'QFML%';]]);
```

The result is:

- the first statement will finish instantaneously,
- the second statement will be noticeably slower but still a fraction of a second.

Cleanup and exit

We're done. We've shown that Tarantool 2.1 has a very reasonable subset of SQL, and it works.

The rest of these commands will simply destroy all the database objects that were created so that you can do the demonstration again. These statements must be entered separately.

```
tarantool> \set language sql
tarantool> DROP TABLE tester;
```

(continues on next page)

(continued from previous page)

```
tarantool> DROP TABLE table1;
tarantool> DROP VIEW v3;
tarantool> DROP TRIGGER tr;
tarantool> DROP TABLE table5;
tarantool> DROP TABLE table4;
tarantool> DROP TABLE table3;
tarantool> DROP TABLE table2;
tarantool> DROP TABLE t6;
tarantool> \set language lua
tarantool> os.exit();
```

5.4 libslave tutorial

libslave is a C++ library for reading data changes done by MysQL and, optionally, writing them to a Tarantool database. It works by acting as a replication slave. The MySQL server writes data-change information to a "binary log", and transfers the information to any client that says "I want to see the information starting with this file and this record, continuously". So, libslave is primarily good for making a Tarantool database replica (much faster than using a conventional MySQL slave server), and for keeping track of data changes so they can be searched.

We will not go into the many details here – the API documentation has them. We will only show an exercise: a minimal program that uses the library.

Note: Use a test machine. Do not use a production machine.

STEP 1: Make sure you have:

- a recent version of Linux (versions such as Ubuntu 14.04 will not do),
- a recent version of MySQL 5.6 or MySQL 5.7 server (MariaDB will not do),
- MySQL client development package. For example, on Ubuntu you can download it with this command:

```
$ sudo apt-get install mysql-client-core-5.7
```

STEP 2: Download libslave.

The recommended source is https://github.com/tarantool/libslave/. Downloads include the source code only.

```
$ sudo apt-get install libboost-all-dev
$ cd ~
$ git clone https://github.com/tarantool/libslave.git tarantool-libslave
$ cd tarantool-libslave
$ git submodule init
$ git submodule update
$ cmake .
$ make
```

If you see an error message mentioning the word "vector", edit field.h and add this line:

```
#include <vector>
```

STEP 3: Start the MySQL server. On the command line, add appropriate switches for doing replication. For example:

```
$ mysqld --log-bin=mysql-bin --server-id=1
```

STEP 4: For purposes of this exercise, we are assuming you have:

- a "root" user with password "root" with privileges,
- a "test" database with a table named "test",
- a binary log named "mysql-bin",
- a server with server id = 1.

The values are hard-coded in the program, though of course you can change the program – it's easy to see their settings.

STEP 5: Look at the program:

```
#include <unistd.h>
#include <iostream>
#include <sstream>
#include "Slave.h"
#include "DefaultExtState.h"

slave::Slave* sl = NULL;

void callback(const slave::RecordSet& event) {
    slave::Slave::binlog_pos_t sBinlogPos = sl->getLastBinlog();
    switch (event.type_event) {
    case slave::RecordSet::Update: std::cout << "UPDATE" << "\n"; break;
    case slave::RecordSet::Delete: std::cout << "DELETE" << "\n"; break;
    case slave::RecordSet::Write:  std::cout << "INSERT" << "\n"; break;
    default: break;
    }
}

bool isStopping()
{
    return 0;
}

int main(int argc, char** argv)
{
    slave::MasterInfo masterinfo;
    masterinfo.conn_options.mysql_host = "127.0.0.1";
    masterinfo.conn_options.mysql_port = 3306;
    masterinfo.conn_options.mysql_user = "root";
    masterinfo.conn_options.mysql_pass = "root";
    bool error = false;
    try {
        slave::DefaultExtState sDefExtState;
        slave::Slave slave(masterinfo, sDefExtState);
        sl = &slave;
        sDefExtState.setMasterLogNamePos("mysql-bin", 0);
        slave.setCallback("test", "test", callback);
        slave.init();
        slave.createDatabaseStructure();
        try {
```

(continues on next page)

(continued from previous page)

```
            slave.get_remote_binlog(isStopping);
        } catch (std::exception& ex) {
            std::cout << "Error reading: " << ex.what() << std::endl;
            error = true;
        }
    } catch (std::exception& ex) {
        std::cout << "Error initializing: " << ex.what() << std::endl;
        error = true;
    }
    return 0;
}
```

Everything unnecessary has been stripped so that you can see quickly how it works. At the start of main(), there are some settings used for connecting – host, port, user, password. Then there is an initialization call with the binary log file name = "mysql-bin". Pay particular attention to the setCallback statement, which passes database name = "test", table name = "test", and callback function address = callback. The program will be looping and invoking this callback function. See how, earlier in the program, the callback function prints "UPDATE" or "DELETE" or "INSERT" depending on what is passed to it.

STEP 5: Put the program in the tarantool-libslave directory and name it example.cpp.

Step 6: Compile and build:

```
$ g++ -I/tarantool-libslave/include example.cpp -o example libslave_a.a -ldl -lpthread
```

Note: Replace tarantool-libslave/include with the full directory name.

Notice that the name of the static library is libslave_a.a, not libslave.a.

Step 7: Run:

```
$ ./example
```

The result will be nothing – the program is looping, waiting for the MySQL server to write to the replication binary log.

Step 8: Start a MySQL client program – any client program will do. Enter these statements:

```
USE test
INSERT INTO test VALUES ('A');
INSERT INTO test VALUES ('B');
DELETE FROM test;
```

Watch what happens in example.cpp output – it displays:

```
INSERT
INSERT
DELETE
DELETE
```

This is row-based replication, so you see two DELETEs, because there are two rows.

What the exercise has shown is:

- the library can be built, and
- programs that use the library can access everything that the MySQL server dumps.

For the many details and examples of usage in the field, see:

- Our downloadable libslave version:
  https://github.com/tarantool/libslave
- The version it was forked from (with a different README):
  https://github.com/vozbu/libslave/wiki/API
- How to speed up your MySQL with replication to in-memory database article
- Replicating data from MySQL to Tarantool article (in Russian)
- Asynchronous replication uncensored article (in Russian)

CHAPTER 6

Release Notes

The Release Notes are summaries of significant changes introduced in Tarantool 2.1.2, 2.1.1, 2.0.4, 1.10.3, 1.10.2, 1.9.0, 1.8.1, 1.7.6, 1.7.5, 1.7.4, 1.7.3, 1.7.2, 1.7.1, 1.6.9, 1.6.8, and 1.6.6.

For smaller feature changes and bug fixes, see closed milestones at GitHub.

6.1 Version 2.x

Tarantool 2.x is backward compatible with Tarantool 1.10.x in binary data layout, client-server protocol and replication protocol. You can upgrade using the box.schema.upgrade() procedure.

Release 2.1.2

Release type: stable. Release date: 2019-04-05.

Announcement: https://github.com/tarantool/tarantool/releases/tag/2.1.2.

This is the first stable release in the 2.x series.

The goal of this release is to significantly extend SQL support and increase stability.

Functionality added or changed:

- (SQL) box.sql.execute() replaced with box.execute(). It now works just like netbox.execute(): returns result set metadata, row count, etc. E.g.:

```
box.execute("CREATE TABLE person(id INTEGER PRIMARY KEY, birth_year INT)")
---
- row_count: 1
...
box.execute("SELECT birth_year FROM person")
---
- metadata:
  - name: birth_year
    type: INTEGER
  rows:
```

(continues on next page)

(continued from previous page)

```
  - [1983]
  - [1984]
...
```

- (SQL) Type system was significantly refactored.
- (SQL) There are cases in SQL when it is possible to do Tarantool's update operation for UPDATE statement, instead of doing delete + insert. However, there are cases where SQL semantics is too complex. E.g.:

```
CREATE TABLE file (id INT PRIMARY KEY, checksum INT);
INSERT INTO stock VALUES (1, 3),(2, 4),(3,5);
CREATE UNIQUE INDEX i ON file (checksum);
SELECT * FROM file;
-- [1, 3], [2, 4], [3, 5]
UPDATE OR REPLACE file SET checksum = checksum + 1;
SELECT * FROM stock;
-- [1, 4], [3, 6]
```

  I.e. [1, 3] tuple is updated as [1, 4] and have replaced tuple [2, 4]. This logic is implemented by preventive tuple deletion from all corresponding indexes in SQL.

- (SQL) Now SQL's integer type is stored as integer in space's format. It was stored as scalar before, which made comarisons slow.
- (SQL) It is now possible to define a constraint within column definition. E.g.:

```
CREATE TABLE person (id INT PRIMARY KEY, age INT, CHECK (age > 10));
```

- (SQL) Syntax for the pragma pragma index_info is now unified with table_info. E.g. to get information on index age_index of table person you can write:

```
pragma index_info(person.age_index);
```

- (Server) It is now possible to index a field specified using JSON. E.g.:

```
person = box.schema.create_space("person")
name_idx = person:create_index('name', {parts = {{'[2]fname', 'str'}, {'[2]sname', 'str'}}})
person:insert({1, {fname='James', sname='Bond'}, {town='London', country='GB', organization=
↪'MI6'}})
```

- (Server) In case of out of space event, Tarantool is now allowed to delete backup WAL files not needed for recovery from the last checkpoint.
- (Server) Add support for tarantoolctl rocks pack / unpack subcommands. The subcommands are used to create / deploy binary rock distributions.
- (Server) string.rstrip and string.lstrip should accept symbols to strip. Add optional 'chars' parameter for specifying the unwanted characters. E.g.:

```
local chars = "#\0"
str = "##Hello world!#"
print(string.strip(str, chars)) -- "Hello world!"
```

- (Server) on_shutdown trigger added. It may be set in a way similar to space:on_replace triggers:

```
box.ctl.on_shutdown(new_trigger, old_trigger)
```

- (Server) on_schema_init trigger added. It may be set before the first call to box.cfg() and is fired during box.cfg() before user data recovery start. To set the trigger, say:

```
box.ctl.on_schema_init(new_trig, old_trig)
```

- (Server) A new option for the snapshot daemon, box.cfg.checkpoint_wal_threshold, allows to limit the maximum disk size of maintained WALs. Once the configured threshold is exceeded, the WAL thread notifies the checkpoint daemon that it's time to make a new checkpoint and delete old WAL files.
- (Server) New types of privileges – to create, alter and drop space – were introduced. In order to create, drop or alter space or index, you should have a corresponding privilege. E.g.:

```
box.schema.user.create("optimizer", { password = 'secret' })
box.schema.user.grant("optimizer", "alter", "space")
person = box.schema.space.create("person")
box.session.su("optimizer")
i = s:create_index("primary") -- success
s:insert{1} -- fail
s:select{} -- fail
s:drop() -- fail
```

  Notice the incompatible change: Tarantool 1.10 requires read/write/execute privileges on an object to allow create, drop or alter. These privileges are no longer sufficient in 2.1. To remedy the problem, Tarantool 2.1 automatically grants create/drop/alter privileges on an object if a user has read/write/execute privileges on it during schema upgrade. But old scripts may stop working if read/write/execute is granted after schema upgrade.

  Additionally, create/drop/alter privileges are already supported in 1.10, which also supports the old semantics of read/write/execute. You are encouraged to grant new privileges in 1.10 before upgrade and modify your scripts.

Release 2.1.1

Release type: beta. Release date: 2018-11-14.

Announcement: https://github.com/tarantool/tarantool/releases/tag/2.1.1.

This release resolves all major bugs since 2.0.4 alpha and extends Tarantool's SQL feature set.

Release 2.0.4

Release type: alpha. Release date: 2018-02-15.

Announcement: https://github.com/tarantool/tarantool/releases/tag/2.0.4.

This is a successor of the 1.8.x releases. It improves the overall stability of the SQL engine and has some new features.

Functionality added or changed:

- Added support for SQL collations by incorporating libICU character set and collation library.
- IPROTO interface was extended to support SQL queries.
- net.box subsystem was extended to support SQL queries.
- Enabled ANALYZE statement to produce correct results, necessary for efficient query plans.
- Enabled savepoints functionality. SAVEPOINT statement works w/o issues.
- Enabled ALTER TABLE ... RENAME statement.
- Improved rules for identifier names: now fully consistent with Lua frontend.

- Enabled support for triggers; trigger bodies now persist in Tarantool snapshots and survive server restart.
- Significant performance improvements.

6.2 Version 1.10

Release 1.10.3

Release type: stable (lts). Release date: 2019-04-01. Tag: 1-10-3.

Announcement: https://github.com/tarantool/tarantool/releases/tag/1.10.3.

Overview

1.10.3 is the next stable (lts) release in the 1.10 series. The label 'stable' means we have had systems running in production without known crashes, bad results or other showstopper bugs for quite a while now.

This release resolves 69 issues since 1.10.2. There may be bugs in less common areas, please feel free to file an issue at GitHub. Compatibility

Tarantool 1.10.x is backward compatible with Tarantool 1.9.x in binary data layout, client-server protocol and replication protocol. Please upgrade using the box.schema.upgrade() procedure to unlock all the new features of the 1.10.x series when migrating from 1.9 version.

Functionality added or changed

- (Engines) Randomize vinyl index compaction Issue 3944.
- (Engines) Throttle tx thread if compaction doesn't keep up with dumps Issue 3721.
- (Engines) Do not apply run_count_per_level to the last level Issue 3657.
- (Server) Report the number of active iproto connections Issue 3905.
- (Replication) Never keep a dead replica around if running out of disk space Issue 3397.
- (Replication) Report join progress to the replica log Issue 3165.
- (Lua) Expose snapshot status in box.info.gc() Issue 3935.
- (Lua) Show names of Lua functions in backtraces in fiber.info() Issue 3538.
- (Lua) Check if transaction opened Issue 3518.

Bugs fixed

- (Engines) Tarantool crashes if DML races with DDL Issue 3420.
- (Engines) Recovery error if DDL is aborted Issue 4066.
- (Engines) Tarantool could commit in the read-only mode Issue 4016.
- (Engines) Vinyl iterator crashes if used throughout DDL Issue 4000.
- (Engines) Vinyl doesn't exit until dump/compaction is complete Issue 3949.
- (Engines) After re-creating secondary index no data is visible Issue 3903.
- (Engines) box.info.memory().tx underflow Issue 3897.
- (Engines) Vinyl stalls on intensive random insertion Issue 3603.
- (Server) Newer version of libcurl explodes fiber stack Issue 3569.
- (Server) SIGHUP crashes tarantool Issue 4063.

- (Server) checkpoint_daemon.lua:49: bad argument #2 to 'format' Issue 4030.
- (Server) fiber:name() show only part of name Issue 4011.
- (Server) Second hot standby switch may fail Issue 3967.
- (Server) Updating box.cfg.readahead doesn't affect existing connections Issue 3958.
- (Server) fiber.join() blocks in 'suspended' if fiber has cancelled Issue 3948.
- (Server) Tarantool can be crashed by sending gibberish to a binary socket Issue 3900.
- (Server) Stored procedure to produce push-messages never breaks on client disconnect Issue 3859.
- (Server) Tarantool crashed in lj_vm_return Issue 3840.
- (Server) Fiber executing box.cfg() may process messages from iproto Issue 3779.
- (Server) Possible regression on nosqlbench Issue 3747.
- (Server) Assertion after improper index creation Issue 3744.
- (Server) Tarantool crashes on vshard startup (lj_gc_step) Issue 3725.
- (Server) Do not restart replication on box.cfg if the configuration didn't change Issue 3711.
- (Replication) Applier times out too fast when reading large tuples Issue 4042.
- (Replication) Vinyl replica join fails Issue 3968.
- (Replication) Error during replication Issue 3910.
- (Replication) Downstream status doesn't show up in replication.info unless the channel is broken Issue 3904.
- (Replication) replication fails: tx checksum mismatch Issue 3993.
- (Replication) Rebootstrap crashes if master has replica's rows Issue 3740.
- (Replication) After restart tuples revert back to their old state which was before replica sync Issue 3722.
- (Replication) Add vclock for safer hot standby switch Issue 3002.
- (Replication) Master row is skipped forever in case of wal write failure Issue 2283.
- (Lua) space:frommap():tomap() conversion fail Issue 4045.
- (Lua) Non-informative message when trying to read a negative count of bytes from socket Issue 3979.
- (Lua) space:frommap raise "tuple field does not match..." even for nullable field Issue 3883.
- (Lua) Tarantool crashes on net.box.call after some uptime with vshard internal fiber Issue 3751.
- (Lua) Heap use after free in lbox_error Issue 1955.
- (Misc) http.client doesn't honour 'connection: keep-alive' Issue 3955.
- (Misc) net.box wait_connected is broken Issue 3856.
- (Misc) Mac build fails on Mojave Issue 3797.
- (Misc) FreeBSD build error: no SSL support Issue 3750.
- (Misc) 'http.client' sets invalid (?) reason Issue 3681.
- (Misc) Http client silently modifies headers when value is not a "string" or a "number" Issue 3679.
- (Misc) yaml.encode uses multiline format for 'false' and 'true' Issue 3662.
- (Misc) yaml.encode encodes 'null' incorrectly Issue 3583.

- (Misc) Error object message is empty Issue 3604.
- (Misc) Log can be flooded by warning messages Issue 2218.

Deprecations: the console=true option for net.box.new is deprecated.

Release 1.10.2

Release type: stable (lts). Release date: 2018-10-13. Tag: 1-10-2.

Announcement: https://github.com/tarantool/tarantool/releases/tag/1.10.2.

This is the first stable (lts) release in the 1.10 series. Also, Tarantool 1.10.2 is a major release that deprecates Tarantool 1.9.2. It resolves 95 issues since 1.9.2.

Tarantool 1.10.x is backward compatible with Tarantool 1.9.x in binary data layout, client-server protocol and replication protocol. You can upgrade using the box.schema.upgrade() procedure.

The goal of this release is to significantly increase vinyl stability and introduce automatic rebootstrap of a Tarantool replica set.

Functionality added or changed:

- (Engines) support ALTER for non-empty vinyl spaces. Issue 1653.
- (Engines) tuples stored in the vinyl cache are not shared among the indexes of the same space. Issue 3478.
- (Engines) keep a stack of UPSERTS in vy_read_iterator. Issue 1833.
- (Engines) box.ctl.reset_stat(), a function to reset vinyl statistics. Issue 3198.
- (Server) configurable syslog destination. Issue 3487.
- (Server) allow different nullability in indexes and format. Issue 3430.
- (Server) allow to back up any checkpoint, not just the last one. Issue 3410.
- (Server) a way to detect that a Tarantool process was started / restarted by tarantoolctl (TARANTOOLCTL and TARANTOOL_RESTARTED env vars). Issues 3384, 3215.
- (Server) net_msg_max configuration parameter to restrict the number of allocated fibers. Issue 3320.
- (Replication) display the connection status if the downstream gets disconnected from the upstream (box.info.replication.downstream.status = disconnected). Issue 3365.
- (Replication) replica-local spaces Issue 3443.
- (Replication) replication_skip_conflict, a new option in box.cfg{} to skip conflicting rows in replication. Issue 3270.
- (Replication) remove old snapshots which are not needed by replicas. Issue 3444.
- (Replication) log records which tried to commit twice. Issue 3105.
- (Lua) new function fiber.join(). Issue 1397.
- (Lua) new option names_only to tuple:tomap(). Issue 3280.
- (Lua) support custom rock servers (server and only-server options for tarantoolctl rocks command). Issue 2640.
- (Lua) expose on_commit/on_rollback triggers to Lua; Issue 857.
- (Lua) new function box.is_in_txn() to check if a transaction is open; Issue 3518.
- (Lua) tuple field access via a json path (by number, name, and path); Issue 1285.
- (Lua) new function space:frommap(); Issue 3282.

- (Lua) new module utf8 that implements libicu's bindings for use in Lua; Issues 3290, 3385.

6.3 Version 1.9

Release 1.9.0

Release type: stable. Release date: 2018-02-26. Tag: 1.9.0-4-g195d446.

Announcement: https://github.com/tarantool/tarantool/releases/tag/1.9.0.

This is the successor of the 1.7.6 stable release. The goal of this release is increased maturity of vinyl and master-master replication, and it contributes a number of features to this cause. Please follow the download instructions at https://tarantool.io/en/download/download.html to download and install a package for your operating system.

Functionality added or changed:

- (Security) it is now possible to block/unblock users. Issue 2898.
- (Security) new function box.session.euid() to return effective user. Effective user can be different from authenticated user in case of setuid functions or box.session.su. Issue 2994.
- (Security) new super role, with superuser access. Grant 'super' to guest to disable access control. Issue 3022.
- (Security) on_auth trigger is now fired in case of both successful and failed authentication. Issue 3039.
- (Replication/recovery) new replication configuration algorithm: if replication doesn't connect to replication_quorum peers in replication_connect_timeout seconds, the server start continues but the server enters the new orphan status, which is basically read-only, until the replicas connect to each other. Issues 3151 and 2958.
- (Replication/recovery) after replication connect at startup, the server does not start processing write requests before syncing up syncing up with all connected peers.
- (Replication/recovery) it is now possible to explicitly set instance_uuid and replica set uuid as configuration parameters. Issue 2967.
- (Replication/recovery) box.once() no longer fails on a read-only replica but waits. Issue 2537.
- (Replication/recovery) force_recovery can now skip a corrupted xlog file. Issue 3076.
- (Replication/recovery) improved replication monitoring: box.info.replication shows peer ip:port and correct replication lag even for idle peers. Issues 2753 and 2689.
- (Application server) new before triggers which can be used for conflict resolution in master-master replication. Issue 2993.
- (Application server) http client now correctly parses cookies and supports http+unix:// paths. Issues 3040 and 2801.
- (Application server) fio rock now supports file_exists(), rename() works across filesystems, and read() without arguments reads the whole file. Issues 2924, 2751 and 2925.
- (Application server) fio rock errors now follow Tarantool function call conventions and always return an error message in addition to the error flag.
- (Application server) digest rock now supports pbkdf2 password hashing algorithm, useful in PCI/DSS compliant applications. Issue 2874.
- (Application server) box.info.memory() provides a high-level overview of server memory usage, including networking, Lua, transaction and index memory. Issue 934.

- (Database) it is now possible to add missing tuple fields to an index, which is very useful when adding an index along with the evolution of the database schema. Issue 2988.
- (Database) lots of improvements in field type support when creating or altering spaces and indexes. Issues 2893, 3011 and 3008.
- (Database) it is now possible to turn on is_nullable property on a field even if the space is not empty, the change is instantaneous. Issue 2973.
- (Database) logging has been improved in many respects: individual messages (issues 1972, 2743, 2900), more logging in cases when it was useful (issues 3096, 2871).
- (Vinyl storage engine) it is now possible to make a unique vinyl index non-unique without index rebuild. Issue 2449.
- (Vinyl storage engine) improved UPDATE, REPLACE and recovery performance in presence of secondary keys. Issues 2289, 2875 and 3154.
- (Vinyl storage engine) space:len() and space:bsize() now work for vinyl (although they are still not exact). Issue 3056.
- (Vinyl storage engine) recovery speed has improved in presence of secondary keys. Issue 2099.
- (Builds) Alpine Linux support. Issue 3067.

6.4 Version 1.8

Release 1.8.1

Release type: alpha. Release date: 2017-05-17. Tag: 1.8.1.

Announcement: https://groups.google.com/forum/#!msg/tarantool-ru/XYaoqJpc544/mSvKrYwNAgAJ.

This is an alpha release which delivers support for a substantial subset of the ISO/IEC 9075:2011 SQL standard, including joins, subqueries and views. SQL is a major feature of the 1.8 release series, in which we plan to add support for ODBC and JDBC connectors, SQL triggers, prepared statements, security and roles, and generally ensure SQL is a first class query language in Tarantool.

Functionality added or changed:

- A new function box.sql.execute() (later changed to box.execute in Tarantool 2.1) was added to query Tarantool databases using SQL statements, e.g.:

```
tarantool> box.sql.execute([[SELECT * FROM _schema]]);
```

- SQL and Lua are fully interoperable.
- New meta-commands introduced to Tarantool's console.

  You can now set input language to either SQL or Lua, e.g.:

```
tarantool> \set language sql
tarantool> SELECT * FROM _schema;
tarantool> \set language lua
tarantool> print("Hello, world!")
```

- Most SQL statements are supported:
  - CREATE/DROP TABLE/INDEX/VIEW

```
tarantool> CREATE TABLE table1 (column1 INTEGER PRIMARY KEY, column2␣
↪VARCHAR(100));
```

– INSERT/UPDATE/DELETE statements e.g.:

```
tarantool> INSERT INTO table1 VALUES (1, 'A');
...
tarantool> UPDATE table1 SET column2 = 'B';
```

– SELECT statements, including complex complicated variants which include multiple JOINs, nested SELECTs etc. e.g.:

```
tarantool> SELECT sum(column1) FROM table1 WHERE column2 LIKE '_B' GROUP BY␣
↪column2;
```

– WITH statements e.g.

```
tarantool> WITH cte AS ( SELECT SUBSTR(column2,1,2), column1 FROM table1 WHERE␣
↪column1 >= 0) SELECT * FROM cte;
```

– SQL schema is persistent, so it is able to survive snapshot()/restore() sequence.

– SQL features are described in a tutorial.

6.5 Version 1.7

Release 1.7.6

Release type: stable. Release date: 2017-11-07. Tag: 1.7.6-0-g7b2945d6c.

Announcement: https://groups.google.com/forum/#!topic/tarantool/hzc7O2YDZUc.

This is the next stable release in the 1.7 series. It resolves more than 75 issues since 1.7.5.

What's new in Tarantool 1.7.6?

- In addition to rollback of a transaction, there is now rollback to a defined point within a transaction – savepoint support.
- There is a new object type: sequences. The older option, auto-increment, will be deprecated.
- String indexes can have collations.

New options are available for:

- net_box (timeouts),
- string functions,
- space formats (user-defined field names and types),
- base64 (urlsafe option), and
- index creation (collation, is-nullable, field names).

Incompatible changes:

- Layout of box.space._index has been extended to support is_nullable and collation features. All new indexes created on columns with is_nullable or collation properties will have the new definition format. Please update your client libraries if you plan to use these new features. Issue 2802

- fiber_name() now raises an exception instead of truncating long fiber names. We found that some Lua modules such as expiration‌d use fiber.name() as a key to identify background tasks. If a name is truncated, this fact was silently missed. The new behavior allows to detect bugs caused by fiber.name() truncation. Please use fiber.name(name, { truncate = true }) to emulate the old behavior. Issue 2622
- space:format() is now validated on DML operations. Previously space:format() was only used by client libraries, but starting from Tarantoool 1.7.6, field types in space:format() are validated on the server side on every DML operation, and field names can be used in indexes and Lua code. If you used space:format() in a non-standard way, please update layout and type names according to the official documentation for space formats.

Functionality added or changed:

- Hybrid schema-less + schemaful data model. Earlier Tarantool versions allowed to store arbitrary MessagePack documents in spaces. Starting from Tarantool 1.7.6, you can use space:format() to define schema restrictions and constraints for tuples in spaces. Defined field types are automatically validated on every DML operation, and defined field names can be used instead of field numbers in Lua code. A new function tuple:tomap() was added to convert a tuple into a key-value Lua dictionary.
- Collation and Unicode support. By default, when Tarantool compares strings, it takes into consideration only the numeric value of each byte in the string. To allow the ordering that you see in phone books and dictionaries, Tarantool 1.7.6 introduces support for collations based on the Default Unicode Collation Element Table (DUCET) and the rules described in Unicode® Technical Standard #10 Unicode Collation Algorithm (UTS #10 UCA) See collations.
- NULL values in unique and non-unique indexes. By default, all fields in Tarantool are "NOT NULL". Starting from Tarantool 1.7.6, you can use is_nullable option in space:format() or inside an index part definition to allow storing NULL in indexes. Tarantool partially implements three-valued logic from the SQL standard and allows storing multiple NULL values in unique indexes. Issue 1557.
- Sequences and a new implementation of auto_increment(). Tarantool 1.7.6 introduces new sequence number generators (like CREATE SEQUENCE in SQL). This feature is used to implement new persistent auto increment in spaces. Issue 389.
- Vinyl: introduced gap locks in Vinyl transaction manager. The new locking mechanism in Vinyl TX manager reduces the number of conflicts in transactions. Issue 2671.
- net.box: on_connect and on_disconnect triggers. Issue 2858.
- Structured logging in JSON format. Issue 2795.
- (Lua) Lua: string.strip() Issue 2785.
- (Lua) added base64_urlsafe_encode() to digest module. Issue 2777.
- Log conflicted keys in master-master replication. Issue 2779.
- Allow to disable backtrace in fiber.info(). Issue 2878.
- Implemented tarantoolctl rocks make *.spec. Issue 2846.
- Extended the default loader to look for .rocks in the parent dir hierarchy. Issue 2676.
- SOL_TCP options support in socket:setsockopt(). Issue 598.
- Partial emulation of LuaSocket on top of Tarantool Socket. Issue 2727.

Developer tools:

- Integration with IntelliJ IDEA with debugging support. Now you can use IntelliJ IDEA as an IDE to develop and debug Lua applications for Tarantool. See Using IDE.
- Integration with MobDebug remote Lua debugger. Issue 2728.

- Configured /usr/bin/tarantool as an alternative Lua interpreter on Debian/Ubuntu. Issue 2730.

New rocks:

- smtp.client - support SMTP via libcurl.

Release 1.7.5

Release type: stable. Release date: 2017-08-22. Tag: 1.7.5.

Announcement: https://github.com/tarantool/doc/issues/289.

This is a stable release in the 1.7 series. This release resolves more than 160 issues since 1.7.4.

Functionality added or changed:

- (Vinyl) a new force_recovery mode to recover broken disk files. Use box.cfg{force_recovery=true} to recover corrupted data files after hardware issues or power outages. Issue 2253.
- (Vinyl) index options can be changed on the fly without rebuild. Now page_size, run_size_ratio, run_count_per_level and bloom_fpr index options can be dynamically changed via index:alter(). The changes take effect in newly created data files only. Issue 2109.
- (Vinyl) improve box.info.vinyl() and index:info() output. Issue 1662.
- (Vinyl) introduce box.cfg.vinyl_timeout option to control quota throttling. Issue 2014.
- Memtx: stable index:pairs() iterators for the TREE index. TREE iterators are automatically restored to a proper position after index's modifications. Issue 1796.
- (Memtx) predictable order for non-unique TREE indexes. Non-unique TREE indexes preserve the sort order for duplicate entries. Issue 2476.
- (Memtx+Vinyl) dynamic configuration of max tuple size. Now box.cfg.memtx_max_tuple_size and box.cfg.vinyl_max_tuple_size configuration options can be changed on the fly without need to restart the server. Issue 2667.
- (Memtx+Vinyl) new implementation. Space truncation doesn't cause re-creation of all indexes any more. Issue 618.
- Extended the maximal length of all identifiers from 32 to 65k characters. Space, user and function names are not limited by 32 characters anymore. Issue 944.
- Heartbeat messages for replication. Replication client now sends the selective acknowledgments for processed records and automatically re-establish stalled connections. This feature also changes box.info.replication[replica_id].vclock. to display committed vclock of remote replica. Issue 2484.
- Keep track of remote replicas during WAL maintenance. Replication master now automatically preserves xlogs needed for remote replicas. Issue 748.
- Enabled box.tuple.new() to work without box.cfg(). Issue 2047.
- box.atomic(fun, ...) wrapper to execute function in a transaction. Issue 818.
- box.session.type() helper to determine session type. Issue 2642.
- Hot code reload for stored C stored procedures. Use box.schema.func.reload('modulename.function') to reload dynamic shared libraries on the fly. Issue 910.
- string.hex() and str:hex() Lua API. Issue 2522.
- Package manager based on LuaRocks. Use tarantoolctl rocks install MODULENAME to install MODULENAME Lua module from https://rocks.tarantool.org/. Issue 2067.
- Lua 5.1 command line options. Tarantool binary now supports '-i', '-e', '-m' and '-l' command line options. Issue 1265.

- Experimental GC64 mode for LuaJIT. GC64 mode allow to operate the full address space on 64-bit hosts. Enable via -DLUAJIT_ENABLE_GC64=ON compile-time configuration option. Issue 2643.
- Syslog logger now support non-blocking mode. box.cfg{log_nonblock=true} now also works for syslog logger. Issue 2466.
- Added a VERBOSE log level beyond INFO. Issue 2467.
- Tarantool now automatically makes snapshots every hour. Please set box.cfg{checkpoint_interval=0 to restore pre-1.7.5 behaviour. Issue 2496.
- Increase precision for percentage ratios provoded by box.slab.info(). Issue 2082.
- Stack traces now contain symbols names on all supported platforms. Previous versions of Tarantool didn't display meaningful function names in fiber.info() on non-x86 platforms. Issue 2103.
- Allowed to create fiber with custom stack size from C API. Issue 2438.
- Added ipc_cond to public C API. Issue 1451.

New rocks:

- http.client (built-in) - libcurl-based HTTP client with SSL/TLS support. Issue 2083.
- iconv (built-in) - bindings for iconv. Issue 2587.
- authman - API for user registration and login in your site using email and social networks.
- document - store nested documents in Tarantool.
- synchronized - critical sections for Lua.

Release 1.7.4

Release type: release candidate. Release date: 2017-05-12. Release tag: Tag: 1.7.4.

Announcement: https://github.com/tarantool/tarantool/releases/tag/1.7.4 or https://groups.google.com/forum/#!topic/tarantool/3x88ATX9YbY

This is a release candidate in the 1.7 series. Vinyl Engine, the flagship feature of 1.7.x, is now feature complete.

Incompatible changes

- box.cfg() options were changed to add Vinyl support:
  - snap_dir renamed to memtx_dir
  - slab_alloc_arena (gigabytes) renamed to memtx_memory (bytes), default value changed from 1Gb to 256MB
  - slab_alloc_minimal renamed to memtx_min_tuple_size
  - slab_alloc_maximal renamed to memtx_max_tuple_size
  - slab_alloc_factor is deprecated, not relevant in 1.7.x
  - snapshot_count renamed to checkpoint_count
  - snapshot_period renamed to checkpoint_interval
  - logger renamed to log
  - logger_nonblock renamed to log_nonblock
  - logger_level renamed to log_level
  - replication_source renamed to replication

  - panic_on_snap_error = true and panic_on_wal_error = true superseded by force_recovery = false

  Until Tarantool 1.8, you can use deprecated parameters for both initial and runtime configuration, but such usage will print a warning in the server log. Issues 1927 and 2042.

- Hot standy mode is now off by default. Tarantool automatically detects another running instance in the same wal_dir and refuses to start. Use box.cfg {hot_standby = true} to enable the hot standby mode. Issue 775.
- UPSERT via a secondary key was banned to avoid unclear semantics. Issue 2226.
- box.info and box.info.replication format was changed to display information about upstream and downstream connections (Issue 723):
  - Added box.info.replication[instance_id].downstream.vclock to display the last sent row to remote replica.
  - Added box.info.replication[instance_id].id.
  - Added box.info.replication[instance_id].lsn.
  - Moved box.info.replication[instance_id].{vclock,status,error} to box.info.replication[instance_id].upstream.{vclock,status,error}.
  - All registered replicas from box.space._cluster are included to box.info.replication output.
  - box.info.server.id renamed box.info.id
  - box.info.server.lsn renamed box.info.lsn
  - box.info.server.uuid renamed box.info.uuid
  - box.info.cluster.signature renamed to box.info.signature
  - box.info.id and box.info.lsn now return nil instead of -1 during initial cluster bootstrap.
- net.box: added per-request options to all requests:
  - conn.call(func_name, arg1, arg2,...) changed to conn.call(func_name, {arg1, arg2, ...}, opts)
  - conn.eval(func_name, arg1, arg2,...) changed to conn.eval(func_name, {arg1, arg2, ...}, opts)
- All requests now support timeout = <seconds>, buffer = <ibuf> options.
- Added connect_timeout option to netbox.connect().
- netbox:timeout() and conn:timeout() are now deprecated. Use netbox.connect(host, port, { call_16 = true }) for 1.6.x-compatible behavior. Issue 2195.
- systemd configuration changed to support Type=Notify / sd_notify(). Now systemctl start tarantool@INSTANCE will wait until Tarantool has started and recovered from xlogs. The recovery status is reported to systemctl status tarantool@INSTANCE. Issue 1923.
- log module now doesn't prefix all messages with the full path to tarantool binary when used without box.cfg(). Issue 1876.
- require('log').logger_pid() was renamed to require('log').pid(). Issue 2917.
- Removed Lua 5.0 compatible defines and functions (Issue 2396):
  - luaL_reg removed in favor of luaL_Reg
  - luaL_getn(L, i) removed in favor of lua_objlen(L, i)
  - luaL_setn(L, i, j) removed (was no-op)
  - lua_ref(L, lock) removed in favor of luaL_ref(L, lock)

  - lua_getref(L,ref) removed in favor of lua_rawgeti(L, LUA_REGISTRYINDEX, (ref))
  - lua_unref(L, ref) removed in favor of luaL_unref(L, ref)
  - math.mod() removed in favor of math.fmod()
  - string.gfind() removed in favor of string.gmatch()

Functionality added or changed:

- (Vinyl) multi-level compaction. The compaction scheduler now groups runs of the same range into levels to reduce the write amplification during compaction. This design allows Vinyl to support 1:100+ ram:disk use-cases. Issue 1821.
- (Vinyl) bloom filters for sorted runs. Bloom filter is a probabilistic data structure which can be used to test whether a requested key is present in a run file without reading the actual file from the disk. Bloom filter may have false-positive matches but false-negative matches are impossible. This feature reduces the number of seeks needed for random lookups and speeds up REPLACE/DELETE with enabled secondary keys. Issue 1919.
- (Vinyl) key-level cache for point lookups and range queries. Vinyl storage engine caches selected keys and key ranges instead of entire disk pages like in traditional databases. This approach is more efficient because the cache is not polluted with raw disk data. Issue 1692.
- (Vinyl) implemented the common memory level for in-memory indexes. Now all in-memory indexes of a space store pointers to the same tuples instead of cached secondary key index data. This feature significantly reduces the memory footprint in case of secondary keys. Issue 1908.
- (Vinyl) new implementation of initial state transfer of JOIN command in replication protocol. New replication protocol fixes problems with consistency and secondary keys. We implemented a special kind of low-cost database-wide read-view to avoid dirty reads in JOIN procedure. This trick wasn't not possible in traditional B-Tree based databases. Issue 2001.
- (Vinyl) index-wide mems/runs. Removed ranges from in-memory and and the stop layer of LSM tree on disk. Issue 2209.
- (Vinyl) coalesce small ranges. Before dumping or compacting a range, consider coalescing it with its neighbors. Issue 1735.
- (Vinyl) implemented transnational journal for metadata. Now information about all Vinyl files is logged in a special .vylog file. Issue 1967.
- (Vinyl) implemented consistent secondary keys. Issue 2410.
- (Memtx+Vinyl) implemented low-level Lua API to create consistent backups. of Memtx + Vinyl data. The new feature provides box.backup.start()/stop() functions to create backups of all spaces. box.backup.start() pauses the Tarantool garbage collector and returns the list of files to copy. These files then can be copied be any third-party tool, like cp, ln, tar, rsync, etc. box.backup.stop() lets the garbage collector continue. Created backups can be restored instantly by copying into a new directory and starting a new Tarantool instance. No special preparation, conversion or unpacking is needed. Issue 1916.
- (Vinyl) added statistics for background workers to box.info.vinyl(). Issue 2005.
- (Memtx+Vinyl) reduced the memory footprint for indexes which keys are sequential and start from the first field. This optimization was necessary for secondary keys in Vinyl, but we optimized Memtx as well. Issue 2046.
- LuaJIT was rebased on the latest 2.1.0b3 with out patches (Issue 2396):
  - Added JIT compiler backend for ARM64
  - Added JIT compiler backend and interpreter for MIPS64

  - Added some more Lua 5.2 and Lua 5.3 extensions
  - Fixed several bugs
  - Removed Lua 5.0 legacy (see incompatible changes above).
- Enabled a new smart string hashing algorithm in LuaJIT to avoid significant slowdown when a lot of collisions are generated. Contributed by Yury Sokolov (@funny-falcon) and Nick Zavaritsky (@mejedi). See https://github.com/tarantool/luajit/pull/2.
- box.snapshot() now updates mtime of a snapshot file if there were no changes to the database since the last snapshot. Issue 2045.
- Implemented space:bsize() to return the memory size utilized by all tuples of the space. Contributed by Roman Tokarev (@rtokarev). Issue 2043.
- Exported new Lua/C functions to public API:
  - luaT_pushtuple, luaT_istuple (issue 1878)
  - luaT_error, luaT_call, luaT_cpcall (issue 2291)
  - luaT_state (issue 2416)
- Exported new Box/C functions to public API: box_key_def, box_tuple_format, tuple_compare(), tuple_compare_with_key(). Issue 2225.
- xlogs now can be rotated based on size (wal_max_size) as well as the number of written rows (rows_per_wal). Issue 173.
- Added string.split(), string.startswith(), string.endswith(), string.ljust(), string.rjust(), string.center() API. Issues 2211, 2214, 2415.
- Added table.copy() and table.deepcopy() functions. Issue 2212.
- Added pwd module to work with UNIX users and groups. Issue 2213.
- Removed noisy "client unix/: connected" messages from logs. Use box.session.on_connect()/on_disconnect() triggers instead. Issue 1938.

  box.session.on_connect()/on_disconnect()/on_auth() triggers now also fired for admin console connections.
- tarantoolctl: eval, enter, connect commands now support UNIX pipes. Issue 672.
- tarantoolctl: improved error messages and added a new man page. Issue 1488.
- tarantoolctl: added filter by replica_id to cat and play commands. Issue 2301.
- tarantoolctl: start, stop and restart commands now redirect to systemctl start/stop/restart when systemd is enabled. Issue 2254.
- net.box: added buffer = <buffer> per-request option to store raw MessagePack responses into a C buffer. Issue 2195.
- net.box: added connect_timeout option. Issue 2054.
- net.box: added on_schema_reload() hook. Issue 2021.
- net.box: exposed conn.schema_version and space.connection to API. Issue 2412.
- log: debug()/info()/warn()/error() now doesn't fail on formatting errors. Issue 889.
- crypto: added HMAC support. Contributed by Andrey Kulikov (@amdei). Issue 725.

Release 1.7.3

Release type: beta. Release date: 2016-12-24. Release tag: Tag: 1.7.3-0-gf0c92aa.

Announcement: https://github.com/tarantool/tarantool/releases/tag/1.7.3

This is the second beta release in the 1.7 series.

Incompatible changes:

- Broken coredump() Lua function was removed. Use gdb -batch -ex "generate-core-file" -p $PID instead. Issue 1886.
- Vinyl disk layout was changed since 1.7.2 to add ZStandard compression and improve the performance of secondary keys. Use the replication mechanism to upgrade from 1.7.2 beta. Issue 1656.

Functionality added or changed:

- Substantial progress on stabilizing the Vinyl storage engine:
  - Fix most known crashes and bugs with bad results.
  - Switch to use XLOG/SNAP format for all data files.
  - Enable ZStandard compression for all data files.
  - Squash UPSERT operations on the fly and merge hot keys using a background fiber.
  - Significantly improve the performance of index:pairs() and index:count().
  - Remove unnecessary conflicts from transactions.
  - In-memory level was mostly replaced by memtx data structures.
  - Specialized allocators are used in most places.
- We're still actively working on Vinyl and plan to add multi-level compaction and improve the performance of secondary keys in 1.7.4. This implies a data format change.
- Support for DML requests for space:on_replace() triggers. Issue 587.
- UPSERT can be used with the empty list of operations. Issue 1854.
- Lua functions to manipulate environment variables. Issue 1718.
- Lua library to read Tarantool snapshots and xlogs. Issue 1782.
- New play and cat commands in tarantoolctl. Issue 1861.
- Improve support for the large number of active network clients. Issue#5#1892.
- Support for space:pairs(key, iterator-type) syntax. Issue 1875.
- Automatic cluster bootstrap now also works without authorization. Issue 1589.
- Replication retries to connect to master indefinitely. Issue 1511.
- Temporary spaces now work with box.cfg { read_only = true }. Issue 1378.
- The maximum length of space names increased to 64 bytes (was 32). Issue 2008.

Release 1.7.2

Release type: beta. Release date: 2016-09-29. Release tag: Tag: 1.7.2-1-g92ed6c4.

Announcement: https://groups.google.com/forum/#!topic/tarantool-ru/qUYUesEhRQg

This is a release in the 1.7 series.

Incompatible changes:

- A new binary protocol command for CALL, which no more restricts a function to returning an array of tuples and allows returning an arbitrary MsgPack/JSON result, including scalars, nil and void (nothing). The old CALL is left intact for backward compatibility. It will be removed in the next major release. All programming language drivers will be gradually changed to use the new CALL. Issue 1296.

Functionality added or changed:

- Vinyl storage engine finally reached the beta stage. This release fixes more than 90 bugs in Vinyl, in particular, removing unpredictable latency spikes, all known crashes and bad/lost result bugs.
  - new cooperative multitasking based architecture to eliminate latency spikes,
  - support for non-sequential multi-part keys,
  - support for secondary keys,
  - support for auto_increment(),
  - number, integer, scalar field types in indexes,
  - INSERT, REPLACE and UPDATE return new tuple, like in memtx.
- We're still actively working on Vinyl and plan to add zstd compression and a new memory allocator for Vinyl in-memory index in 1.7.3. This implies a data format change which we plan to implement before 1.7 becomes generally available.
- Tab-based autocompletion in the interactive console, require('console').connect(), tarantoolctl enter and tarantoolctl connect commands. Issues 86 and 1790. Use the TAB key to auto complete the names of Lua variables, functions and meta-methods.
- A new implementation of net.box improving performance and solving problems when the Lua garbage collector handles dead connections. Issues 799, 800, 1138 and 1750.
- memtx snapshots and xlog files are now compressed on the fly using the fast ZStandard compression algorithm. Compression options are configured automatically to get an optimal trade-off between CPU utilization and disk throughput.
- fiber.cond() - a new synchronization mechanism for cooperative multitasking. Issue 1731.
- Tarantool can now be installed using universal Snappy packages (http://snapcraft.io/) with snap install tarantool --channel=beta.

New rocks and packages:

- curl - non-blocking bindings for libcurl
- prometheus - Prometheus metric collector for Tarantool
- gis - a full-featured geospatial extension for Tarantool
- mqtt - an MQTT protocol client for Tarantool
- luaossl - the most comprehensive OpenSSL module in the Lua universe

Deprecated, removed features and minor incompatibilities:

- num and str fields type names are deprecated, use unsigned and string instead. Issue 1534.
- space:inc() and space:dec() were removed (deprecated in 1.6.x) Issue 1289.
- fiber:cancel() is now asynchronous and doesn't wait for the fiber to end. Issue 1732.
- Implicit error-prone tostring() was removed from digest API. Issue 1591.
- Support for SHA-0 (digest.sha()) was removed due to OpenSSL upgrade.

- net.box now uses one-based indexes for space.name.index[x].parts. Issue 1729.
- Tarantool binary now dynamically links with libssl.so during compile time instead of loading it at the run time.
- Debian and Ubuntu packages switched to use native systemd configuration alongside with old-fashioned sysvinit scripts.

  systemd provides its own facilities for multi-instance management. To upgrade, perform the following steps:

  1. Install new 1.7.2 packages.
  2. Ensure that INSTANCENAME.lua file is present in /etc/tarantool/instace.enabled.
  3. Stop INSTANCENAME using tarantoolctl stop INSTANCENAME.
  4. Start INSTANCENAME using systemctl start tarantool@INSTANCENAME.
  5. Enable INSTANCENAME during system boot using systemctl enable tarantool@INTANCENAME.
  6. Say systemctl disable tarantool; update-rc.d tarantool remove to disable sysvinit-compatible wrappers.

  Refer to issue 1291 comment and the administration chapter for additional information.
- Debian and Ubuntu packages start a ready-to-use example.lua instance on a clean installation of the package. The default instance grants universe permissions for guest user and listens on "locahost:3313".
- Fedora 22 packages were deprecated (EOL).

Release 1.7.1

Release type: alpha. Release date: 2016-07-11.

Announcement: https://groups.google.com/forum/#!topic/tarantool/KGYj3VKJKb8

This is the first alpha in the 1.7 series. The main feature of this release is a new storage engine, called "vinyl". Vinyl is a write optimized storage engine, allowing the amount of data stored exceed the amount of available RAM 10-100x times. Vinyl is a continuation of the Sophia engine from 1.6, and effectively a fork and a distant relative of Dmitry Simonenko's Sophia. Sophia is superseded and replaced by Vinyl. Internally it is organized as a log structured merge tree. However, it takes a serious effort to improve on the traditional deficiencies of log structured storage, such as poor read performance and unpredictable write latency. A single index is range partitioned among many LSM data structures, each having its own in-memory buffers of adjustable size. Range partitioning allows merges of LSM levels to be more granular, as well as to prioritize hot ranges over cold ones in access to resources, such as RAM and I/O. The merge scheduler is designed to minimize write latency while ensuring read performance stays within acceptable limits. Vinyl today only supports a primary key index. The index can consist of up to 256 parts, like in MemTX, up from 8 in Sophia. Partial key reads are supported. Support of non-sequential multi part keys, as well as secondary keys is on the short term todo. Our intent is to remove all limitations currently present in Vinyl, making it a first class citizen in Tarantool.

Functionality added or changed:

- 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 replica set bootstrap (for easier configuration of a new replica set) is supported.
- The space_object:inc() function is removed.

- The space_object:dec() function is removed.
- The space_object:bsize() function is added.
- The box.coredump() function is removed, for an alternative see Core dumps.
- The hot_standby configuration option is added.
- Configuration parameters revised or renamed:
  - slab_alloc_arena (in gigabytes) to memtx_memory (in bytes),
  - slab_alloc_minimal to memtx_min_tuple_size,
  - slab_alloc_maximal to memtx_max_tuple_size,
  - replication_source to replication,
  - snap_dir to memtx_dir,
  - logger to log,
  - logger_nonblock to log_nonblock,
  - snapshot_count to checkpoint_count,
  - snapshot_period to checkpoint_interval,
  - panic_on_wal_error and panic_on_snap_error united under force_recovery.
- Until Tarantool 1.8, you can use deprecated parameters for both initial and runtime configuration, but Tarantool will display a warning. Also, you can specify both deprecated and up-to-date parameters, provided that their values are harmonized. If not, Tarantool will display an error.
- Automatic replication cluster bootstrap; it's now much easier to configure a new replication cluster.
- New indexable data types: INTEGER and SCALAR.
- Code refactoring and performance improvements.
- Updated LuaJIT to 2.1-beta116.

6.6 Version 1.6

Release 1.6.9

Release type: maintenance. Release date: 2016-09-27. Release tag: 1.6.9-4-gcc9ddd7.

Since February 15, 2017, due to Tarantool issue#2040 Remove sophia engine from 1.6 there no longer is a storage engine named sophia. It will be superseded in version 1.7 by the vinyl storage engine.

Incompatible changes:

- Support for SHA-0 (digest.sha()) was removed due to OpenSSL upgrade.
- Tarantool binary now dynamically links with libssl.so during compile time instead of loading it at the run time.
- Fedora 22 packages were deprecated (EOL).

Functionality added or changed:

- Tab-based autocompletion in the interactive console. Issue 86
- LUA_PATH and LUA_CPATH environment variables taken into account, like in PUC-RIO Lua. Issue 1428

- Search for .dylib as well as for .so libraries in OS X. Issue 810.
- A new box.cfg { read_only = true } option to emulate master-slave behavior. Issue 246
- if_not_exists = true option added to box.schema.user.grant. Issue 1683
- clock_realtime()/monotonic() functions added to the public C API. Issue 1455
- space:count(key, opts) introduced as an alias for space.index.primary:count(key, opts). Issue 1391
- Upgrade script for 1.6.4 -> 1.6.8 -> 1.6.9. Issue 1281
- Support for OpenSSL 1.1. Issue 1722

New rocks and packages:

- curl - non-blocking bindings for libcurl
- prometheus - Prometheus metric collector for Tarantool
- gis - full-featured geospatial extension for Tarantool.
- mqtt - MQTT protocol client for Tarantool
- luaossl - the most comprehensive OpenSSL module in the Lua universe

Release 1.6.8

Release type: maintenance. Release date: 2016-02-25. Release tag: 1.6.8-525-ga571ac0.

Incompatible changes:

- RPM packages for CentOS 7 / RHEL 7 and Fedora 22+ now use native systemd configuration without legacy sysvinit shell scripts. Systemd provides its own facilities for multi-instance management. To upgrade, perform the following steps:

  1. Ensure that INSTANCENAME.lua file is present in /etc/tarantool/instace.available.
  2. Stop INSTANCENAME using tarantoolctl stop INSTANCENAME.
  3. Start INSTANCENAME using systemctl start tarantool@INSTANCENAME.
  4. Enable INSTANCENAME during system boot using systemctl enable tarantool@INTANCENAME.

  /etc/tarantool/instance.enabled directory is now deprecated for systemd-enabled platforms.

  See the administration chapter for additional information.
- Sophia was upgraded to v2.1 to fix upsert, memory corruption and other bugs. Sophia v2.1 doesn't support old v1.1 data format. Please use Tarantool replication to upgrade. Issue 1222
- Ubuntu Vivid, Fedora 20, Fedora 21 were deprecated due to EOL.
- i686 packages were deprecated. Please use our RPM and DEB specs to build these on your own infrastructure.
- Please update your yum.repos.d and/or apt sources.list.d according to instructions at http://tarantool.org/download.html

Functionality added or changed:

- Tarantool 1.6.8 fully supports ARMv7 and ARMv8 (aarch64) processors. Now it is possible to use Tarantool on a wide range of consumer devices, starting from popular Raspberry PI 2 to coin-size embedded boards and no-name mini-micro-nano-PCs. Issue 1153. (Also qemu works well, but we don't have real hardware to check.)

- Tuple comparator functions were optimized, providing up to 30% performance boost when an index key consists of 2, 3 or more parts. Issue 969.
- Tuple allocator changes give another 15% performance improvement. Issue 1298
- Replication relay performance was improved by reducing the amount of data directory re-scans. Issue 11150
- A random delay was introduced into snapshot daemon, reducing the chance that multiple instances take a snapshot at the same time. Issue 732.
- Sophia storage engine was upgraded to v2.1:
  - serializable Snapshot Isolation (SSI),
  - RAM storage mode,
  - anti-cache storage mode,
  - persistent caching storage mode,
  - implemented AMQ Filter,
  - LRU mode,
  - separate compression for hot and cold data,
  - snapshot implementation for Faster Recovery,
  - upsert reorganizations and fixes,
  - new performance metrics.

  Please note "Incompatible changes" above.
- Allow to remove servers with non-zero LSN from _cluster space. Issue 1219.
- net.box now automatically reloads space and index definitions. Issue 1183.
- The maximal number of indexes in space was increased to 128. Issue 1311.
- New native systemd configuration with support of instance management and daemon supervision (CentOS 7 and Fedora 22+ only). Please note "Incompatible changes" above. Issue 1264.
- Tarantool package was accepted to the official Fedora repositories (https://apps.fedoraproject.org/packages/tarantool).
- Tarantool brew formula (OS X) was accepted to the official Homebrew repository (http://brewformulas.org/tarantool).
- Clang compiler support was added on FreeBSD. Issue 786.
- Support for musl libc, used by Alpine Linux and Docker images, was added. Issue 1249.
- Added support for GCC 6.0.
- Ubuntu Wily, Xenial and Fedora 22, 23 and 24 are now supported distributions for which we build official packages.
- box.info.cluster.uuid can be used to retrieve cluster UUID. Issue 1117.
- Numerous improvements in the documentation, added documentation for syslog, clock, fiber.storage packages, updated the built-in tutorial.

New rocks and packages:

- Tarantool switched to a new Docker-based cloud build infrastructure The new buildbot significantly decreases commit-to-package time. The official repositories at http://tarantool.org now contain the latest version of the server, rocks and connectors. See http://github.com/tarantool/build
- The repositories at http://tarantool.org/download.html were moved to http://packagecloud.io cloud hosting (backed by Amazon AWS). Thanks to packagecloud.io for their support of open source!
- memcached - memcached text and binary protocol implementation for Tarantool. Turns Tarantool into a persistent memcached with master-master replication. See https://github.com/tarantool/memcached
- migrate - a Tarantool rock for migration from Tarantool 1.5 to 1.6. See https://github.com/bigbes/migrate
- cqueues - a Lua asynchronous networking, threading, and notification framework (contributed by @daurnimator). PR 1204.

Release 1.6.7

Release type: maintenance. Release date: 2015-11-17.

Incompatible changes:

- The syntax of upsert command has been changed and an extra key argument was removed from it. The primary key for look up is now always taken from the tuple, which is the second argument of upsert. upsert() was added fairly late at a release cycle and the design had an obvious bug which we had to fix. Sorry for this.
- fiber.channel.broadcast() was removed since it wasn't used by anyone and didn't work properly.
- tarantoolctl reload command renamed to eval.

Functionality added or changed:

- logger option now accepts a syntax for syslog output. Use uri-style syntax for file, pipe or syslog log destination.
- replication_source now accepts an array of URIs, so each replica can have up to 30 peers.
- RTREE index now accept two types of distance functions: euclid and manhattan.
- fio.abspath() - a new function in fio rock to convert a relative path to absolute.
- The process title now can be set with an on-board title rock.
- This release uses LuaJIT 2.1.

New rocks:

- memcached - makes Tarantool understand Memcached binary protocol. Text protocol support is in progress and will be added to the rock itself, without changes to the server core.

Release 1.6.6

Release type: maintenance. Release date: 2015-08-28.

Tarantool 1.6 is no longer getting major new features, although it will be maintained. The developers are concentrating on Tarantool version 1.9.

Incompatible changes:

- A new schema of _index system space which accommodates multi-dimensional RTREE indexes. Tarantool 1.6.6 works fine with an old snapshot and system spaces, but you will not be able to start Tarantool 1.6.5 with a data directory created by Tarantool 1.6.6, neither will you be able to query Tarantool 1.6.6 schema with 1.6.5 net.box.
- box.info.snapshot_pid is renamed to box.info.snapshot_in_progress

Functionality added or changed:

- Threaded architecture for network. Network I/O has finally been moved to a separate thread, increasing single instance performance by up to 50%.
- Threaded architecture for checkpointing. Tarantool no longer forks to create a snapshot, but uses a separate thread, accessing data via a consistent read view. This eliminates all known latency spikes caused by snapshotting.
- Stored procedures in C/C++. Stored procedures in C/C++ provide speed (3-4 times, compared to a Lua version in our measurements), as well as unlimited extensibility power. Since C/C++ procedures run in the same memory space as the database, they are also an easy tool to corrupt database memory. See The C API description.
- Multidimensional RTREE index. RTREE index type now support a large (up to 32) number of dimensions. RTREE data structure has been optimized to actually use R*-TREE. We're working on further improvements of the index, in particular, configurable distance function. See https://github.com/tarantool/tarantool/wiki/R-tree-index-quick-start-and-usage
- Sophia 2.1.1, with support of compression and multipart primary keys. See https://groups.google.com/forum/#!topic/sophia-database/GfcbEC7ksRg
- New upsert command available in the binary protocol and in stored functions. The key advantage of upsert is that it's much faster with write-optimized storage (sophia storage engine), but some caveats exists as well. See Issue 905 for details. Even though upsert performance advantage is most prominent with sophia engine, it works with all storage engines.
- Better memory diagnostics information for fibers, tuple and index arena Try a new command box.slab.stats(), for detailed information about tuple/index slabs, fiber.info() now displays information about memory used by the fiber.
- Update and delete now work using a secondary index, if the index is unique.
- Authentication triggers. Set box.session.on_auth triggers to catch authentication events. Trigger API is improved to display all defined triggers, easily remove old triggers.
- Manifold performance improvements of net.box built-in package.
- Performance optimizations of BITSET index.
- panic_on_wal_error is a dynamic configuration option now.
- iproto sync field is available in Lua as session.sync().
- box.once() - a new method to invoke code once in an instance and replica set lifetime. Use once() to set up spaces and uses, as well as do schema upgrade in production.
- box.error.last() to return the last error in a session.

New rocks:

- jit.*, jit.dump, jit.util, jit.vmdef modules of LuaJIT 2.0 are now available as built-ins. See http://luajit.org/ext_jit.html
- strict built-in package, banning use of undeclared variables in Lua. Strict mode is on when Tarantool is compiled with debug. Turn on/off with require('strict').on()/require('strict').off().
- pg and mysql rocks, available at http://rocks.tarantool.org - working with MySQL and PostgreSQL from Tarantool.
- gperftools rock, availble at http://rocks.tarantool.org - getting perfromance data using Google's gperf from Tarantool.
- csv built-in rock, to parse and load CSV (comma-separated values) data.

New supported platforms:

- Fedora 22, Ubuntu Vivid

CHAPTER 7

Contributor's Guide

7.1 C API reference

7.1.1 Module box

box_function_ctx_t
: Opaque structure passed to a C stored procedure

int box_return_tuple(box_function_ctx_t *ctx, box_tuple_t *tuple)
: Return a tuple from a C stored procedure.

  The returned tuple is automatically reference-counted by Tarantool. An example program that uses box_return_tuple() is write.c.

  Parameters

  - ctx (box_funtion_ctx_t*) – an opaque structure passed to the C stored procedure by Tarantool
  - tuple (box_tuple_t*) – a tuple to return

  Returns -1 on error (perhaps, out of memory; check box_error_last())

  Returns 0 otherwise

uint32_t box_space_id_by_name(const char *name, uint32_t len)
: Find space id by name.

  This function performs a SELECT request on the _vspace system space.

  Parameters

  - char* name (const) – space name
  - len (uint32_t) – length of name

  Returns BOX_ID_NIL on error or if not found (check box_error_last())

  Returns 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.

This function performs a SELECT request on the _vindex system space.

Parameters

- space_id (uint32_t) – space identifier
- char* name (const) – index name
- len (uint32_t) – length of name

Returns BOX_ID_NIL on error or if not found (check box_error_last())

Returns space_id otherwise

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.

Parameters

- 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. Resulting tuple. Can be set to NULL to discard result

Returns -1 on error (check box_error_last())

Returns 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 a REPLACE request.

Parameters

- 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. Resulting tuple. Can be set to NULL to discard result

Returns -1 on error (check box_error_last())

Returns 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 a DELETE request.

Parameters

- 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. An old tuple. Can be set to NULL to discard result

Returns -1 on error (check box_error_last())

Returns 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.

Parameters

- 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 an ops section
- index_base (int) – 0 if field_ids are zero-based as in C, 1 if field ids are 1-based as in Lua
- result (box_tuple_t**) – output argument. An old tuple. Can be set to NULL to discard result

Returns -1 on error (check box_error_last())

Returns 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.

Parameters

- 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 are zero-based as in C, 1 if field ids are 1-based as in Lua
- result (box_tuple_t**) – output argument. An old tuple. Can be set to NULL to discard result

Returns -1 on error (check :box_error_last())

Returns 0 otherwise

See also space_object.upsert()

int box_truncate(uint32_t space_id)
Truncate a space.

Parameters

- space_id (uint32_t) – space identifier

7.1.2 Module clock

double clock_realtime(void)
double clock_monotonic(void)
double clock_process(void)
double clock_thread(void)

uint64_t clock_realtime64(void)
uint64_t clock_monotonic64(void)
uint64_t clock_process64(void)
uint64_t clock_thread64(void)

7.1.3 Module 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.

Parameters

- 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.

Returns 0 - timeout

Returns >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 may use the worker_pool_threads configuration parameter.

To avoid double error checking, this function does not throw exceptions. In most cases it is 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.

Returns -1 and errno = ENOMEM if failed to create a task

Returns 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).

int coio_close(int fd)

Close the fd and wake any fiber blocked in coio_wait() call on this fd.

Parameters

- fd (int) – non-blocking socket file description

Returns the result of close(fd), see close(2)

7.1.4 Module error

enum box_error_code

enumerator ER_UNKNOWN

enumerator ER_ILLEGAL_PARAMS

enumerator ER_MEMORY_ISSUE

enumerator ER_TUPLE_FOUND

enumerator ER_TUPLE_NOT_FOUND

enumerator ER_UNSUPPORTED

enumerator ER_NONMASTER

enumerator ER_READONLY

enumerator ER_INJECTION

enumerator ER_CREATE_SPACE

enumerator ER_SPACE_EXISTS

enumerator ER_DROP_SPACE

enumerator ER_ALTER_SPACE

enumerator ER_INDEX_TYPE

enumerator ER_MODIFY_INDEX

enumerator ER_LAST_DROP

enumerator ER_TUPLE_FORMAT_LIMIT

enumerator ER_DROP_PRIMARY_KEY

enumerator ER_KEY_PART_TYPE

enumerator ER_EXACT_MATCH

enumerator ER_INVALID_MSGPACK

enumerator ER_PROC_RET

enumerator ER_TUPLE_NOT_ARRAY

enumerator ER_FIELD_TYPE

enumerator ER_FIELD_TYPE_MISMATCH

enumerator ER_SPLICE

enumerator ER_UPDATE_ARG_TYPE

enumerator ER_TUPLE_IS_TOO_LONG

enumerator ER_UNKNOWN_UPDATE_OP

enumerator ER_UPDATE_FIELD

enumerator ER_FIBER_STACK

enumerator ER_KEY_PART_COUNT

enumerator ER_PROC_LUA

enumerator ER_NO_SUCH_PROC

enumerator ER_NO_SUCH_TRIGGER

enumerator ER_NO_SUCH_INDEX

enumerator ER_NO_SUCH_SPACE

enumerator ER_NO_SUCH_FIELD

enumerator ER_EXACT_FIELD_COUNT

enumerator ER_INDEX_FIELD_COUNT

enumerator ER_WAL_IO

enumerator ER_MORE_THAN_ONE_TUPLE

enumerator ER_ACCESS_DENIED

enumerator ER_CREATE_USER

enumerator ER_DROP_USER

enumerator ER_NO_SUCH_USER

enumerator ER_USER_EXISTS

enumerator ER_PASSWORD_MISMATCH

enumerator ER_UNKNOWN_REQUEST_TYPE

enumerator ER_UNKNOWN_SCHEMA_OBJECT

enumerator ER_CREATE_FUNCTION

enumerator ER_NO_SUCH_FUNCTION

enumerator ER_FUNCTION_EXISTS

enumerator ER_FUNCTION_ACCESS_DENIED

enumerator ER_FUNCTION_MAX

enumerator ER_SPACE_ACCESS_DENIED

enumerator ER_USER_MAX

enumerator ER_NO_SUCH_ENGINE

enumerator ER_RELOAD_CFG

enumerator ER_CFG

enumerator ER_UNUSED60

enumerator ER_UNUSED61

enumerator ER_UNKNOWN_REPLICA

enumerator ER_REPLICASET_UUID_MISMATCH

enumerator ER_INVALID_UUID

enumerator ER_REPLICASET_UUID_IS_RO

enumerator ER_INSTANCE_UUID_MISMATCH

enumerator ER_REPLICA_ID_IS_RESERVED

enumerator ER_INVALID_ORDER

enumerator ER_MISSING_REQUEST_FIELD

enumerator ER_IDENTIFIER

enumerator ER_DROP_FUNCTION

enumerator ER_ITERATOR_TYPE

enumerator ER_REPLICA_MAX

enumerator ER_INVALID_XLOG

enumerator ER_INVALID_XLOG_NAME

enumerator ER_INVALID_XLOG_ORDER

enumerator ER_NO_CONNECTION

enumerator ER_TIMEOUT

enumerator ER_ACTIVE_TRANSACTION

enumerator ER_NO_ACTIVE_TRANSACTION

enumerator ER_CROSS_ENGINE_TRANSACTION

enumerator ER_NO_SUCH_ROLE

enumerator ER_ROLE_EXISTS

enumerator ER_CREATE_ROLE

enumerator ER_INDEX_EXISTS

enumerator ER_TUPLE_REF_OVERFLOW

enumerator ER_ROLE_LOOP

enumerator ER_GRANT

enumerator ER_PRIV_GRANTED

enumerator ER_ROLE_GRANTED

enumerator ER_PRIV_NOT_GRANTED

enumerator ER_ROLE_NOT_GRANTED

enumerator ER_MISSING_SNAPSHOT

enumerator ER_CANT_UPDATE_PRIMARY_KEY

enumerator ER_UPDATE_INTEGER_OVERFLOW

enumerator ER_GUEST_USER_PASSWORD

enumerator ER_TRANSACTION_CONFLICT

enumerator ER_UNSUPPORTED_ROLE_PRIV

enumerator ER_LOAD_FUNCTION

enumerator ER_FUNCTION_LANGUAGE

enumerator ER_RTREE_RECT

enumerator ER_PROC_C

enumerator ER_UNKNOWN_RTREE_INDEX_DISTANCE_TYPE

enumerator ER_PROTOCOL

enumerator ER_UPSERT_UNIQUE_SECONDARY_KEY

enumerator ER_WRONG_INDEX_RECORD

enumerator ER_WRONG_INDEX_PARTS

enumerator ER_WRONG_INDEX_OPTIONS

enumerator ER_WRONG_SCHEMA_VERSION

enumerator ER_MEMTX_MAX_TUPLE_SIZE

enumerator ER_WRONG_SPACE_OPTIONS

enumerator ER_UNSUPPORTED_INDEX_FEATURE

enumerator ER_VIEW_IS_RO

enumerator ER_UNUSED114

enumerator ER_SYSTEM

enumerator ER_LOADING

enumerator ER_CONNECTION_TO_SELF

enumerator ER_KEY_PART_IS_TOO_LONG

enumerator ER_COMPRESSION

enumerator ER_CHECKPOINT_IN_PROGRESS

enumerator ER_SUB_STMT_MAX

enumerator ER_COMMIT_IN_SUB_STMT

enumerator ER_ROLLBACK_IN_SUB_STMT

enumerator ER_DECOMPRESSION

enumerator ER_INVALID_XLOG_TYPE

enumerator ER_ALREADY_RUNNING

enumerator ER_INDEX_FIELD_COUNT_LIMIT

enumerator ER_LOCAL_INSTANCE_ID_IS_READ_ONLY

enumerator ER_BACKUP_IN_PROGRESS

enumerator ER_READ_VIEW_ABORTED

enumerator ER_INVALID_INDEX_FILE

enumerator ER_INVALID_RUN_FILE

enumerator ER_INVALID_VYLOG_FILE

enumerator ER_CHECKPOINT_ROLLBACK

enumerator ER_VY_QUOTA_TIMEOUT

enumerator ER_PARTIAL_KEY

enumerator ER_TRUNCATE_SYSTEM_SPACE

enumerator box_error_code_MAX

box_error_t
Error - contains information about error.

const char * box_error_type(const box_error_t *error)
Return the error type, e.g. "ClientError", "SocketError", etc.

Parameters

- error (box_error_t*) – error

Returns not-null string

uint32_t box_error_code(const box_error_t *error)
Return IPROTO error code

Parameters

- error (box_error_t*) – error

Returns enum box_error_code

const char * box_error_message(const box_error_t *error)
Return the error message

Parameters

- error (box_error_t*) – error

Returns 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.

Returns 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.

Parameters

- 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.

7.1.5 Module fiber

struct fiber
Fiber - contains information about a fiber.

typedef int (*fiber_func)(va_list)
Function to run inside a fiber.

struct fiber *fiber_new(const char *name, fiber_func f)
Create a new fiber.

Takes a fiber from the 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.

Parameters

- char* name (const) – string with fiber name
- f (fiber_func) – func for run inside fiber

See also: fiber_start()

struct fiber *fiber_new_ex(const char *name, const struct fiber_attr *fiber_attr, fiber_func f)
Create a new fiber with defined attributes.

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 if has a default stack size when its "main" function completes.

Parameters

- char* name (const) – string with fiber name
- struct fiber_attr* fiber_attr (const) – fiber attributes container
- f (fiber_func) – function to run inside the fiber

See also: fiber_start()

void fiber_start(struct fiber *callee, ...)
Start execution of created fiber.

Parameters

- fiber* callee (struct) – fiber to start
- ... – arguments to start the fiber with

void fiber_yield(void)
Return control to another fiber and wait until it'll be woken.

See also: fiber_wakeup()

void fiber_wakeup(struct fiber *f)
Interrupt a synchronous wait of a fiber

Parameters

- 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_CANCELLABLE 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()).

Parameters

- 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.

Parameters

- fiber* f (struct) – fiber
- yesno (bool) – status to set

Returns previous state

void fiber_set_joinable(struct fiber *fiber, bool yesno)
Set fiber to be joinable (false by default).

Parameters

- 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.

Parameters

- 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.

Parameters

- s (double) – time to sleep

Note: this is a cancellation point.

See also: fiber_is_cancelled()

bool fiber_is_cancelled(void)
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

struct fiber *fiber_self(void)
Return the current fiber.

struct fiber_attr

void fiber_attr_new(void)
Create a new fiber attributes container and initialize it with default parameters.

Can be used for creating many fibers: corresponding fibers will not take ownership.

void fiber_attr_delete(struct fiber_attr *fiber_attr)
Delete the fiber_attr and free all allocated resources. This is safe when fibers created with this attribute still exist.

Parameters

- fiber_attr* fiber_attribute (struct) – fiber attributes container

int fiber_attr_setstacksize(struct fiber_attr *fiber_attr, size_t stack_size)
Set the fiber's stack size in the fiber attributes container.

Parameters

- fiber_attr* fiber_attr (struct) – fiber attributes container
- stack_size (size_t) – stack size for new fibers (in bytes)

Returns 0 on success

Returns -1 on failure (if stack_size is smaller than the minimum allowable fiber stack size)

size_t fiber_attr_getstacksize(struct fiber_attr *fiber_attr)
Get the fiber's stack size from the fiber attributes container.

Parameters

- fiber_attr* fiber_attr (struct) – fiber attributes container, or NULL for default

Returns stack size (in bytes)

struct fiber_cond
A conditional variable: a synchronization primitive that allow fibers in Tarantool's cooperative multitasking environment to yield until some predicate is satisfied.

Fiber conditions have two basic operations – "wait" and "signal", – where "wait" suspends the execution of a fiber (i.e. yields) until "signal" is called.

Unlike pthread_cond, fiber_cond doesn't require mutex/latch wrapping.

struct fiber_cond *fiber_cond_new(void)
Create a new conditional variable.

void fiber_cond_delete(struct fiber_cond *cond)
Delete the conditional variable.

Note: behavior is undefined if there are fibers waiting for the conditional variable.

Parameters

- fiber_cond* cond (struct) – conditional variable to delete

void fiber_cond_signal(struct fiber_cond *cond);
Wake up one (any) of the fibers waiting for the conditional variable.

Does nothing if no one is waiting.

Parameters

- fiber_cond* cond (struct) – conditional variable

void fiber_cond_broadcast(struct fiber_cond *cond);
Wake up all fibers waiting for the conditional variable.

Does nothing if no one is waiting.

Parameters

- fiber_cond* cond (struct) – conditional variable

int fiber_cond_wait_timeout(struct fiber_cond *cond, double timeout)
Suspend the execution of the current fiber (i.e. yield) until fiber_cond_signal() is called.

Like pthread_cond, fiber_cond can issue spurious wake ups caused by explicit fiber_wakeup() or fiber_cancel() calls. It is highly recommended to wrap calls to this function into a loop and check the actual predicate and fiber_is_cancelled() on every iteration.

Parameters

- fiber_cond* cond (struct) – conditional variable
- double timeout (struct) – timeout in seconds

Returns 0 on fiber_cond_signal() call or a spurious wake up

Returns -1 on timeout, and the error code is set to 'TimedOut'

int fiber_cond_wait(struct fiber_cond *cond)
Shortcut for fiber_cond_wait_timeout().

7.1.6 Module 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.

  Parameters

- 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

Returns NULL on error (check box_error_last)

Returns 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.

Parameters

- iterator (box_iterator_t*) – an iterator returned by box_index_iterator
- result (box_tuple_t**) – output argument. result a tuple or NULL if there is no more data.

Returns -1 on error (check box_error_last)

Returns 0 on success. The end of data is not an error.

void box_iterator_free(box_iterator_t *iterator)
Destroy and deallocate iterator.

Parameters

- iterator (box_iterator_t*) – an iterator returned by box_index_iterator

int iterator_direction(enum iterator_type type)
Determine a direction of the given iterator type: -1 for REQ, LT, LE, and +1 for all others.

ssize_t box_index_len(uint32_t space_id, uint32_t index_id)
Return the number of element in the index.

Parameters

- space_id (uint32_t) – space identifier
- index_id (uint32_t) – index identifier

Returns -1 on error (check box_error_last)

Returns >= 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.

Parameters

- space_id (uint32_t) – space identifier
- index_id (uint32_t) – index identifier

Returns -1 on error (check box_error_last)

Returns >= 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).

Parameters

- 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.

Parameters

- 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

Returns -1 on error (check box_error_last)

Returns 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.

Parameters

- 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

Returns -1 on error (check box_error_last())

Returns 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.

Parameters

- 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

Returns -1 on error (check box_error_last())

Returns 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.

Parameters

- 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

Returns -1 on error (check box_error_last())

Returns 0 on success

See also: index_object.count()

const box_key_def_t *box_index_key_def(uint32_t space_id, uint32_t index_id)
Return key definition for an index

Returned object is valid until the next yield.

Parameters

- space_id (uint32_t) – space identifier
- index_id (uint32_t) – index identifier

Returns key definition on success

Returns NULL on error

See also: box_tuple_compare(), box_tuple_format_new()

7.1.7 Module latch

box_latch_t
A lock for cooperative multitasking environment

box_latch_t *box_latch_new(void)
Allocate and initialize the new latch.

Returns allocated latch object

Return type box_latch_t *

void box_latch_delete(box_latch_t *latch)
Destroy and free the latch.

Parameters

- 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.

Parameters

- latch (box_latch_t*) – latch to lock

Returns status of operation. 0 - success, 1 - latch is locked

Return type int

void box_latch_unlock(box_latch_t *latch)
Unlock a latch. The fiber calling this function must own the latch.

Parameters

- latch (box_latch_t*) – latch to unlock

7.1.8 Module 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.

Parameters

- L (lua_State*) – Lua State
- ctypeid (uint32_t) – FFI's CTypeID of this cdata

Returns memory associated with this cdata

See also: luaL_checkcdata()

void *luaL_checkcdata(struct lua_State *L, int idx, uint32_t *ctypeid)
Check whether the function argument idx is a cdata.

Parameters

- L (lua_State*) – Lua State
- idx (int) – stack index
- ctypeid (uint32_t*) – output argument. FFI's CTypeID of returned cdata

Returns memory associated with this cdata

See also: luaL_pushcdata()

void luaL_setcdatagc(struct lua_State *L, int idx)
Set finalizer function on a cdata object.

Equivalent to call ffi.gc(obj, function). Finalizer function must be on the top of the stack.

Parameters

- L (lua_State*) – Lua State
- idx (int) – stack index

uint32_t luaL_ctypeid(struct lua_State *L, const char *ctypename)
Return CTypeID (FFI) of given CDATA type.

Parameters

- L (lua_State*) – Lua State
- char* ctypename (const) – C type name as string (e.g. “struct request” or “uint32_t”)

Returns CTypeID

See also: luaL_pushcdata(), luaL_checkcdata()

int luaL_cdef(struct lua_State *L, const char *ctypename)
Declare symbols for FFI.

Parameters

- L (lua_State*) – Lua State
- char* ctypename (const) – C definitions (e.g. “struct stat”)

Returns 0 on success

Returns 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.

Parameters

- 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.

Parameters

- L (lua_State*) – Lua State
- val (int64_t) – value to push

uint64_t luaL_checkuint64(struct lua_State *L, int idx)
Check whether the argument idx is a uint64 or a convertable string and returns this number.

Throws error if the argument can’t be converted

uint64_t luaL_checkint64(struct lua_State *L, int idx)
Check whether the argument idx is a int64 or a convertable string and returns this number.

Throws error if the argument can’t be converted

uint64_t luaL_touint64(struct lua_State *L, int idx)
Check whether the argument idx is a uint64 or a convertable string and returns this number.

Returns the converted number or 0 of argument can’t be converted

int64_t luaL_toint64(struct lua_State *L, int idx)
Check whether the argument idx is a int64 or a convertable string and returns this number.

Returns the converted number or 0 of argument can't be converted

void luaT_pushtuple(struct lua_State *L, box_tuple_t *tuple)
Push a tuple onto the stack.

Parameters

- L (lua_State*) – Lua State

Throws error on OOM

See also: luaT_istuple

box_tuple_t *luaT_istuple(struct lua_State *L, int idx)
Check whether idx is a tuple.

Parameters

- L (lua_State*) – Lua State
- idx (int) – the stack index

Returns non-NULL if idx is a tuple

Returns NULL if idx is not a tuple

int luaT_error(lua_State *L)
Re-throw the last Tarantool error as a Lua object.

See also: lua_error(), box_error_last().

int luaT_cpcall(lua_State *L, lua_CFunction func, void *ud)
Similar to lua_cpcall(), but with the proper support of Tarantool errors.

lua_State *luaT_state(void)
Get the global Lua state used by Tarantool.

7.1.9 Module say (logging)

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_VERBOSE

enumerator S_DEBUG

say(level, format, ...)
Format and print a message to Tarantool log file.

Parameters

- 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_verbose(format, ...)
say_debug(format, ...)
say_syserror(format, ...)

Format and print a message to Tarantool log file.

Parameters

- char* format (const) – printf()-like format string
- ... – format arguments

See also printf(3), say_level

Example:

```
say_info("Some useful information: %s", status);
```

7.1.10 Module 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_TRIGGER_ID
: Space id of _trigger.

enumerator BOX_TRUNCATE_ID
: Space id of _truncate.

enumerator BOX_SYSTEM_ID_MAX
: End of reserved range of system spaces.

enumerator BOX_ID_NIL
: NULL value, returned on error.

7.1.11 Module trivia/config

API_EXPORT
: Extern modifier for all public functions.

PACKAGE_VERSION_MAJOR
: Package major version - 2 for 2.0.5.

PACKAGE_VERSION_MINOR
: Package minor version - 0 for 2.0.5.

PACKAGE_VERSION_PATCH
: Package patch version - 5 for 2.0.5.

PACKAGE_VERSION
: A string with major-minor-patch-commit-id identifier of the release, e.g. 2.0.5-75-gdd8e14ffb.

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.

7.1.12 Module tuple

box_tuple_format_t

box_tuple_format_t *box_tuple_format_default(void)
: Tuple format.

  Each Tuple has an associated format (class). Default format is used to create tuples which are not attached 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 raw MsgPack Array data.

  Parameters

  - 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

  Returns NULL on out of memory

  Returns tuple otherwise

  See also: box.tuple.new()

> Warning: When working with tuples, it is the developer's responsibility to ensure that enough space is allocated, taking especial caution when writing to them with msgpuck functions such as mp_encode_array().

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 reference counted 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. The Lua garbage collector will not destroy a tuple that has references, even if another fiber removes them from a space. After processing, decrement the reference counter using box_tuple_unref(), otherwise the tuple will leak.

  Parameters

- tuple (box_tuple_t*) – a tuple

Returns -1 on error

Returns 0 otherwise

See also: box_tuple_unref()

void box_tuple_unref(box_tuple_t *tuple)
Decrease the reference counter of tuple.

Parameters

- tuple (box_tuple_t*) – a tuple

Returns -1 on error

Returns 0 otherwise

See also: box_tuple_ref()

uint32_t box_tuple_field_count(const box_tuple_t *tuple)
Return the number of fields in a tuple (the size of MsgPack Array).

Parameters

- 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).

Parameters

- 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.

Returns -1 on error

Returns number of bytes written on success.

box_tuple_format_t *box_tuple_format(const box_tuple_t *tuple)
Return the associated format.

Parameters

- tuple (box_tuple_t*) – a tuple

Returns 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 result is a pointer to raw MessagePack data which can be decoded with mp_decode functions, for an example see the tutorial program read.c.

The buffer is valid until the next call to a box_tuple_* function.

Parameters

- tuple (box_tuple_t*) – a tuple
- field_id (uint32_t) – zero-based index in MsgPack array.

Returns NULL if i >= box_tuple_field_count()

Returns msgpack otherwise

enum field_type

enumerator FIELD_TYPE_ANY

enumerator FIELD_TYPE_UNSIGNED

enumerator FIELD_TYPE_STRING

enumerator FIELD_TYPE_ARRAY

enumerator FIELD_TYPE_NUMBER

enumerator FIELD_TYPE_INTEGER

enumerator FIELD_TYPE_SCALAR

enumerator field_type_MAX

Possible data types for tuple fields.

One cannot use STRS/ENUM macros for types because there is a mismatch between enum name (STRING) and type name literal ("STR"). STR is already used as a type in Objective C.

typedef struct key_def box_key_def_t
Key definition

box_key_def_t *box_key_def_new(uint32_t *fields, uint32_t *types, uint32_t part_count)
Create a key definition with the key fields with passed types on passed positions.

May be used for tuple format creation and/or tuple comparison.

Parameters

- fields (uint32_t*) – array with key field identifiers
- types (uint32_t) – array with key field types
- part_count (uint32_t) – the number of key fields

Returns key definition on success

Returns NULL on error

void box_key_def_delete(box_key_def_t *key_def)
Delete a key definition

Parameters

- key_def (box_key_def_t*) – key definition to delete

box_tuple_format_t *box_tuple_format_new(struct key_def *keys, uint16_t key_count)
Return new in-memory tuple format based on passed key definitions

Parameters

- keys (key_def) – array of keys defined for the format
- key_count (uint16_t) – count of keys

Returns new tuple format on success

Returns NULL on error

void box_tuple_format_ref(box_tuple_format_t *format)
Increment tuple format reference count

Parameters

- tuple_format (box_tuple_format_t) – tuple format to ref

void box_tuple_format_unref(box_tuple_format_t *format)
Decrement tuple format reference count

Parameters

- tuple_format (box_tuple_format_t) – tuple format to unref

int box_tuple_compare(const box_tuple_t *tuple_a, const box_tuple_t *tuple_b, const box_key_def_t *key_def)
Compare tuples using key definition

Parameters

- box_tuple_t* tuple_a (const) – the first tuple
- box_tuple_t* tuple_b (const) – the second tuple
- box_key_def_t* key_def (const) – key definition

Returns 0 if key_fields(tuple_a) == key_fields(tuple_b)

Returns <0 if key_fields(tuple_a) < key_fields(tuple_b)

Returns >0 if key_fields(tuple_a) > key_fields(tuple_b)

See also: enum field_type

int box_tuple_compare_with_key(const box_tuple_t *tuple, const char *key, const box_key_def_t *key_def);
Compare a tuple with a key using key definition

Parameters

- box_tuple_t* tuple (const) – tuple
- char* key (const) – key with MessagePack array header
- box_key_def_t* key_def (const) – key definition

Returns 0 if key_fields(tuple) == parts(key)

Returns <0 if key_fields(tuple) < parts(key)

Returns >0 if key_fields(tuple) > parts(key)

See also: enum field_type

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 allows iterating over fields at the root level of a MsgPack array.

Example:

```
box_tuple_iterator_t* it = box_tuple_iterator(tuple);
if (it == NULL) {
    // error handling using box_error_last()
}
const char* field;
```

(continues on next page)

(continued from previous page)

```
while (field = box_tuple_next(it)) {
    // process raw MsgPack data
}

// rewind the iterator to the 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 returnы the field id of the 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.

Parameters

- it (box_tuple_iterator_t*) – a tuple iterator

Returns position

void box_tuple_rewind(box_tuple_iterator_t *it)
Rewind iterator to the initial position.

Parameters

- 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 result is a pointer to raw MessagePack data which can be decoded with mp_decode functions, for an example see the tutorial program read.c. The returned buffer is valid until the next call to box_tuple_* API. The requested field_no is returned by the next call to box_tuple_next(it).

Parameters

- 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_not 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 result is a pointer to raw MessagePack data which can be decoded with mp_decode functions, for an example see the tutorial program read.c. The returned buffer is valid until next call to box_tuple_* API.

Parameters

- it (box_tuple_iterator_t*) – a tuple iterator

Returns NULL if there are no more fields

Returns 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)

7.1.13 Module 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. See also box.begin().

Returns 0 on success

Returns -1 on error. Perhaps a transaction has already been started.

int box_txn_commit(void)
Commit the current transaction. See also box.commit().

Returns 0 on success

Returns -1 on error. Perhaps a disk write failure

void box_txn_rollback(void)
Roll back the current transaction. See also box.rollback().

box_txn_savepoint_t * savepoint(void)
Return a descriptor of a savepoint.

void box_txn_rollback_to_savepoint(box_txn_savepoint_t *savepoint)
Roll back the current transaction as far as the specified savepoint.

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.

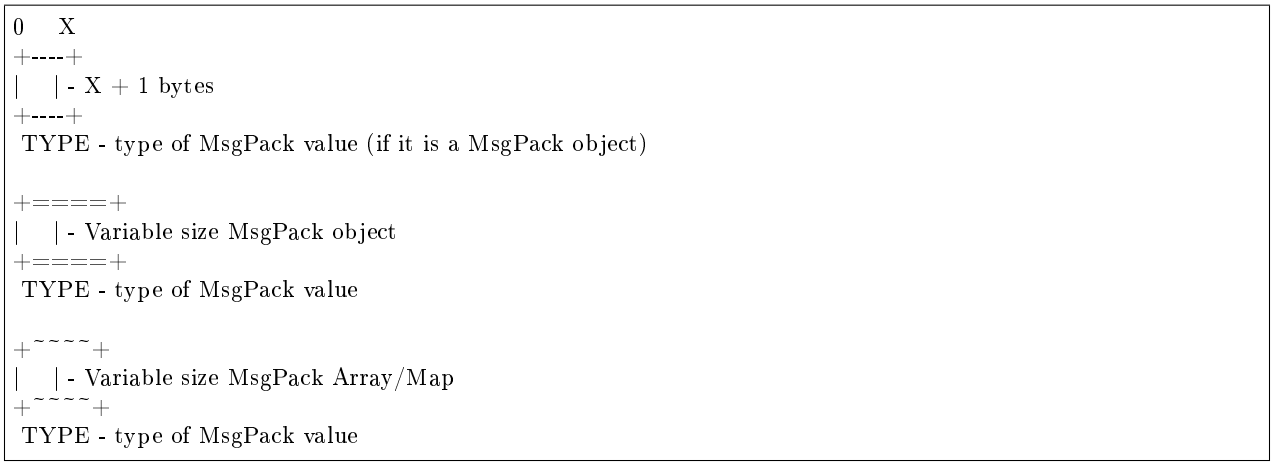
Returns NULL on out of memory

7.2 Internals

7.2.1 Binary protocol

The binary protocol in Tarantool is a binary request/response protocol.

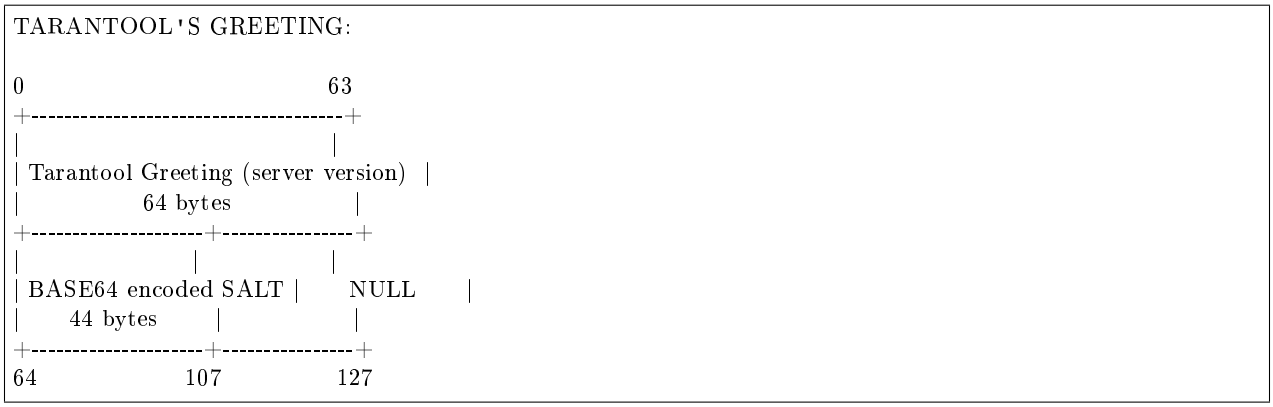
Notation in diagrams



MsgPack data types:

- MP_INT - Integer
- MP_MAP - Map
- MP_ARR - Array
- MP_STRING - String
- MP_FIXSTR - Fixed size string
- MP_OBJECT - Any MsgPack object
- MP_BIN - MsgPack binary format

Greeting packet



The server instance begins the dialogue by sending a fixed-size (128-byte) text greeting to the client. The greeting always contains two 64-byte lines of ASCII text, each line ending with a newline character (\n). The first line contains the instance version and protocol type. The second line contains up to 44 bytes of base64-encoded random string, to use in the authentication packet, and ends with up to 23 spaces.

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

The protocol uses msgpack for data structures and encoding.

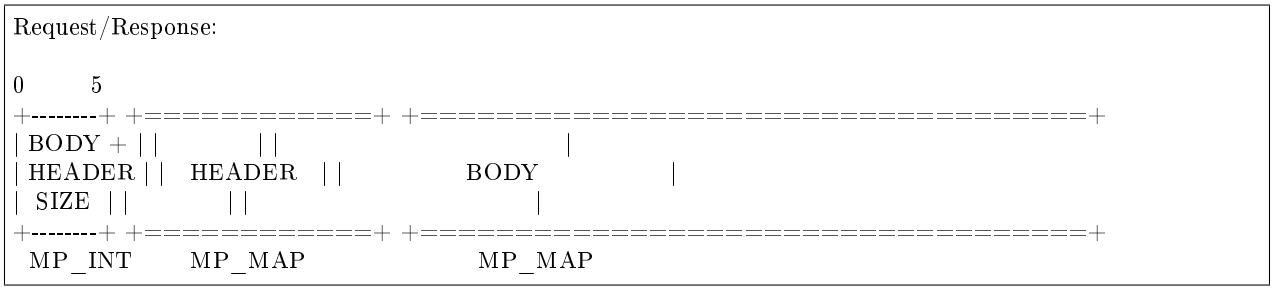
The protocol uses maps that contain some integer constants as keys. These constants are defined in src/box/iproto_constants.h. We list common constants here:

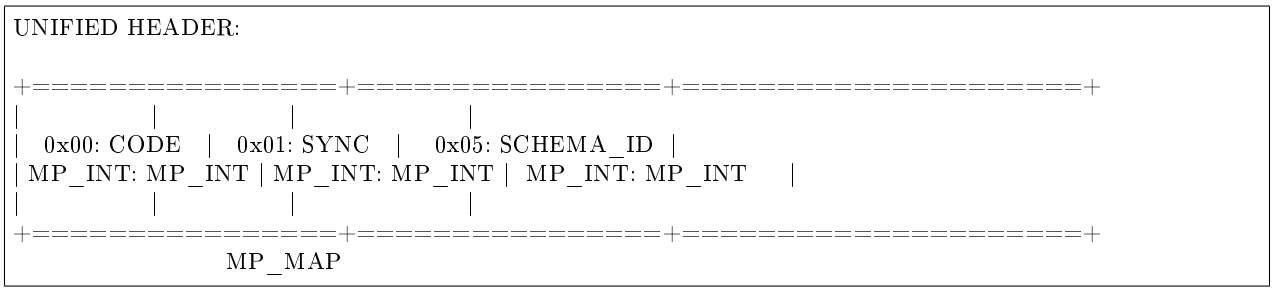
```
-- user keys
<iproto_sync>          ::= 0x01
<iproto_schema_id>     ::= 0x05  /* also known as schema_version */
<iproto_space_id>      ::= 0x10
<iproto_index_id>      ::= 0x11
<iproto_limit>         ::= 0x12
<iproto_offset>        ::= 0x13
<iproto_iterator>      ::= 0x14
<iproto_key>           ::= 0x20
<iproto_tuple>         ::= 0x21
<iproto_function_name> ::= 0x22
<iproto_username>      ::= 0x23
<iproto_expr>          ::= 0x27 /* also known as expression */
<iproto_ops>           ::= 0x28
<iproto_data>          ::= 0x30
<iproto_error>         ::= 0x31
<iproto_sql_text>      ::= 0x40
<iproto_sql_bind>      ::= 0x41
<iproto_sql_info>      ::= 0x42
```

```
-- -- Value for <code> key in request can be:
-- User command codes
<iproto_select>         ::= 0x01
<iproto_insert>         ::= 0x02
<iproto_replace>        ::= 0x03
<iproto_update>         ::= 0x04
<iproto_delete>         ::= 0x05
<iproto_call_16>        ::= 0x06 /* as used in version 1.6 */
<iproto_auth>           ::= 0x07
<iproto_eval>           ::= 0x08
<iproto_upsert>         ::= 0x09
<iproto_call>           ::= 0x0a
<iproto_execute>        ::= 0x0b
<iproto_nop>            ::= 0x0c
<iproto_type_stat_max>  ::= 0x0d
-- Admin command codes
-- (including codes for replica-set initialization and master election)
<iproto_ping>          ::= 0x40
<iproto_join>          ::= 0x41 /* i.e. replication join */
<iproto_subscribe>     ::= 0x42
<iproto_request_vote> ::= 0x43

-- -- Value for <code> key in response can be:
<iproto_ok>            ::= 0x00
<iproto_type_error>    ::= 0x8XXX /* where XXX is a value in errcode.h */
```

Both <header> and <body> are msgpack maps:

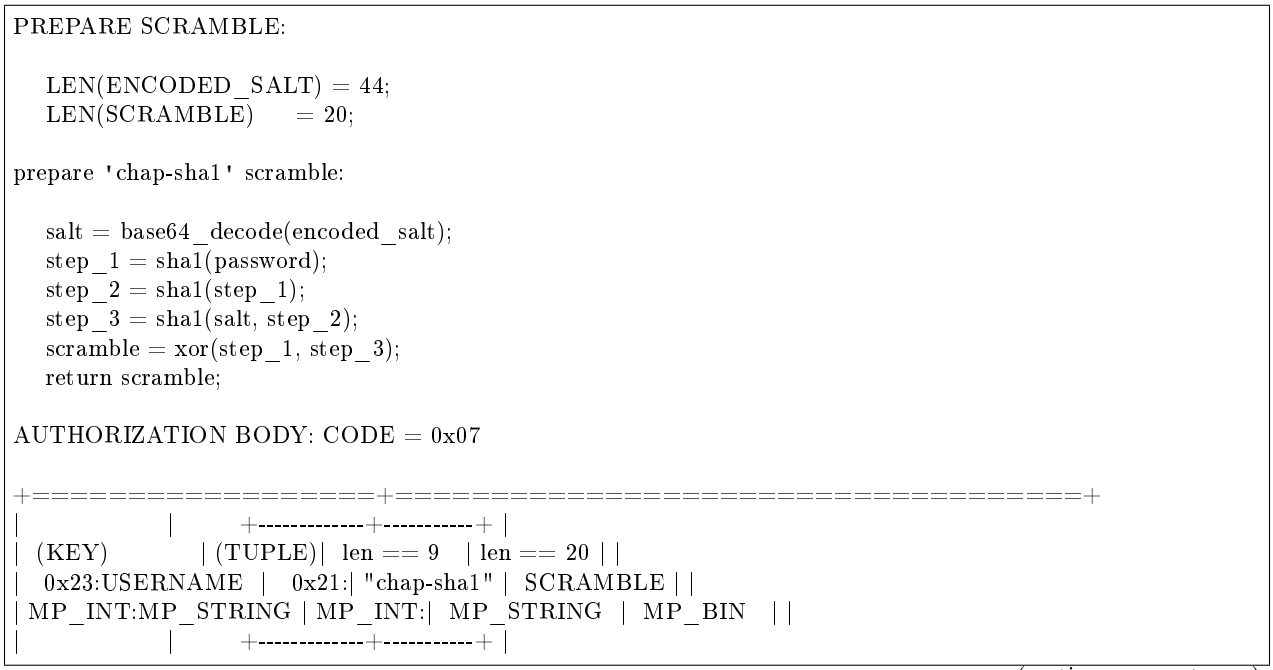




They only differ in the allowed set of keys and values. The key defines the type of value that follows. If a body has no keys, the entire msgpack map for the body may be missing. Such is the case, for example, for a <ping> request. schema_id may be absent in the request's header, meaning that there will be no version checking, but it must be present in the response. If schema_id is sent in the header, then it will be checked.

Authentication

When a client connects to the server instance, the instance 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.



(continues on next page)

(continued from previous page)

```
|                  |                  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 instance responds to authentication packet with a standard response with 0 tuples.

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       |
|                  |                      |
+==================+======================+
|                  |       +~~~~~~~~~~+ |
|                  |       |        | |
```

(continues on next page)

(continued from previous page)

```
|                | (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 is an error to specify an argument of a type that differs from the 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_16: CODE - 0x06 Call a stored function, returning an array of tuples. This is deprecated; CALL (0x0a) is recommended instead.

```
CALL_16 BODY:

+========================+==================+
|                        |                  |
|   0x22: FUNCTION_NAME  |   0x21: TUPLE    |
| MP_INT: MP_STRING      | MP_INT: MP_ARRAY |
|                        |                  |
+========================+==================+
                MP_MAP
```

- EVAL: CODE - 0x08 Evaulate 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
+-----------+==========+==========+
|          |         |         |
```

(continues on next page)

(continued from previous page)

```
|    OP     | FIELD_NO | ARGUMENT |
| MP_FIXSTR |  MP_INT  |  MP_INT  |
|           |          |          |
+-----------+==========+==========+
            MP_ARRAY

Supported operations:

'+' - add a value to a numeric field. If the filed 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).
```

- CALL: CODE - 0x0a Similar to CALL_16, but – like EVAL, CALL returns a list of values, unconverted

```
CALL BODY:

+========================+====================+
|                        |                    |
|   0x22: FUNCTION_NAME  |   0x21: TUPLE      |
| MP_INT: MP_STRING      | MP_INT: MP_ARRAY   |
|                        |                    |
+========================+====================+
                MP_MAP
```

Response packet structure

We will 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
```

(continues on next page)

(continued from previous page)

```
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.
```

An 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

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 initial JOIN
             HEADER                   BODY
+================+================++====================+
|                |                ||  SERVER_UUID      |
|   0x00: 0x41   |   0x01: SYNC   ||   0x24: UUID      |
| MP_INT: MP_INT | MP_INT: MP_INT || MP_INT: MP_STRING |
|                |                ||                    |
+================+================++====================+
             MP_MAP                   MP_MAP

Then instance, 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        |
+================+================++==============================+
```

(continues on next page)

(continued from previous page)

```
          MP_MAP                  MP_MAP

SUBSCRIBE:

Then you must send SUBSCRIBE:

                      HEADER
+====================+====================+
|                    |                    |
|    0x00: 0x42      |    0x01: SYNC      |
|  MP_INT: MP_INT    |  MP_INT: MP_INT    |
|                    |                    |
+====================+====================+
|   SERVER_UUID      |   CLUSTER_UUID     |
|  0x24: UUID        |  0x25: UUID        |
| MP_INT: MP_STRING  | MP_INT: MP_STRING  |
|                    |                    |
+====================+====================+
             MP_MAP

     BODY
+=================+
|                 |
|  0x26: VCLOCK   |
| MP_INT: MP_INT  |
|                 |
+=================+
     MP_MAP

Then you must process every query that'll came through other masters.
Every request between masters will have Additional LSN and SERVER_ID.
```

XLOG / SNAP

XLOG and SNAP files have nearly the same format. The header looks like:

```
<type>\n                SNAP\n or XLOG\n
<version>\n             currently 0.13\n
Server: <server_uuid>\n   where UUID is a 36-byte string
VClock: <vclock_map>\n    e.g. {1: 0}\n
\n
```

After the file header come the data tuples. Tuples begin with a row marker 0xd5ba0bab and the last tuple may be followed by an EOF marker 0xd510aded. Thus, between the file header and the EOF marker, there may be data tuples that have this form:

```
0           3 4                              17
+-------------+========+============+===========+==========+
|             |        |            |           |          |
| 0xd5ba0bab  | LENGTH | CRC32 PREV | CRC32 CUR | PADDING  |
|             |        |            |           |          |
+-------------+========+============+===========+==========+
  MP_FIXEXT2    MP_INT     MP_INT       MP_INT      ---

+=============+ +=====================================+
```

(continues on next page)

(continued from previous page)

```
|          ||                                  |
|  HEADER  ||              BODY                |
|          ||                                  |
+==========+ +==================================+
   MP_MAP                MP_MAP
```

See the example in the following section.

7.2.2 SQL protocol

Tarantool's SQL protocol regulates how to build SQL requests and parse responses using Tarantool's common binary protocol.

Special SQL keys:

```
<metadata>      ::= 0x32
<sql_text>      ::= 0x40
<sql_bind>      ::= 0x41
<sql_info>      ::= 0x42
```

Special SQL commands:

```
<execute> ::= 11
```

Request packet body

An SQL request has the type EXECUTE=11.

```
EXECUTE REQUEST BODY:
                    MAP
+=======================+================================+
|                 |                          |
|   0x40: SQL_TEXT    |     0x41: SQL_BIND           |
| MP_STR: SQL request | MP_ARRAY: array of parameters |
|                 |                          |
+=======================+================================+
```

- SQL_TEXT is a single non-empty SQL statement. For SQL syntax, see https://sqlite.org/lang.html
- SQL_BIND is an optional array of bindings (parameters). Each parameter value is a scalar: number, string, binary, null.

  A parameter can be ordinal or named. An ordinal parameter is encoded as a message pack scalar value (MP_UINT, INT, DOUBLE, FLOAT, STR, BIN, EXT, NIL). A named parameter is encoded as a map with one string key – its name. For bindings syntax, see https://sqlite.org/lang_expr.html#varparam

Examples:

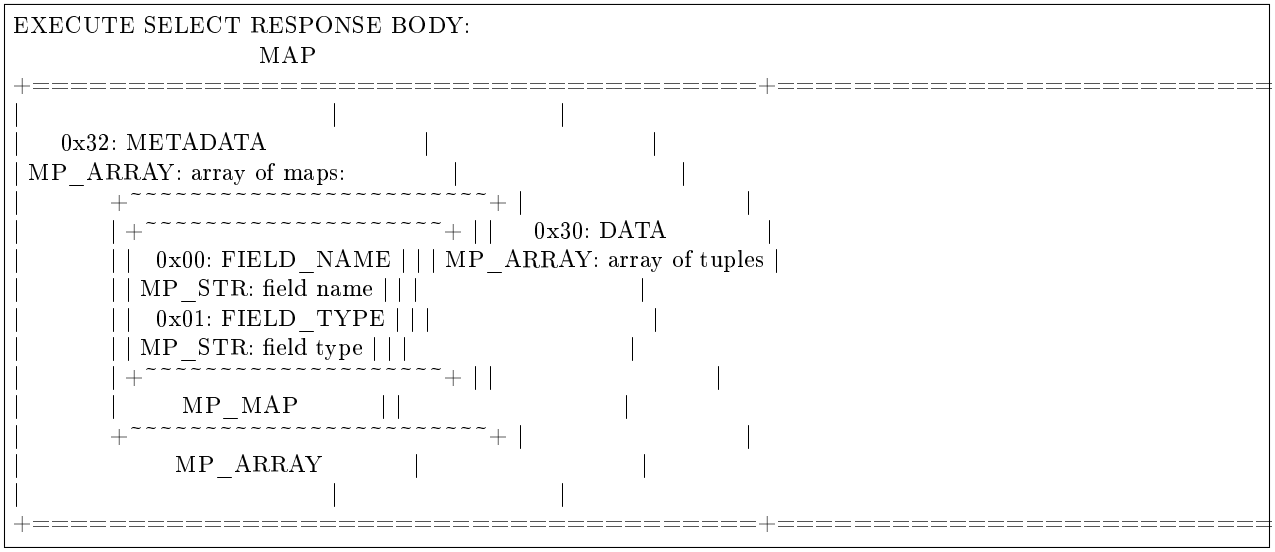
- [100, 'abc', NULL, -345.6] = MP_ARRAY[MP_UINT, MP_STR, MP_NIL, MP_DOUBLE]
- [1, 2, {'name': 300}] = MP_ARRAY[MP_UINT, MP_UINT, MP_MAP{ MP_STR : MP_UINT }]

Response packet body

Body structure depends on the type of the SQL request.

If the SQL request is SELECT, the response contains:

- metadata for columns (metadata for a single column contains only the column's name and type) and
- result rows.



Example:

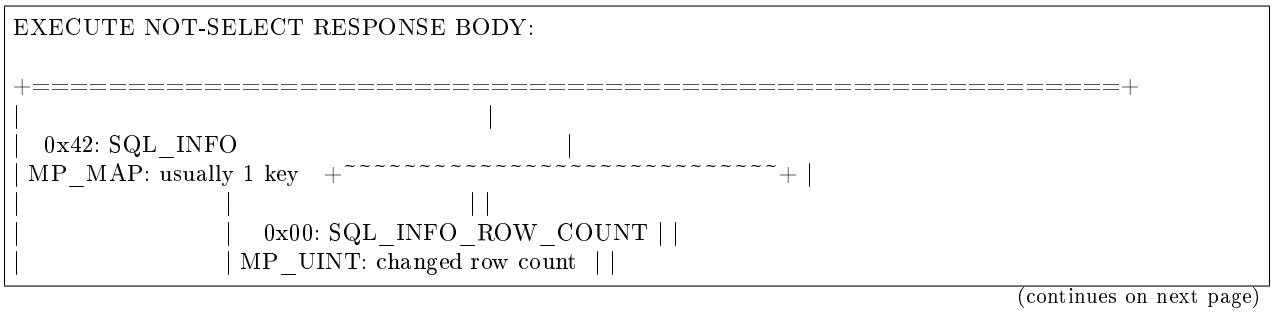
Request: SELECT x, y FROM test_space;

Response:

```
BODY = {
   METADATA = [
   { FIELD_NAME: 'X', FIELD_TYPE: 'TEXT'}, { FIELD_NAME: 'Y', FIELD_TYPE: 'INTEGER'},
   DATA = [ ['a', 1], ['c', 2], ['e', 5], ... ]
}
```

If the SQL request is not SELECT, the response body contains only SQL_INFO. Usually SQL_INFO is a map with only one key – SQL_INFO_ROW_COUNT (0) – which is the number of changed rows. For example, if the request is INSERT INTO test VALUES (1), (2), (3), the response body contains an SQL_INFO map with SQL_INFO_ROW_COUNT = 3. SQL_INFO_ROW_COUNT can be 0 for statements that do not change rows, such as CREATE TABLE.

The SQL_INFO map may contain a second key – SQL_INFO_AUTO_INCREMENT_IDS (1) – which is the new primary-key value for an INSERT in a table defined with PRIMARY KEY AUTOINCREMENT.



(continues on next page)

(continued from previous page)

```
|                   |                      | |
|                   +~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~+ |
|                                            |
+==================================================================+
```

7.2.3 File formats

Data persistence and the WAL file format

To maintain data persistence, Tarantool writes each data change request (insert, update, delete, replace, upsert) into a write-ahead log (WAL) file in the wal_dir directory. A new WAL file is created for every rows_per_wal records, or for every wal_max_size bytes. 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 (formatted as in Tarantool's binary protocol), each WAL record contains a header, some metadata, and then the data formatted according to msgpack rules. For example, this is what the WAL file looks like after the first INSERT request ("s:insert({1})") for the sandbox database created in our "Getting started" exercises. On the left are the hexadecimal bytes that you would see with:

```
$ hexdump 00000000000000000000.xlog
```

and on the right are comments.

```
Hex dump of WAL file       Comment
--------------------       -------
58 4c 4f 47 0a             "XLOG\n"
30 2e 31 33 0a             "0.13\n" = version
53 65 72 76 65 72 3a 20    "Server: "
38 62 66 32 32 33 65 30 2d [Server UUID]\n
36 39 31 34 2d 34 62 35 35
2d 39 34 64 32 2d 64 32 62
36 64 30 39 62 30 31 39 36
0a
56 43 6c 6f 63 6b 3a 20    "Vclock: "
7b 7d                      "{}" = vclock value, initially blank
...                        (not shown = tuples for system spaces)
d5 ba 0b ab                Magic row marker always = 0xab0bbad5
19                         Length, not including length of header, = 25 bytes
00                           Record header: previous crc32
ce 8c 3e d6 70               Record header: current crc32
a7 cc 73 7f 00 00 66 39      Record header: padding
84                         msgpack code meaning "Map of 4 elements" follows
00 02                         element#1: tag=request type, value=0x02=IPROTO_INSERT
02 01                         element#2: tag=server id, value=0x01
03 04                         element#3: tag=lsn, value=0x04
04 cb 41 d4 e2 2f 62 fd d5 d4 element#4: tag=timestamp, value=an 8-byte "Float64"
82                         msgpack code meaning "map of 2 elements" follows
10 cd 02 00                   element#1: tag=space id, value=512, big byte first
21 91 01                      element#2: tag=tuple, value=1-element fixed array={1}
```

Tarantool 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 instance 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 the WAL writer running in a separate thread, requesting that the change be recorded in the WAL. The instance switches to work on the next request until the write is acknowledged.
5. 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.

The snapshot file format

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 instance'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, _space, _index, _func, _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.

7.2.4 The recovery process

The recovery process begins when box.cfg{} happens for the first time after the Tarantool server instance starts.

The recovery process must recover the databases as of the moment when the instance 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, memtx_dir, vinyl_dir and force_recovery.

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 memtx databases, depending on whether the recovery process is "default".

If the recovery process is default (force_recovery is false), 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 the recovery process is non-default (force_recovery is true), 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 3 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 4 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 5 For the memtx engine, re-create all secondary indexes.

7.2.5 Server startup with replication

In addition to the recovery process described above, the server must take additional steps and precautions if replication is enabled.

Once again the startup procedure is initiated by the box.cfg{} request. One of the box.cfg parameters may be replication that specifies replication source(-s). We will refer to this replica, which is starting up due to box.cfg, as the "local" replica to distinguish it from the other replicas in a replica set, which we will refer to as "distant" replicas.

#1. If there is no snapshot .snap file and the ''replication'' parameter is empty: then the local replica assumes it is an unreplicated "standalone" instance, or is the first replica of a new replica set. It will generate new UUIDs for itself and for the replica set. The replica UUID is stored in the _cluster space; the replica set UUID is stored in the _schema space. Since a snapshot contains all the data in all the spaces, that means the local replica's snapshot will contain the replica UUID and the replica set UUID. Therefore, when the local replica restarts on later occasions, it will be able to recover these UUIDs when it reads the .snap file.

#2. If there is no snapshot .snap file and the ''replication'' parameter is not empty and the ''_cluster'' space contains no other replica UUIDs: then the local replica assumes it is not a standalone instance, but is not yet part of a replica set. It must now join the replica set. It will send its replica UUID to the first distant replica which is listed in replication and which will act as a master. This is called the "join request". When a distant replica receives a join request, it will send back:

(1) the distant replica's replica set UUID,

(2) the contents of the distant replica's .snap file. When the local replica receives this information, it puts the replica set UUID in its _schema space, puts the distant replica's UUID and connection information in its _cluster space, and makes a snapshot containing all the data sent by the distant replica. Then,

if the local replica has data in its WAL .xlog files, it sends that data to the distant replica. The distant replica will receive this and update its own copy of the data, and add the local replica's UUID to its _cluster space.

#3. If there is no snapshot .snap file and the "replication" parameter is not empty and the "_cluster" space contains other replica UUIDs: then the local replica assumes it is not a standalone instance, and is already part of a replica set. It will send its replica UUID and replica set UUID to all the distant replicas which are listed in replication. This is called the "on-connect handshake". When a distant replica receives an on-connect handshake:

(1) the distant replica compares its own copy of the replica set UUID to the one in the on-connect handshake. If there is no match, then the handshake fails and the local replica will display an error.

(2) the distant replica looks for a record of the connecting instance in its _cluster space. If there is none, then the handshake fails. Otherwise the handshake is successful. The distant replica will read any new information from its own .snap and .xlog files, and send the new requests to the local replica.

In the end ... the local replica knows what replica set it belongs to, the distant replica knows that the local replica is a member of the replica set, and both replicas have the same database contents.

#4. If there is a snapshot file and replication source is not empty: first the local replica goes through the recovery process described in the previous section, using its own .snap and .xlog files. Then it sends a "subscribe" request to all the other replicas of the replica set. The subscribe request contains the server vector clock. The vector clock has a collection of pairs 'server id, lsn' for every replica in the _cluster system space. Each distant replica, upon receiving a subscribe request, will read its .xlog files' requests and send them to the local replica if (lsn of .xlog file request) is greater than (lsn of the vector clock in the subscribe request). After all the other replicas of the replica set have responded to the local replica's subscribe request, the replica startup is complete.

The following temporary limitations apply for versions 1.7 and 2.1:

- The URIs in the replication parameter should all be in the same order on all replicas. This is not mandatory but is an aid to consistency.
- The maximum number of entries in the _cluster space is 32. Tuples for out-of-date replicas are not automatically re-used, so if this 32-replica limit is reached, users may have to reorganize the _cluster space manually.

7.3 Build and contribute

7.3.1 Building from source

For downloading Tarantool source and building it, the platforms can differ and the preferences can differ. But strategically the steps are always the same.

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 downloading the latest complete set of source files from the Tarantool repository on 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+.
   - A program for managing the build process. For all platforms, this is CMake version 2.8+.
   - ReadLine library, any version

- ncurses library, any version
- OpenSSL library, version 1.0.1+
- cURL library, version 0.725+
- LibYAML library, version 0.1.4+
- ICU library, recent version
- Python and modules. Python interpreter is not necessary for building Tarantool itself, unless you intend to use the "Run the test suite" option in step 5. For all platforms, this is python version 2.7+ (but not 3.x). You need the following Python modules:
  - pyyaml version 3.10
  - argparse version 1.1
  - msgpack-python version 0.4.6
  - gevent version 1.1.2
  - six version 1.8.0

To install all required dependencies, follow the instructions for your OS:

- For Debian/Ubuntu, say:

```
$ apt install -y build-essential cmake coreutils sed \
    libreadline-dev libncurses5-dev libyaml-dev libssl-dev \
    libcurl4-openssl-dev libunwind-dev libicu-dev \
    python python-pip python-setuptools python-dev \
    python-msgpack python-yaml python-argparse python-six python-gevent
```

- For RHEL/CentOS/Fedora, say:

```
$ yum install -y gcc gcc-c++ cmake coreutils sed \
    readline-devel ncurses-devel libyaml-devel openssl-devel \
    libcurl-devel libunwind-devel libicu-devel \
    python python-pip python-setuptools python-devel \
    python-msgpack python-yaml python-argparse python-six python-gevent
```

- For Mac OS X (instructions below are for OS X El Capitan):

  If you're using Homebrew as your package manager, say:

```
$ brew install cmake autoconf binutils zlib \
    readline ncurses libyaml openssl curl libunwind-headers icu4c \
    && pip install python-daemon \
    msgpack-python pyyaml configargparse six gevent
```

  Alternatively, download Apple's default Xcode toolset:

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

- For FreeBSD (instructions below are for FreeBSD 10.1 release), say:

```
$ pkg install -y sudo git cmake gmake gcc coreutils \
    readline ncurses libyaml openssl curl libunwind icu \
    python27 py27-pip py27-setuptools py27-daemon \
    py27-msgpack-python py27-yaml py27-argparse py27-six py27-gevent
```

If some Python modules are not available in a repository, it is best to set up the modules by getting a tarball and doing the setup with python setup.py like this:

```
$ # 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 -r \
    https://raw.githubusercontent.com/tarantool/test-run/master/requirements.txt \
    --user
```

This step is only necessary once, the first time you do a download.

2. Use git to download the latest Tarantool source code from the GitHub repository tarantool/tarantool, branch 2.1, to a local directory named ~/tarantool, for example:

```
$ git clone --recursive https://github.com/tarantool/tarantool.git -b 2.1 ~/tarantool
```

On rare occasions, the submodules need to be updated again with the command:

```
$ git submodule update --init --recursive
```

3. 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.

4. Use make to complete the build.

```
$ make
```

Note: For FreeBSD, use gmake instead.

This creates the 'tarantool' executable in the src/ directory.

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.

5. 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 privileges)
$ ln /usr/bin/python ~/tarantool/bin/python

$ # get to 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
```

6. Make RPM and Debian packages.

   This step is optional. It's only for people who want to redistribute Tarantool. We highly recommend to use official packages from the tarantool.org web-site. However, you can build RPM and Debian packages using PackPack or using the dpkg-buildpackage or rpmbuild tools. Please consult dpkg or rpmbuild documentation for details.

7. Verify your Tarantool installation.

```
$ # if you installed tarantool locally after build
$ tarantool
$ # - OR -
$ # if you didn't install tarantool locally after build
$ ./src/tarantool
```

This starts Tarantool in the interactive mode.

See also:

- Tarantool README.md

7.3.2 Building documentation

Tarantool 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 you can contribute to Tarantool's version.

You need to install these packages:

- 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)
- LaTeX (a system for document preparation; the installable package name usually begins with the word 'texlive' or 'tetex', on Ubuntu the name is 'texlive-latex-base')
- ImageMagick (a system for image conversion; on MacOS install it using brew)

You need to install these Python modules:

- pip, any version
- Sphinx version 1.4.4 or later

  Note: If you encounter the "Missing SPHINX_EXECUTABLE" error message on Mac, manually export the PATH variable:

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

- sphinx-intl version 0.9.9

  Note: If you encounter the "Missing SPHINX_INTL_DIR" error message on Mac, manually export the SPHINX_INTL_DIR variable:

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

- lupa – any version

Note: You should specify --user flag on Mac while installing Python modules for correct installation.

See more details about installation in the build-from-source section of this documentation.

1. Use git to download the latest source code of this documentation from the GitHub repository tarantool/doc, branch 2.1. For example, to download 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 .            # initiate
```

3. Build a local version of the documentation.

   Run the make command with an appropriate option to specify which documentation version to build.

   ```
   $ cd ~/tarantool-doc
   $ 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-ru  # one-page Russian version
   $ make all                   # all versions plus the entire web-site
   ```

   Documentation will be created in subdirectories of /output:

   - /output/en (files of the English version)
   - /output/ru (files of the Russian version)

   The entry point for each version is the index.html file in the appropriate directory.

4. Set up a web server.

   - One way is to say make sphinx-webserver. This will set up and run the web server on port 8000:

     ```
     $ cd ~/tarantool-doc
     $ make sphinx-html       # as an example, build the multi-page English documentation
     $ make sphinx-webserver  # set up and run the web server
     ```

     In case port 8000 is already in use, you can specify any other port number that is bigger than 1000 in the tarantool-doc/CMakeLists.txt file (search it for the sphinx-webserver target) and rebuild cmake files:

     ```
     $ cd ~/tarantool-doc
     $ git clean -qfxd        # clean up old cmake files
     $ cmake .                # rebuild cmake files
     $ make sphinx-html       # as an example, build the multi-page English documentation
     $ make sphinx-webserver  # set up and run the web server on the custom port
     ```

     Or you can release the port:

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

   - The other way is to run the built-in web server in Python. Make sure to run it from the documentation output folder:

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

     In case port 8000 is already in use, you can specify any other port number that is bigger than 1000.

5. Open your browser and enter 127.0.0.1:8000/en/doc/2.1/ into the address box (or 127.0.0.1:8000/ru/doc/2.1/ if you built the Russian documentation). Mind the trailing slash "/" in the address string.

If your local documentation build is valid, the manual will appear in the browser.

6. To contribute to documentation, use the REST format for drafting and submit your updates as a pull request via GitHub.

   To comply with the writing and formatting style, use the guidelines provided in the documentation, common sense and existing documents.

Note:

- If you suggest creating a new documentation section (a whole new page), it has to be saved to the relevant section at GitHub.
- If you want to contribute to localizing this documentation (for example into Russian), add your translation strings to .po files stored in the corresponding locale directory (for example /locale/ru/LC_MESSAGES/ for Russian). See more about localizing with Sphinx at http://www.sphinx-doc.org/en/stable/intl.html

---

7.3.3 Release management

Release policy

A Tarantool release is identified by three digits, for example, 1.7.7. We use these digits according to their definitions provided at http://semver.org:

- The first digit stands for MAJOR release. A major release may contain incompatible changes.
- The second digit stands for MINOR release, it does not contain incompatible changes, and is used for introducing backward-compatible features.
- The third digit is for PATCH releases that contain only backward-compatible bug fixes.

In MINOR digit, we reflect how stable a release is:

- 0 meaning alpha,
- 1 meaning beta,
- anything between 1 and 10 meaning stable, and
- 10 meaning LTS.

So, each MAJOR release series goes through a development-maturity life cycle of MINOR releases, as follows:

1. Alpha. Once in every few months we release a few alpha versions, e.g. 2.0.1, 2.0.2.

   Alpha versions may contain incompatible changes, crashes and other bugs.

2. Beta. Once major changes necessary to introduce new flagship features are ready, we release a few beta versions, e.g. 2.1.3, 2.1.4.

   Beta versions may contain crashes, but do not have incompatible changes, so can be used to develop new applications.

4. Stable. Finally, after we see our beta versions run successfully in production, usually in a few more months, during which we fix all incoming bugs and add some minor features, we declare this MAJOR release series stable.

Like Ubuntu, we distinguish two kinds of stable releases:

- LTS (Long Term Support) releases that are supported for 3 years (community) and up to 5 years (paying customers). LTS release is identified by MINOR version 10.
- Standard stable releases are only supported a few months after the next stable is out.

"Support" means that we continue fixing bugs in a release.

We add commits simultaneously to three MAJOR releases:

- LTS is a stable release which does not receive new features, and only gets backward compatible fixes. Hence, following the rules of semver, LTS release never has its MAJOR or MINOR version increased, and only gets PATCH level releases.
- STABLE is our current stable release, which may receive new features. When the next STABLE version is published, MINOR version is incremented. Between MINOR releases, we may have intermediate PATCH level releases as well, which will contain only bug fixes. We maintain PATCH level releases for two STABLE releases, the current and the previous one, to preserve support continuity.
- NEXT is our next MAJOR release, and it follows the maturity cycle described in the beginning. While NEXT release is in alpha state, its MINOR is frozen at 0 and is only increased when the release reaches BETA status. Once the NEXT release becomes STABLE, we switch the vehicle for delivery of minor features, designating the previous stable release as LTS, and releasing it with MINOR set to 10.

To sum up, once a quarter we release:

- the next LTS release, e.g. 2.10.6, 2.10.7 or 2.10.8
- the next STABLE release, e.g. 3.6, 3.7 or 3.8
- (optionally) an alpha or beta version of the NEXT release, e.g. 4.0.1, 4.0.2 or 4.0.3

In all supported releases, we also release a PATCH release as soon as we find and fix an outstanding CVE/vulnerability.

We also publish nightly builds, and use the fourth slot in the version identifier to designate the nightly build number.

Example version identifier:

- 2.0.3 - third alpha of 2.x release
- 2.1.3 - a beta of 2.x release
- 2.2 - a stable version of 2.x series, but not an LTS yet
- 2.10 - an LTS release

How to make a minor release

```
$ git tag -a 2.4 -m "Next minor in 2.x series"
$ vim CMakeLists.txt # edit CPACK_PACKAGE_VERSION_PATCH
$ git push --tags
```

A tag which is made on a git branch can be taken along with a merge, or left on the branch. The technique to "keep the tag on the branch it was originally set on" is to use --no-fast-forward when merging this branch.

With --no-ff, a merge changeset is created to represent the received changes, and only that merge changeset ends up in the destination branch. This technique can be useful when there are two active lines of development, e.g. "stable" and "next", and it's necessary to be able to tag both lines independently.

To make sure that a tag doesn't end up in the destination branch, it is necessary to have the commit to which the tag is attached, "stay on the original branch". That's exactly what a merge with disabled "fast-forward" does – creates a "merge" commit and adds it to both branches.

Here's what it may look like:

```
kostja@shmita:~/work/tarantool$ git checkout master
Already on 'master'
kostja@shmita:~/work/tarantool$ git tag -a 2.4 -m "Next development"
kostja@shmita:~/work/tarantool$ git describe
2.4
kostja@shmita:~/work/tarantool$ git checkout master-stable
Switched to branch 'master-stable'
kostja@shmita:~/work/tarantool$ git tag -a 2.3 -m "Next stable"
kostja@shmita:~/work/tarantool$ git describe
2.3
kostja@shmita:~/work/tarantool$ git checkout master
Switched to branch 'master'
kostja@shmita:~/work/tarantool$ git describe
2.4
kostja@shmita:~/work/tarantool$ git merge --no-ff master-stable
Auto-merging CMakeLists.txt
Merge made by recursive.
 CMakeLists.txt |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
kostja@shmita:~/work/tarantool$ git describe
2.4.0-0-g0a98576
```

Also, don't forget this:

1. 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.

2. Click 'Release milestone'. Create a milestone for the next minor release. Alert the driver to target bugs and blueprints to the new milestone.

How to release a Docker container

To bump a new version of a Docker container:

1. On the master branch of tarantool/docker repository, find the Dockerfile that corresponds to the commit's major version (e.g. https://github.com/tarantool/docker/blob/master/2.x/Dockerfile for Tarantool version 2.4) and specify the required commit in TARANTOOL_VERSION, for example TARANTOOL_VERSION=2.4.0-11-gcd17b77f9.

   Commit the Dockerfile back to master branch.

3. In the same repository, create a branch named after the commit's <major>.<minor> versions, e.g. branch 2.4 for commit 2.4.0-11-gcd17b77f9.

4. In Tarantool container build settings at hub.docker.com (https://hub.docker.com/r/tarantool/tarantool/~/settings/automated-builds/), add a new line:

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

   where x and y correspond to the commit's major and minor versions.

   Click Save changes.

Shortly after, a new Docker container will be built.

7.4 Guidelines

7.4.1 Developer guidelines

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.

How to write a commit message

Any commit needs a helpful message. Mind the following guidelines when committing to any of Tarantool repositories at GitHub.

1. Separate subject from body with a blank line.
2. Try to limit the subject line to 50 characters or so.
3. Start the subject line with a capital letter unless it prefixed with a subsystem name and semicolon:
    - memtx:
    - vinyl:
    - xlog:
    - replication:
    - recovery:
    - iproto:
    - net.box:
    - lua:
    - sql:
4. Do not end the subject line with a period.
5. Do not put "gh-xx", "closes #xxx" to the subject line.
6. Use the imperative mood in the subject line. A properly formed Git commit subject line should always be able to complete the following sentence: "If applied, this commit will /your subject line here/".
7. Wrap the body to 72 characters or so.
8. Use the body to explain what and why vs. how.

9. Link GitHub issues on the lasts lines (see how).
10. Use your real name and real email address. For Tarantool team members, @tarantool.org email is preferred, but not mandatory.

A template:

```
Summarize changes in 50 characters or less

More detailed explanatory text, if necessary.
Wrap it to 72 characters or so.
In some contexts, the first line is treated as the subject of the
commit, and the rest of the text as the body.
The blank line separating the summary from the body is critical
(unless you omit the body entirely); various tools like `log`,
`shortlog` and `rebase` can get confused if you run the two together.

Explain the problem that this commit is solving. Focus on why you
are making this change as opposed to how (the code explains that).
Are there side effects or other unintuitive consequences of this
change? Here's the place to explain them.

Further paragraphs come after blank lines.

- Bullet points are okay, too.

- Typically a hyphen or asterisk is used for the bullet, preceded
  by a single space, with blank lines in between, but conventions
  vary here.

Fixes: #123
Closes: #456
Needed for: #859
See also: #343, #789
```

Some real-world examples:

- tarantool/tarantool@2993a75
- tarantool/tarantool@ccacba2
- tarantool/tarantool@386df3d
- tarantool/tarantool@076a842

Based on [1] and [2].

How to submit a patch for review

We don't accept GitHub pull requests. Instead, all patches should be sent as plain-text messages to tarantool-patches@freelists.org. Please subscribe to our mailing list at https://www.freelists.org/list/tarantool-patches to ensure that your messages are added to the archive.

1. Preparing a patch

Once you have committed a patch to your local git repository, you can submit it for review.

To prepare an email, use git format-patch command:

```
$ git format-patch -1
```

It will format the commit at the top of your local git repository as a plain-text email and write it to a file in the current directory. The file name will look like 0001-your-commit-subject-line.patch. To specify a different directory, use -o option:

```
$ git format-patch -1 -o ~/patches-to-send
```

Once the patch has been formatted, you can view and edit it with your favorite text editor (after all, it is a plain-text file!). We strongly recommend adding:

- a hyperlink to the branch where this patch can be found at GitHub, and
- a hyperlink to the GitHub issue your patch is supposed to fix, if any.

If there is just one patch, the change log should go right after --- in the message body (it will be ignored by git am then).

If there are multiple patches you want to submit in one go (e.g. this is a big feature which requires some preparatory patches to be committed first), you should send each patch in a separate email in reply to a cover letter. To format a patch series accordingly, pass the following options to git format-patch:

```
$ git format-patch --cover-letter --thread=shallow HEAD~2
```

where:

- --cover-letter will make git format-patch generate a cover letter;
- --thread=shallow will mark each formatted patch email to be sent in reply to the cover letter;
- HEAD~2 (we now use it instead of -1) will make git format-patch format the first two patches at the top of your local git branch instead of just one. To format three patches, use HEAD~3, and so forth.

After the command has been successfully executed, you will find all your patches formatted as separate emails in your current directory (or in the directory specified via -o option):

```
0000-cover-letter.patch
0001-first-commit.patch
0002-second-commit.patch
...
```

The cover letter will have BLURB in its subject and body. You'll have to edit it before submitting (again, it is a plain text file). Please write:

- a short series description in the subject line;
- a few words about each patch of the series in the body.

And don't forget to add hyperlinks to the GitHub issue and branch where your series can be found. In this case you don't need to put links or any additional information to each individual email – the cover letter will cover everything.

Note: To omit --cover-letter and --thread=shallow options, you can add the following lines to your gitconfig:

```
[format]
    thread = shallow
    coverLetter = auto
```

---

2. Sending a patch

Once you have formatted your patches, they are ready to be sent via email. Of course, you can send them with your favorite mail agent, but it is much easier to use git send-email for this. Before using this command, you need to configure it.

If you use a GMail account, add the following code to your .gitconfig:

```
[sendemail]
    smtpencryption = tls
    smtpserver = smtp.gmail.com
    smtpserverport = 587
    smtpuser = your.name@gmail.com
    smtppass = topsecret
```

For mail.ru users, the configuration will be slightly different:

```
[sendemail]
    smtpencryption = ssl
    smtpserver = smtp.mail.ru
    smtpserverport = 465
    smtpuser = your.name@mail.ru
    smtppass = topsecret
```

If your email account is hosted by another service, consult your service provider about your SMTP settings.

Once configured, use the following command to send your patches:

```
$ git send-email --to tarantool-patches@freelists.org 00*
```

(00* wildcard will be expanded by your shell to the list of patches generated at the previous step.)

If you want someone in particular to review your patch, add them to the list of recipients by passing --to or --cc once per each recipient.

Note: It is useful to check that git send-email will work as expected without sending anything to the world. Use --dry-run option for that.

---

3. Review process

After having sent your patches, you just wait for a review. The reviewer will send their comments back to you in reply to the email that contains the patch that in their opinion needs to be fixed.

Upon receiving an email with review remarks, you carefully read it and reply about whether you agree or disagree with. Please note that we use the interleaved reply style (aka “inline reply”) for communications over email.

Upon reaching an agreement, you send a fixed patch in reply to the email that ended the discussion. To send a patch, you can either attach a plain diff (created by git diff or git format-patch) to email and send it with your favorite mail agent, or use --in-reply-to option of git send-email command.

If you feel that the accumulated change set is large enough to send the whole series anew and restart the review process in a different thread, you generate the patch email(s) again with git format-patch, this time adding v2 (then v3, v4, and so forth) to the subject and a change log to the message body. To modify the subject line accordingly, use the --subject-prefix option to git format-patch command:

```
$ git format-patch -1 --subject-prefix='PATCH v2'
```

To add a change log, open the generated email with you favorite text editor and edit the message body. If there is just one patch, the change log should go right after --- in the message body (it will be ignored by

git am then). If there is more than one patch, the change log should be added to the cover letter. Here is an example of a good change log:

```
Changes in v3:
 - Fixed comments as per review by Alex
 - Added more tests
Changes in v2:
 - Fixed a crash if the user passes invalid options
 - Fixed a memory leak at exit
```

It is also a good practice to add a reference to the previous version of your patch set (via a hyperlink or message id).

Note:

- Do not disagree with the reviewer without providing a good argument supporting your point of view.
- Do not take every word the reviewer says for granted. Reviewers are humans too, hence fallible.
- Do not expect that the reviewer will tell you how to do your thing. It is not their job. The reviewer might suggest alternative ways to tackle the problem, but in general it is your responsibility.
- Do not forget to update your remote git branch every time you send a new version of your patch.
- Do follow the guidelines above. If you do not comply, your patches are likely to be silently ignored.

7.4.2 Documentation guidelines

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.

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).

Language and style issues

US vs British spelling

We use English US spelling.

Instance vs server

We say “instance” rather than “server” to refer to an instance of Tarantool server. This keeps the manual terminology consistent with names like /etc/tarantool/instances.enabled in the Tarantool environment.

Wrong usage: “Replication allows multiple Tarantool servers to work on copies of the same databases.”

Correct usage: “Replication allows multiple Tarantool instances to work on copies of the same databases.”

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.

Parameters

- 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.

Parameters

- 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.

7.4.3 C Style Guide

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.

A. 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.

B. The rest of Linux Kernel Coding Style is amended as follows:

General guidelines

We use Git for revision control. The latest development is happening in the default branch (currently 2.1). 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

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, reduced to 66 columns for comments, 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:

(a) 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.

(b) 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.

(c) when you use sparse to literally create a _new_ type for type-checking.

(d) 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.

(e) 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. c 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)   \
    do {                    \
        if (a == 5)         \
            do_this(b, c);  \
    } while (0)
```

Things to avoid when using macros:

1. macros that affect control flow:

   ```
   #define FOO(x)                  \
       do {                        \
           if (blah(x) < 0)        \
               return -EBUGGERED;  \
       } 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
   #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

7.4.4 Python Style Guide

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 Python[1].

This document and PEP 257 (Docstring Conventions) were adapted from Guido's original Python Style Guide essay, with some additions from Barry's style guide[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!

[1] PEP 7, Style Guide for C Code, van Rossum
[2] Barry's GNU Mailman style guide

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).

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,
```

(continues on next page)

(continued from previous page)

```
    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".

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()
```

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.

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.

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 letters[3]). This is also sometimes known as StudlyCaps.

  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).

[3] CamelCase Wikipedia page

- __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

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 clss. (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, not withstanding 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.

References

Copyright

Author:

- Guido van Rossum <guido@python.org>
- Barry Warsaw <barry@python.org>

7.4.5 Lua Style Guide

Inspiration:

- https://github.com/Olivine-Labs/lua-style-guide
- http://dev.minetest.net/Lua_code_style_guidelines
- http://sputnik.freewisdom.org/en/Coding_Standard

Programming style is an art. There is some arbitrariness to the rules, but there are sound rationales for them. It is useful not only to provide sound advice on style but to understand the underlying rationale and human aspect of why the style recommendations are formed:

- http://mindprod.com/jgloss/unmain.html
- http://www.oreilly.com/catalog/perlbp/
- http://books.google.com/books?id=QnghAQAAIAAJ

Zen of Python is good; understand it and use wisely:

> Beautiful is better than ugly.
> Explicit is better than implicit.
> Simple is better than complex.
> Complex is better than complicated.
> Flat is better than nested.
> Sparse is better than dense.
> Readability counts.
> Special cases aren't special enough to break the rules.
> Although practicality beats purity.
> Errors should never pass silently.
> Unless explicitly silenced.
> In the face of ambiguity, refuse the temptation to guess.
> There should be one – and preferably only one – obvious way to do it.
> Although that way may not be obvious at first unless you're Dutch.
> Now is better than never.
> Although never is often better than right now.
> If the implementation is hard to explain, it's a bad idea.
> If the implementation is easy to explain, it may be a good idea.
> Namespaces are one honking great idea – let's do more of those!
>
> https://www.python.org/dev/peps/pep-0020/

Indentation and Formatting

- 4 spaces instead tabs. PIL suggests using of two spaces, but programmer looks at code 4 up to 8 hours a day, so it's simplier to distinguish indentation with 4 spaces. Why spaces? Similar representation everywhere.

  You can use vim modelines:

```
-- vim:ts=4 ss=4 sw=4 expandtab
```

- A file should ends w/ one newline symbol, but shouldn't ends w/ blank line (two newline symbols).
- Every do/while/for/if/function should indent 4 spaces.
- related or/and in if must be enclosed in the round brackets (). Example:

```
if (a == true and b == false) or (a == false and b == true) then
    <...>
end -- good
```

(continues on next page)

(continued from previous page)

```
if a == true and b == false or a == false and b == true then
    <...>
end -- bad

if a ^ b == true then
end -- good, but not explicit
```

- Type conversion

  Do not use concatenation to convert to string or addition to convert to number (use tostring/tonumber instead):

```
local a = 123
a = a .. ''
-- bad

local a = 123
a = tostring(a)
-- good

local a = '123'
a = a + 5 -- 128
-- bad

local a = '123'
a = tonumber(a) + 5 -- 128
-- good
```

- Try to avoid multiple nested if's with common body:

```
if (a == true and b == false) or (a == false and b == true) then
    do_something()
end
-- good

if a == true then
    if b == false then
        do_something()
    end
if b == true then
    if a == false then
        do_something()
    end
end
-- bad
```

- Avoid multiple concatenations in one statement, use string.format instead:

```
function say_greeting(period, name)
    local a = "good  " .. period .. ", " .. name
end
-- bad

function say_greeting(period, name)
    local a = string.format("good %s, %s", period, name)
end
-- good
```

(continues on next page)

(continued from previous page)

```
local say_greeting_fmt = "good %s, %s"
function say_greeting(period, name)
    local a = say_greeting_fmt:format(period, name)
end
-- best
```

- Use and/or for default variable values

```
function(input)
    input = input or 'default_value'
end -- good

function(input)
    if input == nil then
        input = 'default_value'
    end
end -- ok, but excessive
```

- if's and return statements:

```
if a == true then
    return do_something()
end
do_other_thing() -- good

if a == true then
    return do_something()
else
    do_other_thing()
end -- bad
```

- Using spaces:
  - one shouldn't use spaces between function name and opening round bracket, but arguments must be splitted with one whitespace charachter

    ```
    function name (arg1,arg2,...)
    end -- bad

    function name(arg1, arg2, ...)
    end -- good
    ```

  - use space after comment marker

    ```
    while true do -- inline comment
    -- comment
    do_something()
    end
    --[[
      multiline
      comment
    ]]--
    ```

  - surrounding operators

```
local thing=1
thing = thing-1
thing = thing*1
thing = 'string'..'s'
-- bad

local thing = 1
thing = thing - 1
thing = thing * 1
thing = 'string' .. 's'
-- good
```

- use space after commas in tables

```
local thing = {1,2,3}
thing = {1 , 2 , 3}
thing = {1 ,2 ,3}
-- bad

local thing = {1, 2, 3}
-- good
```

- use space in map definitions around equality sign and commas

```
return {1,2,3,4} -- bad
return {
    key1 = val1,key2=val2
} -- bad

return {
    1, 2, 3, 4
    key1 = val1, key2 = val2,
    key3 = vallll
} -- good
```

also, you may use alignment:

```
return {
    long_key  = 'vaaaaalue',
    key       = 'val',
    something = 'even better'
}
```

- 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 function, sparingly, to indicate logical sections

```
if thing then
    -- ...stuff...
end
function derp()
    -- ...stuff...
end
local wat = 7
-- bad
```

(continues on next page)

(continued from previous page)

```
if thing then
    -- ...stuff...
end

function derp()
    -- ...stuff...
end

local wat = 7
-- good
```

- Delete whitespace at EOL (strongly forbidden. Use :s/\s\+$//gc in vim to delete them)

Avoid global variable

You must avoid global variables. If you have an exceptional case, use _G variable to set it, add prefix or add table instead of prefix:

```
function bad_global_example()
end -- very, very bad

function good_local_example()
end
_G.modulename_good_local_example = good_local_example -- good
_G.modulename = {}
_G.modulename.good_local_example = good_local_example -- better
```

Always use prefix to avoid name clash

Naming

- names of variables/"objects" and "methods"/functions: snake_case
- names of "classes": CamelCase
- private variables/methods (properties in the future) of object starts with underscores <object>._<name>. Avoid using of local function private_methods(self) end
- boolean - naming is_<...>, isnt_<...>, has_, hasnt_ is a good style.
- for "very local" variables: - t is for tables - i, j are for indexing - n is for counting - k, v is what you get out of pairs() (are acceptable, _ if unused) - i, v is what you get out of ipairs() (are acceptable, _ if unused) - k/key is for table keys - v/val/value is for values that are passed around - x/y/z is for generic math quantities - s/str/string is for strings - c is for 1-char strings - f/func/cb are for functions - status, <rv>.. or ok, <rv>.. is what you get out of pcall/xpcall - buf, sz is a (buffer, size) pair - <name>_p is for pointers - t0.. is for timestamps - err is for errors
- abbrevations are acceptable if they're unambigous and if you'll document (or they're too common) them.
- global variables are written with ALL_CAPS. If it's some system variable, then they're using underscore to define it (_G/_VERSION/..)
- module naming snake_case (avoid underscores and dashes) - 'luasql', instead of 'Lua-SQL'
- *_mt and *_methods defines metatable and methods table

Idioms and patterns

Always use round brackets in call of functions except multiple cases (common lua style idioms):

- *.cfg{ } functions (box.cfg/memcached.cfg/..)
- ffi.cdef[[ ]] function

Avoid these kind of constructions:

- <func>'<name>' (strongly avoid require'..')
- function object:method() end (use functon object.method(self) end instead)
- do not use semicolon as table separator (only comma)
- semicolons at the end of line (only to split multiple statements on one line)
- try to avoid unnecessary function creation (closures/..)

Modules

Don't start modules with license/authors/descriptions, you can write it in LICENSE/AUTHORS/README files. For writing modules use one of the two patterns (dont use modules()):

```
local M = {}

function M.foo()
...
end

function M.bar()
...
end

return M
```

or

```
local function foo()
...
end

local function bar()
...
end

return {
foo = foo,
bar = bar,
}
```

Commenting

You should write code the way it shouldn't be described, but don't forget about commenting it. You shouldn't comment Lua syntax (assume that reader already knows Lua language). Try to tell about functions/variable names/etc.

Multiline comments: use matching (--[[ ]]--) instead of simple (--[[ ]]).

Public function comments (??):

```
--- Copy any table (shallow and deep version)
-- * deepcopy: copies all levels
-- * shallowcopy: copies only first level
-- Supports __copy metamethod for copying custom tables with metatables
-- @function gsplit
-- @table          inp  original table
-- @shallow[opt]   sep  flag for shallow copy
-- @returns             table (copy)
```

Testing

Use tap module for writing efficient tests. Example of test file:

```
#!/usr/bin/env tarantool

local test = require('tap').test('table')
test:plan(31)

do -- check basic table.copy (deepcopy)
    local example_table = {
        {1, 2, 3},
        {"help, I'm very nested", {{{ }}} }
    }

    local copy_table = table.copy(example_table)

    test:is_deeply(
        example_table,
        copy_table,
        "checking, that deepcopy behaves ok"
    )
    test:isnt(
        example_table,
        copy_table,
        "checking, that tables are different"
    )
    test:isnt(
        example_table[1],
        copy_table[1],
        "checking, that tables are different"
    )
    test:isnt(
        example_table[2],
        copy_table[2],
        "checking, that tables are different"
    )
    test:isnt(
        example_table[2][2],
        copy_table[2][2],
        "checking, that tables are different"
    )
    test:isnt(
        example_table[2][2][1],
        copy_table[2][2][1],
```

(continues on next page)

(continued from previous page)

```
        "checking, that tables are different"
    )
end

<...>

os.exit(test:check() == true and 0 or 1)
```

When you'll test your code output will be something like this:

```
TAP version 13
1..31
ok - checking, that deepcopy behaves ok
ok - checking, that tables are different
ok - checking, that tables are different
ok - checking, that tables are different
ok - checking, that tables are different
ok - checking, that tables are different
...
```

Error Handling

Be generous in what you accept and strict in what you return.

With error handling this means that you must provide an error object as second multi-return value in case of error. The error object can be a string, a Lua table or cdata, in the latter cases it must have __tostring metamethod defined.

In case of error, use nil for the first return value. This makes the error hard to ignore.

When checking function return values, check the first argument first. If it's nil, look for error in the second argument:

```
local data, err = foo()
if not data
    return nil, err
end
return bar(data)
```

Unless performance of your code is paramount, try to avoid using more than two return values.

In rare cases you may want to return nil as a legal return value. In this case it's OK to check for error first, and return second:

```
local data, err = foo()
if not err
    return data
end
return nil, err
```

Lua Module Index

x

y