Tarantool

Release 1.7.6

Mar 14, 2019

Contents

Here is a summary of significant changes introduced in specific versions of Tarantool.

For smaller feature changes and bug fixes, see closed milestones at GitHub.

CHAPTER 1

What's new in Tarantool 1.7.6?

Tarantool 1.7.6 was released on November 7 2017.

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, base64, and index creation.

CHAPTER 2

What's new in Tarantool 1.7?

The disk-based storage engine, which was called sophia or phia in earlier versions, is superseded by the vinyl storage engine.

There are new types for indexed fields.

The LuaJIT version is updated.

Automatic 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:

- Parameters 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.

CHAPTER 3

What's new in Tarantool 1.6.9 after 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 It will be superseded in version 1.7 by the vinyl storage engine.

CHAPTER 4

What's new in Tarantool 1.6?

Tarantool 1.6 is no longer getting major new features, although it will be maintained. The developers are concentrating on Tarantool version 1.7.

CHAPTER 5

Overview

5.1 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.

5.2 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.

CHAPTER 6

User's Guide

6.1 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.

6.1.1 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.

6.1.2 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).

6.1.3 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.

6.2 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:

6.2.1 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 or 1.7) 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:1.7
```

This command runs a new container named 'mytarantool'. Docker starts it from an official image named 'tarantool/tarantool:1.7', with Tarantool version 1.7 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') and the first index (named 'primary'):

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

Next, insert three tuples (our name for "records") into the space:

```
tarantool.sock> t = s:insert({1, 'Roxette'})
tarantool.sock> t = s:insert({2, 'Scorpions', 2015})
tarantool.sock> t = s:insert({3, 'Ace of Base', 1993})
```

To select a tuple from the first space of the database, using the first defined key, say:

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

The terminal screen now looks like this:

```
tarantool.sock> s = box.schema.space.create('tester')
2017-01-17 12:04:18.158 ... creating './00000000000000000000.xlog.inprogress'
---
...
tarantool.sock> s:create_index('primary', {type = 'hash', parts = {1, 'unsigned'}})
---
...
tarantool.sock> t = s:insert{1, 'Roxette'}
---
...
tarantool.sock> t = s:insert{2, 'Scorpions', 2015}
---
...
tarantool.sock> t = s:insert{3, 'Ace of Base', 1993}
```

(continues on next page)

(continued from previous page)

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

To add another index on the second field, say:

```
tarantool.sock> s:create_index('secondary', {
              >   type = 'hash',
              >   parts = {2, 'string'}
              > })
```

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.

6.2.2 Using a binary package

For production purposes, we recommend official binary packages. You can choose from two Tarantool versions: 1.7 (stable) or 1.8 (alpha). An automatic build system creates, tests and publishes packages for every push into a corresponding branch (1.7 or 1.8) 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 installing.

Create a new directory (it's just for tests, so you can delete it when the tests are over):

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

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

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

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

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

Next, insert three tuples (our name for "records") into the space:

```
tarantool> t = s:insert({1, 'Roxette'})
tarantool> t = s:insert({2, 'Scorpions', 2015})
tarantool> t = s:insert({3, 'Ace of Base', 1993})
```

To select a tuple from the first space of the database, using the first defined key, say:

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

The terminal screen now looks like this:

```
tarantool> s = box.schema.space.create('tester')
2017-01-17 12:04:18.158 ... creating './00000000000000000000.xlog.inprogress'
---
...
tarantool>s:create_index('primary', {type = 'hash', parts = {1, 'unsigned'}})
---
...
tarantool> t = s:insert{1, 'Roxette'}
---
...
tarantool> t = s:insert{2, 'Scorpions', 2015}
---
...
tarantool> t = s:insert{3, 'Ace of Base', 1993}
---
...
tarantool> s:select{3}
---
- - [3, 'Ace of Base', 1993]
```

(continues on next page)

(continued from previous page)

```
...
tarantool>
```

To add another index on the second field, say:

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

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:

```
tarantool> 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]
...

localhost:3301>
```

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

6.3 Database

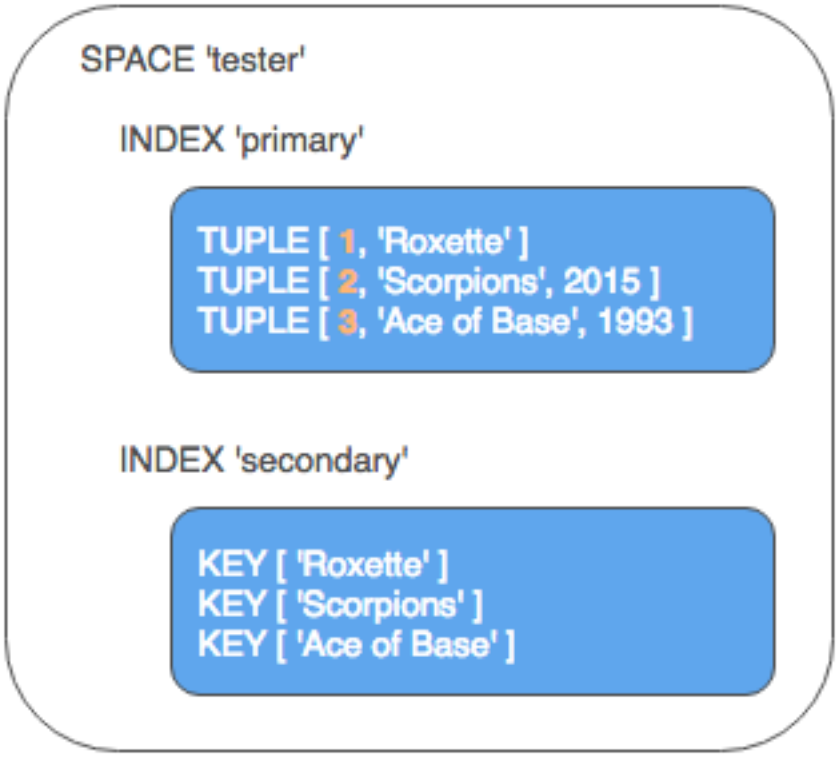
In this chapter, we introduce the basic concepts of working with Tarantool as a database manager.

This chapter contains the following sections:

6.3.1 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
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').

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}
---
- [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'
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	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) Note: When there is a mix of types, the key order is: booleans, then numbers, then strings.	TREE or HASH	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 – unicode and unicode_s1 – that allow for `'A' < 'a' < 'B'` and `'A' = 'a' < 'B'` respectively.

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 four weight levels, from L1 to L4,

- unicode_s1 collation observes only L1 weights.

As an example, let's 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_s1 collation:

```
tarantool> box.space.T:create_index('I', {parts = {{1,'str', collation='unicode_s1'}}})
...
tarantool> box.space.S.index.I:select()
---
- - ['ЕЛЕ']
  - ['елейный']
  - ['ёлка']
  - ['еловый']
  - ['елозить']
  - ['Ёлочка']
  - ['ёлочный']
  - ['ЕЛь']
...
tarantool> box.space.S.index.I:select{'ЁлКа'}
---
- - ['ёлка']
...
```

In fact, though, good collation involves much more than these simple examples of upper case / lower case equivalence in alphabets. We also consider accent marks, non-alphabetic writing systems, and special rules

that apply for combinations of characters.

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

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:

- one data-retrieval operation (SELECT), and

- five data-manipulation operations (INSERT, UPDATE, UPSERT, DELETE, REPLACE).

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 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}
```

All the functions operate on tuples and accept only unique key values. So, the number of tuples in the space is always 0 or 1, since the keys are unique.

Functions insert(), upsert() and replace() accept only primary-key values. Functions select(), delete() and update() may accept either a primary-key value or a secondary-key value.

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.

6.3.2 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 an SQL statement like:

```
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.

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 operations usually do not yield, but it depends on the engine:

- In memtx, reads or writes do not require I/O and do not yield.
- In vinyl, not all data is in memory, and SELECT often incurs a disc I/O, and therefore yields, while a write may stall waiting for memory to free up, thus also causing a yield.

In the "autocommit" mode, all data change operations are followed by an automatic commit, which yields. So does an explicit commit of a multi-statement transaction, box.commit().

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.

6.3.3 Access control

Understanding security details is primarily an issue for administrators. Meanwhile, 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.

In a nutshell:

- 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").

Further on, we explain all of this in more detail.

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 AUTH protocol command, supported by most clients;
- For an admin-console connection and in a Lua initialization script – with box.session.su;
- For a stored function invoked with CALL command over a binary port – with SETUID property enabled for the function, which makes Tarantool temporarily replace the current user with the function's creator, with all 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

In Tarantool, all objects are organized into a hierarchy of ownership. Ordinarily the owner of every object is its creator. The creator of the initial database state (we call it 'universe') – including the database itself, the system spaces, the users – is 'admin'.

An object's owner can share some rights on the object by granting privileges to other users. The following privileges are implemented:

- Read an object,
- Write, i.e. modify contents of an object,
- Execute, i.e. use an object (if the privilege makes sense for the object; for example, spaces can not be "executed", but functions can).

Note: Currently, "drop" and "grant" privileges can not be granted to other users. This possibility will be added in future versions of Tarantool.

This is how the privilege system works under the hood. To be able to create objects, a user needs to have write access to Tarantool's system spaces. The 'admin' user, who is at the top of the hierarchy and who is the ultimate source of privileges, shares write access to a system space (e.g. _space) with some users. Now the users can insert data into the system space (e.g. creating new spaces) and themselves become creators/definers of new objects. For the objects they created, the users can in turn share privileges with other users.

This is why only an object's owner can drop the object, but other ordinary users cannot. Meanwhile, 'admin' can drop any object or delete any other user, because 'admin' is the creator and ultimate owner of them all.

The syntax of all grant()/revoke() commands in Tarantool follows this basic idea.

- Their first argument is the user who gets the grant or whose grant is revoked.
- Their second argument is the type of privilege granted, or a list of privileges.
- Their third argument is the object type on which the privilege is granted, or the word 'universe'. Possible object types are 'space', 'function', 'user', 'role', 'sequence'.
- Their fourth argument is the object name if the object type was specified ('universe' has no name because there is only one 'universe', but otherwise you must specify the name).

Example #1

Here we say that user 'guest' can do common operations on any object.

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

Example #2

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', '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.

Example

```
-- This example will work for a user with many privileges, such as 'admin'
-- 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)
box.schema.role.grant('R1', 'execute', 'role', 'R2')
box.schema.user.grant('U1', 'execute', 'role', '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 details about Tarantool functions related to role management, see reference on box.schema submodule.

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.

6.3.4 Triggers

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

There are three types of triggers in Tarantool:

- connection triggers, which are executed when a session begins or ends,
- authentication triggers, which are executed during authentication, and
- replace triggers, which are for database events.

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(), box.session.on_auth(), box.session.on_disconnect(), or space_object:on_replace().

- 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.
- 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:

- on_connect() – with no arguments – to return a table of all connect-trigger functions;
- on_auth() to return all authentication-trigger functions;
- on_disconnect() to return all disconnect-trigger functions;
- on_replace() to return all replace-trigger functions.

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)
```

6.3.5 Limitations

Number of parts in an index

For TREE or HASH indexes, the maximum is 255 (box.schema.INDEX_PART_MAX). For ref:RTREE <box_index-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 (box.schema.REPLICA_MAX).

6.4 Application server

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

This chapter contains the following sections:

6.4.1 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()
---
- |
 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 without any application:

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

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 tarantool /opt/tarantool/myapp.lua
```

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

- our application file (\`pwd\`/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 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:

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

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:

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 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):

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

6.4.2 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"}
                    }
                }
            }
        }
    }
}
```

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
```

(continues on next page)

(continued from previous page)

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

(continues on next page)

(continued from previous page)

```
        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
        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}}
    )
    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}}
```

(continues on next page)

(continued from previous page)

```
        )
    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
        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
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
```

(continues on next page)

(continued from previous page)

```
    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 1.7 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
    }
}
local player2 = {
    name="Player2",
    id=2,
    location = {
        x=30.123,
        y=40.456
    }
}

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

(continues on next page)

(continued from previous page)

```
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

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 1.7 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.

6.4.3 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> local-name = require('module-name')
```

- 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').

6.4.4 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.

6.4.5 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")
    ...
  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)
```

(continues on next page)

(continued from previous page)

```
  -- 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')

-- 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.

6.4.6 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.

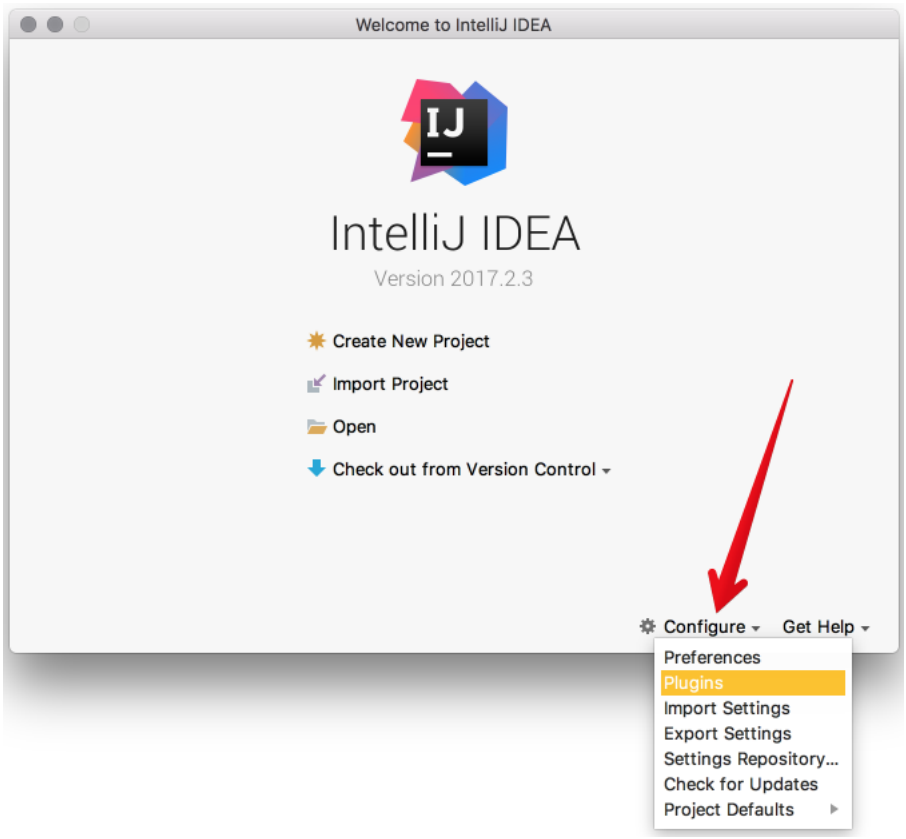
   JetBrains provides specialized editions for particular languages: IntelliJ IDEA (Java), PHPStorm (PHP), PyCharm (Python), RubyMine (Ruby), CLion (C/C++), WebStorm (Web) and others. So, download a version that suits your primary programming language.

   Tarantool integration is supported for all editions.

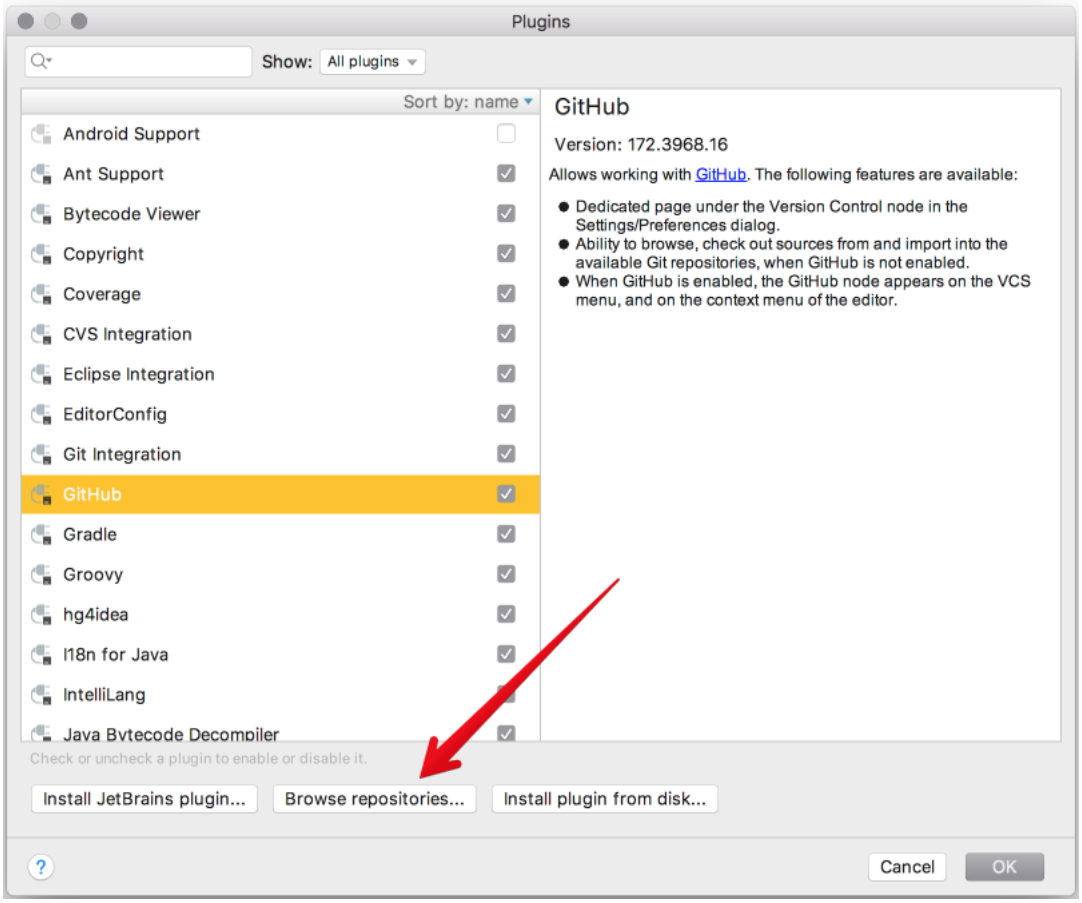
2. Configure the IDE:

   a. Start IntelliJ IDEA.

   b. Click Configure button and select Plugins.

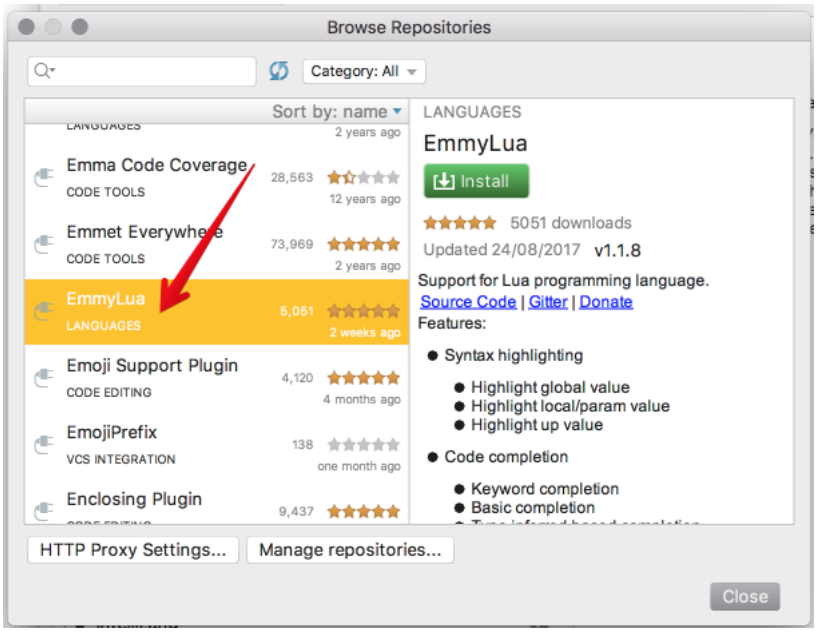


   c. Click Browse repositories.



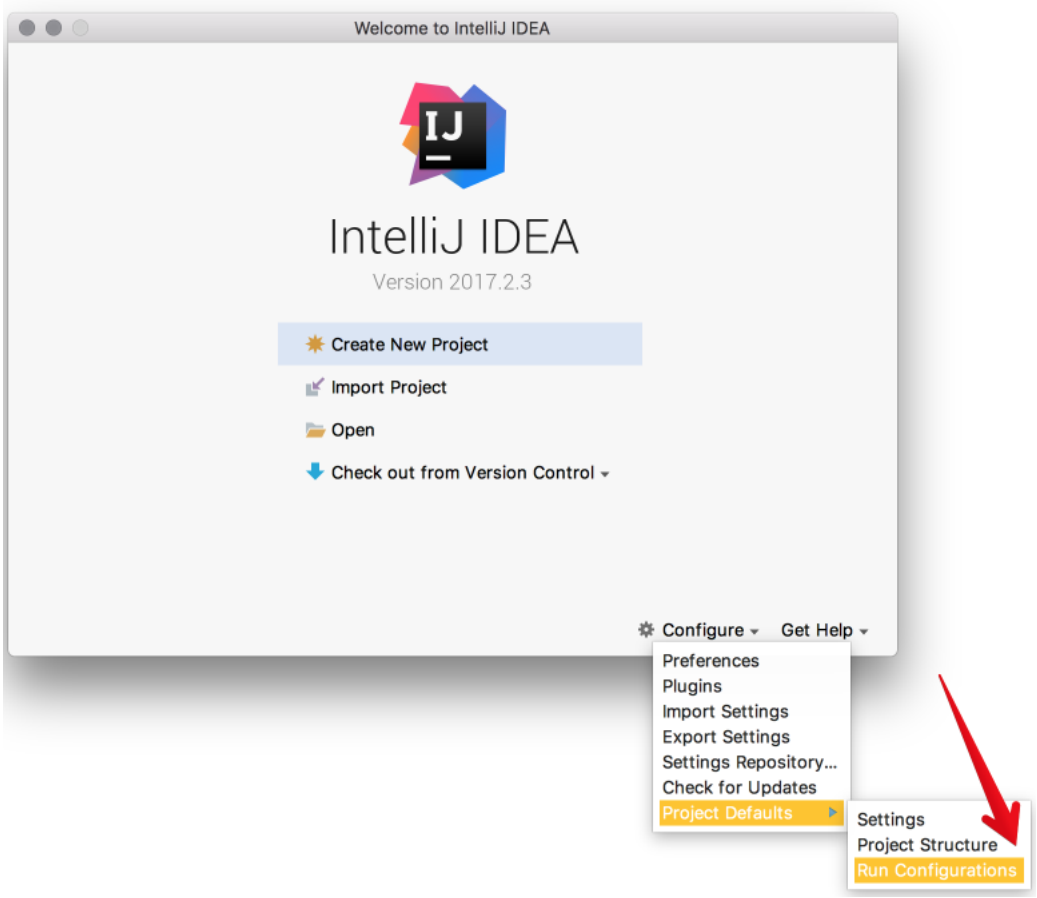
d. Install EmmyLua plugin.

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.

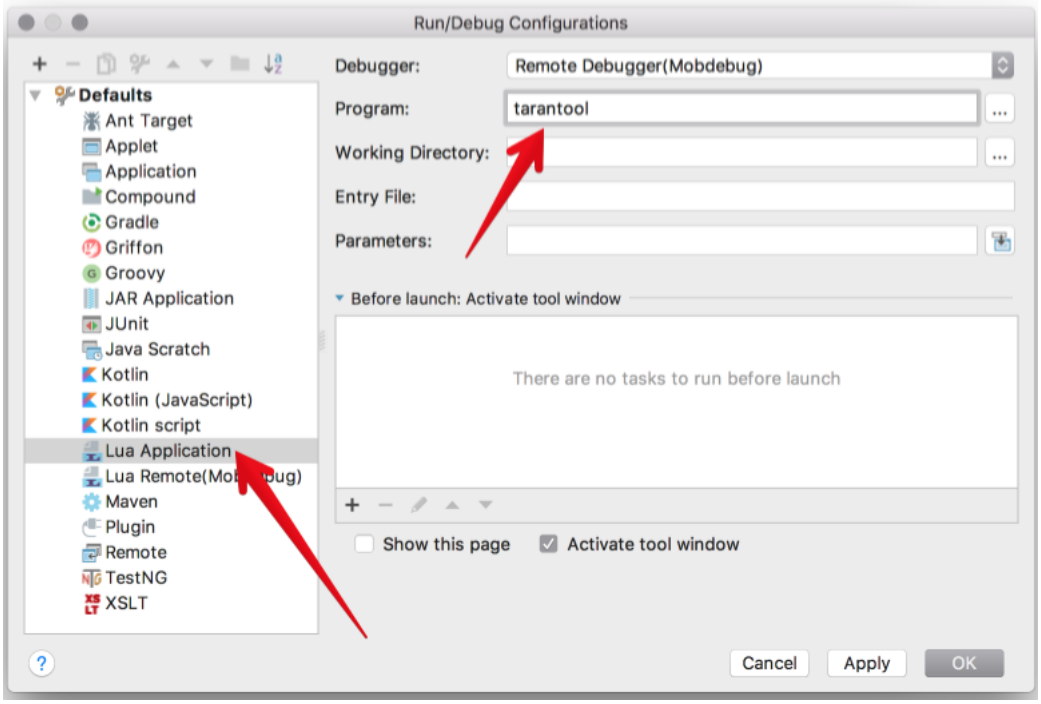


g. Find Lua Application in the sidebar at the left.

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

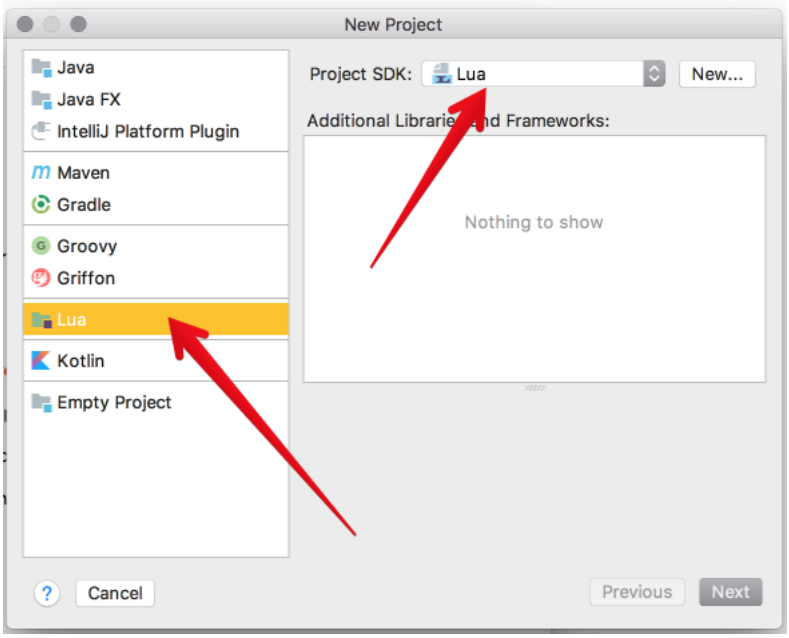
   By default, this is tarantool or /usr/bin/tarantool on most platforms.

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

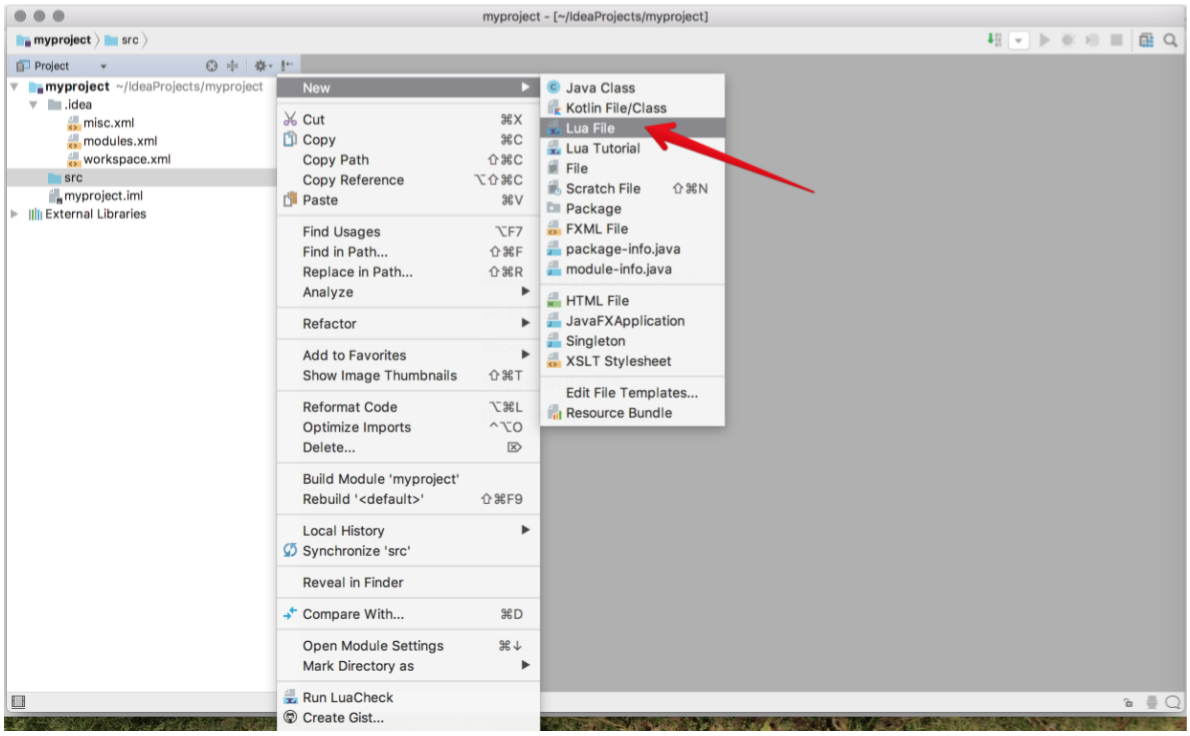


Now IntelliJ IDEA is ready to use with Tarantool.

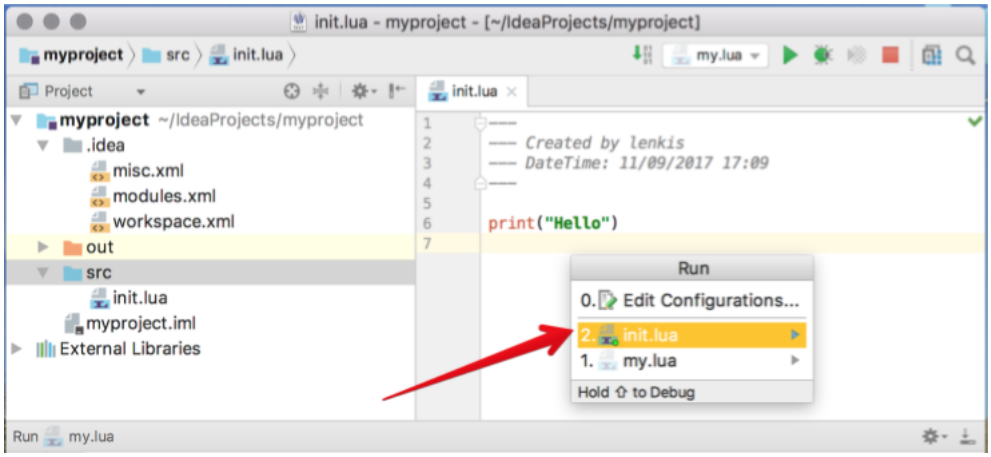
3. Create a new Lua project.



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

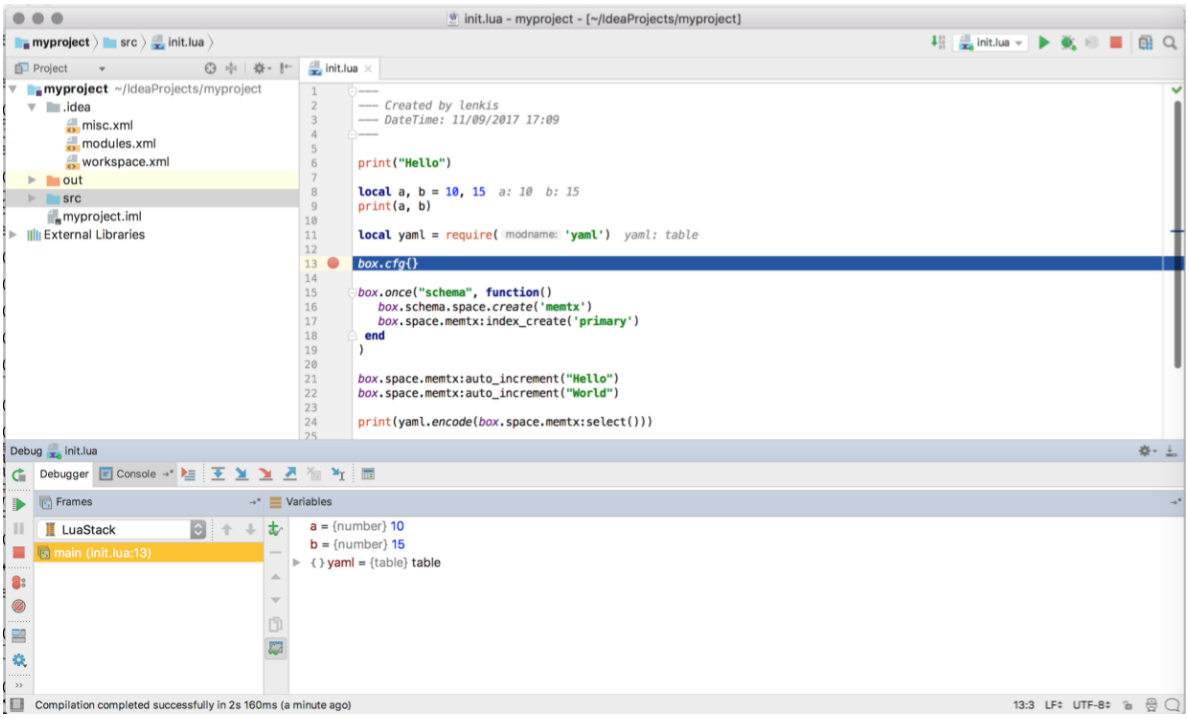


5. Write your code, save the file.
6. To run you application, click Run -> Run in the main menu and select your source file in the list.



Or click Run -> Debug to start debugging.

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



6.4.7 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.

Use freely.

hello_world.lua

The standard example of a simple program.

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

print('Hello, World!')
```

console_start.lua

Use box.once() to initialize a database (creating spaces) if this is the first time the server has been run. Then use console.start() to start interactive mode.

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

-- Configure database
box.cfg {
    listen = 3313
}

box.once("bootstrap", function()
    box.schema.space.create('tweedledum')
    box.space.tweedledum:create_index('primary',
        { type = 'TREE', parts = {1, 'unsigned'}})
end)

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

fio_read.lua

Use the fio module to open, read, and close a file.

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

local fio = require('fio')
local errno = require('errno')
local f = fio.open('/tmp/xxxx.txt', {'O_RDONLY' })
if not f then
    error("Failed to open file: "..errno.strerror())
end
local data = f:read(4096)
f:close()
print(data)
```

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)
```

(continues on next page)

(continued from previous page)

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

-- Lua wrapper for uncompress
local function uncompress(comp, n)
    local buf = ffi.new("uint8_t[?]", n)
    local buflen = ffi.new("unsigned long[1]", n)
    local res = zlib.uncompress(buf, buflen, comp, #comp)
    assert(res == 0)
    return ffi.string(buf, buflen[0])
end

-- Simple test code.
local txt = string.rep("abcd", 1000)
print("Uncompressed size: ", #txt)
local c = compress(txt)
print("Compressed size: ", #c)
local txt2 = uncompress(c, #txt)
assert(txt2 == txt)
```

ffi_meta.lua

Use the LuaJIT ffi library to access a C object via a metamethod (a method which is defined with a metatable).

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

local ffi = require("ffi")
ffi.cdef[[
typedef struct { double x, y; } point_t;
]]

local point
local mt = {
  __add = function(a, b) return point(a.x+b.x, a.y+b.y) end,
  __len = function(a) return math.sqrt(a.x*a.x + a.y*a.y) end,
  __index = {
    area = function(a) return a.x*a.x + a.y*a.y end,
  },
}
point = ffi.metatype("point_t", mt)

local a = point(3, 4)
print(a.x, a.y)  --> 3  4
print(#a)        --> 5
print(a:area())  --> 25
local b = a + point(0.5, 8)
print(#b)        --> 12.5
```

print_arrays.lua

Create Lua tables, and print them. Notice that for the 'array' table the iterator function is ipairs(), while for the 'map' table the iterator function is pairs(). (ipairs() is faster than pairs(), but pairs() is recommended for

map-like tables or mixed tables.) The display will look like: "1 Apple | 2 Orange | 3 Grapefruit | 4 Banana | k3 v3 | k1 v1 | k2 v2".

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

array = { 'Apple', 'Orange', 'Grapefruit', 'Banana'}
for k, v in ipairs(array) do print(k, v) end

map = { k1 = 'v1', k2 = 'v2', k3 = 'v3' }
for k, v in pairs(map) do print(k, v) end
```

count_array.lua

Use the '#' operator to get the number of items in an array-like Lua table. This operation has O(log(N)) complexity.

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

array = { 1, 2, 3}
print(#array)
```

count_array_with_nils.lua

Missing elements in arrays, which Lua treats as "nil"s, cause the simple "#" operator to deliver improper results. The "print(#t)" instruction will print "4"; the "print(counter)" instruction will print "3"; the "print(max)" instruction will print "10". Other table functions, such as table.sort(), will also misbehave when "nils" are present.

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

local t = {}
t[1] = 1
t[4] = 4
t[10] = 10
print(#t)
local counter = 0
for k,v in pairs(t) do counter = counter + 1 end
print(counter)
local max = 0
for k,v in pairs(t) do if k > max then max = k end end
print(max)
```

count_array_with_nulls.lua

Use explicit NULL values to avoid the problems caused by Lua's nil == missing value behavior. Although json.NULL == nil is true, all the print instructions in this program will print the correct value: 10.

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

local json = require('json')
local t = {}
t[1] = 1; t[2] = json.NULL; t[3]= json.NULL;
t[4] = 4; t[5] = json.NULL; t[6]= json.NULL;
```

(continues on next page)

(continued from previous page)

```
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
```

(continues on next page)

(continued from previous page)

```
local myclass_mt = {
    __tostring = myclass_tostring;
    __index = {
        somemethod = myclass_somemethod;
        someothermethod = myclass_someothermethod;
    }
}

-- create a new object of myclass
local object = setmetatable({ data = 'data'}, myclass_mt)
print(object:somemethod())
print(object.data)
```

garbage.lua

Force Lua garbage collection with the collectgarbage function.

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

collectgarbage('collect')
```

fiber_producer_and_consumer.lua

Start one fiber for producer and one fiber for consumer. Use fiber.channel() to exchange data and synchronize. One can tweak the channel size (ch_size in the program code) to control the number of simultaneous tasks waiting for processing.

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

local fiber = require('fiber')
local function consumer_loop(ch, i)
    -- initialize consumer synchronously or raise an error()
    fiber.sleep(0) -- allow fiber.create() to continue
    while true do
        local data = ch:get()
        if data == nil then
            break
        end
        print('consumed', i, data)
        fiber.sleep(math.random()) -- simulate some work
    end
end

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
```

(continues on next page)

(continued from previous page)

```
local function start()
    local consumer_n = 5
    local producer_n = 3

    -- Create a channel
    local ch_size = math.max(consumer_n, producer_n)
    local ch = fiber.channel(ch_size)

    -- Start consumers
    for i=1, consumer_n,1 do
        fiber.create(consumer_loop, ch, i)
    end

    -- Start producers
    for i=1, producer_n,1 do
        fiber.create(producer_loop, ch, i)
    end
end

start()
print('started')
```

socket_tcpconnect.lua

Use socket.tcp_connect() to connect to a remote host via TCP. Display the connection details and the result of a GET request.

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

local s = require('socket').tcp_connect('google.com', 80)
print(s:peer().host)
print(s:peer().family)
print(s:peer().type)
print(s:peer().protocol)
print(s:peer().port)
print(s:write("GET / HTTP/1.0\r\n\r\n"))
print(s:read('\r\n'))
print(s:read('\r\n'))
```

socket_tcp_echo.lua

Use socket.tcp_connect() to set up a simple TCP server, by creating a function that handles requests and echos them, and passing the function to socket.tcp_server(). This program has been used to test with 100,000 clients, with each client getting a separate fiber.

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

local function handler(s, peer)
    s:write("Welcome to test server, " .. peer.host .."\n")
    while true do
        local line = s:read('\n')
        if line == nil then
```

(continues on next page)

(continued from previous page)

```
            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)
```

(continues on next page)

(continued from previous page)

```
        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)
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')
```

(continues on next page)

(continued from previous page)

```
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

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 '*'
```

(continues on next page)

(continued from previous page)

```
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>
```

6.5 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:

6.5.1 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
  ```

(continues on next page)

(continued from previous page)

```
return {
  start = start;
}
```

- An instance file with instance-specific initialization logic and parameters. Put this file, or a symlink to it, into the /etc/tarantool/instances.enabled directory.

  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, tarantoolctl configuration file contains the configuration that tarantoolctl uses to override instance configuration. In other words, it contains system-wide configuration defaults.

Most of the parameters are similar to those used by box.cfg{}. Here are the default settings (installed to /etc/default/tarantool as part of Tarantool distribution):

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

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.
- instance_dir
  The directory where all instance files for this host are stored. Put instance files in this directory, or create symbolic links.

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

6.5.2 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.

   The default tarantoolctl configuration file is installed in /etc/default/tarantool. This file is used when tarantoolctl is invoked by root. When invoked by a local user, tarantoolctl first looks for its defaults

file in the current directory ($PWD/.tarantoolctl), and then in the current user's home directory ($HOME/.config/tarantool/tarantool). If not found, tarantoolctl falls back to built-in defaults.

3. Look up the instance file in the instance directory, e.g. /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. 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)
    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
```

6.5.3 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.

Note: 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.

6.5.4 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 1.7.3 (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.

6.5.5 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
---
```

(continues on next page)

(continued from previous page)

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

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()
---
- 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
```

(continues on next page)

(continued from previous page)

```
  101:
    csw: 284
    backtrace: []
    fid: 101
    memory:
      total: 57656
      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
#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
```

(continues on next page)

(continued from previous page)

```
#7 0x00000000004766b5 in vy_read_iterator_next (itr=itr@entry=0x7f72264af990,␣
↪result=result@entry=0x7f72264afad8) at /usr/src/tarantool/src/box/vinyl.c:8592
#8 0x000000000047689d in vy_index_get (tx=tx@entry=0x7f7226468158, index=index@entry=0x2563860, key=
↪<optimized out>, part_count=<optimized out>, result=result@entry=0x7f72264afad8) at /usr/src/tarantool/
↪src/box/vinyl.c:5705
#9 0x0000000000477601 in vy_replace_impl (request=<optimized out>, request=<optimized out>,␣
↪stmt=0x7f72265a7150, space=0x2567ea0, tx=0x7f7226468158) at /usr/src/tarantool/src/box/vinyl.c:5920
#10 vy_replace (tx=0x7f7226468158, stmt=stmt@entry=0x7f72265a7150, space=0x2567ea0, request=<optimized␣
↪out>) at /usr/src/tarantool/src/box/vinyl.c:6608
#11 0x00000000004615a9 in VinylSpace::executeReplace (this=<optimized out>, txn=<optimized out>, space=
↪<optimized out>, request=<optimized out>) at /usr/src/tarantool/src/box/vinyl_space.cc:108
#12 0x00000000004bd723 in process_rw (request=request@entry=0x7f72265a70f8,␣
↪space=space@entry=0x2567ea0, result=result@entry=0x7f72264afbc8) at /usr/src/tarantool/src/box/box.cc:182
#13 0x00000000004bed48 in box_process1 (request=0x7f72265a70f8, result=result@entry=0x7f72264afbc8) at /
↪usr/src/tarantool/src/box/box.cc:700
#14 0x00000000004bf389 in box_replace (space_id=space_id@entry=513, tuple=<optimized out>, tuple_end=
↪<optimized out>, result=result@entry=0x7f72264afbc8) at /usr/src/tarantool/src/box/box.cc:754
#15 0x00000000004d72f8 in lbox_replace (L=0x413c5780) at /usr/src/tarantool/src/box/lua/index.c:72
#16 0x000000000050f317 in lj_BC_FUNCC ()
#17 0x00000000004d37c7 in execute_lua_call (L=0x413c5780) at /usr/src/tarantool/src/box/lua/call.c:282
#18 0x000000000050f317 in lj_BC_FUNCC ()
#19 0x0000000000529c7b in lua_cpcall ()
#20 0x00000000004f6aa3 in luaT_cpcall (L=L@entry=0x413c5780, func=func@entry=0x4d36d0 <execute_lua_
↪call>, ud=ud@entry=0x7f72264afde0) at /usr/src/tarantool/src/lua/utils.c:962
#21 0x00000000004d3fe7 in box_process_lua (handler=0x4d36d0 <execute_lua_call>,␣
↪out=out@entry=0x7f7213020600, request=request@entry=0x413c5780) at /usr/src/tarantool/src/box/lua/call.
↪c:382
#22 box_lua_call (request=request@entry=0x7f72130401d8, out=out@entry=0x7f7213020600) at /usr/src/
↪tarantool/src/box/lua/call.c:405
#23 0x00000000004c0f27 in box_process_call (request=request@entry=0x7f72130401d8,␣
↪out=out@entry=0x7f7213020600) at /usr/src/tarantool/src/box/box.cc:1074
#24 0x000000000041326c in tx_process_misc (m=0x7f7213040170) at /usr/src/tarantool/src/box/iproto.cc:942
#25 0x0000000000504554 in cmsg_deliver (msg=0x7f7213040170) at /usr/src/tarantool/src/cbus.c:302
#26 0x0000000000504c2e in fiber_pool_f (ap=<error reading variable: value has been optimized out>) at /usr/
↪src/tarantool/src/fiber_pool.c:64
#27 0x000000000041122c in fiber_cxx_invoke(fiber_func, typedef __va_list_tag __va_list_tag *) (f=
↪<optimized out>, ap=<optimized out>) at /usr/src/tarantool/src/fiber.h:645
#28 0x00000000005011a0 in fiber_loop (data=<optimized out>) at /usr/src/tarantool/src/fiber.c:641
#29 0x0000000000688fbf in coro_init () at /usr/src/tarantool/third_party/coro/coro.c:110
```

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
```

(continues on next page)

(continued from previous page)

```
      9 1.5% 50.7% 9 1.5%
::TupleCompare::compare@42efc0
      9 1.5% 52.2% 9 1.5% vy_cache_on_write
      9 1.5% 53.7% 57 9.5% vy_merge_iterator_next_key
      8 1.3% 55.0% 8 1.3% __nss_passwd_lookup
      6 1.0% 56.0% 25 4.2% gc_onestep
      6 1.0% 57.0% 6 1.0% lj_tab_next
      5 0.8% 57.9% 5 0.8% lj_alloc_malloc
      5 0.8% 58.7% 131 21.9% vy_prepare
```

perf

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
                     | | |
```

(continues on next page)

(continued from previous page)

```
| | |--1.33%--lj_tab_free
| | | |
| | | --1.01%--lj_alloc_free
| | |
| | --1.17%--lj_cdata_free
| |
| |--5.41%--gc_finalize
| | |
| | |--1.06%--lj_obj_equal
| | |
| | --0.95%--lj_tab_set
| |
| |--4.97%--rehashtab
| | |
| | --3.65%--lj_tab_resize
| | |
| | |--0.74%--lj_tab_set
| | |
| | --0.72%--lj_tab_newkey
| |
| |--0.91%--propagatemark
| |
| --0.67%--lj_cdata_free
|
 --5.43%--propagatemark
           |
            --0.73%--gc_mark
```

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.

6.5.6 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
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)
```

(continues on next page)

(continued from previous page)

```
       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.

6.5.7 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-lsn 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 automatic garbage collection of 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).

6.5.8 Backups

Tarantool storage architecture is append-only: files are only appended to, and are never overwritten. Old files are removed by garbage collection after a checkpoint. You can configure the amount of past checkpoints preserved by garbage collection by configuring Tarantool's checkpoint daemon. Backups can be taken at any time, with minimal overhead on database performance.

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. Old files are garbage collected 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 backup.
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. Resume garbage collection with box.backup.stop().

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.

6.5.9 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 1.8. Meanwhile Tarantool 1.8 may have incompatible changes when migrating from Tarantool 1.6. to 1.8 directly.

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.

6.5.10 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"
```

6.5.11 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.

6.6 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:

6.6.1 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.

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 case, replicas connect to each other, elect a master, which then creates the starting set of checkpoint files, and distributes it across all 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 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. This UUID is used 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, 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 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 shorter numeric identifiers is also handy to track the state of the entire replica set. For example, box.info.vclock describes the state of replication in regard to each connected peer.

```
box.info.vclock
---
- {1: 827, 2: 584}
...
```

Here vclock contains log sequence numbers (827 and 584) for instances with short identifiers 1 and 2.

Replication setup

To enable replication, you need to specify two parameters in a box.cfg{} request:

- replication parameter which defines the replication source(s), and
- read_only parameter which is true for a replica and false for a master.

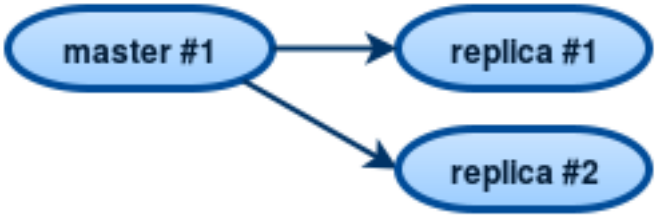
Both these parameters are "dynamic". This allows a replica to become a master and vice versa on the fly with the help of a box.cfg{} request.

Further we're giving a detailed example of bootstrapping a replica set.

Replication roles: master and replica

Replication role (master or replica) is set in read_only configuration parameter. The recommended role for all-but-one instances in a replica set is "read-only" (replica).

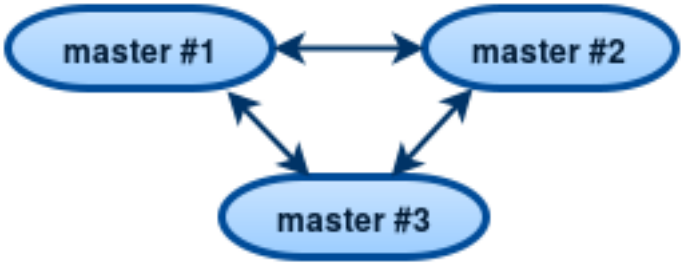
In a master-replica configuration, every change that happens on the master will be visible on the replicas, but not vice versa.



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 mix and apply 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), it is safe to set each instance to a master. 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.

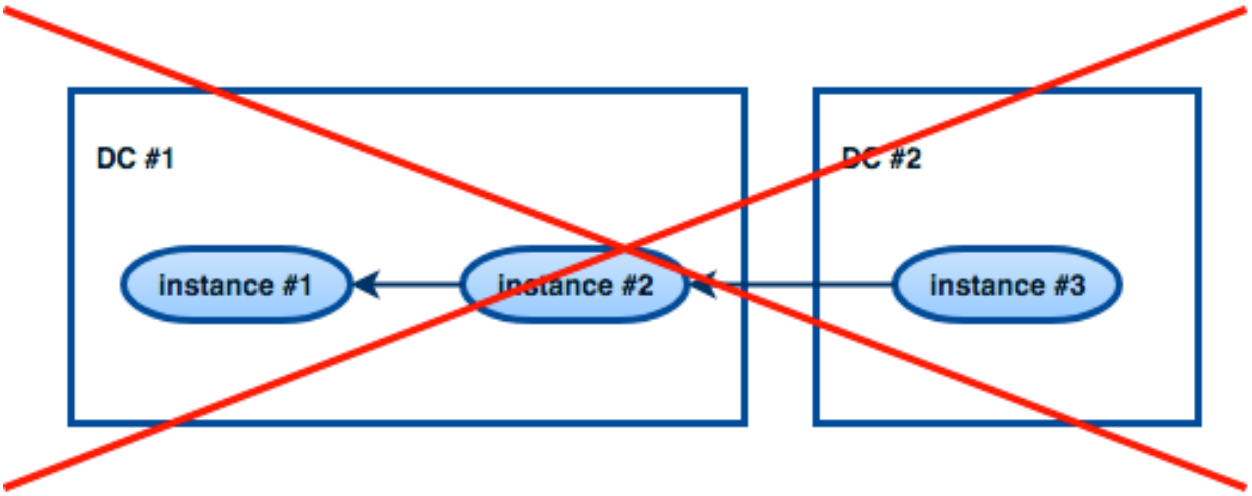
UPDATE operations, however, can easily go out of sync. For example, assignment and increment are not commutative, and may yield different results if applied in different order on different instances.

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.

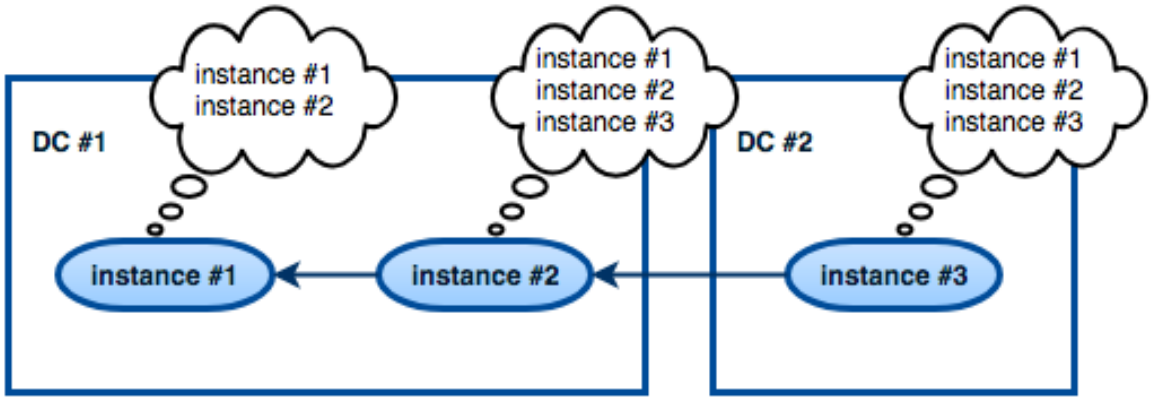
Replication topologies: cascade, ring and full mesh

Replication topology is set in replication configuration parameter. The recommended topology is a full mesh, because it makes potential failover easy.

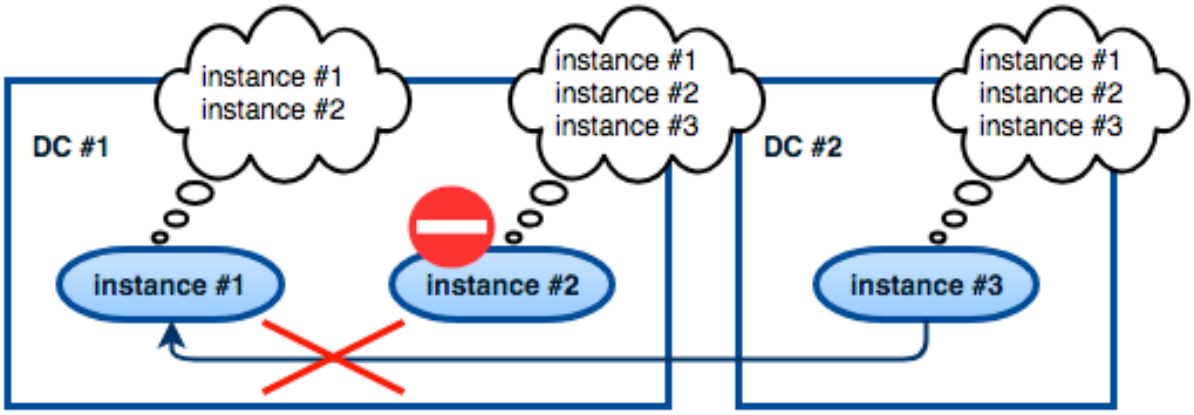
Some database products offer cascading replication topologies: creating a replica on a replica. Tarantool does not recommend such setup.



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 replica set UUID. Without knowing a replica set UUID, a master refuses to accept connections from such instances when replication topology changes. Here is how this can happen:

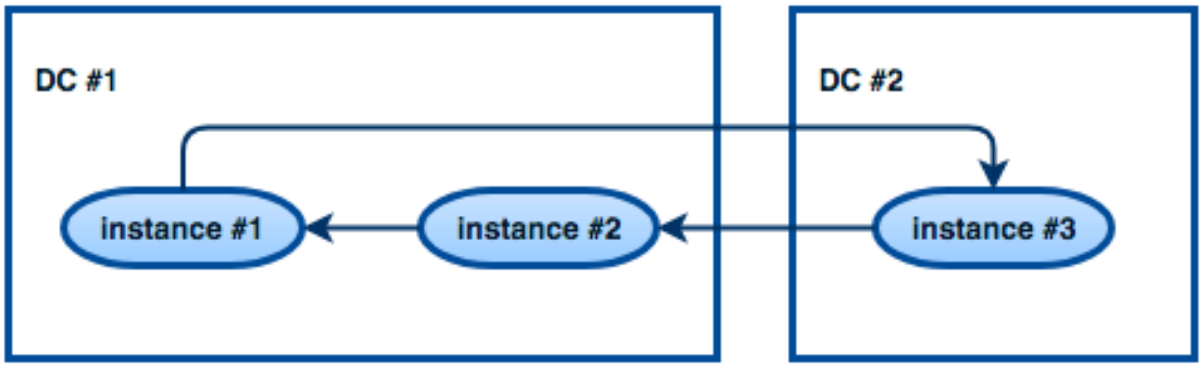


We have a chain of three instances. Instance #1 contains entries for instances #1 and #2 in its _cluster space. Instances #2 and #3 contain entries for instances #1, #2 and #3 in their _cluster spaces.



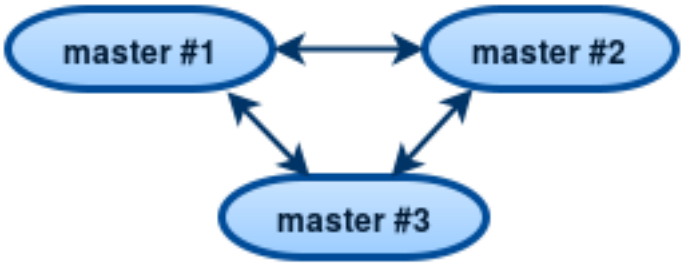
Now instance #2 is faulty. Instance #3 tries connecting to instance #1 as its new master, but the master refuses the connection since it has no entry for instance #3.

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 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.

6.6.2 Bootstrapping a replica set

Master-replica bootstrap

Let's 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:

- listen parameter from box.cfg{} defines a URI (port 3301 in our example), on which the master can accept connections from replicas.
- 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 to 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.

- read_only parameter enables data-change operations on the instance and makes this Tarantool instance act as a master, not as a replica. That's the only parameter in our instance files that will differ.
- 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') to see later in console whether box.once() is 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 further set up privileges for this role as required.

In the replica's instance file, we only set read-only parameter to "true", and say print('box.once executed on replica') to make sure that box.once() is not executed more than once. Otherwise the replica's instance file is fully 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
  read_only = true
}
```

(continues on next page)

(continued from previous page)

```
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, please 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
```

... (yep, box.once() got 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
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 got 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
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'
```

(continues on next page)

(continued from previous page)

```
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 don't want concurrent writes happen 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 4 steps ensure that the replica doesn't accept new writes until it's done fetching writes from the master.

Master-master bootstrap

Now let's 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 replication parameter, we define the URIs of both masters in the replica set and say print('box.once executed on master #1') to see 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
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
```

(continues on next page)

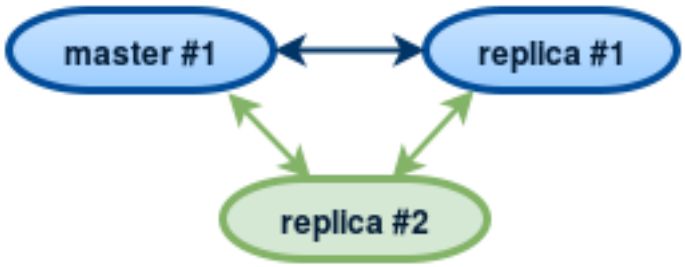
(continued from previous page)

```
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
```

6.6.3 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 replica #2 URI to replication parameter, so now it contains three URIs.

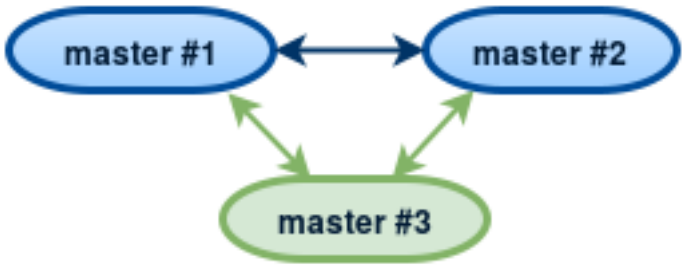
After we launch the new replica instance, it gets connected to the master instance and retrieves the master's write ahead log and snapshot files:

```
$ # 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're adding a read-only instance, there is no need to dynamically update replication parameter on the other running instances. This update would be required if we added a master instance.

However, we recommend to specify replica #3 URI 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 reconfigurations 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 master#3 URI to 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 the third master instance, it gets connected to the other master instances and retrieves their write ahead logs 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
```

(continues on next page)

(continued from previous page)

```
2017-06-14 17:10:00.562 [47121] snapshot/101/main I> saving snapshot `/Users/e.shebunyaeva/work/tarantool-
↪test-repl/master3_dir/00000000000000000009.snap.inprogress'
2017-06-14 17:10:00.562 [47121] snapshot/101/main I> done
2017-06-14 17:10:00.564 [47121] main/101/master3.lua I> vinyl checkpoint done
2017-06-14 17:10:00.564 [47121] main/101/master3.lua I> ready to accept requests
2017-06-14 17:10:00.565 [47121] main/101/master3.lua I> set 'read_only' configuration option to true
2017-06-14 17:10:00.565 [47121] main C> entering the event loop
2017-06-14 17:10:00.565 [47121] main/104/applier/replicator@192.168.0.10 I> authenticated
```

Next, we add master#3 URI to 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.

6.6.4 Removing instances

To politely remove an instance from a replica set, 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. replication parameter at all running instances in the replica set:

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

   b. box.space._cluster on any master instance in the replica set. For example, a record with instance id = 3:

      ```
      tarantool> box.space._cluster:select{}
      ---
      - - [1, '913f99c8-aee3-47f2-b414-53ed0ec5bf27']
        - [2, 'eac1aee7-cfeb-46cc-8503-3f8eb4c7de1e']
      ```

(continues on next page)

(continued from previous page)

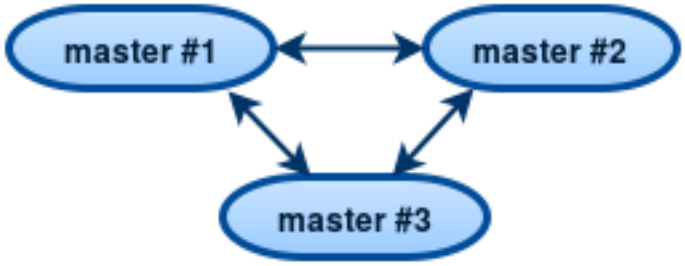
```
  - [3, '97f2d65f-2e03-4dc8-8df3-2469bd9ce61e']
...
tarantool> box.space._cluster:delete(3)
---
- [3, '97f2d65f-2e03-4dc8-8df3-2469bd9ce61e']
...
```

6.6.5 Monitoring a replica set

To learn what instances belong in the replica set, and obtain statistics for all these instances, use box.info.replication request:

```
box.info.replication
---
  replication:
    1:
      id: 1
      uuid: b8a7db60-745f-41b3-bf68-5fcce7a1e019
      lsn: 88
    2:
      id: 2
      uuid: cd3c7da2-a638-4c5d-ae63-e7767c3a6896
      lsn: 31
      upstream:
        status: follow
        idle: 43.187747001648
        lag: 0
      downstream:
     vclock: {1: 31}
    3:
      id: 3
      uuid: e38ef895-5804-43b9-81ac-9f2cd872b9c4
      lsn: 54
      upstream:
        status: follow
        idle: 43.187621831894
        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.

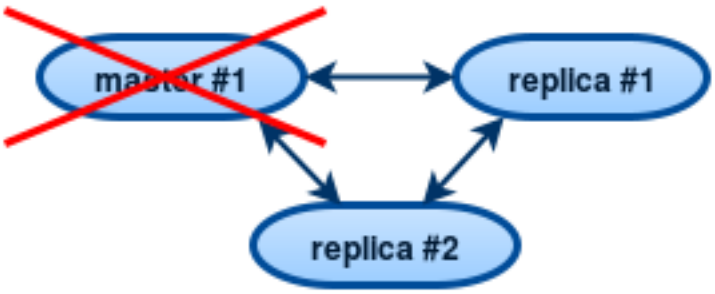


The request was issued at master #1, and the reply includes statistics for the other two masters, given in regard to master #1.

The primary indicators of replication health are idle and lag parameters (see reference on box.info.replication for details).

6.6.6 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:
      status: disconnected
      idle: 182.36929893494
      message: connect, called on fd 13, aka 192.168.0.101:58244
      lag: 0.00026607513427734
  2:
    id: 2
    uuid: fb252ac7-5c34-4459-84d0-54d248b8c87e
    lsn: 0
  3:
    id: 3
    uuid: fd7681d8-255f-4237-b8bb-c4fb9d99024d
    lsn: 0
```

(continues on next page)

(continued from previous page)

```
    downstream:
      vclock: {1: 542}
...
```

```
# report from replica #2
box.info.replication
---
- 1:
    id: 1
    uuid: 70e8e9dc-e38d-4046-99e5-d25419267229
    lsn: 542
    upstream:
      status: disconnected
      idle: 186.76988101006
      message: connect, called on fd 13, aka 192.168.0.101:58253
      lag: 0.00027203559875488
  2:
    id: 2
    uuid: fb252ac7-5c34-4459-84d0-54d248b8c87e
    lsn: 0
    upstream:
      status: follow
      idle: 186.76960110664
      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.

6.6.7 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. replication parameter at all running instances in the replica set.
   b. box.space._cluster 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.

6.6.8 Preventing duplicate actions

Tarantool guarantees that every update is applied only once at every replica. However, due to asynchronous nature of the replication, the order of updates is not guaranteed. Further we analyse this problem in 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 simultaneously 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.

```
$ # 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'll 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
```

(continues on next page)

(continued from previous page)

```
  ro: false
  vclock: {1: 9, 2: 1000000, 3: 3}
  uptime: 557
  lsn: 3
  vinyl: []
  cluster:
    uuid: 34d13b1a-f851-45bb-8f57-57489d3b3c8b
  pid: 30445
  status: running
  signature: 1000012
  replication:
    1:
      id: 1
      uuid: 7ab6dee7-dc0f-4477-af2b-0e63452573cf
      lsn: 9
      upstream:
        status: stopped
        idle: 445.8626639843
        message: Duplicate key exists in unique index 'primary' in space 'tester'
        lag: 0.00050592422485352
    2:
      id: 2
      uuid: 9afbe2d9-db84-4d05-9a7b-e0cbbf861e28
      lsn: 1000000
      upstream:
        status: follow
        idle: 201.99915885925
        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 we get this operation applied on both instances in the replica set:

```
-- at master #1
tarantool> box.space.tester:upsert({1}, {{'=', 2, box.info.uuid}})
-- 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 previous paragraphs represent examples of non-commutative operations, i.e. operations, which result depends on the execution order. On the contrary, for commutative operations, the execution order doesn't 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.

6.7 Connectors

This chapter documents APIs for various programming languages.

6.7.1 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.

6.7.2 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.

6.7.3 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')
```

6.7.4 Java

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

6.7.5 Go

Please see https://github.com/mialinx/go-tarantool.

6.7.6 R

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

6.7.7 Erlang

See Erlang tarantool driver.

6.7.8 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.

6.7.9 PHP

The most commonly used PHP driver is tarantool-php. It is not supplied as part of the Tarantool repository; it must be installed separately, for example with git. See installation instructions. in the driver'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', array(99999, 'BB'));
    echo "Insert succeeded\n";
} catch (Exception $e) {
    echo "Exception: ", $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, you can use an alternative PHP driver from another GitHub project: it includes a client (see tarantool-php/client) and a mapper for that client (see tarantool-php/mapper).

6.7.10 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.

6.7.11 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 user '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.

6.7.12 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.

6.7.13 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");
        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);
    }
```

(continues on next page)

(continued from previous page)

```
    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.

6.7.14 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);
   }
   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);
```

(continues on next page)

(continued from previous page)

```
  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.

6.8 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 7

Reference

7.1 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.

7.1.1 Module box

As well as executing Lua chunks or defining their own functions, you can exploit Tarantool's storage functionality with the box module and its submodules.

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 (see "Configuration reference" for a complete description of all configuration parameters). Use box.cfg without braces to get read-only access to those parameters.

Example:

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

(continues on next page)

(continued from previous page)

```
  background: false
  <...>
...
```

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

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:
  - 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. 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.

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.

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])
  {
    if (iterator is LT or GT) return FALSE
    continue
  }
  if (search-key-part[i] > index-key-part[i])
  {
    if (iterator is EQ or REQ or LE or LT) return FALSE
```

(continues on next page)

(continued from previous page)

```
  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 or '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'	bitset value	If an index key is equal to a bitset value, it matches. Tuples are returned in their order within the space. This is the default.
box.index.BITS_ALL_SET	bitset value	If all of the bits which are 1 in the bitset value are 1 in the index key, it matches. Tuples are returned in their order within the space.
box.index.BITS_ANY_SET	bitset value	If any of the bits which are 1 in the bitset value are 1 in the index key, it matches. Tuples are returned in their order within the space.
box.index.BITS_ALL_NOT_SET	bitset 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 some points of the rectangle-or-box defined by the search value are within the rectangle-or-box defined by the index key, it matches. Tuples are returned in their order within the space.
box.index.NEIGHBOR or 'NEIGHBOR'	search value	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 returns an error.

Example with BITSET index:

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

```
tarantool> s = box.schema.space.create('space_with_bitset')
tarantool> s:create_index('primary_index', {
         >   parts = {1, 'string'},
         >   unique = true,
         >   type = 'TREE'
         > })
tarantool> s:create_index('bitset_index', {
         >   parts = {2, 'unsigned'},
         >   unique = false,
         >   type = 'BITSET'
         > })
tarantool> s:insert{'Tuple with bit value = 01', 0x01}
tarantool> s:insert{'Tuple with bit value = 10', 0x02}
tarantool> s:insert{'Tuple with bit value = 11', 0x03}
tarantool> s.index.bitset_index:select(0x02, {
         >   iterator = box.index.EQ
         > })
---
- - ['Tuple with bit value = 10', 2]
...
tarantool> s.index.bitset_index:select(0x02, {
         >   iterator = box.index.BITS_ANY_SET
         > })
---
- - ['Tuple with bit value = 10', 2]
  - ['Tuple with bit value = 11', 3]
...
tarantool> s.index.bitset_index:select(0x02, {
         >   iterator = box.index.BITS_ALL_SET
         > })
---
- - ['Tuple with bit value = 10', 2]
  - ['Tuple with bit value = 11', 3]
...
tarantool> s.index.bitset_index:select(0x02, {
         >   iterator = box.index.BITS_ALL_NOT_SET
         > })
---
- - ['Tuple with bit value = 01', 1]
```

(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.

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.

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' })
---
```

(continues on next page)

(continued from previous page)

```
- 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.

Parameters

- index_object (index_object) – an object reference.
- options (table) – options list, same as the options list for create_index()

Return nil

Possible errors:

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

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

Example:

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

index_object:drop()

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

Parameters

- index_object (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

Example showing use of the box functions

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

- select a tuple whose key value is 1000;

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

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

```
function example()
  local a, b, c, table_of_selected_tuples, d
  local replaced_tuple, time_field
  local formatted_time_field
  local fiber = require('fiber')
  table_of_selected_tuples = box.space.tester:select{1000}
  if table_of_selected_tuples ~= nil then
    if table_of_selected_tuples[1] ~= nil then
      if #table_of_selected_tuples[1] == 3 then
        box.error({code=1, reason='This tuple already has 3 fields'})
      end
    end
  end
  replaced_tuple = box.space.tester:replace
    {1000,  require('uuid').str(), tostring(fiber.time())}
  time_field = tonumber(replaced_tuple[3])
  formatted_time_field = os.date("%Y-%m-%d %H:%M:%S", time_field)
  c = time_field % 1
  d = string.sub(c, 3, 6)
  formatted_time_field = formatted_time_field .. '.' .. d
  return formatted_time_field
end
```

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

```
tarantool> box.space.tester:delete(1000)
---
- [1000, '264ee2da03634f24972be76c43808254', '1391037015.6809']
...
tarantool> example(1000)
---
- 2014-01-29 16:11:51.1582
...
tarantool> example(1000)
---
- error: 'This tuple already has 3 fields'
...
```

Example showing a user-defined iterator

Here is an example that shows how to build one's own iterator. The paged_iter function is an "iterator function", which will only be understood by programmers who have read the Lua manual section Iterators and Closures. It does paginated retrievals, that is, it returns 10 tuples at a time from a table named "t", whose primary key was defined with create_index('primary',{parts={1,'string'}}).

```
function paged_iter(search_key, tuples_per_page)
  local iterator_string = "GE"
  return function ()
  local page = box.space.t.index[0]:select(search_key,
    {iterator = iterator_string, limit=tuples_per_page})
  if #page == 0 then return nil end
  search_key = page[#page][1]
  iterator_string = "GT"
  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:

```
        X AXIS
        1  2  3  4  5  6  7  8  9  10 11
      1
      2  #-------+                              <-Rectangle#1
Y AXIS  3  |       |
      4  +-------#
      5          #----------------------+       <-Rectangle#2
      6          |                      |
      7          |  #---+               |       <-Rectangle#3
      8          |  |   |               |
      9          |  +---#               |
```

(continues on next page)

(continued from previous page)

```
10      +----------------------#
11                             #              <-Rectangle#4
```

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.

- version is the Tarantool version. This value is also shown by tarantool –version.
- id corresponds to replication.id (see below).
- ro is true if the instance is in "read-only" mode (same as read_only in box.cfg{}).
- vclock corresponds to replication.downstream.vclock (see below).
- uptime is the number of seconds since the instance started. This value can also be retrieved with tarantool.uptime().
- lsn corresponds to replication.lsn (see below).
- vinyl returns runtime statistics for vinyl storage engine.

- 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.
- pid is the process ID. This value is also shown by tarantool module and by the Linux command ps -A.
- status corresponds to replication.upstream.status (see below).
- signature is the sum of all lsn values from the vector clocks (vclock) of all instances in the replica set.
- uuid corresponds to replication.uuid (see below).

replication part contains statistics for all instances in the replica set in regard to the current instance (see an example in the section "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 the instance's role is "replica" (read-only) and 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).
- replication.upstream.idle is the time (in seconds) since the instance received the last event from a master.
- 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.

  Since lag calculation uses operating system clock from two different machines, don't 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, this is the maximal lag.
- replication.downstream contains statistics for the replication data requested and downloaded from the instance.
- replication.downstream.vclock is the instance's vector clock, which contains a pair 'id, lsn'.

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

  Return keys and values in the submodule.

  Rtype table

Example:

```
tarantool> box.info
---
- version: 1.7.4-52-g980d30092
  id: 1
  ro: false
  vclock: {1: 8}
  uptime: 7280
  lsn: 8
  vinyl: []
  cluster:
    uuid: f7c0c1c6-f9d8-4df7-82ff-d4bd00610a6c
  pid: 16162
  status: running
  signature: 8
  replication:
    1:
      id: 1
      uuid: 1899631e-6369-40a1-81c9-7d170e909276
      lsn: 8
    2:
      id: 2
      uuid: bd949e5d-7ff9-413e-b4f2-c9b0149fdda6
      lsn: 0
      upstream:
        status: follow
        idle: 7256.7571430206
        lag: 0
      downstream:
        vclock: {1: 8}
    3:
      id: 3
      uuid: c5cb61d5-fa48-460d-abd7-3f13709d07a7
      lsn: 0
      upstream:
        status: follow
        idle: 7255.7510120869
        lag: 0
      downstream:
        vclock: {1: 8}
  uuid: 1899631e-6369-40a1-81c9-7d170e909276
...
```

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' :

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

Return space object

Rtype userdata

Options for box.schema.space.create

Name	Effect	Type	Default
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
id	unique identifier: users can refer to spaces with the id instead of the name	number	last space's id, +1
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
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
engine	'memtx' or 'vinyl'	string	'memtx'
user	name of the user who is considered to be the space's owner for authorization purposes	string	current user's name
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)

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 not be a number and should not contain special characters
- 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 a combination,
- object-type (string) – 'space' or 'function' or 'sequence'.

- object-name (string) – name of object to grant permissions to
- role-name (string) – name of role to grant to user.
- options (table) – grantor, if_not_exists

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

Variation: instead of object-type, object-name say 'universe' which means 'all object-types and all objects'. 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, privilege, 'role', 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 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. 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. This is optional; if it is not supplied, then the information will be for the user who is currently logged in.

Example:

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

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

Create a role. For explanation of how Tarantool maintains role data, see section Roles.

Parameters

- role-name (string) – name of role, which should not be a number and should not contain special characters
- 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(user-name, privilege, object-type, object-name[, option])

box.schema.role.grant(user-name, privilege, 'universe'[, nil, option])

box.schema.role.grant(role-name, role-name[, nil, nil, option])

Grant privileges to a role.

Parameters

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

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

Example:

```
box.schema.role.grant('Accountant', 'read', 'space', 'tester')
box.schema.role.grant('Accountant', 'execute', 'function', 'f')
box.schema.role.grant('Accountant', 'read,write', 'universe')
box.schema.role.grant('public', 'Accountant')
box.schema.role.grant('role1', 'role2', nil, nil, {if_not_exists=false})
```

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

Revoke privileges from a role.

Parameters

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

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}])

Create a function tuple. This does not create the function itself – that is done with Lua – but if it is necessary to grant privileges for a function, box.schema.func.create must be done first. For explanation of how Tarantool maintains function data, see reference on _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) - with true to make Tarantool treat the function's caller as the function's creator, with full privileges. Remember that SETUID works only over binary ports. SETUID doesn't work if you invoke a function via an admin console or inside a Lua script.
- language = 'LUA'|'C' (default = 'LUA').

Parameters

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

Return nil

Example:

```
box.schema.func.create('calculate')
box.schema.func.create('calculate', {if_not_exists = false})
box.schema.func.create('calculate', {setuid = false})
box.schema.func.create('calculate', {language = 'LUA'})
```

box.schema.func.drop(func-name[, {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 or function 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:

- When a function from a certain module is reloaded, all the other functions from this module are also reloaded.
- Reload will fail if a module was loaded from Lua script with ffi.load().

Parameters

- name (string) – the name of the module or function to reload

Examples:

```
-- reload a function
box.schema.func.reload('module.function')
-- reload the entire module contents
box.schema.func.reload('module')
-- reload everything
box.schema.func.reload()
```

Sequences

An introduction to sequences is in the Sequences section of the "Data model" chapter. Here are the details for each function and option.

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".
- 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 START 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 return 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=200,
        cache=100000,
        cycle=false,
```

(continues on next page)

(continued from previous page)

```
            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=sequence-name (or sequence=sequence-id or sequence=true) 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 would otherwise be nil.

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
  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: If you are using negative numbers for the sequence options, make sure that the index key type is 'integer'. Otherwise the index key type may be either 'integer' or 'unsigned'.

A sequence cannot be dropped if it is associated with an index.

Submodule box.session

Overview

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

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.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.id()

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

Rtype number

box.session.exists(id)

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

Rtype number

box.session.peer(id)

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

Rtype number

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.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"
---
...
```

(continues on next page)

(continued from previous page)

```
tarantool> box.session.storage.radius_of_mars = 3396
---
...
tarantool> m = ''
---
...
tarantool> for k, v in pairs(box.session.storage) do
         >   m = m .. k .. '='.. v .. ' '
         > end
---
...
tarantool> m
---
- 'radius_of_mars=3396 random_memorandum=Don''t forget the eggs. '
...
```

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

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.

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.

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.

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.

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 two 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 box.session.user() and user_name.

The second server instance will connect with console.connect, and then will display 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
box.session.on_auth(function1)
box.session.on_auth(function2)
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 2, box.session.user()=guest
function 2, user_name=admin
function 1, box.session.user()=guest
```

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 memtx_memory parameter, e.g. the default is 1 gigabyte = 2^30 bytes = 1,073,741,824 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
    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: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
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.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
box.space._index	(Metadata) List of indexes
box.space._priv	(Metadata) List of privileges
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._user	(Metadata) List of users

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.

Example:

```
tarantool> box.space.tester:bsize()
---
- 22
...
```

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

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 not be a number and should not contain special characters
- options (table) –

Return index object

Rtype index_object

Options for space_object:create_index()

Name	Effect	Type	Default
type	type of index	string (‘HASH’ or ‘TREE’ or ‘BITSET’ or ‘RTREE’)	‘TREE’
id	unique identifier	number	last index’s id, +1
unique	index is unique	boolean	true
if_not_exists	ignore error if duplicate name	boolean	false
parts	field-numbers + types	{field_no, ‘unsigned’ or ‘string’ or ‘integer’ or ‘number’ or ‘boolean’ or ‘array’ or ‘scalar’, and optional collation, and optional is_nullable value}	{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_level
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

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.

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 seven index field types (unsigned | string | integer | number | boolean | 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.
- array: array of numbers. Legal in memtx RTREE indexes.
- scalar: 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: 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'
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	booleans (true or false), integers between -9223372036854775808 and 18446744073709551615, single-precision floating point numbers, double-precision floating point numbers, strings	memtx TREE or HASH indexes, vinyl TREE indexes	true -1 1.234 '' '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 (the default) or is_nullable=false. If is_nullable is true, then it is legal to insert nil or an equivalent such as msgpack.NULL. 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}}})
```

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}})
```

(continues on next page)

(continued from previous page)

```
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'}}})
```

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

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'
...
```

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

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 {name='...',type='...'} pairs. The name may be any string, provided that two fields do not have the same name.

The type can be any of those allowed for indexed fields: unsigned | string | integer | number | boolean | array | scalar (the same as the requirement in "Options for space_object:create_index").

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 to use format on a space that already has a format, provided that there is no conflict with existing data or index definitions.

Example:

```
box.space.tester:format({{name='surname',type='string'},{name='IDX',type='array'}})
```

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'}})
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'}})
```

Names specified with the format clause can be used in space_object:get() and in space_object:create_index().

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

Example:

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

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 does not support len(). Possible workarounds are to use count() or #select(...).

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 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.

Example #1:

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

The trigger-function can have two parameters: old tuple, new tuple. For example, the following code causes nil to be printed when the insert request is processed, and causes [1, 'Hi'] 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) print(old) end
box.space.space_1:on_replace(on_replace_function)
box.space.space_1:insert{1,'Hi'}
box.space.space_1:delete{1}
```

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()
```

(continues on next page)

(continued from previous page)

```
        >   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: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.

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
---
```

(continues on next page)

(continued from previous page)

```
- 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: If a different tuple with the same unique-key value already exists, returns ER_TUPLE_FOUND. (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'}
```

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])

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.

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

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']
...
```

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

space_object:truncate()

Deletes all tuples.

Parameters

- space_object (space_object) – an object reference

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

Return nil

Note: Note that truncate must be called only by the user who created the space OR under a setuid function created by that user. Read more about setuid functions in reference on box.schema.func.create().

Example:

```
tarantool> box.space.tester:truncate()
---
...
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'.

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'}})
```

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
...
```

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

Rtype table

Example:

```
tarantool> #box.space.tester.index
---
- 1
...
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().

  Tuples in this space contain the following fields:

  - the numeric function id, a number,
  - the function name,
  - flag,
  - a language name (optional): 'LUA' (default) or 'C'.

  The _func space does not include the function's body. 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.

  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._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’ or ‘universe’,
- the numeric id of the object,
- the type of operation: “read” = 1, “write” = 2, “execute” = 4, 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._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]
  - ['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).

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
```

(continues on next page)

(continued from previous page)

```
        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()
---
- - '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._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 four special tuples in the _user space: 'guest', 'admin', 'public' and 'replication'.

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 assigned to new users when they are created with box.schema.user.create(user-name). Therefore, a convenient way to grant 'read' on space 't' to every user that will ever exist is with box.schema.role.grant('public','read','space','t').
replication	3	role	Pre-defined role, assigned by the 'admin' user to users who need to use replication features.

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')
---
...
```

Example: use box.space functions to read _space tuples

This function will illustrate how to look at all the spaces, and for each display: approximately how many tuples it contains, and the first field of its first tuple. The function uses Tarantool box.space functions len() and pairs(). The iteration through the spaces is coded as a scan of the _space system 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: use box.space functions to organize a _space tuple

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

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

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

box.space._space:select{ 288 }
or
box.space._space:select{ box.schema.SPACE_ID }

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

box.space._space.index.name:select{ '_space' }

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
```

(continues on next page)

(continued from previous page)

```
        > end
id, num
owner, num
name, str
engine, str
field_count, num
flags, str
format, *
---
...
```

Submodule box.stat

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

```
tarantool> type(box.stat), type(box.stat.net) -- virtual tables
---
- table
- table
...
tarantool> box.stat, box.stat.net
---
- net: &0 []
- *0
...
tarantool> box.stat()
---
- DELETE:
    total: 1873949
    rps: 123
  SELECT:
    total: 1237723
    rps: 4099
  INSERT:
    total: 0
    rps: 0
  EVAL:
    total: 0
    rps: 0
  CALL:
    total: 0
    rps: 0
  REPLACE:
    total: 1239123
    rps: 7849
  UPSERT:
    total: 0
    rps: 0
  AUTH:
    total: 0
    rps: 0
  ERROR:
    total: 0
```

(continues on next page)

(continued from previous page)

```
    rps: 0
  UPDATE:
    total: 0
    rps: 0
...
tarantool> box.stat().DELETE -- a selected item of the table
---
- total: 0
  rps: 0
...
tarantool> box.stat.net()
---
- SENT:
    total: 0
    rps: 0
  RECEIVED:
    total: 0
    rps: 0
...
```

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, tuples which were allocated before the snapshot has started are not freed until the snapshot has finished. To preserve consistency of the primary key, used to iterate over tuples, a copy-on-write technique is employed. If the master process changes part of a primary key, the corresponding process page is split, and the snapshot process obtains an old copy of the page. In effect, the snapshot process uses multi-version concurrency control in order to avoid copying changes which are superseded while it is running.

Since a snapshot is written sequentially, one can expect a very high write performance (averaging to 80MB/second on modern disks), which means an average database instance gets saved in a matter of minutes.

Note: As long as there are any changes to the parent index memory through concurrent updates, there are going to be page splits, and therefore 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 specific field
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: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.

Return number of bytes

Rtype number

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

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

<tuple_object>(field-number)
: If t is a tuple instance, t[field-number] will return the field numbered field-number in the tuple. The first field is t[1].

  Return field value.

  Rtype lua-value

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

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

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

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

  Return the number of the field in the tuple.

  Rtype number

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

```
tarantool> t = box.tuple.new{'a', 'b', 'c', 'a'}
---
...
tarantool> t:find('a')
---
- 1
...
tarantool> t:findall('a')
---
- 1
- 4
...
tarantool> t:findall(2, 'a')
---
- 4
...
```

tuple_object:transform(start-field-number, fields-to-remove[, field-value, ...])

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

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: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}...).

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'}
---
...
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
...
```

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.

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

box.commit()
: End the transaction, and make all its data-change operations permanent.

box.rollback()
: End the transaction, but cancel all its data-change operations. An explicit call to functions outside box.space that always yield, such as fiber.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.

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.

  Example:

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

(continues on next page)

(continued from previous page)

```
  box.space.t:insert{2} -- this will be rolled back
  box.rollback_to_savepoint(s)
  box.commit()          -- end transaction
end
```

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).

7.1.2 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.ibuf_decode() 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}
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.ibuf_decode(ibuf.rpos)
```

The result of the final request looks like this:

```
tarantool> msgpack.ibuf_decode(ibuf.rpos)
---
- 'cdata<char *>: 0x7f97ba10c041'
- {48: [['ABCDE', 12345]]}
...
```

7.1.3 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)
```

7.1.4 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:

```
console = require('console'); console.delimiter('!')
function f ()
    statement_1 = 'a'
    statement_2 = 'b'
end!
console.delimiter('')!
```

7.1.5 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-0, 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:

```
crypto.cipher.aes192.cbc.encrypt('string', 'key', 'initialization')
crypto.cipher.aes256.ecb.decrypt('string', 'key', 'initialization')
```

crypto.digest.{dss|dss1|md4|md5|mdc2|ripemd160}(string)

crypto.digest.{sha|sha1|sha224|sha256|sha384|sha512}(string)
Pass or return a digest derived from the string. The twelve choices of algorithms:

- dss - dss (using DSS)
- dss1 - dss (using DSS-1)
- md4 - md4 (with 128-bit binary strings using MD4)
- md5 - md5 (with 128-bit binary strings using MD5)
- mdc2 - mdc2 (using MDC2)
- ripemd160 - ripemd (with 160-bit binary strings using RIPEMD-160)
- sha - sha (with 160-bit binary strings using SHA-0)
- sha1 - sha-1 (with 160-bit binary strings using SHA-1)
- sha224 - sha-224 (with 224-bit binary strings using SHA-2)
- sha256 - sha-256 (with 256-bit binary strings using SHA-2)
- sha384 - sha-384 (with 384-bit binary strings using SHA-2)
- sha512 - sha-512(with 512-bit binary strings using SHA-2).

Example:

```
crypto.digest.md4('string')
crypto.digest.sha512('string')
```

Incremental methods in the crypto module

Suppose that a digest is done for a string 'A', then a new part 'B' is appended to the string, then a new digest is required. The new digest could be recomputed for the whole string 'AB', but it is faster to take what was computed before for 'A' and apply changes based on the new part 'B'. This is called multi-step or "incremental" digesting, which Tarantool supports for all crypto functions..

```
crypto = require('crypto')

-- print aes-192 digest of 'AB', with one step, then incrementally
print(crypto.cipher.aes192.cbc.encrypt('AB', 'key'))
c = crypto.cipher.aes192.cbc.encrypt.new()
c:init()
c:update('A', 'key')
c:update('B', 'key')
print(c:result())
c:free()

-- print sha-256 digest of 'AB', with one step, then incrementally
print(crypto.digest.sha256('AB'))
c = crypto.digest.sha256.new()
c:init()
c:update('A')
c:update('B')
print(c:result())
c:free()
```

Getting the same results from digest and crypto modules

The following functions are equivalent. For example, the digest function and the crypto function will both produce the same result.

```
crypto.cipher.aes256.cbc.encrypt('string', 'key') == digest.aes256cbc.encrypt('string', 'key')
crypto.digest.md4('string') == digest.md4('string')
crypto.digest.md5('string') == digest.md5('string')
crypto.digest.sha('string') == digest.sha('string')
crypto.digest.sha1('string') == digest.sha1('string')
crypto.digest.sha224('string') == digest.sha224('string')
crypto.digest.sha256('string') == digest.sha256('string')
crypto.digest.sha384('string') == digest.sha384('string')
crypto.digest.sha512('string') == digest.sha512('string')
```

7.1.6 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
```

(continues on next page)

(continued from previous page)

```
  - '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')
---
...
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
...
```

7.1.7 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-0, SHA-1, SHA-2) as well as a checksum function (CRC32), two functions for base64, and two non-cryptographic hash functions (guava, murmur). Some of the digest functionality is also present in the crypto module.

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.sha()	Get a digest made with SHA-0
digest.sha_hex()	Get a hexadecimal digest made with SHA-0
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.sha(string)
Returns 160-bit binary string = digest made with SHA-0. Not recommended.

digest.sha_hex(string)
Returns 40-byte string = hexadecimal of a digest calculated with sha.

digest.sha1(string)
Returns 160-bit binary string = digest made with SHA-1.

digest.sha1_hex(string)
Returns 40-byte string = hexadecimal of a digest calculated with sha1.

digest.sha224(string)
Returns 224-bit binary string = digest made with SHA-2.

digest.sha224_hex(string)
Returns 56-byte string = hexadecimal of a digest calculated with sha224.

digest.sha256(string)
Returns 256-bit binary string = digest made with SHA-2.

digest.sha256_hex(string)
Returns 64-byte string = hexadecimal of a digest calculated with sha256.

digest.sha384(string)
Returns 384-bit binary string = digest made with SHA-2.

digest.sha384_hex(string)
Returns 96-byte string = hexadecimal of a digest calculated with sha384.

digest.sha512(string)
Returns 512-bit binary tring = digest made with SHA-2.

digest.sha512_hex(string)
Returns 128-byte string = hexadecimal of a digest calculated with sha512.

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 CRC-32C (Castagnoli) polynomial value: 0x1EDC6F41 / 4812730177. If it is necessary to be compatible with other checksum functions in other programming languages, ensure that the other functions use the same polynomial value.

For example, in Python, install the crcmod package and say:

```
>>> import crcmod
>>> fun = crcmod.mkCrcFun('4812730177')
>>> fun('string')
3304160206L
```

In Perl, install the Digest::CRC module and run the following code:

```
use Digest::CRC;
$d = Digest::CRC->new(width => 32, poly => 0x1EDC6F41, init => 0xFFFFFFFF, refin => 1, refout_
↪=> 1);
$d->add('string');
print $d->digest;
```

(the expected output is 3304160206).

digest.crc32.new()
: Initiates incremental crc32. See incremental methods notes.

digest.guava(state, bucket)
: Returns a number made with consistent hash.

  The guava function uses the Consistent Hashing algorithm of the Google guava library. The first parameter should be a hash code; the second parameter should be the number of buckets; the returned value will be an integer between 0 and the number of buckets. For example,

```
tarantool> digest.guava(10863919174838991, 11)
---
- 8
...
```

digest.murmur(string)
: Returns 32-bit binary string = digest made with MurmurHash.

digest.murmur.new([seed])
: Initiates incremental MurmurHash. See incremental methods notes.

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'
...
```

7.1.8 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
  ...
...
```

7.1.9 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(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
```

(continues on next page)

(continued from previous page)

```
...
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
...
```

7.1.10 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.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
fiber_object:cancel()	Cancel a fiber
fiber_object.storage	Local storage within the fiber
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 yields control with fiber.sleep(), it is suspended. When a fiber ends (because the fiber function ends), it is dead.

All fibers are part of the fiber registry. This registry can be searched with fiber.find() - via fiber id (fid), which is a numeric identifier.

A runaway fiber can be stopped with fiber_object.cancel. However, fiber_object.cancel is advisory — it works only if the runaway fiber calls fiber.testcancel() occasionally. Most box.* functions, such as

box.space...delete() or box.space...update(), do call fiber.testcancel() but box.space...select{} does not. In practice, a runaway fiber can only become unresponsive if it does many computations and does not check whether it has been cancelled.

The other potential problem comes from fibers which never get scheduled, because they are not subscribed to any events, or because no relevant events occur. Such morphing fibers can be killed with fiber.kill() at any time, since fiber.kill() sends an asynchronous wakeup event to the fiber, and fiber.testcancel() is checked whenever such a wakeup event occurs.

Like all Lua objects, dead fibers are garbage collected. The garbage collector frees pool allocator memory owned by the fiber, resets all fiber data, and returns the fiber (now called a fiber carcass) to the fiber pool. The carcass can be reused when another fiber is created.

A fiber has all the features of a Lua coroutine and all the programming concepts that apply for Lua coroutines will apply for fibers as well. However, Tarantool has made some enhancements for fibers and has used fibers internally. So, although use of coroutines is possible and supported, use of fibers is recommended.

fiber.create(function[, function-arguments])

Create and start a fiber. The fiber is created and begins to run immediately.

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.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()

Return the status of the current fiber.

Return the status of fiber. One of: “dead”, “suspended”, or “running”.

Rtype string

Example:

```
tarantool> fiber.status()
---
- running
...
```

fiber.info()

Return information about all fibers.

Return number of context switches, backtrace, id, total memory, used memory, name for each fiber.

Rtype table

Example:

```
tarantool> fiber.info()
---
- 101:
    csw: 7
    backtrace: []
    fid: 101
    memory:
      total: 65776
      used: 0
    name: interactive
...
```

fiber.kill(id)

Locate a fiber by its numeric id and cancel it. In other words, fiber.kill() combines fiber.find() and fiber_object:cancel().

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

- self – fiber object, for example the fiber object returned by fiber.create

Return id of the fiber.

Rtype number

Example:

```
tarantool> fiber_object = fiber.self()
---
...
tarantool> fiber_object:id()
---
- 101
...
```

fiber_object:name()

Parameters

- self – fiber object, for example the fiber object returned by fiber.create

Return name of the fiber.

Rtype string

Example:

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

fiber_object:name(name)

Change the fiber name. By default 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

- self – fiber object, for example the fiber object returned by fiber.create
- name (string) – the new name of the fiber.

Return nil

Example:

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

fiber_object:status()

Return the status of the specified fiber.

Parameters

- self – fiber object, for example the fiber object returned by fiber.create

Return the status of fiber. One of: "dead", "suspended", or "running".

Rtype string

Example:

```
tarantool> fiber.self():status()
---
- running
...
```

fiber_object:cancel()

Cancel a fiber. Running and suspended fibers can be cancelled. After a fiber has been cancelled, attempts to operate on it will cause errors, for example fiber_object:id() will cause error: the fiber is dead.

Parameters

- self – fiber object, for example the fiber object returned by fiber.create

Return nil

Possible errors: cancel is not permitted for the specified fiber object.

Example:

```
tarantool> fiber.self():cancel() -- kill self, may make program send
---
- error: fiber is cancelled
...
```

fiber_object.storage

Local storage within the fiber. The storage can contain any number of named values, subject to memory limitations. Naming may be done with fiber_object.storage.name or fiber_object.storage['name']. or with a number fiber_object.storage[number]. Values may be either numbers or strings. The storage is garbage-collected when fiber_object:cancel() happens.

Example:

```
tarantool> fiber = require('fiber')
---
...
tarantool> function f () fiber.sleep(1000); end
---
...
tarantool> fiber_function = fiber:create(f)
---
- error: '[string "fiber_function = fiber:create(f)"]:1: fiber.create(function, ...):
    bad arguments'
...
tarantool> fiber_function = fiber.create(f)
---
...
tarantool> fiber_function.storage.str1 = 'string'
---
...
tarantool> fiber_function.storage['str1']
---
- string
...
tarantool> fiber_function:cancel()
---
...
tarantool> fiber_function.storage['str1']
---
- error: '[string "return fiber_function.storage[''str1'']"]:1: the fiber is dead'
...
```

See also box.session.storage.

fiber.time()

Return current system time (in seconds since the epoch) as a Lua number. The time is taken from the event loop clock, which makes this call very cheap, but still useful for constructing artificial tuple keys.

Rtype num

Example:

```
tarantool> fiber.time(), fiber.time()
---
```

(continues on next page)

(continued from previous page)

```
- 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 (while 0 == 0 is always true). Each iteration of the loop adds 1 to a global variable named gvar, then goes to sleep for 2 seconds. The sleep causes an implicit fiber.yield().

```
tarantool> fiber = require('fiber')
tarantool> function function_x()
         >   gvar = 0
         >   while 0 == 0 do
         >     gvar = gvar + 1
         >     fiber.sleep(2)
         >   end
         > end
---
...
```

Make a fiber, associate function_x with the fiber, and start function_x. It will immediately "detach" so it will be running independently of the caller.

```
tarantool> gvar = 0

tarantool> fiber_of_x = fiber.create(function_x)
---
...
```

Get the id of the fiber (fid), to be used in later displays.

```
tarantool> fid = fiber_of_x:id()
---
...
```

Pause for a while, while the detached function runs. Then ... Display the fiber id, the fiber status, and gvar (gvar will have gone up a bit depending how long the pause lasted). The status is suspended because the fiber spends almost all its time sleeping or yielding.

```
tarantool> print('#', fid, '. ', fiber_of_x:status(), '. gvar=', gvar)
# 102 .  suspended . gvar= 399
---
...
```

Pause for a while, while the detached function runs. Then ... Cancel the fiber. Then, once again ... Display the fiber id, the fiber status, and gvar (gvar will have gone up a bit more depending how long the pause lasted). This time the status is dead because the cancel worked.

```
tarantool> fiber_of_x:cancel()
---
...
tarantool> print('#', fid, '. ', fiber_of_x:status(), '. gvar=', gvar)
# 102 .  dead . gvar= 421
---
...
```

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 channel is garbage collected when no one is using it, as with any other Lua object. Use object-oriented syntax, for example channel:put(message) rather than fiber.channel.put(message).

fiber.channel([capacity])
: Create a new communication channel.

  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()
    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
```

(continues on next page)

(continued from previous page)

```
        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.

7.1.11 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 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.

Name	Use
fio.pathjoin()	Form a path name from one or more partial strings
fio.basename()	Get a file name
fio.dirname()	Get a directory name
fio.umask()	Set mask bits
fio.lstat() fio.stat()	Get information about a file object
fio.mkdir() fio.rmdir()	Create or delete a directory
fio.glob()	Get files whose names match a given string
fio.tempdir()	Get the name of a directory for storing temporary files
fio.cwd()	Get the name of the current working directory
fio.link() fio.symlink() fio.readlink() fio.unlink()	Create and delete links
fio.rename()	Rename a file or directory
fio.chown() fio.chmod()	Manage rights to and ownership of file objects
fio.truncate()	Reduce the file size
fio.sync()	Ensure that changes are written to disk
fio.open()	Open a file
file-handle:close()	Close a file
file-handle:pread() file-handle:pwrite()	Perform random-access read or write on a file
file-handle:read() file-handle:write()	Perform non-random-access read or write on a file
file-handle:truncate()	Change the size of an open file
file-handle:seek()	Change position in a file
file-handle:stat()	Get statistics about an open file
file-handle:fsync() file-handle:fdatasync()	Ensure that changes made to an open file are written to disk
fio.c	Table of constants similar to POSIX flag values

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/'
...
```

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 fields which describe the file's block size, creation time, size, and other attributes.

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 true if success, false if failure.

Rtype boolean

Example:

```
tarantool> fio.mkdir('/etc')
---
- false
...
```

fio.glob(path-name)

Return a list of files that match an input string. The list is constructed with a single flag that controls the behavior of the function: GLOB_NOESCAPE. For details type "man 3 glob".

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.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 fio.link and fio.symlink and fio.unlink return true if success, false if failure. fio.readlink returns the link value if success, nil if failure.

Example:

```
tarantool> fio.link('/home/username/tmp.txt', '/home/username/tmp.txt2')
---
- true
...
```

(continues on next page)

(continued from previous page)

```
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 true if success, false if failure.

Rtype boolean

Example:

```
tarantool> fio.rename('/home/username/tmp.txt', '/home/username/tmp.txt2')
---
- true
...
```

fio.chown(path-name, owner-user, owner-group)

fio.chmod(path-name, new-rights)

Manage the rights to file objects, or ownership of file objects. For details type "man 2 chown" or "man 2 chmod".

Parameters

- owner-user (string) – new user uid.
- owner-group (string) – new group uid.
- new-rights (number) – new permissions

Example:

```
tarantool> fio.chmod('/home/username/tmp.txt', tonumber('0755', 8))
---
- true
...
tarantool> fio.chown('/home/username/tmp.txt', 'username', 'username')
---
- true
...
```

fio.truncate(path-name, new-size)

Reduce file size to a specified value. For details type "man 2 truncate".

Parameters

- path-name (string) –
- new-size (number) –

Return true if success, false if failure.

Rtype boolean

Example:

```
tarantool> fio.truncate('/home/username/tmp.txt', 99999)
---
- true
...
```

fio.sync()

Ensure that changes are written to disk. For details type "man 2 sync".

Return true if success, false if failure.

Rtype boolean

Example:

```
tarantool> fio.sync()
---
- true
...
```

fio.open(path-name[, flags[, mode]])

Open a file in preparation for reading or writing or seeking.

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 file handle (later - fh)

Rtype userdata

Possible errors: nil.

Example:

```
tarantool> fh = fio.open('/home/username/tmp.txt', {'O_RDWR', 'O_APPEND'})
---
...
tarantool> fh -- display file handle returned by fio.open
---
- fh: 11
...
```

object file-handle

file-handle:close()

Close a file that was opened with fio.open. For details type "man 2 close".

Parameters

- fh (userdata) – file-handle as returned by fio.open().

Return true if success, false on failure.

Rtype boolean

Example:

```
tarantool> fh:close() -- where fh = file-handle
---
- true
...
```

file-handle:pread(count, offset)
file-handle:pwrite(new-string, offset)

Perform read/write random-access operation on a file, without affecting the current seek position of the file. For details type "man 2 pread" or "man 2 pwrite".

Parameters

- fh (userdata) – file-handle as returned by fio.open().
- count (number) – number of bytes to read
- new-string (string) – value to write
- offset (number) – offset within file where reading or writing begins

Return fh:pwrite returns true if success, false if failure. fh:pread returns the data that was read, or nil if failure.

Example:

```
tarantool> fh:pread(25, 25)
---
- |
  elete from t8//
  insert in
...
```

file-handle:read(count)
file-handle:write(new-string)

Perform non-random-access read or write on a file. For details type "man 2 read" or "man 2 write".

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().
- count (number) – number of bytes to read
- new-string (string) – value to write

Return fh:write returns true if success, false if failure. fh:read returns the data that was read, or nil if failure.

Example:

```
tarantool> fh:write('new data')
---
- true
...
```

file-handle:truncate(new-size)

Change the size of an open file. Differs from 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()
---
```

(continues on next page)

(continued from previous page)

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

7.1.12 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')
---
```

(continues on next page)

(continued from previous page)

```
...
tarantool> for _k, a in fun.range(3) do
         >   print(a)
         > end
1
2
3
---
...
```

7.1.13 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({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_KEEPALIVE
  - keepalive_interval - the interval, in seconds, that the operating system will wait between sending keepalive probes. See also CURLOPT_TCP_KEEPALIVE
  - 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

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("POST", 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)

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:

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
```

7.1.14 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.

Parameters

- input-string (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'
...
```

7.1.15 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.encode(lua-value)

Convert a Lua object to a JSON string.

Parameters

- lua_value – either a scalar value or a Lua table value.

Return the original value reformatted as a JSON string.

Rtype string

Example:

```
tarantool> json=require('json')
---
...
tarantool> json.encode(123)
---
- '123'
...
```

(continues on next page)

(continued from previous page)

```
tarantool> json.encode({123})
---
- '[123]'
...
tarantool> json.encode({123, 234, 345})
---
- '[123,234,345]'
...
tarantool> json.encode({abc = 234, cde = 345})
---
- '{"cde":345,"abc":234}'
...
tarantool> json.encode({hello = {'world'}})
---
- '{"hello":["world"]}'
...
```

json.decode(string)
: Convert a JSON string to a Lua object.

  Parameters

  - string (string) – a string formatted as JSON.

  Return the original contents formatted as a Lua table.

  Rtype table

  Example:

```
tarantool> json = require('json')
---
...
tarantool> json.decode('123')
---
- 123
...
tarantool> json.decode('[123, "hello"]')
---
- [123, 'hello']
...
tarantool> json.decode('{"hello": "world"}').hello
---
- world
...
```

  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'}
---
- - null
  - a
  - b
```

(continues on next page)

(continued from previous page)

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

Configuration settings

There are configuration settings which affect the way that Tarantool encodes invalid numbers or types. They are all boolean true/false values

- cfg.encode_invalid_numbers (default is true) – allow nan and inf
- cfg.encode_use_tostring (default is false) – use tostring for unrecognizable types
- cfg.encode_invalid_as_nil (default is false) – use null for all unrecognizable types
- cfg.encode_load_metatables (default is false) – load metatables

For example, the following code will interpret 0/0 (which is "not a number") and 1/0 (which is "infinity") as special values rather than nulls or errors:

```
json = require('json')
json.cfg{encode_invalid_numbers = true}
x = 0/0
```

(continues on next page)

(continued from previous page)

```
y = 1/0
json.encode({1, x, y, 2})
```

The result of the json.encode() request will look like this:

```
tarantool> json.encode({1, x, y, 2})
---
- '[1,nan,inf,2]
...
```

The same configuration settings exist for json, for MsgPack, and for YAML.

7.1.16 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.

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.

7.1.17 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 an MsgPack string to a Lua object
msgpack.NULL	Analog of Lua's "nil"

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(string)
Convert a MsgPack string to a Lua object.

Parameters

- string – a string formatted as MsgPack.

- 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.

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.

7.1.18 Module net.box

Overview

The net.box module contains connectors to remote database systems. One variant, to be discussed later, is connecting to MySQL or MariaDB or PostgreSQL (see SQL DBMS modules reference). The other variant, which is discussed in this section, is connecting to Tarantool server instances via a network using the built-in net.box module.

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 a remote box,
- conn:close to disconnect.

All net.box methods are fiber-safe, that is, it is safe to share and use the same connection object across multiple concurrent fibers. In fact, it's perhaps the best programming practice with Tarantool. When multiple fibers use the same connection, all requests are pipelined through the same network socket, but each fiber gets back a correct response. Reducing the number of active sockets lowers the overhead of system calls and increases the overall server performance. There are, however, cases when a single connection is not enough — for example, when it's necessary to prioritize requests or to use different authentication IDs.

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.

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:call()	Call a stored procedure
conn:eval()	Evaluate and execute the expression in a string
conn:timeout()	Set a timeout

net_box.connect(URI[, {option[s]}])

net_box.new(URI[, {option[s]}])

Note: The names connect() and new() are synonymous with the only difference that connect() is the preferred name, while 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 garbage-collected), 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).
- 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 garbage collected just like any other objects in Lua, so an explicit destruction is not mandatory. However, since close() is a system call, it is good programming practice to close a connection explicitly when it is no longer needed, to avoid lengthy stalls of the garbage collector.

Example:

```
conn:close()
```

conn.space.<space-name>:select({field-value, ...} [, {options}])

conn.space.space-name:select({...}) is the remote-call equivalent of the local call box.space.space-name:select{...}.

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(...).

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(...).

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(...).

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(...).

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(...).

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.

Examples:

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

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').

Example:

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

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.

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')
```

(continues on next page)

(continued from previous page)

```
        >   conn.space.tester:delete({800})
        >   table.insert(ta, 'conn delete done on tester.')
        >   conn.space.tester:insert({800, 'data'})
        >   table.insert(ta, 'conn insert done on tester, index 0')
        >   table.insert(ta, '  primary key value = 800.')
        >   wtuple = conn.space.tester:select({800})
        >   table.insert(ta, 'conn select done on tester, index 0')
        >   table.insert(ta, '  number of fields = ' .. #wtuple)
        >   conn.space.tester:delete({800})
        >   table.insert(ta, 'conn delete done on tester')
        >   conn.space.tester:replace({800, 'New data', 'Extra data'})
        >   table.insert(ta, 'conn:replace done on tester')
        >   conn.space.tester:update({800}, {{'=', 2, 'Fld#1'}})
        >   table.insert(ta, 'conn update done on tester')
        >   conn:close()
        >   table.insert(ta, 'conn close done')
        > end
---
...
tarantool> ta = {}
---
...
tarantool> example()
---
...
tarantool> ta
---
- - self:ping() succeeded
  - '  (no surprise -- self connection is pre-established)'
  - The local server listen address = 3301
  - conn delete done on tester.
  - conn insert done on tester, index 0
  - '  primary key value = 800.'
  - conn select done on tester, index 0
  - '  number of fields = 1'
  - conn delete done on tester
  - conn:replace done on tester
  - conn update done on tester
  - conn close done
...
```

7.1.19 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
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")
---
- 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
...
```

7.1.20 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 variable to a 1-byte integer, and stores the integer in the resulting string
s, S	converts Lua variable to a 2-byte integer, and stores the integer in the resulting string, low byte first
i, I	converts Lua variable to a 4-byte integer, and stores the integer in the resulting string, low byte first
l, L	converts Lua variable to an 8-byte integer, and stores the integer in the resulting string, low byte first
n	converts Lua variable to a 2-byte integer, and stores the integer in the resulting string, big endian,
N	converts Lua variable to a 4-byte integer, and stores the integer in the resulting string, big
q, Q	converts Lua variable to an 8-byte integer, and stores the integer in the resulting string, big endian,
f	converts Lua variable to a 4-byte float, and stores the float in the resulting string
d	converts Lua variable to a 8-byte double, and stores the double in the resulting string
a, A	converts Lua variable to a sequence of bytes, and stores the sequence in the resulting string

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

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}})
---
```

(continues on next page)

(continued from previous page)

```
- [0, 4]
...
tarantool> box.space.tester:update({0}, {{'+', 2, 4}})
---
- [0, 8]
...
tarantool> box.space.tester:update({0}, {{'^', 2, 4}})
---
- [0, 12]
...
```

pickle.unpack(format, binary-string)

Counterpart to pickle.pack(). Warning: if format specifier 'A' is used, it must be the last item.

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'))
---
...
```

7.1.21 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:

socket.getaddrinfo('tarantool.org', 'http') will return variable information such as

```
---
- - host: 188.93.56.70
    family: AF_INET
    type: SOCK_STREAM
    protocol: tcp
    port: 80
  - host: 188.93.56.70
    family: AF_INET
    type: SOCK_DGRAM
    protocol: udp
```

(continues on next page)

(continued from previous page)

```
    port: 80
...
```

socket.tcp_server(host, port, handler-function[, 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/table) – what to execute when a connection occurs
- timeout (number) – number of seconds to wait before timing out

The handler-function parameter may be a function name (for example function_55), a function declaration (for example function () print('!') end), or a table including handler = function (for example {handler=function_55, name='A'}).

Example:

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

object socket_object

socket_object:sysconnect(host, port)

Connect an existing socket to a remote host. The argument values are the same as in tcp_connect(). The host must be an IP address.

Parameters:

- Either:
  - host - a string representation of an IPv4 address or an IPv6 address;
  - port - a number.
- Or:
  - host - a string containing "unix/";
  - port - a string containing a path to a unix socket.
- Or:
  - host - a number, 0 (zero), meaning "all local interfaces";
  - port - a number. If a port number is 0 (zero), the socket will be bound to a random local port.

Return the socket object value may change if sysconnect() succeeds.

Rtype boolean

Example:

```
socket = require('socket')
sock = socket('AF_INET', 'SOCK_STREAM', 'tcp')
sock:sysconnect(0, 3301)
```

socket_object:send(data)

socket_object:write(data)
Send data over a connected socket.

Parameters

- data (string) –

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) –

Return a string of the requested length on success.

Rtype string

Possible errors: On error, returns an empty string, followed by status, errno, errstr. In case the writing side has closed its end, returns the remainder read from the socket (possibly an empty string), followed by "eof" status.

socket_object:read(limit[, timeout])
socket_object:read(delimiter[, timeout])
socket_object:read({limit=limit}[, timeout])
socket_object:read({delimiter=delimiter}[, timeout])
socket_object:read({limit=limit, delimiter=delimiter}[, timeout])
Read from a connected socket until some condition is true, and return the bytes that were read. Reading goes on until limit bytes have been read, or a delimiter has been read, or a timeout has expired.

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".

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 –
- port –

Return a socket object on success

Rtype userdata

Possible errors: Returns nil, status, errno, errstr on error.

socket_object:listen(backlog)
Start listening for incoming connections.

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) –
- port (number) –
- data (string) –

Return the number of bytes sent.

Rtype number

Possible errors: on error, returns status, errno, errstr.

socket_object:recvfrom(limit)
Receive a message on a UDP socket.

Parameters

- limit (integer) –

Return message, a table containing “host”, “family” and “port” fields.

Rtype string, table

Possible errors: on error, returns status, errno, errstr.

Example:

After message_content, message_sender = recvfrom(1) the value of message_content might be a string containing ‘X’ and the value of message_sender might be a table containing

```
message_sender.host = '18.44.0.1'
message_sender.family = 'AF_INET'
message_sender.port = 43065
```

socket_object:shutdown(how)

Shutdown a reading end, a writing end, or both ends of a socket.

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 its userdata is garbage collected by Lua.

Return true on success, false on error. For example, if sock is already closed, sock:close() returns false.

Rtype boolean

socket_object:error()
socket_object:errno()

Retrieve information about the last error that occurred on a socket, if any. Errors do not cause throwing of exceptions so these functions are usually necessary.

Return result for sock:errno(), result for sock:error(). If there is no error, then sock:errno() will return 0 and sock:error().

Rtype number, string

socket_object:setsockopt(level, name, value)

Set socket flags. The argument values are the same as in the Linux getsockopt(2) man page. The ones that Tarantool accepts are:

- SO_ACCEPTCONN
- SO_BINDTODEVICE
- SO_BROADCAST
- SO_DEBUG
- SO_DOMAIN
- SO_ERROR
- SO_DONTROUTE
- SO_KEEPALIVE

- SO_MARK
- SO_OOBINLINE
- SO_PASSCRED
- SO_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)

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)
---
```

(continues on next page)

(continued from previous page)

```
- 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')
---
- true
...
tarantool> message = sock_1:recvfrom()
---
...
tarantool> message
---
- X
...
tarantool> sock_1:close()
---
- true
...
```

(continues on next page)

(continued from previous page)

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

The above code means: use tcp_server() to wait for a connection from any host on port 3302. When it happens, enter a loop that reads on the socket and prints what it reads. The "delimiter" for the read function is "\n" so each read() will read a string as far as the next line feed, including the line feed.

On the second shell, create a file that contains a few lines. The contents don't matter. Suppose the first line contains A, the second line contains B, the third line contains C. Call this file "tmp.txt".

On the second shell, use the socat utility to ship the tmp.txt file to the server 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.

7.1.22 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()
```

(continues on next page)

(continued from previous page)

```
---
...
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
---
...
```

7.1.23 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()	Get the hexadecimal value of 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 spaces on the left of a string
string.rstrip()	Remove spaces on the right of a string
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)
---
```

(continues on next page)

(continued from previous page)

```
- ' 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.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.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')
---
...
```

(continues on next page)

(continued from previous page)

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

string.lstrip(input-string)

Return the value of the input string, but without spaces on the left.

Parameters

- input-string – (string) the string to process

Return result after stripping spaces from input string

Rtype string

Example:

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

string.rstrip(input-string)

Return the value of the input string, but without spaces on the right.

Parameters

- input-string – (string) the string to process

Return result after stripping spaces from input string

Rtype string

Example:

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

string.strip(input-string)

Return the value of the input string, but without spaces on the left or the right.

Parameters

- input-string – (string) the string to process

Return result after stripping spaces from input string

Rtype string

Example:

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

7.1.24 Module table

The table module has everything in the standard Lua table library, and some Tarantool extensions.

You can see this by saying "table":

```
tarantool> table
---
- maxn: 'function: builtin#90'
  copy: 'function: 0x41e9d300'
  new: 'function: builtin#94'
  clear: 'function: builtin#95'
  move: 'function: 0x41e918e0'
  foreach: 'function: 0x41e91588'
  sort: 'function: builtin#93'
  remove: 'function: 0x41e917c8'
  foreachi: 'function: 0x41e914b8'
  deepcopy: 'function: 0x41e9d2e0'
  getn: 'function: 0x41e91620'
  concat: 'function: builtin#92'
  insert: 'function: builtin#91'
...
```

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)
---
...

tarantool> output_table
---
- - 1
```

(continues on next page)

(continued from previous page)

```
 - - a
   - b
...
```

7.1.25 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

Below is a list of all tap functions.

Name	Use
tap.test()	Initialize
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 equal
taptest:isnil() taptest:isstring() taptest:isnumber() taptest:istable() taptest:isboolean() taptest:isudata() taptest:iscdata()	Check if a value has a particular type
taptest:is_deeply()	Recursively check if the two arguments are equal

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:plan(count)

Indicate how many tests will be performed.

Parameters

- count (number) –

Return nil

taptest:check()

Checks the number of tests performed. This check should only be done after all planned tests are complete, so ordinarily taptest:check() will only appear at the end of a script.

Will display # bad plan: ... if the number of completed tests is not equal to the number of tests specified by taptest:plan(...).

Return nil

taptest:diag(message)

Display a diagnostic message.

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 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')
```

(continues on next page)

(continued from previous page)

```
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 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 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 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 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 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 test

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
```

(continues on next page)

(continued from previous page)

```
    # Some subtests for test2: end
ok - some subtests for test2
```

7.1.26 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
...
```

7.1.27 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
...
```

7.1.28 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, and 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)

Parameters

- URI-components-table – a series of name:value pairs, one for each component

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
...
```

7.1.29 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}}
```

7.1.30 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
...
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
  ...
...
```

(continues on next page)

(continued from previous page)

```
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.

7.1.31 Miscellaneous

Index

Below is a list of miscellaneous functions.

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

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')
---
...
```

(continues on next page)

(continued from previous page)

```
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>'''
...
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
...
```

7.1.32 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.

7.1.33 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.

7.1.34 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.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.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

7.2 Rocks reference

This reference covers third-party Lua modules for Tarantool.

7.2.1 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/',
```

(continues on next page)

(continued from previous page)

```
    port = '/var/run/mysqld/mysqld.sock'
})
```

Example, creating a function which sets each option in a separate line:

```
tarantool> -- Connection function. Usage: conn = mysql_connect()
tarantool> function mysql_connection()
         >   local p = {}
         >   p.host = 'widgets.com'
         >   p.db = 'test'
         >   conn = mysql.connect(p)
         >   return conn
         > end
---
...
tarantool> conn = mysql_connect()
---
...
```

We will assume that the name is 'conn' in further examples.

How to ping

To ensure that a connection is working, the request is:

connection-name:ping()

Example:

```
tarantool> conn:ping()
---
- true
...
```

Executing a statement

For all MySQL statements, the request is:

connection-name:execute(sql-statement [, parameters])

where sql-statement is a string, and the optional parameters are extra values that can be plugged in to replace any question marks ("?"s) in the SQL statement.

Example:

```
tarantool> conn:execute('select table_name from information_schema.tables')
---
- - table_name: ALL_PLUGINS
  - table_name: APPLICABLE_ROLES
  - table_name: CHARACTER_SETS
  <...>
- 78
...
```

Closing connection

To end a session that began with mysql.connect, the request is:

connection-name:close()

Example:

```
tarantool> conn:close()
---
...
```

For further information, including examples of rarely-used requests, see the README.md file at github.com/tarantool/mysql.

Example

The example was run on an Ubuntu 12.04 ("precise") machine where tarantool had been installed in a /usr subdirectory, and a copy of MySQL had been installed on ~/mysql-5.5. The mysqld server 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) ...
```

(continues on next page)

(continued from previous page)

```
$ # Set up the Tarantool rock list in ~/.luarocks,
$ # following instructions at rocks.tarantool.org
$ mkdir ~/.luarocks
$ echo "rocks_servers = {[[http://rocks.tarantool.org/]]}" >> \
    ~/.luarocks/config.lua

$ # Ensure that the next "install" will get files from Tarantool
$ # master repository. The resultant display is normal for Ubuntu
$ # 12.04 precise
$ cat /etc/apt/sources.list.d/tarantool.list
deb http://tarantool.org/dist/1.7/ubuntu/ precise main
deb-src http://tarantool.org/dist/1.7/ubuntu/ precise main

$ # Install tarantool-dev. The displayed line should show version = 1.6
$ sudo apt-get -y install tarantool-dev | grep -E "Setting up|already"
Setting up tarantool-dev (1.6.6.222.g48b98bb~precise-1) ...
$

$ # Use luarocks to install locally, that is, relative to $HOME
$ luarocks install mysql MYSQL_LIBDIR=/usr/local/mysql/lib --local
Installing http://rocks.tarantool.org/mysql-scm-1.rockspec...
... (more info about building the Tarantool/MySQL driver appears here)
mysql scm-1 is now built and installed in ~/.luarocks/

$ # Ensure driver.so now has been created in a place
$ # tarantool will look at
$ find ~/.luarocks -name "driver.so"
~/.luarocks/lib/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 1.7.0-222-g48b98bb
type 'help' for interactive help
tarantool>
```

Configure tarantool and load mysql module. Make sure that tarantool doesn't reply "error" for the call to "require()".

```
tarantool> box.cfg{}
...
tarantool> mysql = require('mysql')
---
...
```

Create a Lua function that will connect to the MySQL server 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
---
...
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
---
...
tarantool> conn = pg_connect()
---
...
```

We will assume that the name is 'conn' in further examples.

How to ping

To ensure that a connection is working, the request is:

connection-name:ping()

Example:

```
tarantool> conn:ping()
---
- true
...
```

Executing a statement

For all PostgreSQL statements, the request is:

connection-name:execute(sql-statement [, parameters])

where sql-statement is a string, and the optional parameters are extra values that can be plugged in to replace any question marks ("?"s) in the SQL statement.

Example:

```
tarantool> conn:execute('select tablename from pg_tables')
---
- - tablename: pg_statistic
  - tablename: pg_type
  - tablename: pg_authid
  <...>
...
```

Closing connection

To end a session that began with pg.connect, the request is:

connection-name:close()

Example:

```
tarantool> conn:close()
---
...
```

For further information, including examples of rarely-used requests, see the README.md file at github.com/tarantool/pg.

Example

The example was run on an Ubuntu 12.04 ("precise") machine where tarantool had been installed in a /usr subdirectory, and a copy of PostgreSQL had been installed on /usr. The PostgreSQL server 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/1.7/ubuntu/ precise main
deb-src http://tarantool.org/dist/1.7/ubuntu/ precise main
```

(continues on next page)

(continued from previous page)

```
$ # Install tarantool-dev. The displayed line should show version = 1.7
$ sudo apt-get -y install tarantool-dev | grep -E "Setting up|already"
Setting up tarantool-dev (1.7.0.222.g48b98bb~precise-1) ...
$

$ # Use luarocks to install locally, that is, relative to $HOME
$ luarocks install pg POSTGRESQL_LIBDIR=/usr/lib/x86_64-linux-gnu --local
Installing http://rocks.tarantool.org/pg-scm-1.rockspec...
... (more info about building the Tarantool/PostgreSQL driver appears here)
pg scm-1 is now built and installed in ~/.luarocks/

$ # Ensure driver.so now has been created in a place
$ # tarantool will look at
$ find ~/.luarocks -name "driver.so"
~/.luarocks/lib/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 1.7.0-412-g803b15c
type 'help' for interactive help
tarantool>
```

Configure tarantool and load pg module. Make sure that tarantool doesn't reply "error" for the call to "require()".

```
tarantool> box.cfg{}
...
tarantool> pg = require('pg')
---
...
```

Create a Lua function that will connect to 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 .. ' '
```

(continues on next page)

(continued from previous page)

```
        >       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.

7.2.2 Module expiration

For a commercial-grade example of a Lua rock that works with Tarantool, let us look at expiration, which Tarantool supplies on GitHub with an Artistic license. The expiration.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(expiration.constants.check_interval)
...
```

Whenever one hears "daemon" in Tarantool, one should suspect it's being done with a fiber. The program is making a fiber and turning control over to it so it runs occasionally, goes to sleep, then comes back for more.

```
for _, tuple in scan_space.index[0]:pairs(nil, {iterator = box.index.ALL}) do
...
        if task.is_tuple_expired(task.args, tuple) then
        task.expired_tuples_count = task.expired_tuples_count + 1
        task.process_expired_tuple(task.space_id, task.args, tuple)
...
```

The "for" instruction can be translated as "iterate through the index of the space that is being scanned", and within it, if the tuple is "expired" (for example, if the tuple has a timestamp field which is less than the current time), process the tuple as an expired tuple.

```
-- default process_expired_tuple function
local function default_tuple_drop(space_id, args, tuple)
    local key = fun.map(
        function(x) return tuple[x.fieldno] end,
        box.space[space_id].index[0].parts
    ):totable()
    box.space[space_id]:delete(key)
end
```

Ultimately the tuple-expiry process leads to default_tuple_drop() which does a "delete" of a tuple from its original space. First the fun fun module is used, specifically fun.map. Remembering that index[0] is always the space's primary key, and index[0].parts[N].fieldno is always the field number for key part N, fun.map() is creating a table from the primary-key values of the tuple. The result of fun.map() is passed to space_object:delete().

```
local function expirationd_run_task(name, space_id, is_tuple_expired, options)
...
```

At this point, if the above explanation is worthwhile, it's clear that expirationd.lua starts a background routine (fiber) which iterates through all the tuples in a space, sleeps cooperatively so that other fibers can operate at the same time, and - whenever it finds a tuple that has expired - deletes it from this space. Now the "expirationd_run_task()" function can be used in a test which creates sample data, lets the daemon run for a while, and prints results.

For those who like to see things run, here are the exact steps to get expirationd through the test.

1. Get expirationd.lua. There are standard ways - it is after all part of a standard rock - but for this purpose just copy the contents of expirationd.lua to a default directory.
2. Start the Tarantool server as described before.
3. Execute these requests:

```
fiber = require('fiber')
expd = require('expirationd')
box.cfg{}
e = box.schema.space.create('expirationd_test')
e:create_index('primary', {type = 'hash', parts = {1, 'unsigned'}})
e:replace{1, fiber.time() + 3}
e:replace{2, fiber.time() + 30}
function is_tuple_expired(args, tuple)
  if (tuple[2] < fiber.time()) then return true end
  return false
  end
expd.run_task('expirationd_test', e.id, is_tuple_expired)
retval = {}
fiber.sleep(2)
expd.task_stats()
fiber.sleep(2)
expd.task_stats()
expd.kill_task('expirationd_test')
e:drop()
os.exit()
```

The database-specific requests (cfg, space.create, create_index) should already be familiar.

The function which will be supplied to expirationd is is_tuple_expired, which is saying "if the second field of the tuple is less than the current time , then return true, otherwise return false".

The key for getting the rock rolling is expd = require('expirationd'). The "require" function is what reads in the program; it will appear in many later examples in this manual, when it's necessary to get a module that's not part of the Tarantool kernel. After the Lua variable expd has been assigned the value of the expirationd module, it's possible to invoke the module's run_task() function.

After sleeping for two seconds, when the task has had time to do its iterations through the spaces, expd.task_stats() will print out a report showing how many tuples have expired – "expired_count: 0". After sleeping for two more seconds, expd.task_stats() will print out a report showing how many tuples have expired – "expired_count: 1". This shows that the is_tuple_expired() function eventually returned "true" for one of the tuples, because its timestamp field was more than three seconds old.

Of course, expirationd can be customized to do different things by passing different parameters, which will be evident after looking in more detail at the source code.

7.2.3 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 search all shards.

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
```

(continues on next page)

(continued from previous page)

```
$ ~/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
```

(continues on next page)

(continued from previous page)

```
$ 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.

7.2.4 Module tdb

The Tarantool Debugger (abbreviation = tdb) can be used with any Lua program. The operational features include: setting breakpoints, examining variables, going forward one line at a time, backtracing, and showing information about fibers. The display features include: using different colors for different situations, including line numbers, and adding hints.

It is not supplied as part of the Tarantool repository; it must be installed separately. Here is the usual way:

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

To initiate tdb within a Lua program and set a breakpoint, edit the program to include these lines:

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

To start the debugging session, execute the Lua program. Execution will stop at the breakpoint, and it will be possible to enter debugging commands.

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.

7.3 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 the following command:

```
$ tarantool
# OR
$ tarantool options
# OR
$ tarantool lua-initialization-file [ arguments ]
```

7.3.1 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.

7.3.2 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.

7.3.3 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().

7.3.4 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, and for logging.

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

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

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.

  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

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: no

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 can be overridden by a create_index option.

  Type: float
  Default = 0.05
  Dynamic: no

vinyl_cache
: The maximal cache size for the vinyl storage engine, in bytes.

  Type: integer
  Default = 128 * 1024 * 1024 = 134217728
  Dynamic: no

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: no

vinyl_page_size
: Page size, in bytes. Page is a read/write unit for vinyl disk operations. The vinyl_page_size setting can be overridden by a create_index option.

Type: integer
Default = 8 * 1024 = 8192
Dynamic: no

vinyl_range_size
: The maximal range size for vinyl, in bytes. The vinyl_range_size setting can be overridden by a create_index option.

Type: integer
Default = 1024 * 1024 * 1024 = 1073741824
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. This can be overridden by a create_index option.

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 can be overridden by a create_index option.

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

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. If the checkpoint daemon deletes an old snapshot file, then it 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.

Exceptions: the checkpoint daemon will not delete a file if a backup is ongoing and the file has not been backed up (see "Hot backup"), or if replication is ongoing and the file has not been relayed to a replica (see "Replication architecture"), or if a replica is connecting.

The checkpoint_interval and checkpoint_count configuration settings determine how long the intervals are, and how many snapshots should exist before deletions occur.

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.

Type: integer
Default: 2
Dynamic: yes

Binary logging and snapshots

- force_recovery,
- rows_per_wal,
- 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 re-builds the file.

  Otherwise, Tarantool aborts recovery on read errors.

  Type: boolean
  Default: true
  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

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.

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: yes

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 synch), 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
: 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.

  The default user name is 'guest'. A 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

Networking

- io_collect_interval,
- 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

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:

```
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:

```
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:

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

If the log string has the prefix "syslog:", then the string is interpreted as a message for the syslogd program which normally is running in the background of any Unix-like platform. One can optionally specify an identity, a facility, or both. The identity is an arbitrary string, default value = tarantool, which will be placed at the beginning of all messages. The facility is an abbreviation for the name of one of the syslog facilities, default value = user, which tell syslogd where the message should go.

Possible values for facility are: auth, authpriv, cron, daemon, ftp, kern, lpr, mail, news, security, syslog, user, uucp, local0, local1, local2, local3, local4, local5, local6, local7.

The facility setting is currently ignored but will be used in the future.

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 on the log file descriptor when it's not ready for write, and drops the message instead. If log_level is high, and many messages go to the log file, setting log_nonblock to true may improve logging performance at the cost of some log messages getting lost.

Type: boolean

Default: true
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.

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
```

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.

7.4 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.

7.4.1 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.

7.4.2 Commands for managing Tarantool instances

tarantoolctl start NAME Start a Tarantool instance (if not started; fail otherwise).

tarantoolctl stop NAME Stop a Tarantool instance (if not stopped; fail otherwise).

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 (if started; fail otherwise).

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 Enter an instance's interactive Lua console.

tarantoolctl eval NAME FILE Evaluate a local Lua file on a Tarantool instance (if started; fail otherwise).

tarantoolctl connect URI Connect to a Tarantool instance on an admin-console port. Supports both TCP/Unix sockets.

7.4.3 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.

7.4.4 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.

7.5 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.

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:).

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.

CHAPTER 8

Tutorials

8.1 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.

8.1.1 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.

8.1.2 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.

8.1.3 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 <box_index-iterator-types> 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']
...
```

8.2 C tutorial

Here is one C tutorial: C stored procedures.

8.2.1 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 four tasks. The first – easy.c – prints "hello world". The second – harder.c – decodes a passed parameter value. The third – hardest.c – uses the C API to do a DBMS insert. The fourth – read.c – uses the C API to do a DBMS select.

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 1.7 * 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 1.7 was installed from source. Otherwise Tarantool's "developer" package must be installed. For example on Ubuntu say sudo apt-get install tarantool-dev or on Fedora say dnf -y install tarantool-devel

The "msgpuck.h" file will exist if Tarantool 1.7 was installed from source. Otherwise the "msgpuck" package must be installed from https://github.com/rtsisyk/msgpuck.

Both module.h and msgpuck.h must be on the include path for the C compiler to see them. For example, if module.h address is /usr/local/include/tarantool/module.h, and msgpuck.h address is /usr/local/include/msgpuck/msgpuck.h, and they are not currently on the include path, say export CPATH=/usr/local/include/tarantool:/usr/local/include/msgpuck

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;
}
```

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 and box.schema.user.grant and conn:call.

The function that matters is capi_connection:call('easy').

Its first job is to find the 'easy' function, which should be easy because by default Tarantool looks on the current directory for a file named easy.so.

Its second job is to call the 'easy' function. Since the easy() function in easy.c begins with printf("hello world\n"), the words "hello world" will appear on the screen.

Its third job is to check that the call was successful. Since the easy() function in easy.c ends with return 0, there is no error message to display and the request is over.

The result should look like this:

```
tarantool> capi_connection:call('easy')
hello world
---
- []
...
```

Conclusion: calling a C function is easy.

harder.c

Go back to the shell where the easy.c program was created.

Create a file. Name it harder.c. Put these 17 lines in it:

```
#include "module.h"
#include "msgpuck.h"
int harder(box_function_ctx_t *ctx, const char *args, const char *args_end)
{
  uint32_t arg_count = mp_decode_array(&args);
  printf("arg_count = %d\n", arg_count);
```

(continues on next page)

(continued from previous page)

```
  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)
table.insert(passable_table, 2)
table.insert(passable_table, 3)
capi_connection:call('harder', passable_table)
```

This time the call is passing a Lua table (passable_table) to the harder() function. The harder() function will see it, it's in the char *args parameter.

At this point the harder() function will start using functions defined in msgpuck.h, which are documented in http://rtsisyk.github.io/msgpuck. The routines that begin with "mp" are msgpuck functions that handle data formatted according to the MsgPack specification. Passes and returns are always done with this format so one must become acquainted with msgpuck to become proficient with the C API.

For now, though, it's enough to know that mp_decode_array() returns the number of elements in an array, and mp_decode_uint returns an unsigned integer, from args. And there's a side effect: when the decoding finishes, args has changed and is now pointing to the next element.

Therefore the first displayed line will be "arg_count = 1" because there was only one item passed: passable_table. The second displayed line will be "field_count = 3" because there are three items in the table. The next three lines will be "1" and "2" and "3" because those are the values in the items in the table.

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];
  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.

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. The whole C API is documented here. The function box_space_id_by_name() is documented here. The function box_insert() is documented here.

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);
```

(continues on next page)

(continued from previous page)

```
  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);
  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

came from the C API. The function box_index_get() is documented here. The function box_tuple_field() is documented here.

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, and get rid of the capi_test space with box.schema.capi_test:drop(), and remove the .c and .so files that were created for this tutorial.

An example in the test suite

Download the source code of Tarantool. Look in a subdirectory test/box. Notice that there is a file named tuple_bench.test.lua and another file named tuple_bench.c. Examine the Lua file and observe that it is calling a function in the C file, using the same techniques that this tutorial has shown.

Conclusion: parts of the standard test suite use C stored procedures, and they must work, because releases don't happen if Tarantool doesn't pass the tests.

8.3 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
```

(continues on next page)

(continued from previous page)

```
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;
```

(continues on next page)

(continued from previous page)

```
    try {
        slave::DefaultExtState sDefExtState;
        slave::Slave slave(masterinfo, sDefExtState);
        sl = &slave;
        sDefExtState.setMasterLogNamePos("mysql-bin", 0);
        slave.setCallback("test", "test", callback);
        slave.init();
        slave.createDatabaseStructure();
        try {
            slave.get_remote_binlog(isStopping);
        } catch (std::exception& ex) {
            std::cout << "Error reading: " << ex.what() << std::endl;
            error = true;
        }
    } catch (std::exception& ex) {
        std::cout << "Error initializing: " << ex.what() << std::endl;
        error = true;
    }
    return 0;
}
```

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 9

Contributor's Guide

9.1 C API reference

9.1.1 Module box

box_function_ctx_t
: Opaque structure passed to the stored C procedure

int box_return_tuple(box_function_ctx_t *ctx, box_tuple_t *tuple)
: Return a tuple from stored C procedure.

  Returned tuple is automatically reference counted by Tarantool.

  Parameters

  - ctx (box_funtion_ctx_t*) – an opaque structure passed to the stored C 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 SELECT request to _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.

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

This function performs SELECT request to _vindex system space.

See also: box_space_id_by_name

int box_insert(uint32_t space_id, const char *tuple, const char *tuple_end, box_tuple_t **result)
Execute an INSERT/REPLACE request.

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. Resulted 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 an 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. Resulted 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. Result 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 Arrat format, e.g. [['=', field_id, value], ['!', 2, 'xxx']]
- char* ops_end (const) – end of a ops
- index_base (int) – 0 if field_ids in update operation are zero-based indexed (like C) or 1 if for one-based indexed field ids (like Lua).
- result (box_tuple_t**) – output argument. Result an old tuple. Can be set to NULL to discard result

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 Arrat format, e.g. [['=', field_id, value], ['!', 2, 'xxx']]
- char* ops_end (const) – end of a ops
- index_base (int) – 0 if field_ids in update operation are zero-based indexed (like C) or 1 if for one-based indexed field ids (like Lua).
- result (box_tuple_t**) – output argument. Result an old tuple. Can be set to NULL to discard result

Returns -1 on error (check :box_error_last())

Returns 0 otherwise

See also space_object.upsert()

int box_truncate(uint32_t space_id)
Truncate space.

Parameters

- space_id (uint32_t) – space identifier

9.1.2 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)

9.1.3 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)

9.1.4 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.

9.1.5 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().

9.1.6 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()

9.1.7 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

9.1.8 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.

9.1.9 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);
```

9.1.10 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_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.

9.1.11 Module trivia/config

API_EXPORT
Extern modifier for all public functions.

PACKAGE_VERSION_MAJOR
Package major version - 1 for 1.7.0.

PACKAGE_VERSION_MINOR
Package minor version - 7 for 1.7.0.

PACKAGE_VERSION_PATCH
Package patch version - 0 for 1.7.0.

PACKAGE_VERSION
A string with major-minor-patch-commit-id identifier of the release, e.g. 1.7.0-1216-g73f7154.

SYSCONF_DIR
System configuration dir (e.g /etc)

INSTALL_PREFIX
Install prefix (e.g. /usr)

BUILD_TYPE
Build type, e.g. Debug or Release

BUILD_INFO
CMake build type signature, e.g. Linux-x86_64-Debug

BUILD_OPTIONS
Command line used to run CMake.

COMPILER_INFO
Pathes to C and CXX compilers.

TARANTOOL_C_FLAGS
C compile flags used to build Tarantool.

TARANTOOL_CXX_FLAGS
CXX compile flags used to build Tarantool.

MODULE_LIBDIR
A path to install *.lua module files.

MODULE_LUADIR
A path to install *.so/*.dylib module files.

MODULE_INCLUDEDIR
: A path to Lua includes (the same directory where this file is contained)

MODULE_LUAPATH
: A constant added to package.path in Lua to find *.lua module files.

MODULE_LIBPATH
: A constant added to package.cpath in Lua to find *.so module files.

9.1.12 Module tuple

box_tuple_format_t

box_tuple_format_t *box_tuple_format_default(void)
: Tuple format.

  Each Tuple has associated format (class). Default format is used to create tuples which are not attach to any particular space.

box_tuple_t
: Tuple

box_tuple_t *box_tuple_new(box_tuple_format_t *format, const char *tuple, const char *tuple_end)
: Allocate and initialize a new tuple from a raw MsgPack Array data.

  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()

int box_tuple_ref(box_tuple_t *tuple)
: Increase the reference counter of tuple.

  Tuples are reference counted. All functions that return tuples guarantee that the last returned tuple is refcounted internally until the next call to API function that yields or returns another tuple.

  You should increase the reference counter before taking tuples for long processing in your code. Such tuples will not be garbage collected even if another fiber remove they from space. After processing please decrement the reference counter using box_tuple_unref(), otherwise the tuple will leak.

  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 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 next call to box_tuple_* functions.

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.

Can't use STRS/ENUM macros for them, since there is a mismatch between enum name (STRING) and type name literal ("STR"). STR is already used as Objective C type.

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 key definition with the key fields with passed typed 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 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 ref 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 ref 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 allow to iterate over fields at root level of MsgPack array.

Example:

```
box_tuple_iterator_t* it = box_tuple_iterator(tuple);
if (it == NULL) {
    // error handling using box_error_last()
}
const char* field;
while (field = box_tuple_next(it)) {
    // process raw MsgPack data
}

// rewind iterator to first position
box_tuple_rewind(it)
assert(box_tuple_position(it) == 0);

// rewind three fields
field = box_tuple_seek(it, 3);
assert(box_tuple_position(it) == 4);
```

(continues on next page)

(continued from previous page)

```
box_iterator_free(it);
```

void box_tuple_iterator_free(box_tuple_iterator_t *it)
Destroy and free tuple iterator

uint32_t box_tuple_position(box_tuple_iterator_t *it)
Return zero-based next position in iterator. That is, this function return the field id of field that will be returned by the next call to box_tuple_next(). Returned value is zero after initialization or rewind and box_tuple_field_count() after the end of iteration.

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 next call to box_tuple_* API. Requested field_no returned by 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)

9.1.13 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.

Returns 0 on success

Returns -1 on error. Perhaps a transaction has already been started

int box_txn_commit(void)
Commit the current transaction.

Returns 0 on success

Returns -1 on error. Perhaps a disk write failure

void box_txn_rollback(void)
Roll back the current transaction.

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

9.2 Internals

9.2.1 Tarantool's binary protocol

Tarantool's binary protocol is a binary request/response protocol.

Notation in diagrams

```
0    X
+----+
|    | - X bytes
+----+
 TYPE - type of MsgPack value (if it is a MsgPack object)

+====+
|    | - Variable size MsgPack object
+====+
 TYPE - type of MsgPack value
```

(continues on next page)

(continued from previous page)

```
+~~~~+
|    | - Variable size MsgPack Array/Map
+~~~~+
 TYPE - type of MsgPack value
```

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

```
TARANTOOL'S GREETING:

0                                  63
+----------------------------------+
|                                  |
| Tarantool Greeting (server version)  |
|              64 bytes            |
+--------------------+-------------+
|                    |             |
| BASE64 encoded SALT |     NULL      |
|      44 bytes      |             |
+--------------------+-------------+
64                  107           127
```

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

For data structuring and encoding, the protocol uses msgpack data format, see http://msgpack.org

The Tarantool protocol mandates use of a few integer constants serving as keys in maps used in the protocol. These constants are defined in src/box/iproto_constants.h

We list them here too:

```
-- user keys
<code>          ::= 0x00
<sync>          ::= 0x01
<schema_id>     ::= 0x05
<space_id>      ::= 0x10
<index_id>      ::= 0x11
<limit>         ::= 0x12
<offset>        ::= 0x13
<iterator>      ::= 0x14
<key>           ::= 0x20
<tuple>         ::= 0x21
<function_name> ::= 0x22
<username>      ::= 0x23
<expression>    ::= 0x27
<ops>           ::= 0x28
<data>          ::= 0x30
<error>         ::= 0x31
```

```
-- -- Value for <code> key in request can be:
-- User command codes
<select>  ::= 0x01
<insert>  ::= 0x02
<replace> ::= 0x03
<update>  ::= 0x04
<delete>  ::= 0x05
<call_16> ::= 0x06
<auth>    ::= 0x07
<eval>    ::= 0x08
<upsert>  ::= 0x09
<call>    ::= 0x0a
-- Admin command codes
<ping>    ::= 0x40

-- -- Value for <code> key in response can be:
<OK>      ::= 0x00
<ERROR>   ::= 0x8XXX
```

Both <header> and <body> are msgpack maps:

```
Request/Response:

0       5
+--------+ +============+ +====================================+
| BODY + | |         | |                      |
| HEADER | |  HEADER  | |          BODY            |
| SIZE | |         | |                      |
+--------+ +============+ +====================================+
 MP_INT      MP_MAP               MP_MAP
```

```
UNIFIED HEADER:

+================+================+====================+
|            |           |                |
|  0x00: CODE  |  0x01: SYNC  |   0x05: SCHEMA_ID  |
| MP_INT: MP_INT | MP_INT: MP_INT |  MP_INT: MP_INT    |
|            |           |                |
```

(continues on next page)

(continued from previous page)

```
+================+================+====================+
                  MP_MAP
```

They only differ in the allowed set of keys and values. The key defines the type of value that follows. If a body has no keys, 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.

```
PREPARE SCRAMBLE:

   LEN(ENCODED_SALT) = 44;
   LEN(SCRAMBLE)     = 20;

prepare 'chap-sha1' scramble:

   salt = base64_decode(encoded_salt);
   step_1 = sha1(password);
   step_2 = sha1(step_1);
   step_3 = sha1(salt, step_2);
   scramble = xor(step_1, step_3);
   return scramble;

AUTHORIZATION BODY: CODE = 0x07

+==================+=====================================+
|                  |       +-------------+-----------+ |
|  (KEY)           | (TUPLE)|  len == 9    | len == 20 | |
|   0x23:USERNAME  |   0x21:| "chap-sha1" |  SCRAMBLE | |
| MP_INT:MP_STRING | MP_INT:|  MP_STRING  |  MP_BIN   | |
|                  |       +-------------+-----------+ |
|                  |                 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:

+==================+==================+==================+
```

(continues on next page)

(continued from previous page)

```
|                 |                   |                   |
|  0x10: SPACE_ID |  0x11: INDEX_ID   |  0x12: LIMIT      |
| MP_INT: MP_INT  | MP_INT: MP_INT    | MP_INT: MP_INT    |
|                 |                   |                   |
+=================+===================+===================+
|                 |                   |                   |
|  0x13: OFFSET   |  0x14: ITERATOR   |  0x20: KEY        |
| MP_INT: MP_INT  | MP_INT: MP_INT    | MP_INT: MP_ARRAY  |
|                 |                   |                   |
+=================+===================+===================+
                    MP_MAP
```

- INSERT: CODE - 0x02 Inserts tuple into the space, if no tuple with same unique keys exists. Otherwise throw duplicate key error.
- REPLACE: CODE - 0x03 Insert a tuple into the space or replace an existing one.

```
INSERT/REPLACE BODY:

+=================+==================+
|                 |                  |
|  0x10: SPACE_ID |  0x21: TUPLE     |
| MP_INT: MP_INT  | MP_INT: MP_ARRAY |
|                 |                  |
+=================+==================+
              MP_MAP
```

- UPDATE: CODE - 0x04 Update a tuple

```
UPDATE BODY:

+=================+========================+
|                 |                        |
|  0x10: SPACE_ID |  0x11: INDEX_ID        |
| MP_INT: MP_INT  | MP_INT: MP_INT         |
|                 |                        |
+=================+========================+
|                 |        +~~~~~~~~~~+ |
|                 |        |         | |
|                 | (TUPLE)  |   OP    | |
|  0x20: KEY      |   0x21:  |         | |
| MP_INT: MP_ARRAY |  MP_INT: +~~~~~~~~~~+ |
|                 |          MP_ARRAY    |
+=================+========================+
              MP_MAP
```

```
OP:
   Works only for integer fields:
   * Addition    OP = '+' . space[key][field_no] += argument
   * Subtraction OP = '-' . space[key][field_no] -= argument
   * Bitwise AND OP = '&' . space[key][field_no] &= argument
   * Bitwise XOR OP = '^' . space[key][field_no] ^= argument
   * Bitwise OR  OP = '|' . space[key][field_no] |= argument
   Works on any fields:
   * Delete      OP = '#'
     delete <argument> fields starting
```

(continues on next page)

(continued from previous page)

```
    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
+------------+============+============+
|            |            |            |
|    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.
```

(continues on next page)

(continued from previous page)

```
'!' - 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

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      |
+=================+================++=============================+
           MP_MAP                  MP_MAP

SUBSCRIBE:

Then you must send SUBSCRIBE:

                     HEADER
+====================+====================+
|             |             |
|    0x00: 0x41    |    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
```

(continues on next page)

(continued from previous page)

```
    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      ---

+=============+ +=====================================+
|         | |                       |
|  HEADER   | |           BODY          |
|         | |                       |
+=============+ +=====================================+
   MP_MAP               MP_MAP
```

See the example in the following section.

9.2.2 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. 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}
```

A tool for reading .xlog files is Tarantool's xlog module.

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 writer process running in the WAL 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.

9.2.3 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.

9.2.4 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.

9.2.5 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.

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 apply for version 1.7:

- 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 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.

9.3 Build and contribute

9.3.1 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+
   - 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 \
    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 \
    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 \
    && 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 \
    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 1.7, to a local directory named ~/tarantool, for example:

```
$ git clone --recursive https://github.com/tarantool/tarantool.git -b 1.7 ~/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)
```

(continues on next page)

(continued from previous page)

```
$ 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

9.3.2 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)

You need to install these Python modules:

- pip, any version
- Sphinx version 1.4.4 or later
- sphinx-intl version 0.9.9
- lupa – any version

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 1.7. 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.

   Run the following command to set up a web-server. The example below is for Ubuntu, but the procedure is similar for other supported operating systems. Make sure to run it from the documentation output folder, output/en or output/ru, as in the example below:

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

5. Open your browser and enter 127.0.0.1:8000/doc/1.7 into the address box. If your local documentation build is valid, the manual will appear in the browser.
6. To contribute to documentation, use the .rst 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

---

9.3.3 Release management

How to make a minor release

```
$ git tag -a 1.4.4 -m "Next minor in 1.4 series"
$ vim CMakeLists.txt # edit CPACK_PACKAGE_VERSION_PATCH
$ git push --tags
```

Update the Web site in doc/www

Update all issues, upload the ChangeLog based on git log output. The ChangeLog must only include items which are mentioned as issues on github. If anything significant is there, which is not mentioned, something went wrong in release planning and the release should be held up until this is cleared.

Click 'Release milestone'. Create a milestone for the next minor release. Alert the driver to target bugs and blueprints to the new milestone.

9.4 Guidelines

9.4.1 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:
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.
```

(continues on next page)

(continued from previous page)

```
- 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].

9.4.2 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.

9.4.3 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 'master' branch. Our git repository is hosted on github, and can be checked out with git clone git://github.com/tarantool/tarantool.git # anonymous read-only access

If you have any questions about Tarantool internals, please post them on the developer discussion list, https://groups.google.com/forum/#!forum/tarantool. However, please be warned: Launchpad silently deletes posts from non-subscribed members, thus please be sure to have subscribed to the list prior to posting. Additionally, some engineers are always present on #tarantool channel on irc.freenode.net.

Commenting style

Use Doxygen comment format, Javadoc flavor, i.e. @tag rather than tag. The main tags in use are @param, @retval, @return, @see, @note and @todo.

Every function, except perhaps a very short and obvious one, should have a comment. A sample function comment may look like below:

```
/** Write all data to a descriptor.
 *
 * This function is equivalent to 'write', except it would ensure
 * that all data is written to the file unless a non-ignorable
 * error occurs.
 *
 * @retval 0  Success
 *
 * @reval  1  An error occurred (not EINTR)
 * /
static int
write_all(int fd, void \*data, size_t len);
```

Public structures and important structure members should be commented as well.

Header files

Use header guards. Put the header guard in the first line in the header, before the copyright or declarations. Use all-uppercase name for the header guard. Derive the header guard name from the file name, and append _INCLUDED to get a macro name. For example, core/log_io.h -> CORE_LOG_IO_H_INCLUDED. In .c (implementation) file, include the respective declaration header before all other headers, to ensure that the header is self- sufficient. Header "header.h" is self-sufficient if the following compiles without errors:

```
#include "header.h"
```

Allocating memory

Prefer the supplied slab (salloc) and pool (palloc) allocators to malloc()/free() for any performance-intensive or large memory allocations. Repetitive use of malloc()/free() can lead to memory fragmentation and should therefore be avoided.

Always free all allocated memory, even allocated at start-up. We aim at being valgrind leak-check clean, and in most cases it's just as easy to free() the allocated memory as it is to write a valgrind suppression. Freeing all allocated memory is also dynamic-load friendly: assuming a plug-in can be dynamically loaded and unloaded multiple times, reload should not lead to a memory leak.

Other

Select GNU C99 extensions are acceptable. It's OK to mix declarations and statements, use true and false.

The not-so-current list of all GCC C extensions can be found at: http://gcc.gnu.org/onlinedocs/gcc-4.3.5/gcc/C-Extensions.html

Linux kernel coding style

This is a short document describing the preferred coding style for the linux kernel. Coding style is very personal, and I won't _force_ my views on anybody, but this is what goes for anything that I have to be able to maintain, and I'd prefer it for most other things too. Please at least consider the points made here.

First off, I'd suggest printing out a copy of the GNU coding standards, and NOT read it. Burn them, it's a great symbolic gesture.

Anyway, here goes:

Chapter 1: Indentation

Tabs are 8 characters, and thus indentations are also 8 characters. There are heretic movements that try to make indentations 4 (or even 2!) characters deep, and that is akin to trying to define the value of PI to be 3.

Rationale: The whole idea behind indentation is to clearly define where a block of control starts and ends. Especially when you've been looking at your screen for 20 straight hours, you'll find it a lot easier to see how the indentation works if you have large indentations.

Now, some people will claim that having 8-character indentations makes the code move too far to the right, and makes it hard to read on a 80-character terminal screen. The answer to that is that if you need more than 3 levels of indentation, you're screwed anyway, and should fix your program.

In short, 8-char indents make things easier to read, and have the added benefit of warning you when you're nesting your functions too deep. Heed that warning.

The preferred way to ease multiple indentation levels in a switch statement is to align the "switch" and its subordinate "case" labels in the same column instead of "double-indenting" the "case" labels. e.g.:

```
switch (suffix) {
case 'G':
case 'g':
    mem <<= 30;
    break;
case 'M':
case 'm':
    mem <<= 20;
    break;
case 'K':
case 'k':
    mem <<= 10;
    /* fall through */
default:
    break;
}
```

Don't put multiple statements on a single line unless you have something to hide:

```
if (condition) do_this;
  do_something_everytime;
```

Don't put multiple assignments on a single line either. Kernel coding style is super simple. Avoid tricky expressions.

Outside of comments, documentation and except in Kconfig, spaces are never used for indentation, and the above example is deliberately broken.

Get a decent editor and don't leave whitespace at the end of lines.

Chapter 2: Breaking long lines and strings

Coding style is all about readability and maintainability using commonly available tools.

The limit on the length of lines is 80 columns and this is a strongly preferred limit.

Statements longer than 80 columns will be broken into sensible chunks. Descendants are always substantially shorter than the parent and are placed substantially to the right. The same applies to function headers with a long argument list. Long strings are as well broken into shorter strings. The only exception to this is where exceeding 80 columns significantly increases readability and does not hide information.

```
void fun(int a, int b, int c)
{
    if (condition)
        printk(KERN_WARNING "Warning this is a long printk with "
                    "3 parameters a: %u b: %u "
                    "c: %u \n", a, b, c);
    else
        next_statement;
}
```

Chapter 3: Placing Braces and Spaces

The other issue that always comes up in C styling is the placement of braces. Unlike the indent size, there are few technical reasons to choose one placement strategy over the other, but the preferred way, as shown

to us by the prophets Kernighan and Ritchie, is to put the opening brace last on the line, and put the closing brace first, thusly:

```
if (x is true) {
    we do y
}
```

This applies to all non-function statement blocks (if, switch, for, while, do). e.g.:

```
switch (action) {
case KOBJ_ADD:
    return "add";
case KOBJ_REMOVE:
    return "remove";
case KOBJ_CHANGE:
    return "change";
default:
    return NULL;
}
```

However, there is one special case, namely functions: they have the opening brace at the beginning of the next line, thus:

```
int function(int x)
{
    body of function;
}
```

Heretic people all over the world have claimed that this inconsistency is ... well ... inconsistent, but all right-thinking people know that (a) K&R are _right_ and (b) K&R are right. Besides, functions are special anyway (you can't nest them in C).

Note that the closing brace is empty on a line of its own, _except_ in the cases where it is followed by a continuation of the same statement, ie a "while" in a do-statement or an "else" in an if-statement, like this:

```
do {
    body of do-loop;
} while (condition);
```

and

```
if (x == y) {
    ..
} else if (x > y) {
    ...
} else {
    ....
}
```

Rationale: K&R.

Also, note that this brace-placement also minimizes the number of empty (or almost empty) lines, without any loss of readability. Thus, as the supply of new-lines on your screen is not a renewable resource (think 25-line terminal screens here), you have more empty lines to put comments on.

Do not unnecessarily use braces where a single statement will do.

```
if (condition)
    action();
```

This does not apply if one branch of a conditional statement is a single statement. Use braces in both branches.

```
if (condition) {
    do_this();
    do_that();
} else {
    otherwise();
}
```

Chapter 3.1: Spaces

Linux kernel style for use of spaces depends (mostly) on function-versus-keyword usage. Use a space after (most) keywords. The notable exceptions are sizeof, typeof, alignof, and __attribute__, which look somewhat like functions (and are usually used with parentheses in Linux, although they are not required in the language, as in: "sizeof info" after "struct fileinfo info;" is declared).

So use a space after these keywords: if, switch, case, for, do, while but not with sizeof, typeof, alignof, or __attribute__. E.g.,

```
s = sizeof(struct file);
```

Do not add spaces around (inside) parenthesized expressions. This example is bad:

```
s = sizeof( struct file );
```

When declaring pointer data or a function that returns a pointer type, the preferred use of '*' is adjacent to the data name or function name and not adjacent to the type name. Examples:

```
char *linux_banner;
unsigned long long memparse(char *ptr, char **retptr);
char *match_strdup(substring_t *s);
```

Use one space around (on each side of) most binary and ternary operators, such as any of these:

    = + - < > * / % | & ^ <= >= == != ? :

but no space after unary operators:

    & * + - ~ ! sizeof typeof alignof __attribute__ defined

no space before the postfix increment & decrement unary operators:

    ++ --

no space after the prefix increment & decrement unary operators:

    ++ --

and no space around the '.' and "->" structure member operators.

Do not leave trailing whitespace at the ends of lines. Some editors with "smart" indentation will insert whitespace at the beginning of new lines as appropriate, so you can start typing the next line of code right away. However, some such editors do not remove the whitespace if you end up not putting a line of code there, such as if you leave a blank line. As a result, you end up with lines containing trailing whitespace.

Git will warn you about patches that introduce trailing whitespace, and can optionally strip the trailing whitespace for you; however, if applying a series of patches, this may make later patches in the series fail by changing their context lines.

Chapter 4: Naming

C is a Spartan language, and so should your naming be. Unlike Modula-2 and Pascal programmers, C programmers do not use cute names like ThisVariableIsATemporaryCounter. A C programmer would call that variable "tmp", which is much easier to write, and not the least more difficult to understand.

HOWEVER, while mixed-case names are frowned upon, descriptive names for global variables are a must. To call a global function "foo" is a shooting offense.

GLOBAL variables (to be used only if you _really_ need them) need to have descriptive names, as do global functions. If you have a function that counts the number of active users, you should call that "count_active_users()" or similar, you should _not_ call it "cntusr()".

Encoding the type of a function into the name (so-called Hungarian notation) is brain damaged - the compiler knows the types anyway and can check those, and it only confuses the programmer. No wonder MicroSoft makes buggy programs.

LOCAL variable names should be short, and to the point. If you have some random integer loop counter, it should probably be called "i". Calling it "loop_counter" is non-productive, if there is no chance of it being mis-understood. Similarly, "tmp" can be just about any type of variable that is used to hold a temporary value.

If you are afraid to mix up your local variable names, you have another problem, which is called the function-growth-hormone-imbalance syndrome. See chapter 6 (Functions).

Chapter 5: Typedefs

Please don't use things like "vps_t".

It's a _mistake_ to use typedef for structures and pointers. When you see a

```
vps_t a;
```

in the source, what does it mean?

In contrast, if it says

```
struct virtual_container *a;
```

you can actually tell what "a" is.

Lots of people think that typedefs "help readability". Not so. They are useful only for:

(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

9.4.4 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>

9.4.5 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
...
```

Lua Module Index