Tarantool Enterprise manual

Release 1.10.4-1

Mail.Ru, Tarantool team

Dec 24, 2019

Contents

This product is the Enterprise edition of Tarantool software – a Lua application server integrated with a DBMS for deploying fault-tolerant distributed data storages.

The Enterprise edition provides an extended feature set for developing and managing clustered Tarantool applications:

- Static package for standalone Linux systems.
- Tracing module for debugging performance issues (based on OpenTracing, Zipkin, Jaeger)
- Background task manager for cluster applications
- Visual explorer for browsing Tarantool data
- Tarantool bindings to OpenLDAP
- Security audit log
- Enterprise database connectivity: Oracle and any ODBC-supported DBMS (MySQL, Microsoft SQL Server, etc.)

This document primarily concentrates on distinctive features of the Tarantool Enterprise edition. For information on the underlying open-source edition, see Tarantool manual.

CHAPTER 1

Changelog

1.1 1.10.4-1 - 2019-10-23

Major news:

- The framework for developing cluster applications (Tarantool Cartridge) has gone open source as two modules: cartridge (successor of cluster rock) and cartridge-cli (successor of tarantoolapp utility).
- We added another open source module named luacheck – a static analyzer and linter for Lua, preconfigured for Tarantool.
- We added documentation for all closed source modules.
- Meet the redesigned administrative web interface!

Changed:

- The underlying Tarantool Community version upgraded to 1.10.4-21-g9349237.
- vshard-zookeeper module updated.

Added:

- quickfix module v1.0.0.
- luarapidxml module v2.0.0.
- cartridge module v1.0.0, successor of cluster module.
- cartridge-cli module v1.0.0-1, successor of tarantoolapp utility.
- task module v0.4.0.
- membership module v2.1.2 with improved stability on large clusters.
- membership module v2.1.4.
- odbc module v0.3.0.
- cluster module v0.10.0.

- task module v0.3.0.
- odbc module v0.4.0.
- kafka module v1.0.2.
- frontend-core module v6.0.1.
- space-explorer module v1.1.0.
- oracle module v1.2.2.
- http module v1.1.0.
- avro-schema module v3.0.3.
- vshard module v0.1.12.
- icu-date module v1.3.1.
- luatest module v0.2.2.
- metrics module v0.1.6.
- queue module v1.0.4.

Removed:

- tarantoolapp utility.

1.2 1.10.3-71 - 2019-07-12

Changed:

- The underlying Tarantool Community version upgraded to 1.10.3-89-g412e943.

Added:

- Unit / integration test templates for a Tarantool cluster; available via tarantoolctl rocks test.
- .tarantoolapp.ignore files to exclude specific resources from build.
- kafka module v1.0.1 (Apache Kafka connector for Tarantool).
- tracing module v0.1.0 (for debugging performance issues in applications; based on OpenTracing/Zipkin).
- task module v0.2.0 (for managing background tasks in cluster applications).
- odbc module v0.1.0 (outgoing ODBC connector for Tarantool).
- luatest module v0.2.0 (testing framework for Tarantool).
- checks module v3.0.1.
- errors module v2.1.1.
- frontend-core module v5.0.2.
- metrics module v0.1.5.
- oracle module v1.1.6.
- cluster module v0.9.2 with new features:
  - Users, a new tab in the web interface for managing cluster admin users.

  - Multiple isolated groups for storage replica sets in a single cluster, e.g. hot or cold groups to process hot and cold data independently.
  - Integration tests helpers for luatest.

Removed:

- cluster module v0.2.0, v0.3.0.
- oracle module v1.0.0, v1.1.0.
- vshard module v0.1.5, v0.1.6.

1.3 1.10.3-29 - 2019-05-28

Changed:

- The underlying Tarantool Community version upgraded to 1.10.3-57-gd2efb0d.

Added:

- tarantoolapp pack now puts VERSION file inside of packed project.
- Ability to set up logging level per route.
- http module v0.1.6 with a new feature: setting up logging level per route.
- space-explorer module v1.0.2.
- cron-parser module v1.0.0.
- argon2 module v3.0.1 with PCI DSS grade hashing algorithms.
- metrics module v0.1.3.
- vshard module v0.1.9
- oracle module v1.1.5.
- frontend-core module v5.0.0, v5.0.1
- cluster module v0.8.0 with some bug fixes and new features:
  - role dependencies;
  - cluster cookies;
  - labels for servers.

1.4 1.10.3-5 - 2019-04-05

- The underlying Tarantool Community version upgraded to 1.10.3-6-gfbf53b9.
- The following features added:
  - Failover priority configuration via the web interface.
  - RPC module (remote calls between cluster instances).
  - Space explorer via the web interface.
  - OpenLDAP client for Tarantool.
- Instance restart now triggers configuration validation before the roles initialization.

- The web interface design has been updated.

1.5 1.10.2-41 - 2019-02-08

- The underlying Tarantool Community version has been upgraded to 1.10.2-131-g6f96bfe.
- The cluster template has been updated to allow the application to operate across several hosts (virtual machines).
- The following closed-source modules of latest versions have been added: queue, front, errors, membership, cluster, and oracle. These rocks include:
  - a new front-end core with cosmetic changes;
  - active master indication during failover;
  - the ability to disable the vshard-storage role after the end of the rebalancing process;
  - dependencies updates and minor improvements.

1.6 1.10.2-15 - 2018-12-13

- The underlying Tarantool Community version has been upgraded to 1.10.2-84-g19d471bd4.
- The checks module has been updated in the bundle.
- The custom (user-defined) cluster roles API has been added to the cluster module.
- The vshard replica set's weight parameter is now supported.

1.7 1.10.2-4 — 2018-10-31

- Sample applications demonstrating how to run Tarantool in Docker and a write-through cache to PostgreSQL have been added.
- The abilities to manually switch the replica set's master and enable automatic failover have been added to Web interface.
- The tarantoolapp utility that helps set up a development environment and pack the application in an environment-independent way has been added.

1.8 1.10.1-29 — 2018-10-04

- The sample application is now cluster-based and the cluster is orchestrated via a Web interface.
- Tarantool Enterprise release comes in an archive that includes an offline rocks repository from which you can install all necessary modules.
- Oracle connector is now included in the rocks repository. With this package, Lua applications can access Oracle databases.

1.9 1.10.1 — 2018-04-09

- The Tarantool build is now static: all dependencies are linked into a static binary. This simplifies deploy of Tarantool in the Linux environment.
- The underlying Tarantool Community version has been upgraded to 1.10.1. For all new features and bugfixes of the Community version please refer to https://github.com/tarantool/tarantool/releases.
- The modules required for integration with ZooKeeper and orchestrator are now deprecated and no longer supported.

CHAPTER 2

Setup

This chapter explains how to download and set up Tarantool Enterprise and run a sample application provided with it.

2.1 System requirements

The recommended system requirements for running Tarantool Enterprise are as follows.

2.1.1 Hardware requirements

To fully ensure the fault tolerance of a distributed data storage system, at least three physical computers or virtual servers are required.

For testing/development purposes, the system can be deployed using a smaller number of servers; however, it is not recommended to use such configurations for production.

2.1.2 Software requirements

1. As host operating systems, Tarantool Enterprise supports Red Hat Enterprise Linux and CentOS versions 7.5 and higher.

   Note: Tarantool Enterprise can run on other systemd-based Linux distributions but it is not tested on them and may not work as expected.

2. glibc 2.17-260.el7_6.6 and higher is required. Take care to check and update, if needed:

```
$ rpm -q glibc
glibc-2.17-196.el7_4.2
$ yum update glibc
```

2.1.3 Network requirements

Hereinafter, "storage servers" or "Tarantool servers" are the computers used to store and process data, and "administration server" is the computer used by the system operator to install and configure the product.

The Tarantool cluster has a full mesh topology, therefore all Tarantool servers should be able to communicate and send traffic from any TCP port to TCP ports 3000:4000.

To configure remote monitoring or to connect via the administrative console, the administration server should be able to access the following TCP ports on Tarantool servers:

- 22 to use the SSH protocol,
- ports specified in instance configuration (http_port parameter) to monitor the HTTP-metrics.

Additionally, it is recommended to apply the following settings for sysctl on all Tarantool servers:

```
$ # TCP KeepAlive setting
$ sysctl -w net.ipv4.tcp_keepalive_time=60
$ sysctl -w net.ipv4.tcp_keepalive_intvl=5
$ sysctl -w net.ipv4.tcp_keepalive_probes=5
```

This optional setup of the Linux network stack helps speed up the troubleshooting of network connectivity when the server physically fails. To achieve the maximum performance, you may also need to configure other network stack parameters that are not specific to the Tarantool DBMS. For more information, please refer to the Network Performance Tuning Guide section of the RHEL7 user documentation.

2.2 Package contents

The latest release packages of Tarantool Enterprise are available in the customer zone. at Tarantool website. Please contact support@tarantool.org for access.

Each package is distributed as a tar + gzip archive and includes the following components and features:

- static Tarantool binary for simplified deployment in Linux environments,
- selection of open and closed source modules,
- sample application walking you through all included modules.

Archive contents:

- tarantool is the main executable of Tarantool.
- tarantoolctl is the utility script for installing supplementary modules and connecting to the administrative console.
- cartridge is the utility script to help you set up a development environment for applications and pack them for easy deployment.
- examples/ is the directory containing sample applications:
  - pg_writethrough_cache/ is an application showcasing how Tarantool can cache data written to, for example, a PostgreSQL database;
  - ora_writebehind_cache/ is an application showcasing how Tarantool can cache writes and queue them to, for example, an Oracle database;
  - docker/ is an application designed to be easily packed into a Docker container;
- rocks/ is the directory containing a selection of additional open and closed source modules included in the distribution as an offline rocks repository. See the rocks reference for details.

- templates/ is the directory containing template files for your application development environment.
- deprecated/ is a set of modules that are no longer actively supported:
  - vshard-zookeeper-orchestrator is a Python application for launching orchestrator,
  - zookeeper-scm files are the ZooKeeper integration modules (require usr/ libraries).

2.3 Installation

The delivered tar + gzip archive should be uploaded to a server and unpacked:

```
$ tar xvf tarantool-enterprise-bundle-<version>.tar.gz
```

No further installation is required as the unpacked binaries are almost ready to go. Go to the directory with the binaries (tarantool-enterprise) and add them to the executable path by running the script provided by the distribution:

```
$ source ./env.sh
```

Next, set up your development environment as described in the developer's guide.

CHAPTER 3

Developer's guide

To develop an application, use Tarantool Cartridge framework that is installed as part of Tarantool Enterprise.

Here is a summary of the commands you need:

1. Create a cluster-aware application from template:

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

2. Develop your application:

```
$ cd /path/to/<app_name>
$ ...
```

3. Package your application:

```
$ cartridge pack [rpm|tgz] /path/to/<app_name>
```

4. Deploy your application:
   - For rpm package:
     1. Upload the package to all servers dedicated to Tarantool.
     2. Install the package:

        ```
        $ yum install <app_name>-<version>.rpm
        ```

     3. Launch the application.

        ```
        $ systemctl start <app_name>
        ```

   - For tgz archive:
     1. Upload the archive to all servers dedicated to Tarantool.
     2. Unpack the archive:

```
$ tar -xzvf <app_name>-<version>.tar.gz -C /home/<user>/apps
```

3. Launch the application

```
$ tarantool init.lua
```

For details and examples, please consult the open-source Tarantool documentation:

- a getting started guide that walks you through developing and deploying a simple clustered application using Tarantool Cartridge,
- a detailed manual on creating and managing clustered Tarantool applications using Tarantool Cartridge.

Further on, this guide focuses on Enterprise-specific developer features available on top of the open-source Tarantool version with Tarantool Cartridge framework:

- LDAP authorization in the web interface,
- environment-independent applications,
- sample applications with Enterprise flavors.

3.1 Implementing LDAP authorization in the web interface

If you run an LDAP server in your organization, you can connect Tarantool Enterprise to it and let it handle the authorization. In this case, follow the general recipe where in the first step add the ldap module to the .rockspec file as a dependency and consider implementing the check_password function the following way:

```
-- auth.lua
-- Require the LDAP module at the start of the file
local ldap = require('ldap')
...
-- Add a function to check the credentials
local function check_password(username, password)

    -- Configure the necessary LDAP parameters
    local user = string.format("cn=%s,ou=superheros,dc=glauth,dc=com", username)

    -- Connect to the LDAP server
    local ld, err = ldap.open("localhost:3893", user, password)

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

3.2 Delivering environment-independent applications

Tarantool Enterprise allows you to build environment-independent applications.

An environment-independent application is an assembly (in one directory) of:

- files with Lua code,
- tarantool executable,
- plugged external modules (if necessary).

When started by the tarantool executable, the application provides a service.

The modules are Lua rocks installed into a virtual environment (under the application directory) similar to Python's virtualenv and Ruby's bundler.

Such an application has the same structure both in development and production-ready phases. All the application-related code resides in one place, ready to be packed and copied over to any server.

3.2.1 Packaging applications

Once custom cluster role(s) are defined and the application is developed, pack it and all its dependencies (module binaries) together with the tarantool executable.

This will allow you to upload, install, and run your application on any server in one go.

To pack the application, say:

```
$ cartridge pack [rpm|tgz] /path/to/<app_name>
```

where specify a path to your development environment – the Git repository containing your application code, – and one of the following build options:

- rpm to build an RPM package (recommended), or
- tgz to build a tar + gz archive (choose this option only if you do not have root privileges on servers dedicated for Tarantool Enterprise).

This will create a package (or compressed archive) named <app_name>-<version_tag>-<number_of_commits> (e.g., myapp-1.2.1-12.rpm) containing your environment-independent application.

Next, proceed to deploying packaged applications (or archived ones) on your servers.

3.2.2 Deploying packaged applications

To deploy your packaged application, do the following on every server dedicated for Tarantool Enterprise:

1. Upload the package created in the previous step.
2. Install:

   ```
   $ yum install <app_name>-<version>.rpm
   ```

3. Start one or multiple Tarantool instances with the corresponding services as described below.
   - A single instance:

     ```
     $ systemctl start <app_name>
     ```

     This will start an instantiated systemd service that will listen to port 3301.
   - Multiple instances on one or multiple servers:

```
$ systemctl start <app_name>@instance_1
$ systemctl start <app_name>@instance_2
...
$ systemctl start <app_name>@instance_<number>
```

where <app_name>@instance_<number> is the instantiated service name for systemd with an incremental <number> (unique for every instance) to be added to the 3300 port the instance will listen to (e.g., 3301, 3302, etc.).

4. In case it is a cluster-aware application, proceed to deploying the cluster.

To stop all services on a server, use the systemctl stop command and specify instance names one by one. For example:

```
$ systemctl stop <app_name>@instance_1 <app_name>@instance_2 ... <app_name>@instance_<N>
```

3.2.3 Deploying archived applications

While the RPM package places your application to /usr/share/tarantool/<app_name> on your server by default, the tar + gz archive does not enforce any structure apart from just the <app_name>/ directory, so you are responsible for placing it appropriately.

Note: RPM packages are recommended for deployment. Deploy archives only if you do not have root privileges.

To place and deploy the application, do the following on every server dedicated for Tarantool Enterprise:

1. Upload the archive, decompress, and extract it to the /home/<user>/apps directory:

   ```
   $ tar -xzvf <app_name>-<version>.tar.gz -C /home/<user>/apps
   ```

2. Start Tarantool instances with the corresponding services.

   To manage instances and configuration, use tools like ansible, systemd, and supervisord.

3. In case it is a cluster-aware application, proceed to deploying the cluster.

3.2.4 Upgrading code

All instances in the cluster are to run the same code. This includes all the components: custom roles, applications, module binaries, tarantool and tarantoolctl (if necessary) executables.

Pay attention to possible backward incompatibility that any component may introduce. This will help you choose a scenario for an upgrade in production. Keep in mind that you are responsible for code compatibility and handling conflicts should inconsistencies occur.

To upgrade any of the components, prepare a new version of the package (archive):

1. Update the necessary files in your development environment (directory):
   - Your own source code: custom roles and/or applications.
   - Module binaries.
   - Executables. Replace them with ones from the new bundle.
2. Increment the version as described in application versioning.

3. Repack the updated files as described in packaging applications.
4. Choose an upgrade scenario as described in production upgrade section.

3.3 Running sample applications

The Enterprise distribution package includes sample applications in the examples/ directory that showcase basic Tarantool functionality.

Sample applications:

- Write-through cache application for PostgreSQL
- Write-behind cache application for Oracle
- Hello-world application in Docker

3.3.1 Write-through cache application for PostgreSQL

The example in pg_writethrough_cache/ shows how Tarantool can cache data written through it to a PostgreSQL database to speed up the reads.

The sample application requires a deployed PostgreSQL database and the following rock modules:

```
$ tarantoolctl rocks install http
$ tarantoolctl rocks install pg
$ tarantoolctl rocks install argparse
```

Look through the code in the files to get an understanding of what the application does.

To run the application for a local PostgreSQL database, say:

```
$ tarantool cachesrv.lua --binary-port 3333 --http-port 8888 --database postgresql://localhost/postgres
```

3.3.2 Write-behind cache application for Oracle

The example in ora-writebehind-cache/ shows how Tarantool can cache writes and queue them to an Oracle database to speed up both writes and reads.

Application requirements

The sample application requires:

- deployed Oracle database;
- Oracle tools: Instant Client and SQL Plus, both of version 12.2;

Note: In case the Oracle Instant Client errors out on .so files (Oracle's dynamic libraries), put them to some directory and add it to the LD_LIBRARY_PATH environment variable.

For example: export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$PWD/<path_to_so_files>

- rock modules listed in the rockspec file.

To install the modules, run the following command in the examples/ora_writebehind_cache directory:

```
$ tarantoolctl rocks make oracle_rb_cache-0.1.0-1.rockspec
```

If you do not have a deployed Oracle instance at hand, run a dummy in a Docker container:

1. In browser, log in to Oracle container registry, click Database, and accept the Oracle's Enterprise Terms and Restrictions.
2. In the ora-writebehind-cache/ directory, log in to the repository under the Oracle account, pull, and run an image using the prepared scripts:

```
$ docker login container-registry.oracle.com
Login:
Password:
Login Succeeded
$ docker pull container-registry.oracle.com/database/enterprise:12.2.0.1
$ docker run -itd \
  -p 1521:1521 \
  -p 5500:5500 \
  --name oracle \
  -v "$(pwd)"/setupdb/configDB.sh:/home/oracle/setup/configDB.sh \
  -v "$(pwd)"/setupdb/runUserScripts.sh:/home/oracle/setup/runUserScripts.sh \
  -v "$(pwd)"/startupdb:/opt/oracle/scripts/startup \
  container-registry.oracle.com/database/enterprise:12.2.0.1
```

When all is set and done, run the example application.

Running write-behind cache

To launch the application, run the following in the examples/ora_writebehind_cache directory:

```
$ tarantool init.lua
```

The application supports the following requests:

- Get: GET http://<host>:<http_port>/account/id;
- Add: POST http://<host>:<http_port>/account/ with the following data:

```
{"clng_clng_id":1,"asut_asut_id":2,"creation_data":"01-JAN-19","navi_user":"userName"}
```

- Update: POST http://<host>:<http_port>/account/id with the same data as in the add request;
- Remove: DELETE http://<host>:<http_port>/account/id where id is an account identifier.

Look for sample CURL scripts in the examples/ora_writebehind_cache/testing directory and check the README.md for more information on implementation.

3.3.3 Hello-world application in Docker

The example in the docker/ directory contains a hello-world application that you can pack in a Docker container and run on CentOS 7.

The hello.lua file is the entry point and it is very bare-bones, so you can add your own code here.

1. To build the container, say:

```
$ docker build -t tarantool-enterprise-docker -f Dockerfile ../..
```

2. To run it:

```
$ docker run --rm -t -i tarantool-enterprise-docker
```

CHAPTER 4

Cluster administrator's guide

This guide focuses on Enterprise-specific administration features available on top of the open-source Tarantool version with Tarantool Cartridge framework:

- space explorer
- upgrade of environment-independent applications in production

Otherwise, please consult the open-source Tarantool documentation for:

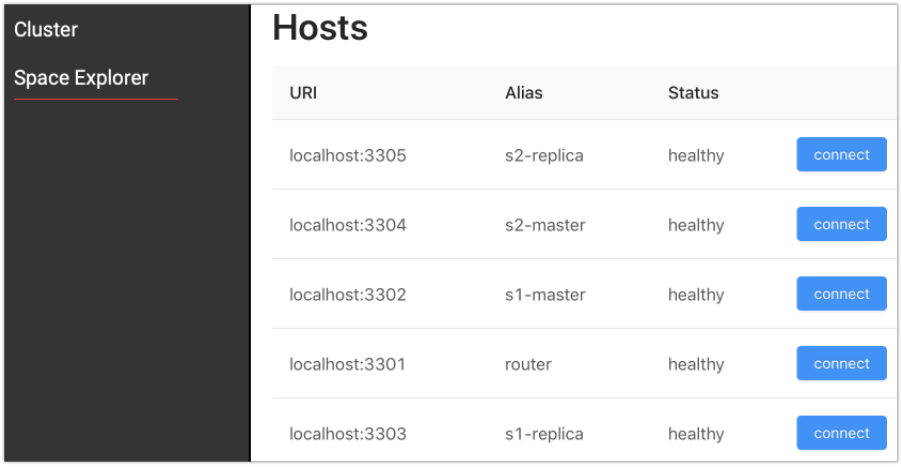
- basic information on deploying and managing a Tarantool cluster
- more information on managing Tarantool instances.

4.1 Exploring spaces

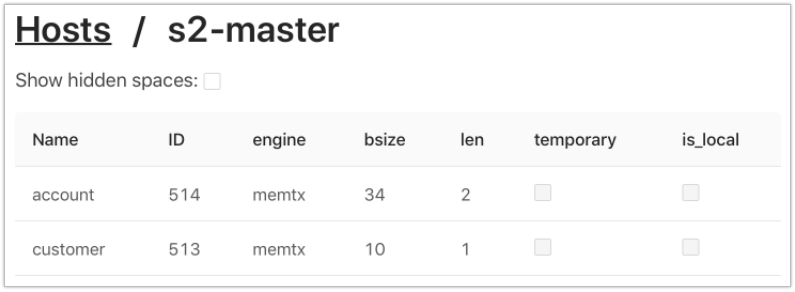
The web interface lets you connect (in the browser) to any instance in the cluster and see what spaces it stores (if any) and their contents.

To explore spaces:

1. Open the Space Explorer tab in the menu on the left:

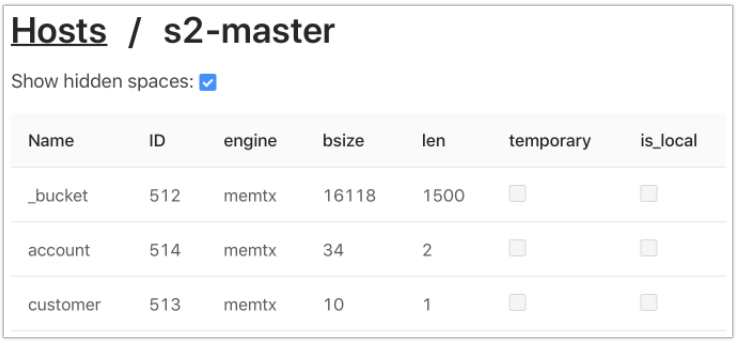


2. Click connect next to an instance that stores data. The basic sanity-check (test.py) of the example application puts sample data to one replica set (shard), so its master and replica store the data in their spaces:

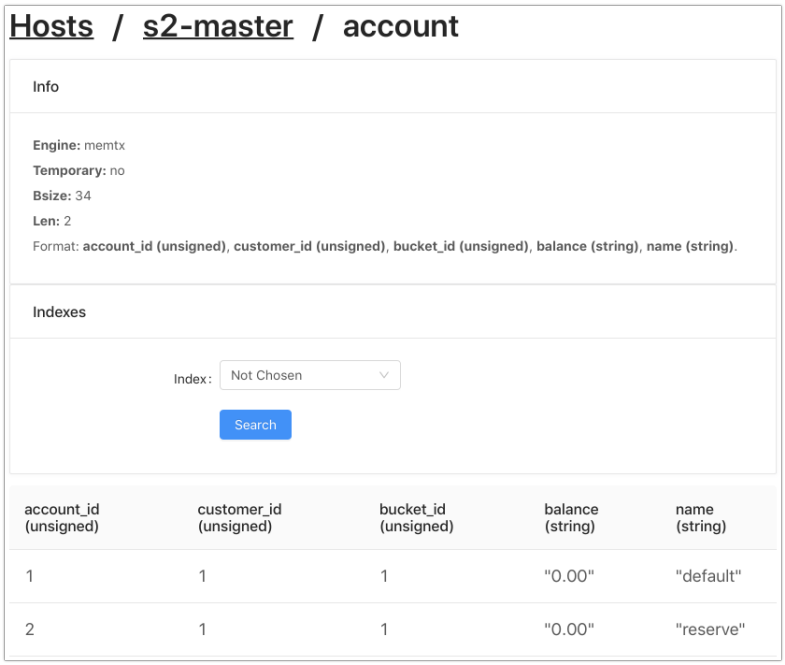


When connected to a instance, the space explorer shows a table with basic information on its spaces. For more information, see the box.space reference.

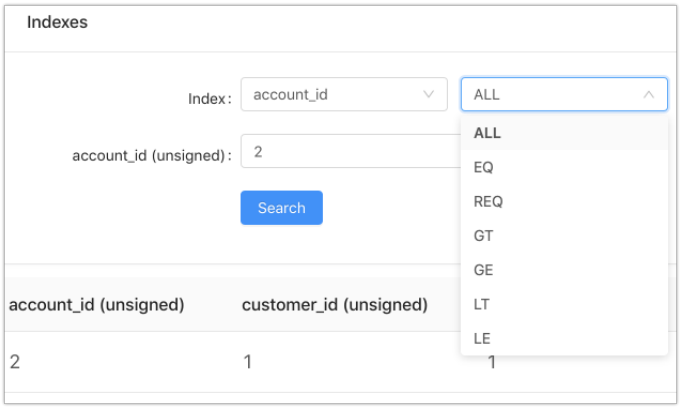
To see hidden spaces, tick the corresponding checkbox:



3. Click the space's name to see its format and contents:



To search the data, select an index and, optionally, its iteration type from the drop-down lists, and enter the index value:



4.2 Upgrading in production

To upgrade either a single instance or a cluster, you need a new version of the packaged (archived) application.

A single instance upgrade is simple:

1. Upload the package (archive) to the server.
2. Stop the current instance.
3. Deploy the new one as described in deploying packaged applications (or archived ones).

4.2.1 Cluster upgrade

To upgrade a cluster, choose one of the following scenarios:

- Cluster shutdown. Recommended for backward-incompatible updates, requires downtime.
- Instance by instance. Recommended for backward-compatible updates, does not require downtime.

To upgrade the cluster, do the following:

1. Schedule a downtime or plan for the instance-by-instance upgrade.
2. Upload a new application package (archive) to all servers.

Next, execute the chosen scenario:

- Cluster shutdown:
  1. Stop all instances on all servers.
  2. Deploy the new package (archive) on every server.
- Instance by instance. Do the following in every replica set in succession:
  1. Stop a replica on any server.
  2. Deploy the new package (archive) in place of the old replica.
  3. Promote the new replica to a master (see Switching the replica set's master section in the Tarantool manual).
  4. Redeploy the old master and the rest of the instances in the replica set.
  5. Be prepared to resolve possible logic conflicts.

CHAPTER 5

Security hardening guide

This guide explains how to enhance security in your Tarantool Enterprise cluster using built-in features and provides general recommendations on security hardening.

Tarantool Enterprise does not provide a dedicated API for security control. All the necessary configuration can be done via an administrative console or initialization code.

5.1 Built-in security features

Tarantool Enterprise has the following built-in security features:

- authentication,
- access control,
- audit log;

And backup functionality:

- snapshotting (physical),
- dumping (logical).

The following sections describe the features and provide links to detailed documentation.

5.1.1 Authentication

Tarantool Enterprise supports password-based authentication and allows for two types of connections:

- via an administrative console and
- over a binary port for read and write operations and procedure invocation.

For more information on authentication and connection types, see the security section of the Tarantool manual.

In addition, Tarantool provides the following functionality:

- sessions – states which associate connections with users and make Tarantool API available to them after authentication,
- authentication triggers which execute actions on authentication events.
- third-party (external) authentication protocols and services such as LDAP or Active Directory – supported in the web interface, but unavailable on the binary-protocol level.

5.1.2 Access control

Tarantool Enterprise provides the means for administrators to prevent unauthorized access to the database and to certain functions.

Tarantool recognizes:

- different users (guests and administrators),
- privileges associated with users,
- roles (containers for privileges) granted to users;

And divides system space into:

- _user space to store usernames and hashed passwords for authentication,
- _priv space to store privileges for access control.

For more information, see the access control section of the Tarantool manual.

Users who create objects (spaces, indexes, users, roles, sequences, and functions) in the database become their owners and automatically acquire privileges for what they create. For more information, see the owners and privileges section of the Tarantool manual.

5.1.3 Audit log

Tarantool Enterprise has a built-in audit log that records events such as:

- authentication successes and failures;
- connection closures;
- creation, removal, enabling, and disabling of users;
- changes of passwords, privileges, and roles;
- denials of access to database objects;

Audit log contains:

- timestamps,
- usernames of users who performed actions,
- event types (e.g. user_create, user_enable, disconnect, etc),
- descriptions.

Audit log has two configuration parameters:

- audit_log = <PATH_TO_FILE> which is similar to the log parameter; it tells Tarantool to record audit events to a specific file;
- audit_nonblock which is similar to the log_nonblock parameter.

For more information on logging, see the following:

- logs section of the Tarantool manual,
- logging section of the configuration reference,
- appendix A in this document.

Access permissions to audit log files can be set up as to any other Unix file system object – via chmod.

5.2 Recommendations on security hardening

This section lists recommendations that can help you harden the cluster's security.

5.2.1 Traffic encryption

Tarantool Enterprise does not encrypt traffic over binary connections (i.e., between servers in the cluster). To secure such connections, consider:

- setting up connection tunneling, or
- encrypting the actual data stored in the database.

For more information on data encryption, see the crypto module reference.

The HTTP server module provided by rocks does not support the HTTPS protocol. To set up a secure connection for a client (e.g., REST service), consider hiding the Tarantool instance (router if it is a cluster of instances) behind a Nginx server and setting up an SSL certificate for it.

To make sure that no information can be intercepted 'from the wild', run Nginx on the same physical server as the instance and set up their communication over a Unix socket. For more information, see the socket module reference.

5.2.2 Firewall configuration

To protect the cluster from any unwanted network activity 'from the wild', configure the firewall on each server to allow traffic on ports listed in network requirements.

If you are using static IP addresses, whitelist them, again, on each server as the cluster has a full mesh network topology. Consider blacklisting all the other addresses on all servers except the router (running behind the Nginx server).

Tarantool Enterprise does not provide defense against DoS or DDoS attacks. Consider using third-party software instead.

5.2.3 Data integrity

Tarantool Enterprise does not keep checksums or provide the means to control data integrity. However, it ensures data persistence using a write ahead log, regularly snapshots the entire data set to disk, and checks the data format whenever it reads the data back from the disk. For more information, see the data persistence section of the Tarantool manual.

CHAPTER 6

Modules reference

6.1 ldap

6.1.1 LDAP client library for tarantool

This library allows you to authenticate in a LDAP server and perform searches.

Usage example

First, download glauth, a simple Go-based LDAP server using the following command:

```
./download_glauth.sh
```

Then run glauth:

```
./glauth -c glauth_test.cfg
```

Then run the following tarantool script in a separate terminal

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

local ldap = require('ldap')
local yaml = require('yaml')

local user = "cn=johndoe,ou=superheros,dc=glauth,dc=com"
local password = "dogood"

local ld = assert(ldap.open("localhost:3893", user, password))

local iter = assert(ldap.search(ld,
    {base="dc=glauth,dc=com",
     scope="subtree",
```

(continues on next page)

(continued from previous page)

```
    sizelimit=10,
    filter="(objectclass=*)"}))

for entry in iter do
   print(yaml.encode(entry))
end
```

Usage ldap for authorization in the web interface

See this doc page

6.1.2 Changelog

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog and this project adheres to Semantic Versioning.

[Unreleased]

Added

- Gitlab CI testing

[1.0.0] - 2019-04-03

Added

- Basic functionality
- Luarock-based packaging
- Build without any dependencies but Tarantool Enterprise

6.2 task

6.2.1 Module *scheduler*

Task scheduler (a cartridge role).

Functions

init (opts)

Initialize the scheduler, start cron background fiber.

Parameters:

- opts:

  - runner: (string) name of the runner module name, default is task.runner.local
  - storage: (string) name of the storage module, default is task.storage.local

get_tasks ()

List registered tasks.

get_task_log (opts)

List task execution log, ordered by creation time.

Parameters:

- opts:
  - filter: (table) must contain either an id number, or an array of names
  - limit: (number) the maximum length of a single task log fetched from storage
  - created: (string) ISO 8601 timestamp, acts as offset for pagination

start (name)

Start a task.

Parameters:

- name: (string) name of the task

stop (id)

Stop a running or pending task.

Parameters:

- id: (string) name of the task

forget (id)

Remove task execution log record from storage.

Parameters:

- id: (string) name of the task

start_periodical_task (name)

Start a periodical task.

Parameters:

- name: (string) name of the task

register (tasks)

Register available tasks. Starts launching periodical and continuous tasks, allows to start single_shot tasks.

Parameters:

- tasks: (table) names of tasks

6.2.2 Module *roles.scheduler*

Task manager (a cartridge role).

Handles setting available tasks from the cluster config, handles scheduler fibers (including failover). In a basic case, it sets up a scheduler, a task storage, and a task runner on the same node.

6.2.3 Module *roles.runner*

Local task runner module, used by default.

You must provide the same interface if you want to write your own one. It is expected to take tasks from storage, run them, and complete them.

6.2.4 Module *roles.storage*

Local task storage module, used by default.

You must provide the same interface if you want to write your own one.

Functions

select (index, key, opts)

Select task log.

Parameters:

- index: (string) index to iterate over; default is id
- key: (string) index key value
- opts: (optional table) compatible with[vanilla Tarantool iterator parameters](https://www.tarantool.io/en/doc/1.10/book/box/box_index/#lua-function.index_object.select)

6.2.5 Task Manager for Tarantool Enterprise

@lookup README.md

Task manager module allows you to automate several types of background jobs:

- periodical, that need to be launched according to cron-like schedule;
- continuous, that need to be working at all times;
- single_shot, that are launched manually by the operations team.

You get the following features out-of-the-box:

- configurable schedule of periodic tasks;
- guarding and restarting continuous tasks;
- stored log of task launches;
- API and UI for launching tasks and observing launch results.

Task manager comes with several built-in cluster roles:

- task.roles.scheduler, a module which allows to configure launchable tasks;
- task.roles.runner, a cluster-aware stateless task runner;
- task.roles.storage, a cluster-aware dedicated task contents storage;
- plugin-based API which allows you to provide you own storage module (e. g. distributed, or external to Tarantool cluster), or your own runner module, providing more tooling for your needs.

Basic usage (single-node application)

1) Embed the following to the instance file:

```
...
local task = require('task')
...
cartridge.cfg({
  roles = {'my_role', ...}
})
task.init_webui()
```

2) Add to your role dependency on task scheduler, runner and storage roles:

```
return {
    ...
    dependencies = {
        'task.roles.storage',
        'task.roles.scheduler',
        'task.roles.runner'
    }
}
```

3) Add tasks section to your cluster configuration:

```
tasks:
  my_task:
    kind: periodical
    func_name: my_module.my_task
    schedule: "*/* 1 * * * *"
```

4) That's it! my_task function will be launched every minute.

Advanced usage (multi-node installation)

1) Embed the following to the instance file:

```
...

...
local task = require('task')
cartridge.cfg({
  roles = {
   ...
   'task.roles.scheduler',
   'task.roles.storage',
   'task.roles.runner'
  }
})

task.init_webui()
```

2) Enable the task scheduler role on a dedicated node in your cluster (after deployment). If you set up a big cluster, don't set up more than one replica set with the scheduler.

3) Enable the task storage role on a dedicated node in your cluster (after deployment), possibly on the same node as task scheduler. If you set up a big cluster, don't set up more than one replica set with the storage.

4) Enable the task runner role on dedicated stateless nodes in your cluster (after deployment) - as many as you may need.

Advanced usage (sharded storage)

1) Embed the following to the instance file:

```
...
local task = require('task')
cartridge.cfg({
  roles = {
   ...
   'task.roles.sharded.scheduler',
   'task.roles.sharded.storage',
   'task.roles.sharded.runner'
  }
})

task.init_webui()
```

2) Enable the task scheduler role on a dedicated node in your cluster (after deployment). If you set up a big cluster, don't set up more than one replica set with the scheduler.

3) Enable the task storage role on the nodes of some vshard group (or an all storage nodes). Set up cartridge built-in vshard-storage role on these nodes.

4) Enable the task runner role on dedicated stateless nodes in your cluster (after deployment) - as many as you may need.

Tasks configuration

Tasks are configured via the scheduler cluster role. An example of valid role configuration:

```
tasks:
  my_reload:
    kind: periodical
    func_name: my_module.cache_reload
    schedule: "*/* 1 * * * *"
    time_to_resolve: 180
  my_flush:
    kind: single_shot
    func_name: my_module.cache_flush
    args:

      - some_string1
      - some_string2
  push_metrics:
    kind: continuous
    func_name: my_module.push_metrics
    pause_sec: 30
```

- Every task must have a unique name (subsection name in config).
- Each task must have a kind: periodical, continuous, single_shot.
- Each task must have a func_name - name of the function (preceded by the name of the module) which will be invoked.
- Each task may have time_to_resolve - timeout after which a running task is considered lost (failed).
- Each task may have args - an array of arguments which will be passed to the function (to allow basic parametrization)
- Periodical tasks also must have a schedule, conforming with ccronexpr (basically, cron with seconds).
- Tasks may have a pause_sec - pause between launches (60 seconds by default).

You may set up default task config for your application in task.init() call:

```
task.init({
  default_config = {
    my_task = {
      kind = 'single_shot',
      func_name = 'dummy_task.dummy',
    }
  }
})
```

Default config will be applied if no tasks are set in clusterwide config. task.init() should be called prior to cluster.cfg().

Advanced usage

Running a task via API

Everything visible from the UI is available via the API. You may look up requests in the UI or in the cartridge graphql schema.

```
curl -w "\n" -X POST http://127.0.0.1:8080/admin/api --fail -d@- <<'QUERY'
{"query": "mutation { task { start(name: "my_reload") } }"}
QUERY
```

Supplying custom runner and storage

Embed the following to your instance file

```
task.init({
    runner = 'my_tasks.my_runner',
    storage = 'my_tasks.my_storage'
})
...
cartridge.cfg{...}
task.init_webui()
```

Be sure to call task.init() it prior to cartridge.cfg, so that custom options would be provided by the time role initialization starts.

You may set up then only task scheduler role, and handle storages and runners yourself.

Writing your own runner and storage

Runner module must expose api member with the following functions:

- stop_task

storage module must expose api member with the following functions:

- select
- get
- delete
- put
- complete
- take
- cancel
- wait

For more details refer to built-in runner and storage documentation

6.2.6 Changelog

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog and this project adheres to Semantic Versioning.

[Unreleased]

[0.6.1]

- Bugfixes and minor improvements

[0.6.0]

Added:

- Ability to specify task args in config
- several bugfixes

[0.5.0]

Added:

- task storages can now be set up on vshard storages
- default task configuration can be set up in task.init()

[0.4.0]

Changed:

- task now depends on cartridge instead of cluster
- cartridge dependency bumped to 1.0.0

[0.3.0]

Added: - Cluster-aware task storage role - Cluster-aware task runner role

Changed: - Module initialization API changed: now task scheduler role does not start runner and storage by default; you must handle it via role dependencies or via respective cluster roles

[0.2.0]

Added: - Basic functionality - scheduler, local task storage, local task runner, cluster role, cluster UI

6.3 tracing

6.3.1 Module *opentracing*

OpenTracing API module entrypoint

Functions

set_global_tracer (tracer)

Set global tracer

Parameters:

- tracer: (table)

get_global_tracer ()

Get global tracer

Returns:

(table) tracer

start_span (name, opts)

Start root span

Parameters:

- name: (string)
- opts: table specifying modifications to make to thenewly created span. The following parameters are supported: trace_id , references ,a list of referenced spans; start_time , the time to mark when the spanbegins (in microseconds since epoch); tags , a table of tags to add tothe created span.
  - trace_id: (optional string)
  - child_of: (optional table)
  - references: (optional table)
  - tags: (optional table)

Returns:

(table) span

start_span_from_context (context, name)

Start new child span from context

Parameters:

- context: (table)
- name: (string)

Returns:

(table) span

trace_with_context (name, ctx, fun, ...)

Trace function with context by global tracer This function starts span from global tracer and finishes it after execution

Parameters:

- name: (string) span name
- ctx: (table) context
- fun: (function) wrapped function
- ...: (vararg) function's arguments

Returns:

(vararg) result

Or

(nil) nil

(string) err error message

Usage:

```
-- Process HTTP request
local ctx = opentracing.extractors.http(req.headers)
local result, err = opentracing.trace_with_context('process_data', ctx, process_data, req.body)
-- Wrap functions. In example we create root span that generates two child spans
local span = opentracing.start_span()
local result, err = opentracing.trace_with_context('format_string', span:context(), format, str)
if not result ~= nil then
    print('Error: ', err)
end
opentracing.trace_with_context('print_string', span:context(), print, result)
span:finish()
```

trace (name, fun, ...)

Trace function by global tracer This function starts span from global tracer and finishes it after execution

Parameters:

- name: (string) span name
- fun: (function) wrapped function
- ...: (vararg) function's arguments

Returns:

(vararg) result

Or

(nil) nil

(string) err error message

Usage:

```
local result, err = opentracing.trace_with_context('process_data', process_data, req.body)
```

6.3.2 Module *opentracing.span*

Span represents a unit of work executed on behalf of a trace.

Examples of spans include a remote procedure call, or a in-process method call to a sub-component. Every span in a trace may have zero or more causal parents, and these relationships transitively form a DAG. It is

common for spans to have at most one parent, and thus most traces are merely tree structures. The internal data structure is modeled off the ZipKin Span JSON Structure This makes it cheaper to convert to JSON for submission to the ZipKin HTTP api, which Jaegar also implements.

You can find it documented in this OpenAPI spec: https://github.com/openzipkin/zipkin-api/blob/7e33e977/zipkin2-api.yaml#L280

Functions

new (tracer, context, name, start_timestamp)

Create new span

Parameters:

- tracer: (table)
- context: (table)
- name: (string)
- start_timestamp: (optional number)

Returns:

(table) span

context (self)

Provides access to the SpanContext associated with this Span The SpanContext contains state that propagates from Span to Span in a larger tracer.

Parameters:

- self: (table)

Returns:

(table) context

tracer (self)

Provides access to the Tracer that created this span.

Parameters:

- self: (table)

Returns:

(table) tracer the Tracer that created this span.

set_operation_name (self, name)

Changes the operation name

Parameters:

- self: (table)

- name: (string)

Returns:

(table) tracer

start_child_span (self, name, start_timestamp)

Start child span

Parameters:

- self: (table)
- name: (string)
- start_timestamp: (optional number)

Returns:

(table) child span

finish (self, opts)

Indicates the work represented by this Span has completed or terminated.

If finish is called a second time, it is guaranteed to do nothing.

Parameters:

- self: (table)
- opts:
  - finish_timestamp: (number) a timestamp represented by microsecondssince the epoch to mark when the span ended. If unspecified, the currenttime will be used.
  - error: (string) add error tag

Returns:

(boolean) true

Or

(boolean) false

(string) error

set_tag (self, key, value)

Attaches a key/value pair to the Span .

The value must be a string, bool, numeric type, or table of such values.

Parameters:

- self: (table)
- key: (string) key or name of the tag. Must be a string.
- value: (any) value of the tag

Returns:

(boolean) true

get_tag (self, key)

Get span's tag

Parameters:

- self: (table)
- key: (string)

Returns:

(any) tag value

each_tag (self)

Get tags iterator

Parameters:

- self: (table)

Returns:

(function) iterator

(table) tags

get_tags (self)

Get copy of span's tags

Parameters:

- self: (table)

Returns:

(table) tags

log (self, key, value, timestamp)

Log some action

Parameters:

- self: (table)
- key: (table)
- value: (table)
- timestamp: (optional number)

Returns:

(boolean) true

log_kv (self, key_values, timestamp)

Attaches a log record to the Span .

Parameters:

- self: (table)
- key_values: (table) a table of string keys and values of string, bool, ornumeric types
- timestamp: (optional number) an optional timestamp as a unix timestamp.defaults to the current time

Returns:

(boolean) true

Usage:

```
span:log_kv({
  ["event"] = "time to first byte",
  ["packet.size"] = packet:size()})
```

each_log (self)

Get span's logs iterator

Parameters:

- self: (table)

Returns:

(function) log iterator

(table) logs

set_baggage_item (self, key, value)

Stores a Baggage item in the Span as a key/value pair.

Enables powerful distributed context propagation functionality where arbitrary application data can be carried along the full path of request execution throughout the system.

Note 1: Baggage is only propagated to the future (recursive) children of this Span .

Note 2: Baggage is sent in-band with every subsequent local and remote calls, so this feature must be used with care.

Parameters:

- self: (table)
- key: (string) Baggage item key
- value: (string) Baggage item value

Returns:

(boolean) true

get_baggage_item (self, key)

Retrieves value of the baggage item with the given key.

Parameters:

- self: (table)
- key: (string)

Returns:

(string) value

each_baggage_item (self)

Returns an iterator over each attached baggage item

Parameters:

- self: (table)

Returns:

(function) iterator

(table) baggage

set_component (self, component)

Set component tag (The software package, framework, library, or module that generated the associated Span.)

Parameters:

- self: (table)
- component: (string)

set_http_method (self, method)

Set HTTP method of the request for the associated Span

Parameters:

- self: (table)
- method: (string)

set_http_status_code (self, status_code)

Set HTTP response status code for the associated Span

Parameters:

- self: (table)
- status_code: (number)

set_http_url (self, url)

Set URL of the request being handled in this segment of the trace, in standard URI format

Parameters:

- self: (table)
- url: (string)

set_http_host (self, host)

Set the domain portion of the URL or host header. Used to filter by host as opposed to ip address.

Parameters:

- self: (table)
- host: (string)

set_http_path (self, path)

Set the absolute http path, without any query parameters. Used as a filter or to clarify the request path for a given route. For example, the path for a route "/objects/:objectId" could be "/objects/abdc-ff". This does not limit cardinality like HTTP_ROUTE("http.route") can, so is not a good input to a span name.

The Zipkin query api only supports equals filters. Dropping query parameters makes the number of distinct URIs less. For example, one can query for the same resource, regardless of signing parameters encoded in the query line. Dropping query parameters also limits the security impact of this tag.

Parameters:

- self: (table)
- path: (string)

set_http_route (self, route)

Set the route which a request matched or "" (empty string) if routing is supported, but there was no match. Unlike HTTP_PATH("http.path"), this value is fixed cardinality, so is a safe input to a span name function or a metrics dimension. Different formats are possible. For example, the following are all valid route templates: "/users" "/users/:userId" "/users/*"

Route-based span name generation often uses other tags, such as HTTP_METHOD("http.method") and HTTP_STATUS_CODE("http.status_code"). Route-based names can look like "get /users/{userId}", "post /users", "get not_found" or "get redirected".

Parameters:

- self: (table)
- route: (string)

set_http_request_size (self, host)

Set the size of the non-empty HTTP request body, in bytes. Large uploads can exceed limits or contribute directly to latency.

Parameters:

- self: (table)
- host: (string)

set_response_size (self, host)

Set the size of the non-empty HTTP response body, in bytes. Large downloads can exceed limits or contribute directly to latency.

Parameters:

- self: (table)
- host: (string)

set_peer_address (self, address)

Set remote "address", suitable for use in a networking client library. This may be a "ip:port", a bare "hostname", a FQDN, or even a JDBC substring like "mysql://prod-db:3306"

Parameters:

- self: (table)
- address: (string)

set_peer_hostname (self, hostname)

Set remote hostname

Parameters:

- self: (table)
- hostname: (string)

set_peer_ipv4 (self, IPv4)

Set remote IPv4 address as a .-separated tuple

Parameters:

- self: (table)
- IPv4: (string)

set_peer_ipv6 (self, IPv6)

Set remote IPv6 address as a string of colon-separated 4-char hex tuples

Parameters:

- self: (table)
- IPv6: (string)

set_peer_port (self, port)

Set remote port

Parameters:

- self: (table)
- port: (number)

set_peer_service (self, service_name)

Set remote service name (for some unspecified definition of "service")

Parameters:

- self: (table)
- service_name: (string)

set_sampling_priority (self, priority)

Set sampling priority If greater than 0, a hint to the Tracer to do its best to capture the trace. If 0, a hint to the trace to not-capture the trace. If absent, the Tracer should use its default sampling mechanism.

Parameters:

- self: (table)
- priority: (number)

set_kind (self, kind)

Set span's king Either "client" or "server" for the appropriate roles in an RPC, and "producer" or "consumer" for the appropriate roles in a messaging scenario.

Parameters:

- self: (table)
- kind: (string)

set_client_kind (self)

Set client kind to span

Parameters:

- self: (table)

set_server_kind (self)

Set server kind to span

Parameters:

- self: (table)

set_producer_kind (self)

Set producer kind to span

Parameters:

- self: (table)

set_consumer_kind (self)

Set consumer kind to span

Parameters:

- self: (table)

6.3.3 Module *opentracing.span_context*

SpanContext represents Span state that must propagate to descendant Span 's and across process boundaries.

SpanContext is logically divided into two pieces: the user-level "Baggage" (see Span.set_baggage_item and Span.get_baggage_item) that propagates across Span boundaries and any tracer-implementation-specific fields that are needed to identify or otherwise contextualize the associated Span (e.g., a (trace_id, span_id, sampled) tuple).

Functions

new (opts)

Create new span context

Parameters:

- opts: options
  - trace_id: (optional string)
  - span_id: (optional string)
  - parent_id: (optional string)

  - should_sample: (optional boolean)
  - baggage: (optional table)

Returns:

(table) span context

child ()

Create span child span context

Returns:

(table) child span context

clone_with_baggage_item (self, key, value)

New from existing but with an extra baggage item Clone context and add item to its baggage

Parameters:

- self: (table)
- key: (string)
- value: (string)

Returns:

(table) context

get_baggage_item (self, key)

Get item from baggage

Parameters:

- self: (table)
- key: (string)

Returns:

(string) value

each_baggage_item (self)

Get baggage item iterator

Parameters:

- self: (table)

Returns:

(function) iterator

(table) baggage

6.3.4 Module *opentracing.tracer*

Tracer is the entry point API between instrumentation code and the tracing implementation.

This implementation both defines the public Tracer API, and provides a default no-op behavior.

Functions

new (reporter, sampler)

Init new tracer

Parameters:

- reporter:
  - report: (function)
- sampler:
  - sample: (function)

Returns:

(table) tracer

start_span (self, name, opts)

Starts and returns a new Span representing a unit of work.

Example usage:

Create a root Span (a Span with no causal references):

```
tracer:start_span("op-name")
```

Create a child Span :

```
tracer:start_span(
      "op-name",
      {["references"] = {{"child_of", parent_span:context()}}})
```

Parameters:

- self: (table)
- name: (string) operation_name name of the operation represented by the new.. code-block:: lua Span from the perspective of the current service.
- opts: table specifying modifications to make to thenewly created span. The following parameters are supported: trace_id , references ,a list of referenced spans; start_time , the time to mark when the spanbegins (in microseconds since epoch); tags , a table of tags to add tothe created span.
  - trace_id: (optional string)
  - child_of: (optional table)
  - references: (optional table)
  - tags: (optional table)
  - start_timestamp: (optional number)

Returns:

(table) span a Span instance

register_injector (self, format, injector)

Register injector for tracer

Parameters:

- self: (table)
- format: (string)
- injector: (function)

Returns:

(boolean) true

register_extractor (self, format, extractor)

Register extractor for tracer

Parameters:

- self: (table)
- format: (string)
- extractor: (function)

Returns:

(boolean) true

inject (self, context, format, carrier)

Inject context into carrier with specified format. See https://opentracing.io/docs/overview/inject-extract/

Parameters:

- self: (table)
- context: (table)
- format: (string)
- carrier: (table)

Returns:

(table) carrier

Or

(nil)

(string) error

extract (self, format, carrier)

Extract context from carrier with specified format. See https://opentracing.io/docs/overview/inject-extract/

Parameters:

- self: (table)
- format: (string)
- carrier: (table)

Returns:

(table) context

Or

(nil)

(string) error

http_headers_inject (self, context, carrier)

Injects span_context into carrier using a format appropriate for HTTP headers.

Parameters:

- self: (table)
- context: (table) the SpanContext instance to inject
- carrier: (table)

Returns:

(table) context a table to contain the span context

Usage:

```
carrier = {}
tracer:http_headers_inject(span:context(), carrier)
```

text_map_inject (self, context, carrier)

Injects span_context into carrier .

Parameters:

- self: (table)
- context: (table) the SpanContext instance to inject
- carrier: (table)

Returns:

(table) context a table to contain the span context

Usage:

```
carrier = {}
tracer:text_map_inject(span:context(), carrier)
```

http_headers_extract (self, carrier)

Returns a SpanContext instance extracted from the carrier or nil if no such SpanContext could be found. http_headers_extract expects a format appropriate for HTTP headers and uses case-sensitive comparisons for the keys.

Parameters:

- self: (table)
- carrier: (table) the format-specific carrier object to extract from

Returns:

(table) context

Or

(nil)

(string) error

text_map_extract (self, carrier)

Returns a SpanContext instance extracted from the carrier or nil if no such SpanContext could be found.

Parameters:

- self: (table)
- carrier: (table) the format-specific carrier object to extract from

Returns:

(table) context

Or

(nil)

(string) error

6.3.5 Module *zipkin.tracer*

Client for Zipkin

Functions

new (config, sampler)

Init new Zipkin Tracer

Parameters:

- config: Table with Zipkin configuration
  - base_url: (table) Zipkin API base url
  - api_method: (table) API method to send spans to zipkin
  - report_interval: (table) Interval of reports to zipkin
  - spans_limit: (optional number) Limit of a spans buffer (1k by default)
  - on_error: (optional function) On error callback that apply error in string format
  - spans_limit: (optional number) Limit of a spans buffer (1k by default)
- sampler: (table) Table that contains function samplethat is apply span name and mark this span for further report

Returns:

(table) context

Or

(nil) nil

(string) error

6.3.6 Changelog

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog and this project adheres to Semantic Versioning.

[Unreleased]

[0.1.0] - 2019-06-05

- OpenTracing API implementation
- Zipkin client
- Lua API documentation, which you can read with tarantoolctl rocks doc tracing command

6.3.7 Tracing for Tarantool

Tracing module for Tarantool includes the following parts:

- OpenTracing API
- Zipkin tracer

Table of contents

OpenTracing

This library is a Tarantool platform API for OpenTracing.

Required Reading

To fully understand this platform API, it's helpful to be familiar with the OpenTracing project and terminology more specifically.

Conventions

- All timestamps are in microseconds

Span

> The "span" is the primary building block of a distributed trace, representing an individual unit of work done in a distributed system. Traces in OpenTracing are defined implicitly by their Spans. In particular, a Trace can be thought of as a directed acyclic graph (DAG) of Spans, where the edges between Spans are called References.

```
local opentracing_span = require('opentracing.span')
-- tracer - External tracer
-- context - Span context
-- name - Name of span
-- start_timestamp (optional) - Time of span's start in microseconds (by default current time)
local span = opentracing_span.new(tracer, context, name, start_timestamp)
```

SpanContext

> The SpanContext carries data across process boundaries.

```
local opentracing_span_context = require('opentracing.span_context')
-- trace_id (optional) - Trace ID (by default generates automatically)
-- span_id (optional) - Span ID (by default generates automatically)
-- parent_id (optional) - Span ID of parent span (by default is empty)
-- should_sample (optional) - Flag is enable collecting data of this span (by default false)
-- baggage (optional) - Table with trace baggage (by default is empty table)
```

(continues on next page)

(continued from previous page)

```
local context = opentracing_span_context.new({
                    tracer_id = trace_id,
                    span_id = span_id,
                    parent_id = parent_id,
                    should_sample = should_sample,
                    baggage = baggage,
                })
```

Tracer

> The Tracer interface creates Spans and understands how to Inject (serialize) and Extract (deserialize) their metadata across process boundaries.

An interface for custom tracers

```
local opentracing_tracer = require('opentracing.tracer')
-- reporter (optional) - Table with `report` method to process finished spans (by default no-op table)
-- sampler (optional) - Table with `sample` method to select traces to send to distributing tracing system (by␣
↪default random selection)
-- But you can implement your own sampler with appropriate sampling strategy
-- For more information see: https://www.jaegertracing.io/docs/1.11/sampling/
local tracer = opentracing_tracer.new(reporter, sampler)
```

Basic usage

```
local zipkin = require('zipkin.tracer')
local opentracing = require('opentracing')

-- Create client to Zipkin and set it global for easy access from any part of app
local tracer = zipkin.new(config)
opentracing.set_global_tracer(tracer)

-- Create and manage spans manually
local span = opentracing.start_span('root span')
-- ... your code ...
span:finish()

-- Simple wrappers via user's function

-- Creates span before function call and finishes it after
local result = opentracing.trace('one span', func, ...)

-- Wrappers with context passing
local span = opentracing.start_span('root span')

-- Pass your function as third argument and then its arguments
opentracing.trace_with_context('child span 1', span:context(), func1, ...)
opentracing.trace_with_context('child span 2', span:context(), func2, ...)
span:finish()
```

Zipkin

Zipkin is a distributed tracing system.

It helps gather timing data needed to troubleshoot latency problems in microservice architectures. It manages both the collection and lookup of this data.

This module allows you to instance Zipkin Tracer that can start spans and will report collected spans to Zipkin Server.

Basic usage

```
local zipkin = require('zipkin.tracer')
-- First argument is config that contains url of Zipkin API,
--  method to send collected traces and interval of reports in seconds
-- Second optional argument is Sampler (see OpenTracing API description), by default random sampler
local tracer = zipkin.new({
    base_url = 'localhost:9411/api/v2/spans',
    api_method = 'POST',
    report_interval = 0,
}, Sampler)

local span = tracer:start_span('example')
-- ...
span:finish()
```

Examples

HTTP

This example is a Lua port of Go OpenTracing tutorial.

Description

The example demonstrates trace propagation through two services: formatter that formats the source string to “Hello, world” and publisher that prints it in the console.

Add data to these services via HTTP; initially it sends client.

Note: example requires http rock (version >= 2.0.1) Install it using ‘tarantoolctl rocks install http 2.0.1‘

How to run

- Create docker-compose.zipkin.yml

```
---
version: '3.5'


==========================================================================================
Initially got from https://github.com/openzipkin/docker-zipkin/blob/master/docker-compose.yml
==========================================================================================
```

(continues on next page)

(continued from previous page)

```
services:
  storage:
    image: openzipkin/zipkin-mysql
    container_name: mysql
    networks:
      - zipkin
    ports:
      - 3306:3306

  # The zipkin process services the UI, and also exposes a POST endpoint that
  # instrumentation can send trace data to. Scribe is disabled by default.
  zipkin:
    image: openzipkin/zipkin
    container_name: zipkin
    networks:
      - zipkin
    # Environment settings are defined here https://github.com/openzipkin/zipkin/tree/1.19.0/zipkin-server
↪#environment-variables
    environment:
      - STORAGE_TYPE=mysql
      # Point the zipkin at the storage backend
      - MYSQL_HOST=mysql
      # Enable debug logging
      - JAVA_OPTS=-Dlogging.level.zipkin=DEBUG -Dlogging.level.zipkin2=DEBUG
    ports:
      # Port used for the Zipkin UI and HTTP Api
      - 9411:9411
    depends_on:
      - storage

  # Adds a cron to process spans since midnight every hour, and all spans each day
  # This data is served by http://192.168.99.100:8080/dependency
  #
  # For more details, see https://github.com/openzipkin/docker-zipkin-dependencies
  dependencies:
    image: openzipkin/zipkin-dependencies
    container_name: dependencies
    entrypoint: crond -f
    networks:
      - zipkin
    environment:
      - STORAGE_TYPE=mysql
      - MYSQL_HOST=mysql
      # Add the baked-in username and password for the zipkin-mysql image
      - MYSQL_USER=zipkin
      - MYSQL_PASS=zipkin
      # Dependency processing logs
      - ZIPKIN_LOG_LEVEL=DEBUG
    depends_on:
      - storage

networks:
  zipkin:
```

- Start Zipkin docker-compose -f docker-compose.zipkin.yml up

- Run mock applications from separate consoles: consumer, formatter and client

Formatter HTTP server

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

local http_server = require('http.server')
local http_router = require('http.router')
local fiber = require('fiber')
local log = require('log')
local zipkin = require('zipkin.tracer')
local opentracing = require('opentracing')

local app = {}

local Sampler = {
    sample = function() return true end,
}

local HOST = '0.0.0.0'
local PORT = '33302'

local function handler(req)

    -- Extract content from request's http headers
    local ctx, err = opentracing.http_extract(req:headers())
    if ctx == nil then
        local resp = req:render({ text = err })
        resp.status = 400
        return resp
    end

    local hello_to = req:query_param('helloto')
    -- Start new child span
    local span = opentracing.start_span_from_context(ctx, 'format_string')
    -- Set service type
    span:set_component('formatter')
    span:set_server_kind()
    span:set_http_method(req:method())
    span:set_http_path(req:path())
    local greeting = span:get_baggage_item('greeting')
    local result = ('%s, %s!'):format(greeting, hello_to)
    local resp = req:render({ text = result })

    -- Simulate long request processing
    fiber.sleep(2)
    span:log_kv({
        event = 'String format',
        value = result,
    })
    resp.status = 200
    span:set_http_status_code(resp.status)
    span:finish()
    return resp
end

function app.init()
    -- Initialize zipkin client that will be send spans every 5 seconds
```

(continues on next page)

(continued from previous page)

```
    local tracer = zipkin.new({
        base_url = 'localhost:9411/api/v2/spans',
        api_method = 'POST',
        report_interval = 5,
        on_error = function(err) log.error(err) end,
    }, Sampler)
    opentracing.set_global_tracer(tracer)

    local httpd = http_server.new(HOST, PORT)
    local router = http_router.new()
        :route({ path = '/format', method = 'GET' }, handler)
    httpd:set_router(router)
    httpd:start()
end

app.init()

return app
```

Publisher HTTP server

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

local http_server = require('http.server')
local http_router = require('http.router')
local fiber = require('fiber')
local log = require('log')
local zipkin = require('zipkin.tracer')
local opentracing = require('opentracing')

local app = {}

local Sampler = {
    sample = function() return true end,
}

local HOST = '0.0.0.0'
local PORT = '33303'

local function handler(req)
    local ctx, err = opentracing.http_extract(req:headers())

    if ctx == nil then
        local resp = req:render({ text = err })
        resp.status = 400
        return resp
    end

    local hello = req:query_param('hello')
    local span = opentracing.start_span_from_context(ctx, 'print_string')
    span:set_component('publisher')
    span:set_server_kind()
    span:set_http_method(req:method())
    span:set_http_path(req:path())

    -- Simulate long request processing
```

(continues on next page)

(continued from previous page)

```
    fiber.sleep(3)

    io.write(hello, '\n')
    local resp = req:render({text = '' })
    resp.status = 200
    span:set_http_status_code(resp.status)
    span:finish()
    return resp
end

function app.init()
    local tracer = zipkin.new({
        base_url = 'localhost:9411/api/v2/spans',
        api_method = 'POST',
        report_interval = 5,
        on_error = function(err) log.error(err) end,
    }, Sampler)
    opentracing.set_global_tracer(tracer)

    local httpd = http_server.new(HOST, PORT)
    local router = http_router.new()
        :route({ path = '/print', method = 'GET' }, handler)
    httpd:set_router(router)
end

app.init()

return app
```

Client

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

local http_client = require('http.client')
local json = require('json')
local log = require('log')
local fiber = require('fiber')
local zipkin = require('zipkin.tracer')
local opentracing = require('opentracing')

local app = {}

-- Process all requests
local Sampler = {
    sample = function() return true end,
}

local function url_encode(str)
    local res = string.gsub(str, '[^a-zA-Z0-9_]',
        function(c)
            return string.format('%%%02X', string.byte(c))
        end
    )
    return res
end
```

(continues on next page)

(continued from previous page)

```
-- Client part to formatter
local formatter_url = 'http://localhost:33302/format'
local function format_string(ctx, str)
    local span = opentracing.start_span_from_context(ctx, 'format_string')
    local httpc = http_client.new()
    span:set_component('client')
    span:set_client_kind()
    span:set_http_method('GET')
    span:set_http_url(formatter_url)

    -- Use http headers as carrier
    local headers = {
        ['content-type'] = 'application/json'
    }
    opentracing.http_inject(span:context(), headers)

    -- Simulate problems with network
    fiber.sleep(1)
    local resp = httpc:get(formatter_url .. '?helloto=' .. url_encode(str),
            { headers = headers })
    fiber.sleep(1)

    span:set_http_status_code(resp.status)
    if resp.status ~= 200 then
        error('Format string error: ' .. json.encode(resp))
    end
    local result = resp.body
    -- Log result
    span:log_kv({
        event = 'String format',
        value = result
    })
    span:finish()
    return result
end

-- Client part to publisher
local printer_url = 'http://localhost:33303/print'
local function print_string(ctx, str)
    local span = opentracing.start_span_from_context(ctx, 'print_string')
    local httpc = http_client.new()
    span:set_component('client')
    span:set_client_kind()
    span:set_http_method('GET')
    span:set_http_url(printer_url)

    local headers = {
        ['content-type'] = 'application/json'
    }
    opentracing.http_inject(span:context(), headers)

    -- Simulate problems with network
    fiber.sleep(1)
    local resp = httpc:get(printer_url .. '?hello=' .. url_encode(str),
            { headers = headers })
    fiber.sleep(1)
```

(continues on next page)

(continued from previous page)

```
    span:set_http_status_code(resp.status)
    if resp.status ~= 200 then
        error('Print string error: ' .. json.encode(resp))
    end
    span:finish()
end

function app.init()
    -- Initialize Zipkin tracer
    local tracer = zipkin.new({
        base_url = 'localhost:9411/api/v2/spans',
        api_method = 'POST',
        report_interval = 0,
        on_error = function(err) log.error(err) end,
    }, Sampler)
    opentracing.set_global_tracer(tracer)

    -- Initialize root span
    local span = opentracing.start_span('Hello-world')

    local hello_to = 'world'
    local greeting = 'my greeting'
    span:set_component('client')
    -- Set service type
    span:set_client_kind()
    -- Set tag with metadata
    span:set_tag('hello-to', hello_to)
    -- Add data to baggage
    span:set_baggage_item('greeting', greeting)

    local ctx = span:context()
    local formatted_string = format_string(ctx, hello_to)
    print_string(ctx, formatted_string)
    span:finish()
end

app.init()

os.exit(0)
```

- Check results on http://localhost:9411/zipkin

Tarantool Cartridge

Opentracing could be used with Tarantool Cartridge.

This example is pretty similar to previous. We will have several roles that communicate via rpc_call.

Basics

Before describing let's define some restrictions of "tracing in Tarantool". Remote communications between tarantools are made using net.box module. It allows to send only primitive types (except functions) and doesn't have containers for request context (as headers in HTTP). Then you should transfer span context explicitly as raw table as additional argument in your function.

```
-- Create span
local span = opentracing.start_span('span')

-- Create context carrier
local rpc_context = {}
opentracing.map_inject(span:context(), rpc_context)

-- Pass context explicitly as additional argument
local res, err = cartridge.rpc_call('role', 'fun', {rpc_context, ...})
```

Using inside roles

The logic of tracing fits into a separate role. Let's define it:

```
local opentracing = require('opentracing')
local zipkin = require('zipkin.tracer')

local log = require('log')

-- config = {
--     base_url = 'localhost:9411/api/v2/spans',
--     api_method = 'POST',
--     report_interval = 5,     -- in seconds
--     spans_limit = 1e4,       -- amount of spans that could be stored locally
-- }

local function apply_config(config)
    -- sample all requests
    local sampler = { sample = function() return true end }

    local tracer = zipkin.new({
        base_url = config.base_url,
        api_method = config.api_method,
        report_interval = config.report_interval,
        spans_limit = config.spans_limit,
        on_error = function(err) log.error('zipkin error: %s', err) end,
    }, sampler)

    -- Setup global tracer for easy access from another modules
    opentracing.set_global_tracer(tracer)

    return true
end

return {
    role_name = 'tracing',
    apply_config = apply_config,
    dependencies = {},
}
```

Then you can use this role as dependency:

```
local opentracing = require('opentracing')
local membership = require('membership')
```

(continues on next page)

(continued from previous page)

```
local role_name = 'formatter'
local template = 'Hello, %s'

local service_uri = ('%s@%s'):format(role_name, membership.myself().uri)

local function format(ctx, input)
    -- Extract tracing context from request context
    local context = opentracing.map_extract(ctx)
    local span = opentracing.start_span_from_context(context, 'format')
    span:set_component(service_uri)

    local result, err
    if input == '' then
        err = 'Empty string'
        span:set_error(err)
    else
        result = template:format(input)
    end

    span:finish()

    return result, err
end

local function init(_)
    return true
end

local function stop()
end

local function validate_config(_, _)
    return true
end

local function apply_config(_, _)
    return true
end

return {
    format = format,

    role_name = role_name,
    init = init,
    stop = stop,
    validate_config = validate_config,
    apply_config = apply_config,
    -- Setup tracing role as dependency
    dependencies = {'app.roles.tracing'},
}
```

6.4 odbc

6.4.1 ODBC connector for Tarantool

Based on unixODBC

Examples

Use a single connection

```
local odbc = require 'odbc'
local yaml = require 'yaml'

local env, err = odbc.create_env()
local conn, err = env:connect("DSN=odbc_test")

local result, err = conn:execute("SELECT 1 as a, 2 as b")
print(yaml.encode(result))

conn:close()
```

Use ODBC transactions

```
local odbc = require 'odbc'
local yaml = require 'yaml'

local env, err = odbc.create_env()
local conn, err = env:connect("DSN=odbc_test")
conn:execute("CREATE TABLE t(id INT, value TEXT)")

conn:set_autocommit(false)
conn:execute("INSERT INTO t VALUES (1, 'one')")
conn:execute("INSERT INTO t VALUES (2, 'two')")
local result, err = conn:execute("SELECT * FROM t")
print(yaml.encode(result))

conn:commit()
conn:close()
```

Use connection pool

```
local odbc = require 'odbc'
local yaml = require 'yaml'

local pool, err = odbc.create_pool({
    size = 5
})
local _, err = pool:connect()

local conn = pool:get()
```

(continues on next page)

(continued from previous page)

```
local res, err = conn:execute("SELECT 1 as a, 2 as b")
print(yaml.encode(res))

pool:put(conn)

pool:close()
```

Use connection pool for ad-hoc requests

Pool implements :execute(), :drivers(), :datasources() and :tables() methods that acquire and release a connection object for you.

```
local odbc = require 'odbc'
local yaml = require 'yaml'

local pool, err = odbc.create_pool({
    size = 5
})
local _, err = pool:connect()

local res, err = pool:execute("SELECT 1 as a, 2 as b")
print(yaml.encode(res))

pool:close()
```

API Reference

Creates ODBC environment.

Options

date_as_table - configures behaviour of odbc package of how to deal with dates. If date_as_table is false then dates are represented in the default way for the driver (e.g. represents as strings for PostgreSQL). If date_as_table is true then dates are represented as tables compatible with os.date('*t').

Creates a connection pool.

Options

1. All options for odbc.create_env
2. dsn - connection string
3. size - number of connections in the pool

Environment methods

env:connect(dsn)

Parameters

1. dsn - Connection string (documentation).

Connection methods

conn:execute(query, params)

Executes an arbitrary SQL query

Parameters

1. query - SQL query
2. param - table with parameters binding

Example

```
conn:execute("SELECT * FROM t WHERE id > ? and value = ?", {1, "two"})
```

conn:set_autocommit(flag)

Sets autocommit of connection to a specified value. Used to achieve transaction behaviour. Set autocommit to false to execute multiple statements in one transactions.

Parameters

1. flag - true/false.

conn:commit()

Commit a transaction

conn:rollback()

Rollback a transaction

conn:set_isolation(level)

Sets isolation level of a transaction. Cannot be run in an active transaction.

Parameters

1. level - isolation level. One of the values defined in the odbc.isolation table.

Isolation levels

1. odbc.isolation.READ_UNCOMMITTED
2. odbc.isolation.READ_COMMITTED
3. odbc.isolation.REPEATABLE_READ
4. odbc.isolation.SERIALIZABLE

conn:is_connected()

Returns true if connection is active.

conn:state()

Returns an internal state of a connection.

conn:close()

Disconnect and close the connection.

conn:drivers()

Returns a list of drivers available in the system (contents of odbcinst.ini file).

Example

conn:datasources()

Returns a list of data sources available in the system (contents of odbc.ini file).

Example

conn:tables()

Returns a list of tables of a connected data source.

Example

conn:cursor(query, params)

Creates a cursor object for the specified query.

Parameters

1. query - SQL query
2. param - table with parameters binding

conn:prepare(query)

Create object and prepare query.

Parameters

1. query - SQL query

Cursor methods

cursor:fetchrow()

Fetch one row from the data frame and return as a single table value.

Example

cursor:fetch(n)

Fetch multiple rows.

Parameters 1. n - number of rows to fetch

Example

cursor:fetchall()

Fetch all available rows in the data frame.

cursor:is_open()

Returns true if cursor is open.

cursor:close()

Close cursor discarding available data.

Prepare methods

prepare:execute()

Execute prepared SQL query

Parameters 1. param - table with parameters binding

Example

prepare:is_open()

Returns true if prepare is open.

prepare:close()

Close prepare discarding prepared query.

Pool methods

pool:connect()

Connect to all size connections.

pool:acquire()

Acquire a connection. The connection must be either returned to pool with pool:release() method or closed.

pool:release(conn)

Release the connection.

pool:available()

Returns the number of available connections.

pool:close()

Close pool and all underlying connections.

pool:execute(query, params)

Acquires a connection, executes query on it and releases the connection.

1. query - SQL query
2. param - table with parameters binding

pool:tables()

Acquires a connection, executes tables() on it and releases.

pool:drivers()

Acquires a connection, executes drivers() on it and releases.

pool:datasources()

Acquires a connection, executes datasources() on it and releases.

Installation

Prerequisites:

1. unixODBC driver
2. Driver for the database of your choice. Currently this module is tested only with PostgreSQL, MySQL and MS SQL Server databases.
3. Datasource for the database in odbc.ini file

PostgreSQL

Linux (Ubuntu)

```
$ sudo apt-get install odbc-postgresql
```

Add to file /etc/odbcinst.ini:

```
[PostgreSQL ANSI]
Description=PostgreSQL ODBC driver (ANSI version)
Driver=psqlodbca.so
Setup=libodbcpsqlS.so
Debug=0
CommLog=1
UsageCount=1
```

Add to file /etc/odbc.ini:

```
[<dsn_name>]
Description=PostgreSQL
Driver=PostgreSQL ANSI
Trace=No
TraceFile=/tmp/psqlodbc.log
Database=<Database>
Servername=localhost
username=<username>
password=<password>
port=
readonly=no
rowversioning=no
showsystemtables=no
showoidcolumn=no
fakeoidindex=no
connsettings=
```

MacOS

Use brew to install:

```
$ brew install psqlodbc
```

/usr/local/etc/odbcinst.ini contents:

```
[PostgreSQL ANSI]
Description=PostgreSQL ODBC driver (ANSI version)
Driver=/usr/local/lib/psqlodbca.so
Debug=0
CommLog=1
UsageCount=1
```

/usr/local/etc/odbc.ini contents:

```
[<dsn_nam>]
Description=PostgreSQL
```

(continues on next page)

(continued from previous page)

```
Driver=PostgreSQL ANSI
Trace=No
TraceFile=/tmp/psqlodbc.log
Database=<database>
Servername=<host>
UserName=<username>
Password=<password>
ReadOnly=No
RowVersioning=No
ShowSystemTables=No
ShowOidColumn=No
FakeOidIndex=No
```

MSSQL

Linux

Please follow to the official installation guide.

/etc/odbcinst.ini contents:

```
[ODBC Driver 17 for SQL Server]
Description=Microsoft ODBC Driver 17 for SQL Server
Driver=/opt/microsoft/msodbcsql17/lib64/libmsodbcsql-17.2.so.0.1
UsageCount=1
```

/etc/odbc.ini contents:

```
[<dsn_name>]
Driver=ODBC Driver 17 for SQL Server
Database=<Database>
Server=localhost
```

MacOS

For El Capitan, Sierra and High Sierra use brew to install:

```
$ brew tap microsoft/mssql-release https://github.com/Microsoft/homebrew-mssql-release
$ brew install --no-sandbox msodbcsql17 mssql-tools
```

For El Capitan and Sierra use brew to install:

```
$ brew tap microsoft/mssql-release https://github.com/Microsoft/homebrew-mssql-release
$ brew install --no-sandbox msodbcsql@13.1.9.2 mssql-tools@14.0.6.0
```

Examples below are fair to msodbcsql 13

/usr/local/etc/odbcinst.ini contents:

```
[ODBC Driver 13 for SQL Server]
Description=Microsoft ODBC Driver 13 for SQL Server
Driver=/usr/local/lib/libmsodbcsql.13.dylib
UsageCount=1
```

/usr/local/etc/odbc.ini contents:

```
[<dsn_name>]
Description=SQL Server
Driver=ODBC Driver 13 for SQL Server
Server=<host>,<port>
```

FYI:

Uid, Pwd etc are placed into connstring

Example

MySQL

Linux

Please follow to the official installation guide.

MacOS

Download and install:

1. http://www.iodbc.org/dataspace/doc/iodbc/wiki/iodbcWiki/Downloads
2. https://dev.mysql.com/downloads/connector/odbc/

Add to file /usr/local/etc/odbcinst.ini:

```
[MySQL ODBC 8.0 Driver]
Driver=/usr/local/mysql-connector-odbc-8.0.12-macos10.13-x86-64bit/lib/libmyodbc8a.so
UsageCount=1
```

Add to file /usr/local/etc/odbc.ini:

```
[<dsn>]
Driver = MySQL ODBC 8.0 Driver
Server = <host>
PORT = <port>
```

FYI:

USER, DATABASE etc are placed into connstring

Example

Sybase ASE

MacOS

Run brew install freetds

Add to file /usr/local/etc/freetds.conf

```
[sybase]
host = localhost
port = 8000
tds version = auto
```

Add to file /usr/local/etc/odbcinst.ini:

```
[Sybase Driver]
Driver=/usr/local/lib/libtdsodbc.so
UsageCount=1
```

Add to file /usr/local/etc/odbc.ini:

```
[default]
Driver=/usr/local/lib/libtdsodbc.so
Port=8000

[sybase]
Driver=Sybase Driver
Description=Sybase ASE
DataSource=<datasource>
ServerName=sybase
Database=<database>
```

Example

References

1. Tarantool - in-memory database and application server.
2. PostgreSQL ODBC
3. MS SQL Server ODBC
4. MySQL ODBC

6.5 oracle

6.5.1 Oracle connector

The oracle package exposes some functionality of OCI. With this package, Tarantool Lua applications can send and receive data over Oracle protocol.

The advantage of integrating oracle with Tarantool, which is an application server plus a DBMS, is that anyone can handle all of the tasks associated with Oracle (control, manipulation, storage, access) with the same high-level language (Lua) and with minimal delay.

Table of contents

- API reference

Prerequisites

- An operating system with developer tools including cmake, C compiler with gnu99 support, git and Lua.
- Tarantool 1.6.5+ with header files (tarantool and tarantool-dev packages).
- Oracle OCI 10.0+ header files and dynamic libs.

Automatic build

Important: Builder requires Oracle Instant Client zip archives. You need to download them from Oracle into the source tree:

```
curl -O https://raw.githubusercontent.com/bumpx/oracle-instantclient/master/instantclient-basic-linux.x64-12.2.0.
↪1.0.zip
curl -O https://raw.githubusercontent.com/bumpx/oracle-instantclient/master/instantclient-sdk-linux.x64-12.2.0.1.
↪0.zip
sha256sum -c instantclient.sha256sum
```

To build a complete oracle package, you need to run package.sh script first (depends on docker). Packages will be available in build/ directory. Example:

```
wget <oracle-client.rpm>
wget <oracle-devel.rpm>
$./package.sh
...
done
$ls -1 build
oracle-instantclient12.2-basic-12.2.0.1.0-1.x86_64.rpm
oracle-instantclient12.2-devel-12.2.0.1.0-1.x86_64.rpm
tarantool-oracle-1.0.0.0-1.el7.centos.src.rpm
tarantool-oracle-1.0.0.0-1.el7.centos.x86_64.rpm
tarantool-oracle-debuginfo-1.0.0.0-1.el7.centos.x86_64.rpm
```

After that you can install oracle package on the target machine:

```
rpm -Uvh tarantool-oracle-1.0.0.0-1.el7.centos.x86_64.rpm
```

Getting started

Start Tarantool in the interactive mode. Execute these requests:

```
tarantool> oracle = require('oracle')
tarantool> env, errmsg = oracle.new()
tarantool> if not env then error("Failed to create environment: "..errmsg) end
tarantool> c, errmsg = env:connect({username='system', password='oracle', db='localhost:1511/myspace'})
tarantool> if not c then error("Failed to connect: "..errmsg) end
tarantool> c:exec('CREATE TABLE test(i int, s varchar(20))')
tarantool> c:exec('INSERT INTO test(i, s) VALUES(:I, :S)', {I=1, S='Hello!'})
tarantool> rc, result_set = c:exec('SELECT * FROM test')
```

If all goes well, you should see:

```
tarantool> result_set[1][2] -- 'Hello!'
```

This means that you have successfully installed tarantool/oracle and successfully executed an instruction that brought data from an Oracle database.

API reference

function new([opts])

Create Oracle connection environment.

Accepts parameters:

- [optional] table of options:
  - charset - client-side character and national character set. If not set or set improperly, NLS_LANG setting is used.

Returns: * env - environment object in case of success, nil otherwise, * err [OPTIONAL] - error string in case of error.

function env:connect(credentials [, additional options])

Connect to the Oracle database.

Accepts parameters: * credentials (table):

- username (str) - user login,
- password (str) - user password,
- db (str) - database URL.
- additional options (as table):
  - prefetch_count (int) - prefetch row count amount from Oracle,
  - prefetch_size (int) - memory limit for prefetching (in MB),
  - batch_size (int) - the size of each SELECT loop batch on exec() and cursor:fetchall().

Returns: * conn - connection object in case of success, nil otherwise, * err [OPTIONAL] - error string or table structure in case of error.

function env:version()

Get version string.

function conn:exec(sql [, args])

Execute an operation.

Accepts parameters:

- sql - SQL statement,
- [optional] statement arguments.

Returns:

- rc - result code (0 - Success, 1 - Error)
- result_set - result table, err - table with error (see below) in case of error
- row_count - number of rows in result_set
- err [OPTIONAL] - table with error in case of warning from Oracle

Examples:

```
-- Schema - create table(a int, b varchar(25), c number)
conn:exec("insert into table(a, b, c) values(:A, :B, :C)", {A=1, B='string', C=0.1})
```

```
-- Schema - create table(a int, b varchar(25), c number)
rc, res = conn:exec("SELECT a, b, c FROM table")
res[1][1] -- a
res[1][2] -- b
res[1][3] -- c
```

function conn:cursor(sql [, args, opts])

Create cursor to fetch SELECT results.

Accepts parameters:

- sql - SELECT SQL statement,
- [optional] statement arguments,
- [optional] table of options:
  - scrollable - enable cursor scrollable mode (false by default).

Returns:

- cursor - cursor object in case of success, nil otherwise
- err [OPTIONAL] - table with error, same format as exec function one

function conn:close()

Close connection and all associated cursors.

function cursor:fetch_row()

Fetch one row of resulting set.

Returns:

- rc - result code (0 - Success, 1 - Error),
- result_set - result table, err - table with error in case of error,
- row_count - number of rows in result_set.

function cursor:fetch(fetch_size)

Fetch fetch_size rows of resulting set.

Accepts parameters:

- fetch_size - number of rows (positive integer).

Returns:

- rc - result code (0 - Success, 1 - Error),
- result_set - result table, err - table with error in case of error,
- row_count - number of rows in result_set.

function cursor:fetch_all()

Fetch all remaining rows of resulting set.

Returns:

- rc - result code (0 - Success, 1 - Error),
- result_set - result table, err - table with error in case of error,
- row_count - number of rows in result_set.

function cursor:fetch_first(fetch_size)

Scrollable only.

Fetch first fetch_size rows of resulting set.

Accepts parameters:

- fetch_size - number of rows (positive integer).

Returns:

- rc - result code (0 - Success, 1 - Error),
- result_set - result table, err - table with error in case of error,
- row_count - number of rows in result_set.

function cursor:fetch_last()

Scrollable only.

Fetch last row of resulting set.

Returns:

- rc - result code (0 - Success, 1 - Error),
- result_set - result table, err - table with error in case of error,
- row_count - number of rows in result_set.

function cursor:fetch_absolute(fetch_size, offset)

Scrollable only.

Fetch fetch_size rows of resulting set, starting from offset absolute position (including offset row).

Accepts parameters:

- fetch_size - number of rows (positive integer),
- offset - absolute cursor offset (positive integer).

Returns:

- rc - result code (0 - Success, 1 - Error),
- result_set - result table, err - table with error in case of error,
- row_count - number of rows in result_set.

function cursor:fetch_relative(fetch_size, offset)

Scrollable only.

Fetch fetch_size rows of resulting set, starting from current + offset absolute position(including current + offset row).

Accepts parameters:

- fetch_size - number of rows (positive integer),
- offset - relative cursor offset (signed integer).

Returns:

- rc - result code (0 - Success, 1 - Error),
- result_set - result table, err - table with error in case of error,
- row_count - number of rows in result_set.

function cursor:fetch_current(fetch_size)

Scrollable only.

Fetch fetch_size rows of resulting set, starting from current position (including current row).

Accepts parameters:

- fetch_size - number of rows (positive integer).

Returns:

- rc - result code (0 - Success, 1 - Error),
- result_set - result table, err - table with error in case of error,
- row_count - number of rows in result_set.

function cursor:fetch_prior(fetch_size)

Scrollable only.

Fetch fetch_size rows of resulting set, starting from previous row from the current position (including previous row).

Accepts parameters:

- fetch_size - number of rows (positive integer).

Returns:

- rc - result code (0 - Success, 1 - Error),
- result_set - result table, err - table with error in case of error,
- row_count - number of rows in result_set.

function cursor:get_position()

Scrollable only.

Get current cursor position.

Returns:

- rc - result code (0 - Success, 1 - Error)
- position - current cursor position, err - table with error in case of error

function cursor:close()

Closes cursor. After this was executed, cursor is no longer available for fetching results.

function cursor:is_closed()

Returns:

- is_closed - true if closed, false otherwise.

function cursor:ipairs()

Lua 5.1 version of ipairs(cursor) operator.

Example:

```
-- Foo(row) is some function for row processing
for k, row in cursor:ipairs() do
    Foo(row)
end
```

function conn:close()

Close connection and all associated cursors.

Error handling

In case of error returns nil, err where err is table with the next fields:

- type - type of error (1 - Oracle error is occured, 0 - Connector error is occured)
- msg - message with text of error
- code - error code (now defined ONLY for Oracle error codes)

6.5.2 Deploy Oracle in Docker

Description (For Oracle EE v12.2):

1. You should have working Docker and Oracle accounts.
2. instantclient-sqlplus.
3. Go to this page and follow instructions. You need to follow steps 1.a-1.d steps. For short here they are:

- 1.a Log in to Docker Store with your Docker credentials.
- 1.b Search for 'oracle database' and select the Oracle Database Enterprise Edition image.
- 1.c Click through and accept terms if needed.
- 1.d View Setup Instructions.
- 1.d.1 Download image with

```
docker pull store/oracle/database-enterprise:12.2.0.1
```

- 1.d.2 Run container with -P option, it will allocate port to access database outside docker container.

```
docker run -d -it --name OraDBEE -P store/oracle/database-enterprise:12.2.0.1
```

Note: type docker ps to get allocated port, you need PORTS section (or docker port CONTAINER_NAME), also check that status is healthy (if not, repeat 1.d.2 - in my case probability to create working container was 50/50 :P). Assume port is 32771 for farther instructions.

4. Go inside container, create a user, grant all necessary permissions:

```
docker exec -it OraDBEE bash -c "source /home/oracle/.bashrc; sqlplus sys/Oradoc_db1@ORCLCDB as sysdba"
...

SQL> alter session set "_ORACLE_SCRIPT"=true;
SQL> CREATE USER user1 IDENTIFIED BY qwerty123;
SQL> GRANT CONNECT, RESOURCE, DBA TO user1;
```

5. Check connection and access rights outside container for a new user:

```
> sqlplus user1/qwerty123@localhost:32771/ORCLCDB.localdomain
```

Try to create a table, insert some rows, select them, then drop table.

6. Follow Getting started part. Use this user credentials in connect method:

```
tarantool> c, err = ora.connect({username='user1', password='qwerty123', db='localhost:32771/ORCLCDB.
↪localdomain'})
```

Troubleshooting

If Docker can't get Oracle image with

```
docker pull store/oracle/database-enterprise:12.2.0.1
```

try to login with 'docker login'. If still nothing is happening, go to docker store and check license (Terms of Service).

6.5.3 Changelog

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog and this project adheres to Semantic Versioning.

[Unreleased]

Changed

- Removed get_position for non-scrollable cursors

[1.3.0] - 2019-11-28

Added

- Added support for connection environment

Changed

- Direct call of 'oracle.connect' is deprecated and will print warnings on call
- Added explicit error return for fiber cancelled errors
- Fixed segmentation fault on fiber cancel in the middle of fetch, cursor and connection create
- Fixed freeze on wrong credentials

[1.2.2] - 2019-09-30

Added

- Added support for scrollable cursors

Changed

- Fixed bug when cursor had no explicit Lua link to connection object and collecting connection object by gc made cursor unusable

[1.2.1] - 2019-09-16

Changed

- Fixed bug when connection close affected other connections cursors
- Fixed bug when cursors remained open on the Oracle side when closed manually

[1.2.0] - 2019-08-22

Added

- Added support for non-scrollable cursors
- Added support for connection caching parameters

[1.1.6] - 2019-07-08

Changed

- Removed memory leak when fiber with connection has been killed
- Fixed unresponsive event loop when fiber with long-running request is killed

[1.1.5] - 2019-05-27

Changed

- Bugfixes and other improvements

[1.1.4] - 2019-04-04

Changed

- Improved error handling
- Fixed pushing double value into Lua stack
- Fixed Lua VM freeze while connecting to Oracle DB
- Various bugfixes

Added

- Added OCI libs as a separate rock with dependency

[1.1.0] - 2019-02-01

Changed

- Bugfixes
- Several stability improvements

Added

- Oracle OCCI switched to 11.2
- Detect and bind params from statement
- Removed autocommit
- Added more data conversions for returning dataset
- Use one method for read and write requests
- exec_once() removed
- Tests added
- CentOS 7 Dockerfile added
- Add support for blob parameters

[1.0.0] - 2017-04-12

Added

- Basic functionality

6.6 space-explorer

6.6.1 Space explorer

Rock for exploring tarantool spaces in cartridge

Installation

- Add 'space-explorer == ...' to rockspec dependencies
- Run tarantoolctl rocks make
- Add require('space-explorer').init() after cartridge.cfg call

Example

example-app-scm-1.rockspec

```
package = 'example-app'
version = 'scm-1'
source  = {
    url = '/dev/null',
}
dependencies = {
    'tarantool',
    'lua >= 5.1',
    'luatest == 0.2.0-1',
    'ldecnumber == 1.1.3-1',
    'cartridge == scm-1',
    'space-explorer == scm-1'
}
build = {
    type = 'none';
}
```

init.lua

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

require('strict').on()

local cartridge = require('cartridge')

local ok, err = cartridge.cfg({
    roles = {
        'cartridge.roles.vshard-storage',
        'cartridge.roles.vshard-router',
        'app.roles.api',
        'app.roles.storage',
    },
    cluster_cookie = 'example-app-cluster-cookie',
})

require('space-explorer').init()

assert(ok, tostring(err))
```

Guide

Space explore guide can be seen here.

This chapter covers open and closed source Lua modules for Tarantool Enterprise included in the distribution as an offline rocks repository.

6.7 Open source modules

- avro-schema is an assembly of Apache Avro schema tools;
- cartridge is a high-level cluster management interface that contains several modules:
  - rpc implements remote procedure calls between cluster instances and allows roles running on some instances to interact with other roles on other instances.

  - service-registry implements inter-role interaction and allows different roles to interact with each other in the scope of one instance.
  - confapplier implements cluster-wide configuration validation and application via a two-phase commit.
  - auth manages authentication.
  - pool reuses Tarantool's net.box connections.
  - admin implements administration functions.
- cartridge-cli is the command-line interface for the cartridge module.
- checks is a type checker of functional arguments. This library that declares a checks() function and checkers table that allow to check the parameters passed to a Lua function in a fast and unobtrusive way.
- http is an on-board HTTP-server, which comes in addition to Tarantool's out-of-the-box HTTP client, and must be installed as described in the installation section.
- icu-date is a date-and-time formatting library for Tarantool based on International Components for Unicode;
- kafka is a full-featured high-performance kafka library for Tarantool based on librdkafka;
- ldecnumber is the decimal arithmetic library;
- luacheck is a static analyzer and linter for Lua, preconfigured for Tarantool.
- luarapidxml is a fast XML parser.
- luatest is a Tarantool test framework written in Lua.
- membership builds a mesh from multiple Tarantool instances based on gossip protocol. The mesh monitors itself, helps members discover everyone else in the group and get notified about their status changes with low latency. It is built upon the ideas from Consul or, more precisely, the SWIM algorithm.
- stat is a selection of useful monitoring metrics.
- vshard is an automatic sharding system that enables horizontal scaling for Tarantool DBMS instances.

6.8 Closed source modules

- ldap allows you to authenticate in a LDAP server and perform searches.
- odbc is an ODBC connector for Tarantool based on unixODBC.
- oracle is an Oracle connector for Lua applications through which they can send and receive data to and from Oracle databases. The advantage of the Tarantool-Oracle integration is that anyone can handle all the tasks with Oracle DBMSs (control, manipulation, storage, access) with the same high-level language (Lua) and with minimal delay.
- task is a module for managing background tasks in a Tarantool cluster.
- tracing is a module for debugging performance issues.
- space-explorer is a module for exploring Tarantool spaces in cartridge.

6.9 Installing and using modules

To use a module, install the following:

1. All the necessary third-party software packages (if any). See the module's prerequisites for the list.
2. The module itself on every Tarantool instance:

```
$ tarantoolctl rocks install <module_name> [<module_version>]
```

See also other useful tarantoolctl commands for managing Tarantool modules.

CHAPTER 7

Appendixes

7.1 Appendix A. Audit log

Audit log provides records on the Tarantool DBMS events in the JSON-format. The following event logs are available:

- successful/failed user authentication and authorization,
- closed connection,
- password change,
- creation/deletion of a user/role,
- enabling/disabling a user,
- changing privileges of a user/role.

7.1.1 Log structure

Key	Type	Description	Example
type	string	type of event	<“access_denied”>
type_id	number	id of event	<8>
description	string	description of event	<“Authentication failed”>
time	string	time of event	<“YYYY-MM-DDTHH:MM:SS.03f[+|-]GMT”>
peer	string	remote client	<“ip:port”>
user	string	user	<“user”>
param	string	parameters of event	see below

7.1.2 Events description

Event	Key	Parameters
user authorized successfully	auth_ok	{“name”: “user”}
user authorization failed	auth_fail	{“name”: “user”}
user logged out or quit the session	disconnect	
failed access attempts to secure data (personal records, details, geolocation, etc.)	access_denied	{“name”: “obj_name”, “obj_type“: “space”, “access_type”: “read”}
creating a user	user_create	{“name”: “user”}
dropping a user	user_drop	{“name”: “user”}
disabling a user	user_disable	{“name”: “user”}
enabling a user	user_enable	{“name”: “user”}
granting (changing) privileges (roles, profiles, etc.) for the user	user_priv	{“name”: “user”} “obj_name”: “obj_name”, “obj_type”: “space”, “old_priv”: “”, “new_priv”: “read,write”}
resetting password of the user (the user making changes should be specified)	password_change	{“name”: “user”}
creating a role	role_create	{“name”: “role”}
granting (changing) privileges for the role	role_priv	{“name”: “role”} “obj_name”: “obj_name”, “obj_type”: “space”, “old_priv”: “”, “new_priv”: “read,write”}

7.2 Appendix B. Useful Tarantool parameters

- box.info

- box.info.replication
- box.info.memory
- box.stat
- box.stat.net
- box.slab.info
- box.slab.stats

For details, please see reference on module 'box' in the main Tarantool documentation.

7.3 Appendix C. Monitoring system metrics

Option	Description	SNMP type	Units of measure	Threshold
Version	Tarantool version	DisplayString		
IsAlive	instance availability indicator	Integer (listing)		0 - unavailable 1 - available
MemoryLua	storage space used by Lua	Gauge32	Mbyte	900
MemoryData	storage space used for storing data	Gauge32	Mbyte	set the value manually
MemoryNet	storage space used for network I/O	Gauge32	Mbyte	1024
MemoryIndex	storage space used for storing indexes	Gauge32	Mbyte	set the value manually
MemoryCache	storage space used for storing caches (for vinyl engine only)	Gauge32	Mbyte	
ReplicationLag	lag time since the last sync between (the maximum value in case there are multiple fibers)	Integer32	sec.	5
FiberCount	number of fibers	Gauge32	pc.	1000
CurrentTime	current time, in seconds, starting at January, 1st, 1970	Unsigned32	Unix timestamp, in sec.	
StorageStatus	status of a replica set	Integer	listing	> 1
StorageAlerts	number of alerts for storage nodes	Gauge32	pc.	>= 1
StorageTotalBkts	total number of buckets in the storage	Gauge32	pc.	< 0
StorageActiveBkts	number of buckets in the ACTIVE state	Gauge32	pc.	< 0

Continued on next page

Table 1 – continued from previous page

Option	Description	SNMP type	Units of measure	Threshold
StorageGarbageBkts	number of buckets in the GARBAGE state	Gauge32	pc.	< 0
StorageReceivingBkts	number of buckets in the RECEIVING state	Gauge32	pc.	< 0
StorageSendingBkts	number of buckets in the SENDING state	Gauge32	pc.	< 0
RouterStatus	status of the router	Integer	listing	> 1
RouterAlerts	number of alerts for the router	Gauge32	pc.	>= 1
RouterKnownBkts	number of buckets within the known destination replica sets	Gauge32	pc.	< 0
RouterUnknownBkts	number of buckets that are unknown to the router	Gauge32	pc.	< 0
RequestCount	total number of requests	Counter64	pc.	
InsertCount	total number of insert requests	Counter64	pc.	
DeleteCount	total number of delete requests	Counter64	pc.	
ReplaceCount	total number of replace requests	Counter64	pc.	
UpdateCount	total number of update requests	Counter64	pc.	
SelectCount	total number of select requests	Counter64	pc.	
EvalCount	number of calls made via Eval	Counter64	pc.	
CallCount	number of calls made via call	Counter64	pc.	
ErrorCount	number of errors in Tarantool	Counter64	pc.	
AuthCount	number of completed authentication operations	Counter64	pc.	

7.4 Appendix D. Deprecated features

The ZooKeeper along with orchestrator are no longer supported. However, they still can be used, if necessary. The following sections describe the corresponding functionality.

7.4.1 Controlling the cluster via API

To control the cluster, use the orchestrator included in the delivery package. The orchestrator uses ZooKeeper to store and distribute the configuration. The orchestrator provides the REST API for controlling the cluster. Configurations in the ZooKeeper are changed as a result of calling the orchestrator's API-functions, which in turn leads to changes in configurations of the Tarantool nodes.

We recommend using a curl command line interface to call the API-functions of the orchestrator.

The following example shows how to register a new availability zone (DC):

```
$ curl -X POST http://HOST:PORT/api/v1/zone \
    -d '{
  "name": "Caucasian Boulevard"
  }'
```

To check whether the DC registration was successful, try the following instruction. It retrieves the list of all registered nodes in the JSON format:

```
$ curl http://HOST:PORT/api/v1/zone| python -m json.tool
```

To apply the new configuration directly on the Tarantool nodes, increase the configuration version number after calling the API function. To do this, use the POST request to /api/v1/version:

```
$ curl -X POST http://HOST:PORT/api/v1/version
```

Altogether, to update the cluster configuration:

1. Call the POST/PUT method of the orchestrator. As a result, the ZooKeeper nodes are updated, and a subsequent update of the Tarantool nodes is initiated.
2. Update the configuration version using the POST request to /api/v1/version. As a result, the configuration is applied to the Tarantool nodes.

See Appendix E for the detailed orchestrator API.

7.4.2 Setting up geo redundancy

Logically, cluster nodes can belong to some availability zone. Physically, an availability zone is a separate DC, or a rack inside a DC. You can specify a matrix of weights (distances) for the availability zones.

New zones are added by calling a corresponding API method of the orchestrator.

By default, the matrix of weights (distances) for the zones is not configured, and geo-redundancy for such configurations works as follows:

- Data is always written to the master.
- If the master is available, then it is used for reading.
- If the master is unavailable, then any available replica is used for reading.

When you define a matrix of weights (distances) by calling /api/v1/zones/weights, the automatic scale-out system of the Tarantool DBMS finds a replica which is the closest to the specified router in terms of weights, and starts using this replica for reading. If this replica is not available, then the next nearest replica is selected, taking into account the distances specified in the configuration.

7.5 Appendix E. Orchestrator API reference

7.5.1 Configuring the zones

- POST /api/v1/zone
- GET /api/v1/zone/
- PUT /api/v1/zone/
- DELETE /api/v1/zone/

POST /api/v1/zone
Create a new zone.

Request

```
{
"name": "zone 1"
}
```

Response

```
{
"error": {
    "code": 0,
    "message": "ok"
},
"data": {
    "id": 2,
    "name": "zone 2"
},
"status": true
}
```

Potential errors

- zone_exists - the specified zone already exists

GET /api/v1/zone/{zone_id: optional}
Return information on the specified zone or on all the zones.

Response

```
{
    "error": {
        "code": 0,
        "message": "ok"
    },
    "data": [
        {
            "id": 1,
            "name": "zone 11"
        },
        {
            "id": 2,
            "name": "zone 2"
        }
    ],
```

(continues on next page)

(continued from previous page)


```
    "status": true
}
```


Potential errors

- zone_not_found - the specified zone is not found

PUT /api/v1/zone/{zone_id}
Update information on the zone.

Body


```
{
    "name": "zone 22"
}
```


Response


```
{
    "error": {
        "code": 0,
        "message": "ok"
    },
    "data": {},
    "status": true
}
```


Potential errors

- zone_not_found - the specified zone is not found

DELETE /api/v1/zone/{zone_id}
Delete a zone if it doesn't store any nodes.

Response


```
{
    "error": {
        "code": 0,
        "message": "ok"
    },
    "data": {},
    "status": true
}
```


Potential errors

- zone_not_found - the specified zone is not found
- zone_in_use - the specified zone stores at least one node

7.5.2 Configuring the zone weights

- GET /api/v1/zones/weights
- POST /api/v1/zones/weights

POST /api/v1/zones/weights
Set the zone weights configuration.

Body


```
{
   "weights": {
      "1": {
         "2": 10,
         "3": 11
      },
      "2": {
         "1": 10,
         "3": 12
      },
      "3": {
         "1": 11,
         "2": 12
      }
   }
}
```


Response


```
{
   "error": {
      "code": 0,
      "message": "ok"
   },
   "data": {},
   "status": true
}
```


Potential errors

- zones_weights_error - configuration error

GET /api/v1/zones/weights
Return the zone weights configuration.

Response


```
{
   "error": {
      "code": 0,
      "message": "ok"
   },
   "data": {
      "1": {
         "2": 10,
         "3": 11
      },
      "2": {
         "1": 10,
         "3": 12
      },
      "3": {
         "1": 11,
         "2": 12
      }
   },
```


(continues on next page)

(continued from previous page)


```
    "status": true
}
```


Potential errors

- zone_not_found - the specified zone is not found

7.5.3 Configuring registry

- GET /api/v1/registry/nodes/new
- POST /api/v1/registry/node
- PUT /api/v1/registry/node/
- GET /api/v1/registry/node/
- DELETE /api/v1/registry/node/

GET /api/v1/registry/nodes/new
Return all the detected nodes.

Response


```
{
    "error": {
        "code": 0,
        "message": "ok"
    },
    "data": [
        {
            "uuid": "uuid-2",
            "hostname": "tnt2.public.i",
            "name": "tnt2"
        }
    ],
    "status": true
}
```


POST /api/v1/registry/node
Register the detected node.

Body


```
{
    "zone_id": 1,
    "uuid": "uuid-2",
    "uri": "tnt2.public.i:3301",
    "user": "user1:pass1",
    "repl_user": "repl_user1:repl_pass1",
    "cfg": {
        "listen": "0.0.0.0:3301"
    }
}
```


Response

```
{
   "error": {
      "code": 0,
      "message": "ok"
   },
   "data": {},
   "status": true
}
```


Potential errors

- node_already_registered - the specified node is already registered
- zone_not_found - the specified zone is not found
- node_not_discovered - the specified node is not detected

PUT /api/v1/registry/node/{node_uuid}
Update the registered node parameters.

Body

Pass only those parameters that need to be updated.


```
{
   "zone_id": 1,
   "repl_user": "repl_user2:repl_pass2",
   "cfg": {
      "listen": "0.0.0.0:3301",
      "memtx_memory": 100000
   }
}
```


Response


```
{
   "error": {
      "code": 0,
      "message": "ok"
   },
   "data": {},
   "status": true
}
```


Potential errors

- node_not_registered - the specified node is not registered

GET /api/v1/registry/node/{node_uuid: optional}
Return information on the nodes in a cluster. If node_uuid is passed, information on this node only is returned.

Response


```
{
   "error": {
      "code": 0,
      "message": "ok"
   },
   "data": {
```


(continues on next page)

(continued from previous page)


```
        "uuid-1": {
            "user": "user1:pass1",
            "hostname": "tnt1.public.i",
            "repl_user": "repl_user2:repl_pass2",
            "uri": "tnt1.public.i:3301",
            "zone_id": 1,
            "name": "tnt1",
            "cfg": {
                "listen": "0.0.0.0:3301",
                "memtx_memory": 100000
            },
            "zone": 1
        },
        "uuid-2": {
            "user": "user1:pass1",
            "hostname": "tnt2.public.i",
            "name": "tnt2",
            "uri": "tnt2.public.i:3301",
            "repl_user": "repl_user1:repl_pass1",
            "cfg": {
                "listen": "0.0.0.0:3301"
            },
            "zone": 1
        }
    },
    "status": true
}
```


Potential errors

- node_not_registered - the specified node is not registered

DELETE /api/v1/registry/node/{node_uuid}

Delete the node if it doesn't belong to any replica set.

Response


```
{
    "error": {
        "code": 0,
        "message": "ok"
    },
    "data": {},
    "status": true
}
```


Potential errors

- node_not_registered - the specified node is not registered
- node_in_use - the specified node is in use by a replica set

7.5.4 Routers API

- GET /api/v1/routers
- POST /api/v1/routers

- DELETE /api/v1/routers/{uuid}

GET /api/v1/routers
: Return the list of all nodes that constitute the router.

  Response

```
{
   "data": [
      "uuid-1"
   ],
   "status": true,
   "error": {
      "code": 0,
      "message": "ok"
   }
}
```

POST /api/v1/routers
: Assign the router role to the node.

  Body

```
{
   "uuid": "uuid-1"
}
```

  Response

```
{
   "error": {
      "code": 0,
      "message": "ok"
   },
   "data": {},
   "status": true
}
```

  Potential errors

  - node_not_registered - the specified node is not registered

DELETE /api/v1/routers/{uuid}
: Release the router role from the node.

  Response

```
{
   "error": {
      "code": 0,
      "message": "ok"
   },
   "data": {},
   "status": true
}
```

7.5.5 Configuring replica sets

- POST /api/v1/replicaset

- PUT /api/v1/replicaset/
- GET /api/v1/replicaset/
- DELETE /api/v1/replicaset/
- POST /api/v1/replicaset/{replicaset_uuid}/master
- POST /api/v1/replicaset/{replicaset_uuid}/node
- DELETE /api/v1/zone/

POST /api/v1/replicaset
Create a replica set containing all the registered nodes.

Body

```
{
   "uuid": "optional-uuid",
   "replicaset": [
      {
         "uuid": "uuid-1",
         "master": true
      }
   ]
}
```

Response

```
{
   "error": {
      "code": 0,
      "message": "ok"
   },
   "data": {
      "replicaset_uuid": "cc6568a2-63ca-413d-8e39-704b20adb7ae"
   },
   "status": true
}
```

Potential errors

- replicaset_exists – the specified replica set already exists
- replicaset_empty – the specified replica set doesn't contain any nodes
- node_not_registered – the specified node is not registered
- node_in_use – the specified node is in use by another replica set

PUT /api/v1/replicaset/{replicaset_uuid}
Update the replica set parameters.

Body

```
{
   "replicaset": [
      {
         "uuid": "uuid-1",
         "master": true
      },
      {
```

(continues on next page)

(continued from previous page)

```
            "uuid": "uuid-2",
            "master": false,
            "off": true
        }
    ]
}
```

Response

```
{
    "error": {
        "code": 0,
        "message": "ok"
    },
    "data": {},
    "status": true
}
```

Potential errors

- replicaset_empty – the specified replica set doesn't contain any nodes
- replicaset_not_found – the specified replica set is not found
- node_not_registered – the specified node is not registered
- node_in_use – the specified node is in use by another replica set

GET /api/v1/replicaset/{replicaset_uuid: optional}

Return information on all the cluster components. If replicaset_uuid is passed, information on this replica set only is returned.

Body

```
{
    "name": "zone 22"
}
```

Response

```
{
    "error": {
        "code": 0,
        "message": "ok"
    },
    "data": {
        "cc6568a2-63ca-413d-8e39-704b20adb7ae": {
            "uuid-1": {
                "hostname": "tnt1.public.i",
                "off": false,
                "repl_user": "repl_user2:repl_pass2",
                "uri": "tnt1.public.i:3301",
                "master": true,
                "name": "tnt1",
                "user": "user1:pass1",
                "zone_id": 1,
                "zone": 1
            },
```

(continues on next page)

(continued from previous page)


```
            "uuid-2": {
                "hostname": "tnt2.public.i",
                "off": true,
                "repl_user": "repl_user1:repl_pass1",
                "uri": "tnt2.public.i:3301",
                "master": false,
                "name": "tnt2",
                "user": "user1:pass1",
                "zone": 1
            }
        }
    },
    "status": true
}
```


Potential errors

- replicaset_not_found – the specified replica set is not found

DELETE /api/v1/replicaset/{replicaset_uuid}

Delete a replica set.

Response


```
{
    "error": {
        "code": 0,
        "message": "ok"
    },
    "data": {},
    "status": true
}
```


Potential errors

- replicaset_not_found - the specified replica set is not found

POST /api/v1/replicaset/{replicaset_uuid}/master

Switch the master in the replica set.

Body


```
{
    "instance_uuid": "uuid-1",
    "hostname_name": "hostname:instance_name"
}
```


Response


```
{
    "error": {
        "code": 0,
        "message": "ok"
    },
    "data": {},
    "status": true
}
```


Potential errors

- replicaset_not_found – the specified replica set is not found
- node_not_registered – the specified node is not registered
- node_not_in_replicaset – the specified node is not in the specified replica set

POST /api/v1/replicaset/{replicaset_uuid}/node
Add a node to the replica set.

Response


```
{
   "error": {
      "code": 0,
      "message": "ok"
   },
   "data": {},
   "status": true
}
```


Body


```
{
   "instance_uuid": "uuid-1",
   "hostname_name": "hostname:instance_name",
   "master": false,
   "off": false
}
```


Potential errors

- replicaset_not_found – the specified replica set is not found
- node_not_registered – the specified node is not registered
- node_in_use – the specified node is in use by another replica set

GET /api/v1/replicaset/status
Return statistics on the cluster.

Response


```
{
   "error": {
      "code": 0,
      "message": "ok"
   },
   "data": {
      "cluster": {
         "routers": [
            {
               "zone": 1,
               "name": "tnt1",
               "repl_user": "repl_user1:repl_pass1",
               "hostname": "tnt1.public.i",
               "status": null,
               "uri": "tnt1.public.i:3301",
               "user": "user1:pass1",
               "uuid": "uuid-1",
               "total_rps": null
            }
```


(continues on next page)

(continued from previous page)


```
        ],
        "storages": [
            {
                "hostname": "tnt1.public.i",
                "repl_user": "repl_user2:repl_pass2",
                "uri": "tnt1.public.i:3301",
                "name": "tnt1",
                "total_rps": null,
                "status": 'online',
                "replicas": [
                    {
                        "user": "user1:pass1",
                        "hostname": "tnt2.public.i",
                        "replication_info": null,
                        "repl_user": "repl_user1:repl_pass1",
                        "uri": "tnt2.public.i:3301",
                        "uuid": "uuid-2",
                        "status": 'online',
                        "name": "tnt2",
                        "total_rps": null,
                        "zone": 1
                    }
                ],
                "user": "user1:pass1",
                "zone_id": 1,
                "uuid": "uuid-1",
                "replicaset_uuid": "cc6568a2-63ca-413d-8e39-704b20adb7ae",
                "zone": 1
            }
        ]
    }
  },
  "status": true
}
```


Potential errors

- zone_not_found - the specified zone is not found
- zone_in_use - the specified zone stores at least one node

7.5.6 Setting up configuration versions

- POST /api/v1/version
- GET /api/v1/version

POST /api/v1/version

Set the configuration version.

Response


```
{
  "error": {
    "code": 0,
    "message": "ok"
  },
```


(continues on next page)

(continued from previous page)


```
    "status": true,
    "data": {
        "version": 2
    }
}
```


Potential errors

- cfg_error - configuration error

GET /api/v1/version
Return the configuration version.

Response


```
{
    "error": {
        "code": 0,
        "message": "ok"
    },
    "status": true,
    "data": {
        "version": 2
    }
}
```


7.5.7 Configuring sharding

- POST /api/v1/sharding/cfg
- GET /api/v1/sharding/cfg

POST /api/v1/sharding/cfg
Add a new sharding configuration.

Response


```
{
    "error": {
        "code": 0,
        "message": "ok"
    },
    "status": true,
    "data": {}
}
```


GET /api/v1/sharding/cfg
Return the current sharding configuration.

Response


```
{
    "error": {
        "code": 0,
        "message": "ok"
    },
    "status": true,
```


(continues on next page)

(continued from previous page)

```
    "data": {}
}
```

7.5.8 Resetting cluster configuration

- POST /api/v1/clean/cfg
- POST /api/v1/clean/all

POST /api/v1/clean/cfg
: Reset the cluster configuration.

  Response

```
{
    "error": {
        "code": 0,
        "message": "ok"
    },
    "status": true,
    "data": {}
}
```

POST /api/v1/clean/all
: Reset the cluster configuration and delete information on the cluster nodes from the ZooKeeper catalogues.

  Response

```
{
    "error": {
        "code": 0,
        "message": "ok"
    },
    "status": true,
    "data": {}
}
```