Postgres Pro Shardman 14.17.2 Documentation

Postgres Professional
The PostgreSQL Global Development Group
https://postgrespro.com

Postgres Pro Shardman 14.17.2 Documentation

Postgres Professional
The PostgreSQL Global Development Group

Chapter 1. Get Started with Shardman

Shardman is a PostgreSQL-based distributed database management system (DBMS) that implements sharding. *Sharding* is a database design principle where rows of a table are held separately in different databases that are potentially managed by different DBMS instances. The main purpose of Shardman is to make querying sharded distributed databases efficient and ease the complexity of managing them.

This chapter provides an introduction to the Shardman distributed DBMS.

1.1. What is Shardman

The database size in modern enterprises and in highload web applications is constantly growing. The only working approach to accommodate this growth is horizontal scaling. The Shardman distributed DBMS is intended to enable horizontal scaling of online transaction processing (OLTP) databases while preserving the strong ACID semantics.

Shardman provides the following advantages, compatibility features to your applications:

- Strong ACID guarantees.
- Compatibility with *Postgres Pro Enterprise*.
- Trust level 4 and security class 4 certificates.
- Several clusters support.
- Transparent *horizontal scaling* without a need in adopting NoSQL DBMS.
- *Built-in support of replication with no single point of failure*, with any node being able to become coordinator that requires no system shut down and prevents any data loss.
- Capacity of up to 100 cluster nodes.
- *High availability* with primary and stand-by modes, along with the synchronous solution for the failover scenarios, and asynchronous solution that has a minimal performance impact.
- *Support* of planning and execution statistics of all SQL statements.
- *Utility* to discover most resource-intensive activities in your database.
- *Tools* to support `REPEATABLE READ` isolation level in a distributed system.
- *Work with cluster* as with a fully functional DBMS.
- *Hot standby* and *backup and recovery* tools that support full and incremental backup with logs.
- *Point-in-time recovery (PITR)*.
- *Streaming replication*.
- High-availability cluster creation with multiple primary nodes with special *utilities*.
- *ANSI* standard.
- *SQL arrays*.
- *Stored procedures*.
- *Big data* storing and processing.
- *Full text search*.
- *Covering indexes*.
- *B-tree, hash, GiST, GIN, SP-GiST, BRIN* indexes.
- *Perl* and *Python* procedural languages.
- Interfaces for *C++, Ruby*, *C*, *ODBC Perl, Python, Tcl*, and *Java*.
- *EUC, UTF-8, and Mule* character set.
- *Adaptive query optimization*.
- *Compressed file system*.
- Access data stored in external PostgreSQL servers with *postgres_fdw*, e.g. *Microsoft Active Directory*, *Mysql server*, *Oracle*, and *Postgres Pro Enterprise*.

- Long queries monitoring with the *pg_query_state module*.
- *In-built monitoring agent*.
- *No limits* for the number of records or indexes, with the maximum table size of 32 TB, maximum attribute size of 1 GB, and maximum number of attributes of 1600.
- *Detailed access management* with different access levels and roles.
- *Secure password storing*.
- Detailed *memory purge* configuration.

1.2. When to use

Shardman provides horizontal scalability with a view and consistency of a single database. Applications can use every node to access the distributed database and operate mostly the same way as with a single PostgreSQL instance. Still internally it is a distributed system that imposes certain rules on designing schema and writing queries. The main direction of adoption is to localize the data and the computations.

The following properties of a database or workload should be marks to consider a distributed system:

- The working set of data does not fit in RAM of one server. Sharded systems can have much bigger total size of RAM.
- Maintenance operations such as vacuum take too long. Shardman utilizes partitioned tables under the hood. Maintenance operations can be parallelized by nodes and partitions of tables.
- Number of read sessions is too large for one instance of PostgreSQL. Shardman allows to distribute read sessions across the cluster and handle internal connections very efficiently with multiplexing transport.
- Intensive write operations. Sharded systems can have much bigger total number of disk IOPS.
- CPU intensive queries. Shardman allows to distribute calculations by nodes and reduce execution time for complex queries.

When Shardman is not appropriate:

- Vertical scaling is economically and technically possible.
- Data model and workload require a lot of cross-shard transactions.
- Complex analytics, in particular joins of sharded tables when conditions don't include the sharding key.
- Multi-DC/Multi-region deployments.

1.3. Quickstart Guide

Shardman is composed of several software components:

- PostgreSQL 14 DBMS with a set of patches.
- Shardman extension.
- Management tools and services, including built-in stolon manager to provide high availability.

Postgres Pro Shardman and stolon store their configuration in an etcd cluster. Therefore, we can use an existing etcd cluster, or we can deploy a simple one-node etcd cluster.

The shardmand daemon monitors the cluster configuration and manages stolon clusters, which are used to guarantee high availability of all shards. The common Shardman configuration (shardmand, stolon) is stored in an etcd cluster.

Currently Shardman packages are available for

- Ubuntu 20.04/22.04
- Debian 10/11/12
- Red Hat Enterprise Linux 7/8/9
- Red OS 7.3/7.3.1/7.3.2
- Alt 9/10
- AstraLinux 1.7 (Smolensk)

1.3.1. Cluster Configuration

Assume that we have three nodes for deploying Postgres Pro Shardman. Let’s make the first one for the etcd one-node cluster and the other two nodes for the Postgres Pro Shardman two-node cluster.

Let’s suppose that we have the following node names and IP addresses:

```
192.0.1.1 etcd  - etcd one-node cluster
192.0.1.20 sdm01 - Shardman node1
192.0.1.21 sdm02 - Shardman node2
```

Each node has 4Gb RAM, 20GB HDD, 2CPU and Ubuntu 22.04 installed.

1.3.2. Preparation

1.3.2.1. Add host names to `/etc/hosts`

This step must be performed on all nodes.

```
sudo /bin/sh -c 'cat << EOF >> /etc/hosts
192.0.1.1 etcd
192.0.1.20 sdm01
192.0.1.21 sdm02
EOF'
```

1.3.2.2. Time Synchronization

This step must be performed on all nodes.

Deploy and start chrony daemon on all hosts.

```
sudo apt install -y chrony
```

By default, chrony gets the time from available servers on internet or the local time server. You can check available time servers as follows:

```
chronyc sources
MS Name/IP address         Stratum Poll Reach LastRx Last sample
===============================================================================
^? 192.0.1.100                   1   6     7     1    -98us[  -98us] +/-   11ms
^* time.cloudflare.com           3   6     7     1   +139us[ +163us] +/-   11ms
^+ tms04.deltatelesystems.ru     1   6     7     1   -381us[ -357us] +/-   17ms
```

It is desirable to synchronize time with your server or the local server for the cluster. To do this, make changes similar to the following to chrony configuration:

```
cat /etc/chrony/chrony.conf

server 192.0.1.100 iburst
keyfile /etc/chrony.keys
driftfile /var/lib/chrony/chrony.drift
log tracking measurements statistics
logdir /var/log/chrony

systemctl restart chrony
```

Check that chrony is connected to the appropriate server.

```
chronyc sources
MS Name/IP address         Stratum Poll Reach LastRx Last sample
===============================================================================
^? 192.0.1.100                   8   6    17    37    +14us[  +70us] +/-  161us
chronyc tracking
Reference ID    : 0A80000C (ntp.local)
```

```
Stratum         : 9
Ref time (UTC)  : Wed Nov 15 11:58:52 2023
System time     : 0.000000004 seconds slow of NTP time
Last offset     : -0.000056968 seconds
RMS offset      : 0.000056968 seconds
Frequency       : 10.252 ppm fast
Residual freq   : -2.401 ppm
Skew            : 364.419 ppm
Root delay      : 0.000455358 seconds
Root dispersion : 0.010503666 seconds
Update interval : 2.1 seconds
Leap status     : Normal
```

1.3.3. Deploy an etcd One-Node Cluster

Note also a Deploy a Multi-Node etcd cluster section.

Install the following packages:

```
sudo apt install -y vim curl
```

To connect a Postgres Pro Shardman repository:

- Run

  ```
  curl -fsSL -u "<user>:<password>" https://repo.postgrespro.ru/sdm/sdm-14/keys/pgpro-repo-add.sh > pgpro-repo-add.sh
  chmod +x pgpro-repo-add.sh
  ```

- Open the file `pgpro-repo-add.sh` and specify the repository password in the `PASSWORD` variable.
- Run `sudo pgpro-repo-add.sh`.

Install etcd-sdm packages:

```
sudo apt install -y etcd-sdm
```

In the file that lists environment variables, insert specific values for them:

```
sudo vim /etc/default/etcd-sdm
ETCD_NAME=etcd
ETCD_LISTEN_CLIENT_URLS=http://0.0.0.0:2379
ETCD_ADVERTISE_CLIENT_URLS=http://192.0.1.1:2379
ETCD_MAX_SNAPSHOTS=5
ETCD_MAX_WALS=5
ETCD_AUTO_COMPACTION_MODE=periodic
ETCD_AUTO_COMPACTION_RETENTION=5m
ETCD_QUOTA_BACKEND_BYTES=6442450944
ETCD_DATA_DIR=/var/lib/etcd-sdm/sdm-14
```

This file will be loaded at etcd start.

Clear the etcd data directory:

```
sudo rm -rf /var/lib/etcd-sdm/sdm-14/*
```

Restart the etcd-sdm service:

```
sudo systemctl restart etcd-sdm
```

For your user, add `/opt/pgpro/sdm-14/bin` to the `PATH` environment variable:

```
echo "export PATH=$PATH:/opt/pgpro/sdm-14/bin" >> .bashrc
source .bashrc
```

Check that etcd is properly configured:

```
etcdctl endpoint --endpoints=http://192.0.1.1:2379 status health -w table
+-----------------------+-----------------+---------+---------
+-----------+-----------+-----------+-----------+-------------------
+-------------------------------+
|        ENDPOINT        |        ID        | VERSION | DB SIZE | IS LEADER | IS
 LEARNER | RAFT TERM | RAFT INDEX | RAFT APPLIED INDEX |             ERRORS
 |
+-----------------------+-----------------+---------+---------
+-----------+-----------+-----------+-----------+-------------------
+-------------------------------+
| http://192.0.1.1:2379 | 9324a99282752a09 |   3.5.9 |  2.1 GB |      true |      false
 |        14 |   91459207 |           91459207 |  memberID:10602785869456026121 |
|                       |                    |         |         |           |
  |           |            |                    |                 alarm:NOSPACE  |
+-----------------------+-----------------+---------+---------
+-----------+-----------+-----------+-----------+-------------------
+-------------------------------+
```

etcd one-node cluster is properly configured and ready to serve requests.

To prevent bloat when etcd is intensively used, add a defragmentation command to cron:

```
sudo sh -c '
{ crontab -l; echo "@hourly /opt/pgpro/sdm-14/bin/etcdctl defrag"; }
| crontab'
```

1.3.4. Deploy Shardman Nodes

Let’s add a Postgres Pro Shardman repository on each node:

- Set a user and password as in Section 1.3.3.
- Run

  ```
  curl -fsSL -u "<user>:<password>" https://repo.postgrespro.ru/sdm/sdm-14/keys/pgpro-repo-add.sh > pgpro-repo-add.sh | bash
  chmod +x pgpro-repo-add.sh
  ```

- Open the file `pgpro-repo-add.sh` and specify the repository password in the `PASSWORD` variable.
- Run `pgpro-repo-add.sh`.

Next step is installation of packages (on each node):

```
  sudo apt update
  sudo apt install -y postgrespro-sdm-14-server postgrespro-sdm-14-client postgrespro-sdm-14-contrib postgrespro-sdm-14-libs pg-probackup-sdm-14  shardman-services shardman-tools
```

Suppose we have chosen a default cluster name of `cluster0`. The next step is to put Shardman environment vars into the `/etc/shardman` directory (on each node):

```
sudo sh -c 'cat << EOF > /etc/shardman/shardmand-cluster0.env
SDM_CLUSTER_NAME=cluster0
SDM_LOG_LEVEL=info
SDM_STORE_ENDPOINTS=http://etcd:2379
EOF'
```

The file and directory are created with `sudo`, but later `shardmanctl` does not use `sudo`, thus cannot access the file with the environment variables. To access it, either add the variables to the system with `export`, or grant user with access rights to the file and the directory.

For your user, add `/opt/pgpro/sdm-14/bin` to the `PATH` environment variable:

```
echo "export PATH=$PATH:/opt/pgpro/sdm-14/bin" >> .bashrc
source .bashrc
```

Let’s generate a sample configuration with the Shardman utilities (only on one node).

```
shardmanctl config generate > spec.json
```

In this step, you can make some changes to the cluster specification (configuration), i.e., change the password or PostgreSQL `shared_buffers` parameter and so on.

1.3.5. Initialize the Shardman Cluster

Now we have some final steps. First, let's initialize the cluster configuration in etcd (only on one [any] node).

```
shardmanctl init -f spec.json
```

The expected output is:

```
2023-04-18T12:30:03.043Z DEBUG cmd/common.go:100 Waiting for metadata lock...
2023-04-18T12:30:03.048Z DEBUG cluster/cluster.go:365 DataDir is not specified,
setting to default /var/lib/pgpro/sdm-14/data
```

Enable and start the shardmand service (on each node):

```
sudo systemctl enable --now shardmand@cluster0
sudo systemctl status shardmand@cluster0
```

```
# shardmand@cluster0.service - deployment daemon for shardman
     Loaded: loaded (/lib/systemd/system/shardmand@.service; enabled; vendor preset:
enabled)
     Active: active (running) since Tue 2023-04-18 12:28:18 UTC; 2min 13s ago
       Docs: https://github.com/postgrespro/shardman
   Main PID: 618 (shardmand)
      Tasks: 10 (limit: 4571)
     Memory: 32.0M
        CPU: 422ms
     CGroup: /system.slice/system-shardmand.slice/shardmand@cluster0.service
             ##618 /opt/pgpro/sdm-14/bin/shardmand --cluster-name cluster0 --system-
bus --user postgres
```

1.3.6. Add Nodes to the Shardman Cluster

In this step we assume that all previous steps were executed successfully: etcd cluster is working properly, the time on all hosts is synchronized, and the daemon is launched on `sdm01` and `sdm02`. The final step should be executed with `shardmanctl` command as follows:

```
shardmanctl nodes add -n sdm01,sdm02 \
           --cluster-name cluster0 \
           --log-level debug \
           --store-endpoints=http://etcd:2379
```

The expected output should be:

```
2023-04-18T12:43:11.300Z DEBUG cmd/common.go:100 Waiting for metadata lock...
2023-04-18T12:43:11.306Z INFO cluster/store.go:277 Checking if shardmand on all nodes
have applied current cluster configuration
# Waiting for shardmand on node sdm01 to apply current configuration: success 0.000s
# Waiting for shardmand on node sdm02 to apply current configuration: success 0.000s
2023-04-18T12:43:11.307Z INFO add/case.go:112 Initting Stolon instances...
2023-04-18T12:43:11.312Z INFO add/case.go:170 Waiting for Stolon daemons to start...
make sure shardmand daemons are running on the nodes
# Waiting for Stolon daemons of rg clover-1-sdm01: success 31.012s
```

```
 # Waiting for Stolon daemons of rg clover-1-sdm02: success 0.012s
 2023-04-18T12:43:42.336Z INFO add/case.go:187 Adding repgroups...
 # waiting rg 1 config apply: done 7.014s
 2023-04-18T12:43:49.444Z DEBUG broadcaster/worker.go:33 start broadcaster worker for
repgroup id=1
 2023-04-18T12:43:49.453Z DEBUG broadcaster/worker.go:51 repgroup 1 connect
established
 2023-04-18T12:43:49.453Z DEBUG commands/addrepgroup.go:575 waiting for extension
lock...
 2023-04-18T12:43:49.453Z DEBUG commands/addrepgroup.go:137 Loading schema into
replication group rg 1
 ...
 2023-04-18T12:44:25.665Z DEBUG rebalance/service.go:528 wait all tasks finish
 2023-04-18T12:44:25.666Z DEBUG broadcaster/worker.go:75 finish broadcaster worker for
repgroup id=1
 2023-04-18T12:44:25.666Z DEBUG broadcaster/worker.go:75 finish broadcaster worker for
repgroup id=2
 2023-04-18T12:44:25.666Z INFO add/case.go:221 Successfully added nodes sdm01, sdm02
to the cluster
```

The “Successfully added nodes sdm01, sdm02 to the cluster” message means that everything is fine and nodes `sdm01` and `sdm02` are working properly.

1.3.7. Check the Shardman Cluster Status

Let's check the status of the cluster nodes

```
 shardmanctl status

 ##############################################################################################
 #                                          == STORE STATUS ==
             #

 ##############################################################################################
 #   STATUS  #                   MESSAGE                 #  REPLICATION GROUP  #
 NODE        #

 ##############################################################################################
 #  Warning  # Store has only one member, consider      #                     #
             #
 #           # deploying store cluster                 #                     #
             #

 ##############################################################################################

 ##############################################################################################
 #                                       == TOPOLOGY STATUS ==
            #

 ##############################################################################################
 #   STATUS  #                  MESSAGE                 #  REPLICATION GROUP  #
NODE        #

 ##############################################################################################
 #   CROSS   # Topology placement policy is CROSS      #                     #
            #

 ##############################################################################################
```

```
#################################################################################################
 #                                        == METADATA STATUS ==
            #

#################################################################################################
 #   STATUS  #                  MESSAGE                  #  REPLICATION GROUP  #
NODE         #

#################################################################################################
 #     OK    # Metadata is OK                           #                     #
            #

#################################################################################################

#################################################################################################
 #                                        == SHARDMAND STATUS ==
            #

#################################################################################################
 #   STATUS  #                  MESSAGE                  #  REPLICATION GROUP  #
NODE         #

#################################################################################################
 #     OK    # shardmand on node sdm01 is OK            #                     #
sdm01        #

#################################################################################################
 #     OK    # shardmand on node sdm02 is OK            #                     #
sdm02        #

#################################################################################################

#################################################################################################
 #                                    == REPLICATION GROUP STATUS ==
            #

#################################################################################################
 #   STATUS  #                  MESSAGE                  #  REPLICATION GROUP  #
NODE         #

#################################################################################################
 #     OK    # Replication group clover-1-sdm01 is OK   #    clover-1-sdm01   #
            #

#################################################################################################
 #     OK    # Replication group clover-1-sdm02 is OK   #    clover-1-sdm02   #
            #

#################################################################################################

#################################################################################################
 #                                          == MASTER STATUS ==
             #

#################################################################################################
 #   STATUS  #                   MESSAGE                  #  REPLICATION GROUP  #
 NODE        #
```

```
#################################################################################################
 #     OK    # Replication group clover-1-sdm01 master  #    clover-1-sdm01   #
sdm01:5432     #
 #           # is running on sdm01:5432                #                     #
             #

#################################################################################################
 #     OK    # Replication group clover-1-sdm02 master  #    clover-1-sdm02   #
sdm02:5432     #
 #           # is running on sdm02:5432                #                     #
             #

#################################################################################################

#################################################################################################
 #                                      == DICTIONARY STATUS ==
             #

#################################################################################################
 #   STATUS  #                  MESSAGE                #  REPLICATION GROUP  #
 NODE        #

#################################################################################################
 #     OK    # Replication group clover-1-sdm01         #    clover-1-sdm01   #
             #
 #           # dictionary is OK                        #                     #
             #

#################################################################################################
 #     OK    # Replication group clover-1-sdm02         #    clover-1-sdm02   #
             #
 #           # dictionary is OK                        #                     #
             #

#################################################################################################
```

1.3.8. Connect to the Shardman Cluster

To connect to the cluster we should get the cluster connection string on any cluster node (`sdm01` or `sdm02`):

```
 shardmanctl getconnstr

 dbname=postgres host=sdm01,sdm02 password=!!!CHANGE_ME!!! port=5432,5432
user=postgres
```

And then let’s try to connect:

```
 psql -d 'dbname=postgres host=sdm01,sdm02 password=!!!CHANGE_ME!!! port=5432,5432
user=postgres'

 psql (14.7)
 Type "help" for help.

 postgres=#
```

1.3.9. Create Sharded Tables

Let's try to create a sharded table and check if everything is working properly.

```
 postgres=# create table x(id int primary key, t text) with
(distributed_by='id',num_parts=2);
 CREATE TABLE

 postgres=# \d
                 List of relations
  Schema |  Name   |       Type        |  Owner
 --------+---------+-------------------+----------
  public | x       | partitioned table | postgres
  public | x_0     | table             | postgres
  public | x_1_fdw | foreign table     | postgres
 (3 rows)

 postgres=# \d x_0
  Table "public.x_0"
  Column |  Type   | Collation | Nullable | Default
 --------+---------+-----------+----------+---------
  id     | integer |           | not null |
  t      | text    |           |          |
 Partition of: x FOR VALUES WITH (modulus 2, remainder 0)
 Indexes:
     "x_0_pkey" PRIMARY KEY, btree (id)

 postgres=# \d x_1_fdw
                 Foreign table "public.x_1_fdw"
 Column |  Type   | Collation | Nullable | Default | FDW options
--------+---------+-----------+----------+---------+-------------
 id     | integer |           | not null |         |
 t      | text    |           |          |         |
Partition of: x FOR VALUES WITH (modulus 2, remainder 1)
Server: shardman_rg_2
FDW options: (table_name 'x_1')

 postgres=# insert into x values (1,'t'),(2,'t'),(3,'t');
 INSERT 0 3

 postgres=# select * from x_0;
  id | t
 ----+---
   1 | t
   2 | t
 (2 rows)

 postgres=# select * from x_1_fdw;
  id | t
 ----+---
   3 | t
 (1 row)
```

Everything works as expected.

1.3.10. Example: Deploy a Multi-Node etcd Cluster

The process is described for the following servers:

```
192.0.1.1 etcd1
192.0.1.2 etcd2
192.0.1.3 etcd3
```

Install the needed packages on each server:

```
sudo apt install -y vim curl
```

To connect the repository, on each server, run:

```
sudo curl -fsSL https://repo.postgrespro.ru/sdm/sdm-14/keys/pgpro-repo-add.sh | bash
```

Install etcd-sdm packages on each server:

```
sudo apt install -y etcd-sdm
```

For each server, edit the file that lists environment variables, replacing placeholders in angle brackets with specific values:

```
sudo vim /etc/default/etcd-sdm
ETCD_NAME=<hostname>
ETCD_LISTEN_PEER_URLS=http://0.0.0.0:2380
ETCD_LISTEN_CLIENT_URLS=http://0.0.0.0:2379
ETCD_ADVERTISE_CLIENT_URLS=http://<host ip address>:2379
ETCD_INITIAL_ADVERTISE_PEER_URLS=http://<host ip address>:2380
ETCD_INITIAL_CLUSTER_TOKEN=etcd-cluster-1
ETCD_INITIAL_CLUSTER_STATE=new
ETCD_MAX_SNAPSHOTS=5
ETCD_MAX_WALS=5
ETCD_AUTO_COMPACTION_MODE=periodic
ETCD_AUTO_COMPACTION_RETENTION=5m
ETCD_QUOTA_BACKEND_BYTES=6442450944
ETCD_DATA_DIR=/var/lib/etcd-sdm/sdm-14
ETCD_INITIAL_CLUSTER=etcd1=http://<ip etcd1>:2380,etcd2=http://<ip
 etcd2>:2380,etcd3=http://<ip etcd3>:2380
```

This file will be loaded at etcd start with its own start settings on each server.

Clear the etcd data directory:

```
sudo rm -rf /var/lib/etcd-sdm/sdm-14/*
```

Restart the etcd-sdm service on each server:

```
sudo systemctl restart etcd-sdm
```

For your user, add `/opt/pgpro/sdm-14/bin` to the `PATH` environment variable:

```
echo "export PATH=$PATH:/opt/pgpro/sdm-14/bin" >> .bashrc
source .bashrc
```

Check that etcd is properly configured:

```
etcdctl member list -w table
+------------------+---------+-------+---------------------------
+---------------------------+------------+
|        ID        | STATUS  | NAME  |         PEER ADDRS         |        CLIENT ADDRS
        | IS LEARNER |
+------------------+---------+-------+---------------------------
+---------------------------+------------+
|  318be6342e6d9ac | started | etcd1 | http://192.0.1.1:2380      |
 http://192.0.1.1:2379      |      false |
| 9e49480544aedb89 | started | etcd2 | http://192.0.1.2:2380      |
 http://192.0.1.2:2379      |      false |
| bb3772bfa22482d7 | started | etcd3 | http://192.0.1.3:2380      |
 http://192.0.1.3.4:2379    |      false |
+------------------+---------+-------+---------------------------
+---------------------------+------------+
```

```
$ etcdctl --endpoints=http://192.0.1.1:2380,http://192.0.1.2:2380,http://192.0.1.3:2380
 endpoint status health  -w table
+---------------------------+-----------------+---------+---------+-----------
+------------+-----------+------------+--------------------+--------+
|          ENDPOINT         |        ID        | VERSION | DB SIZE | IS LEADER | IS
 LEARNER | RAFT TERM | RAFT INDEX | RAFT APPLIED INDEX | ERRORS |
+---------------------------+-----------------+---------+---------+-----------
+------------+-----------+------------+--------------------+--------+
| http://192.0.1.1:2380      |  318be6342e6d9ac |   3.5.9 |  5.7 MB |      true |
 false |        13 |     425686 |             425686 |        |
| http://192.0.1.2:2380      | 9e49480544aedb89 |   3.5.9 |  5.7 MB |     false |
 false |        13 |     425686 |             425686 |        |
|  http://192.0.1.3:2380     | bb3772bfa22482d7 |   3.5.9 |  5.7 MB |     false |
 false |        13 |     425686 |             425686 |        |
+---------------------------+-----------------+---------+---------+-----------
+------------+-----------+------------+--------------------+--------+
-------+
```

The etcd cluster is properly configured and ready to serve requests.

To prevent bloat when etcd is intensively used, add a defragmentation command to cron:

```
sudo { crontab -l; echo "@hourly /opt/pgpro/sdm-14/bin/etcdctl defrag"; } | crontab
```

The final endpoints string of the etcd cluster:

```
etcd1=http://<ip etcd1>:2380,etcd2=http://<ip etcd2>:2380,etcd3=http://<ip etcd3>:2380
```

It should be specified in `/etc/shardman` configuration file and as a `--store-endpoints` parameter of shardmanctl.

Chapter 2. Manage

2.1. Cluster Services

The Shardman cluster configuration is stored in etcd. Shardman cluster services are organized as systemd services. The Shardman configuration daemon shardmand monitors the cluster configuration and manages PostgreSQL instances through integrated stolon. Each node has one shardmand service, whose typical name is `shardmand@CLUSTER_NAME.service`. Here *`CLUSTER_NAME`* is the Shardman cluster name, `cluster0` by default.

Each shardmand includes several integrated stolon `keeper` and stolon `sentinel` threads.

Each registered DBMS instance has an associated stolon `keeper` thread that directly manages this PostgreSQL instance. The `keeper` starts, stops, initializes and resyncs PostgreSQL instances according to the desired stolon cluster state.

Each registered DBMS instance has an associated stolon `sentinel` thread. For each replication group, stolon `sentinels` elect the leader among existing `sentinels`. This leader makes decisions about the desired cluster state (for example, which `keeper` should become a new master when the existing one fails). When the new master in a replication group is selected, the leader selects the `keeper` with the minimal lag. When all replicas are synchronous, the `keeper` with the maximal priority is selected to become a new master even when the master in the replication group is alive. Shardman only uses synchronous replicas (otherwise, there is a chance to lose data when a node fails).

shardmand is a systemd unit, its logs are written to journald. You can use `journalctl` to examine it. For example, to get all logs since `2023-05-09 10:00` for the shardmand service of the `cluster0` cluster, you can use the following command:

```
$ journalctl -u shardmand@cluster0 --since '2023-05-09 10:00'
```

To control the log verbosity for all Shardman services, set `SDM_LOG_LEVEL` in the shardmand configuration file.

2.2. Scaling the Cluster

The Shardman architecture allows you to scale out your cluster without any downtime. This section describes how you can add more nodes to your Shardman cluster in order to improve query performance/scalability. If a Shardman cluster does not meet your performance expectations or storage capacity, you can add new nodes to the cluster.

2.2.1. Adding and Removing a Node

How nodes are added to a cluster and where replicas will be located depends on the type of a highly available configuration. Shardman supports two types of configurations: cross-replication mode and manual-topology mode. The `PlacementPolicy` parameter in `sdmspec.json` allows you to select the cluster behavior. The parameter supports two values: `cross` and `manual`. The default is `cross`. For example:

```
{
    "PlacementPolicy": "cross",
    "Repfactor": 1,
    ...
}
```

2.2.1.1. Cross Replication

The `shardmanctl nodes add` command is used to add new nodes to a Shardman cluster. With *cross* placement policy, nodes are added to a cluster by *clovers*. Each node in a clover runs the primary DBMS instance and replicas of other nodes in the clover. The number of replicas is determined by the Repfactor configuration parameter. So, each clover consists of `Repfactor + 1` nodes and can stand loss of `Repfactor` nodes. An example of creating a cluster of four nodes with `Repfactor`=1 and cross replication is shown below:

```
$ shardmanctl --store-endpoints http://etcd1:2379,http://
etcd2:2379,http://etcd3:2379 init -f sdmspec.json
```

```
                        $ shardmanctl --store-endpoints http://etcd1:2379,http://
etcd2:2379,http://etcd3:2379 nodes add -n n1,n2,n3,n4
```

View the topology of a cluster:

```
                        $ shardmanctl --store-endpoints http://etcd1:2379,http://
etcd2:2379,http://etcd3:2379 cluster topology
```

The command output is as follows:

```
 ###########################################################################
                        #              == REPLICATION GROUP clover-1-n1, RGID - 1 ==
       #

 ###########################################################################
                        #            HOST           #       PORT      #           STATUS
       #

 ###########################################################################
                        #              n1           #       5432      #          PRIMARY
       #

 ###########################################################################
                        #              n2           #       5433      #          STANDBY
       #

 ###########################################################################

 ###########################################################################
                        #              == REPLICATION GROUP clover-1-n2, RGID - 2 ==
       #

 ###########################################################################
                        #            HOST           #       PORT      #           STATUS
       #

 ###########################################################################
                        #              n1           #       5433      #          STANDBY
       #

 ###########################################################################
                        #              n2           #       5432      #          PRIMARY
       #

 ###########################################################################

 ###########################################################################
                        #              == REPLICATION GROUP clover-2-n3, RGID - 1 ==
       #

 ###########################################################################
                        #            HOST           #       PORT      #           STATUS
       #

 ###########################################################################
                        #              n3           #       5432      #          PRIMARY
       #

 ###########################################################################
```

```
                        #             n4            #       5433      #          STANDBY
      #

 ###########################################################################

 ###########################################################################
                        #              == REPLICATION GROUP clover-2-n4, RGID - 2 ==
      #

 ###########################################################################
                        #            HOST           #       PORT      #           STATUS
      #

 ###########################################################################
                        #             n3            #       5433      #          STANDBY
      #

 ###########################################################################
                        #             n4            #       5432      #          PRIMARY
      #

 ###########################################################################
```

The `shardmanctl nodes rm` command is used to remove nodes from a Shardman cluster. This command removes clovers containing the specified nodes from the cluster. The last clover in the cluster cannot be removed. Any data (such as partitions of sharded relations) on removed replication groups is migrated to the remaining replication groups using logical replication, and all references to the removed replication groups (including definitions of foreign servers) are removed from the metadata of the remaining replication groups. Finally, the metadata in etcd is updated.

```
                        $ shardmanctl --store-endpoints http://etcd1:2379,http://
etcd2:2379,http://etcd3:2379 nodes rm -n n3
```

View the topology of a cluster:

```
                        $ shardmanctl --store-endpoints http://etcd1:2379,http://
etcd2:2379,http://etcd3:2379 cluster topology
```

The command output is as follows:

```
 ###########################################################################
                        #              == REPLICATION GROUP clover-1-n1, RGID - 1 ==
      #

 ###########################################################################
                        #            HOST           #       PORT      #           STATUS
      #

 ###########################################################################
                        #             n1            #       5432      #          PRIMARY
      #

 ###########################################################################
                        #             n2            #       5433      #          STANDBY
      #

 ###########################################################################

 ###########################################################################
                        #              == REPLICATION GROUP clover-1-n2, RGID - 2 ==
      #
```

```
##########################################################################
                        #            HOST           #       PORT      #          STATUS
      #
##########################################################################
                        #             n1            #       5433      #          STANDBY
      #
##########################################################################
                        #             n2            #       5432      #          PRIMARY
      #
##########################################################################
```

2.2.1.2. Manual Topology

In the *manual-topology* mode, to add a primar# to a cluster, use the `shardmanctl nodes add` command, which adds the list of nodes to the cluster as primaries with a separate replication group for each primary. Create a cluster with three primary nodes and manual topology (`PlacementPolicy=manual` in `sdmspec.json`):

```
                        $ shardmanctl --store-endpoints http://etcd1:2379,http://
etcd2:2379,http://etcd3:2379 init -f sdmspec.json
                        $ shardmanctl --store-endpoints http://etcd1:2379,http://
etcd2:2379,http://etcd3:2379 nodes add -n n1,n2,n3
```

To view the topology of a cluster, use the `shardmanctl cluster topology` command:

```
                        $ shardmanctl --store-endpoints http://etcd1:2379,http://
etcd2:2379,http://etcd3:2379 cluster topology
```

The command output is as follows:

```
##########################################################################
                        #              == REPLICATION GROUP clover-1-n1, RGID - 1 ==
         #
##########################################################################
                        #            HOST           #       PORT       #          STATUS
         #
##########################################################################
                        #             n1            #       5432       #
PRIMARY          #
##########################################################################

##########################################################################
                        #              == REPLICATION GROUP clover-2-n2, RGID - 2 ==
         #
##########################################################################
                        #            HOST           #       PORT       #          STATUS
         #
##########################################################################
                        #             n2            #       5432       #
PRIMARY          #
##########################################################################
```

```
################################################################################
                    #              == REPLICATION GROUP clover-3-n3, RGID - 3 ==
        #

################################################################################
                    #           HOST           #       PORT       #          STATUS
        #

################################################################################
                    #            n3            #       5432       #
PRIMARY         #

################################################################################
```

Add `n4`, `n5`, `n6` nodes as replicas using the `shardmanctl shard add` command:

```
                    $ shardmanctl --store-endpoints http://etcd1:2379,http://
etcd2:2379,http://etcd3:2379 shard --shard clover-1-n1 add -n n4
                    $ shardmanctl --store-endpoints http://etcd1:2379,http://
etcd2:2379,http://etcd3:2379 shard --shard clover-2-n2 add -n n5
                    $ shardmanctl --store-endpoints http://etcd1:2379,http://
etcd2:2379,http://etcd3:2379 shard --shard clover-3-n3 add -n n6
```

In manual-topology mode, one node can be added to more than one replication group.

As a result, we get the following cluster configuration:

```
################################################################################
                    #              == REPLICATION GROUP clover-1-n1, RGID - 1 ==
      #

################################################################################
                    #           HOST           #       PORT      #          STATUS
      #

################################################################################
                    #            n1            #       5432      #         PRIMARY
      #

################################################################################
                    #            n4            #       5432      #         STANDBY
      #

################################################################################

################################################################################
                    #              == REPLICATION GROUP clover-2-n2, RGID - 2 ==
      #

################################################################################
                    #           HOST           #       PORT      #          STATUS
      #

################################################################################
                    #            n2            #       5432      #         PRIMARY
      #

################################################################################
```

```
                        #             n5            #       5432      #          STANDBY
      #
 ###########################################################################
 ###########################################################################
                        #              == REPLICATION GROUP clover-3-n3, RGID - 3 ==
      #
 ###########################################################################
                        #            HOST           #       PORT      #           STATUS
      #
 ###########################################################################
                        #             n3            #       5432      #          PRIMARY
      #
 ###########################################################################
                        #             n6            #       5432      #          STANDBY
      #
 ###########################################################################
```

To remove a replica, just run the `shardmanctl shard rm` command. For example:

```
                        $ shardmanctl --store-endpoints http://etcd1:2379,http://
etcd2:2379,http://etcd3:2379 shard --shard clover-1-n1 rm -n n4
```

To remove the master, first run the `shardmanctl shard switch` command to switch the master to the replica; then delete the old master.

```
                        $ shardmanctl --store-endpoints http://etcd1:2379,http://
etcd2:2379,http://etcd3:2379 shard --shard clover-1-n1 switch --new-primary n4
```

2.3. Rebalancing the Data

2.3.1. Automatically Rebalancing the Data

Automatic rebalancing is used as the default mode. A rebalance process starts automatically after adding nodes (by default if `--no-rebalance` is not set) or before deleting a node. Rebalance can also be started manually. The essence of the rebalancing process is to evenly distribute partitions for each sharded table between replication groups.

The rebalancing process for each sharded table iteratively determines the replication group with the maximum and minimum number of partitions and creates a task to move one partition to the replication group with the minimum number of partitions. This process is repeated while `max - min > 1`. To move partitions, we use logical replication. Partitions of colocated tables are moved together with partitions of the sharded tables to which they refer.

It is important to remember that `max_logical_replication_workers` should be rather high since the rebalance process uses up to `max(max_replication_slots, max_logical_replication_workers, max_worker_processes, max_wal_senders)/3` concurrent threads. In practice, you can use `max_logical_replication_workers = Rep-factor + 3 * task_num` (`task_num` is the number of parallel rebalance tasks).

To rebalance sharded tables in the `cluster0` cluster manually, run the command (where etcd1, etcd2, etcd3 are etcd cluster nodes):

```
                    $ shardmanctl --store-endpoints http://etcd1:2379,http://
etcd2:2379,http://etcd3:2379 rebalance
```

If the process ends with an error, then you need to call the `shardmanctl cleanup` command with the `--after-rebalance` option.

2.3.2. Manually Rebalancing the Data

There are times when you need to place partitions of sharded tables in a specific way across the cluster nodes. To solve this problem, Shardman supports the manual data rebalancing mode.

How it works:

1. Get a list of sharded tables using the `shardmanctl tables sharded list` command. As a result, we get an answer similar to the following:

```
$ shardmanctl shardmanctl tables sharded list

Sharded tables:

public.doc
public.resolution
public.users
```

2. Request information about the selected sharded tables. Example:

```
$ shardmanctl shardmanctl tables sharded info -t public.users

Table public.users

Partitions:

Partition    RgID     Shard                          Master
0            1        clover-1-shrn1                 shrn1:5432
1            2        clover-2-shrn2                 shrn2:5432
2            3        clover-3-shrn3                 shrn3:5432
3            1        clover-1-shrn1                 shrn1:5432
4            2        clover-2-shrn2                 shrn2:5432
5            3        clover-3-shrn3                 shrn3:5432
6            1        clover-1-shrn1                 shrn1:5432
7            2        clover-2-shrn2                 shrn2:5432
8            3        clover-3-shrn3                 shrn3:5432
9            1        clover-1-shrn1                 shrn1:5432
10           2        clover-2-shrn2                 shrn2:5432
11           3        clover-3-shrn3                 shrn3:5432
12           1        clover-1-shrn1                 shrn1:5432
13           2        clover-2-shrn2                 shrn2:5432
14           3        clover-3-shrn3                 shrn3:5432
15           1        clover-1-shrn1                 shrn1:5432
16           2        clover-2-shrn2                 shrn2:5432
17           3        clover-3-shrn3                 shrn3:5432
18           1        clover-1-shrn1                 shrn1:5432
19           2        clover-2-shrn2                 shrn2:5432
20           3        clover-3-shrn3                 shrn3:5432
21           1        clover-1-shrn1                 shrn1:5432
22           2        clover-2-shrn2                 shrn2:5432
23           3        clover-3-shrn3                 shrn3:5432
```

3. Move a partition to a new shard, as shown below:

```
$ shardmanctl --log-level debug tables sharded partmove -t public.users --partnum 1
 --shard clover-1-shrn1
```

```
2023-07-26T06:00:36.900Z        DEBUG   cmd/common.go:105       Waiting for metadata
 lock...
2023-07-26T06:00:36.936Z        DEBUG   rebalance/service.go:256        take
 extension lock
2023-07-26T06:00:36.938Z        DEBUG   broadcaster/worker.go:33        start
 broadcaster worker for repgroup id=3
2023-07-26T06:00:36.938Z        DEBUG   broadcaster/worker.go:33        start
 broadcaster worker for repgroup id=2
2023-07-26T06:00:36.938Z        DEBUG   broadcaster/worker.go:33        start
 broadcaster worker for repgroup id=1
2023-07-26T06:00:36.951Z        DEBUG   broadcaster/worker.go:51        repgroup 3
 connect established
2023-07-26T06:00:36.951Z        DEBUG   broadcaster/worker.go:51        repgroup 2
 connect established
2023-07-26T06:00:36.952Z        DEBUG   broadcaster/worker.go:51        repgroup 1
 connect established
2023-07-26T06:00:36.952Z        DEBUG   extension/lock.go:35    Waiting for
 extension lock...
2023-07-26T06:00:36.976Z        INFO    rebalance/service.go:276        Performing
 move partition...
2023-07-26T06:00:36.977Z        DEBUG   broadcaster/worker.go:33        start
 broadcaster worker for repgroup id=3
2023-07-26T06:00:36.978Z        DEBUG   broadcaster/worker.go:33        start
 broadcaster worker for repgroup id=2
2023-07-26T06:00:36.978Z        DEBUG   broadcaster/worker.go:33        start
 broadcaster worker for repgroup id=1
2023-07-26T06:00:36.987Z        DEBUG   broadcaster/worker.go:51        repgroup 1
 connect established
2023-07-26T06:00:36.989Z        DEBUG   broadcaster/worker.go:51        repgroup 2
 connect established
2023-07-26T06:00:36.992Z        DEBUG   broadcaster/worker.go:51        repgroup 3
 connect established
2023-07-26T06:00:36.992Z        DEBUG   rebalance/service.go:71 Performing cleanup
 after possible rebalance operation failure
2023-07-26T06:00:37.077Z        DEBUG   broadcaster/worker.go:75        finish
 broadcaster worker for repgroup id=3
2023-07-26T06:00:37.077Z        DEBUG   broadcaster/worker.go:75        finish
 broadcaster worker for repgroup id=1
2023-07-26T06:00:37.077Z        DEBUG   broadcaster/worker.go:75        finish
 broadcaster worker for repgroup id=2
2023-07-26T06:00:37.082Z        DEBUG   rebalance/service.go:422        Rebalance
 will run 1 tasks
2023-07-26T06:00:37.095Z        DEBUG   rebalance/service.go:452        Guessing
 that rebalance() can use 3 workers
2023-07-26T06:00:37.096Z        DEBUG   rebalance/job.go:352    state: Idle
 {"worker_id": 1, "table": "users", "partition num": 1, "source rgid": 2, "dest
 rgid": 1, "kind": "move"}
2023-07-26T06:00:37.111Z        DEBUG   rebalance/job.go:352    state:
 ConnsEstablished {"worker_id": 1, "table": "users", "partition num": 1, "source
 rgid": 2, "dest rgid": 1, "kind": "move"}
2023-07-26T06:00:37.171Z        DEBUG   rebalance/job.go:352    state: WaitInitCopy
     {"worker_id": 1, "table": "users", "partition num": 1, "source rgid": 2, "dest
 rgid": 1, "kind": "move"}
2023-07-26T06:00:38.073Z        DEBUG   rebalance/job.go:347    current state
 {"worker_id": 1, "table": "users", "partition num": 1, "source rgid": 2, "dest
 rgid": 1, "kind": "move", "state": "WaitInitialCatchup"}
```

```
2023-07-26T06:00:38.073Z        DEBUG   rebalance/job.go:352    state:
 WaitInitialCatchup       {"worker_id": 1, "table": "users", "partition num": 1,
 "source rgid": 2, "dest rgid": 1, "kind": "move"}
2023-07-26T06:00:38.084Z        DEBUG   rebalance/job.go:347    current state
 {"worker_id": 1, "table": "users", "partition num": 1, "source rgid": 2, "dest
 rgid": 1, "kind": "move", "state": "WaitFullSync"}
2023-07-26T06:00:38.084Z        DEBUG   rebalance/job.go:352    state: WaitFullSync
     {"worker_id": 1, "table": "users", "partition num": 1, "source rgid": 2, "dest
 rgid": 1, "kind": "move"}
2023-07-26T06:00:38.108Z        DEBUG   rebalance/job.go:347    current state
 {"worker_id": 1, "table": "users", "partition num": 1, "source rgid": 2, "dest
 rgid": 1, "kind": "move", "state": "Committing"}
2023-07-26T06:00:38.108Z        DEBUG   rebalance/job.go:352    state: Committing
     {"worker_id": 1, "table": "users", "partition num": 1, "source rgid": 2, "dest
 rgid": 1, "kind": "move"}
2023-07-26T06:00:38.254Z        DEBUG   rebalance/job.go:352    state: Complete
 {"worker_id": 1, "table": "users", "partition num": 1, "source rgid": 2, "dest
 rgid": 1, "kind": "move"}
2023-07-26T06:00:38.258Z        DEBUG   rebalance/service.go:583        Produce and
 process tasks on destination replication groups...
2023-07-26T06:00:38.258Z        DEBUG   rebalance/service.go:594        Produce and
 process tasks on source replication groups...
2023-07-26T06:00:38.258Z        DEBUG   rebalance/service.go:606        wait all
 tasks finish
2023-07-26T06:00:38.258Z        DEBUG   rebalance/service.go:531        Analyzing
 table public.users in rg 1    {"table": "public.users", "rgid": 1, "action":
 "analyze"}
2023-07-26T06:00:38.573Z        DEBUG   rebalance/service.go:531        Analyzing
 table public.users in rg 2    {"table": "public.users", "rgid": 2, "action":
 "analyze"}
2023-07-26T06:00:38.833Z        DEBUG   broadcaster/worker.go:75        finish
 broadcaster worker for repgroup id=1
2023-07-26T06:00:38.833Z        DEBUG   broadcaster/worker.go:75        finish
 broadcaster worker for repgroup id=2
2023-07-26T06:00:38.833Z        DEBUG   broadcaster/worker.go:75        finish
 broadcaster worker for repgroup id=3
```

In this example, partition number `1` of the `public.users` table will be moved to the `clover-1-shrn1` shard.

After manually moving a partition of a sharded table and for all tables collocated with it, automatic data rebalancing for these tables will be disabled.

To get the list of tables with disabled automatic rebalancing, call the `shardmanctl tables sharded norebalance` command. Example:

```
$ shardmanctl tables sharded norebalance

public.users
```

To enable automatic data rebalancing for a selected sharded table, call the `shardmanctl tables sharded rebalance` command, as shown in the example below:

```
$ shardmanctl tables sharded rebalance -t public.users

2023-07-26T07:07:00.657Z        DEBUG   cmd/common.go:105       Waiting for metadata
 lock...
```

```
2023-07-26T07:07:00.687Z        DEBUG   broadcaster/worker.go:33        start
 broadcaster worker for repgroup id=1
2023-07-26T07:07:00.687Z        DEBUG   broadcaster/worker.go:33        start
 broadcaster worker for repgroup id=2
2023-07-26T07:07:00.687Z        DEBUG   broadcaster/worker.go:33        start
 broadcaster worker for repgroup id=3
2023-07-26T07:07:00.697Z        DEBUG   broadcaster/worker.go:51        repgroup 1
 connect established
2023-07-26T07:07:00.698Z        DEBUG   broadcaster/worker.go:51        repgroup 2
 connect established
2023-07-26T07:07:00.698Z        DEBUG   broadcaster/worker.go:51        repgroup 3
 connect established
2023-07-26T07:07:00.698Z        DEBUG   extension/lock.go:35    Waiting for extension
 lock...
2023-07-26T07:07:00.719Z        DEBUG   rebalance/service.go:381        Planned moving
 pnum 21 for table users from rg 1 to rg 2
2023-07-26T07:07:00.719Z        INFO    rebalance/service.go:244        Performing
 rebalance...
2023-07-26T07:07:00.720Z        DEBUG   broadcaster/worker.go:33        start
 broadcaster worker for repgroup id=1
2023-07-26T07:07:00.720Z        DEBUG   broadcaster/worker.go:33        start
 broadcaster worker for repgroup id=2
2023-07-26T07:07:00.720Z        DEBUG   broadcaster/worker.go:33        start
 broadcaster worker for repgroup id=3
2023-07-26T07:07:00.732Z        DEBUG   broadcaster/worker.go:51        repgroup 3
 connect established
2023-07-26T07:07:00.732Z        DEBUG   broadcaster/worker.go:51        repgroup 1
 connect established
2023-07-26T07:07:00.734Z        DEBUG   broadcaster/worker.go:51        repgroup 2
 connect established
2023-07-26T07:07:00.734Z        DEBUG   rebalance/service.go:71 Performing cleanup
 after possible rebalance operation failure
2023-07-26T07:07:00.791Z        DEBUG   broadcaster/worker.go:75        finish
 broadcaster worker for repgroup id=1
2023-07-26T07:07:00.791Z        DEBUG   broadcaster/worker.go:75        finish
 broadcaster worker for repgroup id=2
2023-07-26T07:07:00.791Z        DEBUG   broadcaster/worker.go:75        finish
 broadcaster worker for repgroup id=3
2023-07-26T07:07:00.795Z        DEBUG   rebalance/service.go:422        Rebalance will
 run 1 tasks
2023-07-26T07:07:00.809Z        DEBUG   rebalance/service.go:452        Guessing that
 rebalance() can use 3 workers
2023-07-26T07:07:00.809Z        DEBUG   rebalance/job.go:352    state: Idle
 {"worker_id": 1, "table": "users", "partition num": 21, "source rgid": 1, "dest rgid":
 2, "kind": "move"}
2023-07-26T07:07:00.823Z        DEBUG   rebalance/job.go:352    state: ConnsEstablished
 {"worker_id": 1, "table": "users", "partition num": 21, "source rgid": 1, "dest rgid":
 2, "kind": "move"}
2023-07-26T07:07:00.880Z        DEBUG   rebalance/job.go:352    state: WaitInitCopy
 {"worker_id": 1, "table": "users", "partition num": 21, "source rgid": 1, "dest rgid":
 2, "kind": "move"}
2023-07-26T07:07:01.886Z        DEBUG   rebalance/job.go:347    current state
 {"worker_id": 1, "table": "users", "partition num": 21, "source rgid": 1, "dest rgid":
 2, "kind": "move", "state": "WaitInitialCatchup"}
2023-07-26T07:07:01.886Z        DEBUG   rebalance/job.go:352    state:
 WaitInitialCatchup      {"worker_id": 1, "table": "users", "partition num": 21,
 "source rgid": 1, "dest rgid": 2, "kind": "move"}
```

```
2023-07-26T07:07:01.904Z        DEBUG   rebalance/job.go:347    current state
 {"worker_id": 1, "table": "users", "partition num": 21, "source rgid": 1, "dest rgid":
 2, "kind": "move", "state": "WaitFullSync"}
2023-07-26T07:07:01.905Z        DEBUG   rebalance/job.go:352    state: WaitFullSync
 {"worker_id": 1, "table": "users", "partition num": 21, "source rgid": 1, "dest rgid":
 2, "kind": "move"}
2023-07-26T07:07:01.932Z        DEBUG   rebalance/job.go:347    current state
 {"worker_id": 1, "table": "users", "partition num": 21, "source rgid": 1, "dest rgid":
 2, "kind": "move", "state": "Committing"}
2023-07-26T07:07:01.932Z        DEBUG   rebalance/job.go:352    state: Committing
 {"worker_id": 1, "table": "users", "partition num": 21, "source rgid": 1, "dest rgid":
 2, "kind": "move"}
2023-07-26T07:07:02.057Z        DEBUG   rebalance/job.go:352    state: Complete
 {"worker_id": 1, "table": "users", "partition num": 21, "source rgid": 1, "dest rgid":
 2, "kind": "move"}
2023-07-26T07:07:02.060Z        DEBUG   rebalance/service.go:583        Produce and
 process tasks on destination replication groups...
2023-07-26T07:07:02.060Z        DEBUG   rebalance/service.go:594        Produce and
 process tasks on source replication groups...
2023-07-26T07:07:02.060Z        DEBUG   rebalance/service.go:531        Analyzing table
 public.users in rg 2    {"table": "public.users", "rgid": 2, "action": "analyze"}
2023-07-26T07:07:02.060Z        DEBUG   rebalance/service.go:606        wait all tasks
 finish
2023-07-26T07:07:02.321Z        DEBUG   rebalance/service.go:531        Analyzing table
 public.users in rg 1    {"table": "public.users", "rgid": 1, "action": "analyze"}
2023-07-26T07:07:02.587Z        DEBUG   broadcaster/worker.go:75        finish
 broadcaster worker for repgroup id=3
2023-07-26T07:07:02.587Z        DEBUG   broadcaster/worker.go:75        finish
 broadcaster worker for repgroup id=2
2023-07-26T07:07:02.587Z        DEBUG   broadcaster/worker.go:75        finish
 broadcaster worker for repgroup id=1
```

To enable automatic data rebalancing for all sharded tables, run the `shardmanctl rebalance` command with the `--force` option.

```
$ shardmanctl rebalance --force
```

2.4. Analyzing and Vacuuming

Shardman databases require periodic maintenance, known as vacuuming. For many installations, it is sufficient to let vacuuming be performed by the autovacuum daemon. As in PostgreSQL installation, autovacuum daemon will automatically issue ANALYZE commands whenever the content of a table has changed sufficiently. When ANALYZE is run by the autovacuum daemon or manually on the whole database, statistics from foreign partitions is transferred from remote nodes.

Rebalance process can move or copy data between cluster nodes. After this operation, all transferred objects are automatically analyzed. As usual, local statistics is gathered, and remote statistics is fetched from foreign servers.

> **Note**
>
> Database-wide ANALYZE relies on statistics being available on remote shards. But statistics on remote shards may be missing, and it is not enough to just broadcast ANALYZE for cluster-wide update of statistics. Instead, shardman.global_analyze() function can be used. It performs gathering of statistics for sharded and global tables.
>
> Database-wide VACUUM command can be broadcast to perform cluster-wide vacuuming. It can be done when the `shardman.broadcast_ddl` configuration parameter is on.

> **Note**
>
> When ANALYZE is run on a global table, only statistics on corresponding local table is updated. When ANALYZE is run on a sharded table, statistics on local partitions is updated, statistics for foreign partitions is transferred from remote nodes, if remote nodes have it. When ANALYZE is run on a foreign table directly and remote node doesn't have any statistics for the corresponding local table, local table is analyzed remotely. Then statistics is transferred from the remote node.
>
> When VACUUM is run on a sharded or global table, the statement is broadcast. For a sharded table, it is efficiently run on all table partitions.

2.5. Access Management

A Shardman cluster emulates a usual PostgreSQL security model, which, however, has features inherent to a distributed DBMS. This section describes these features and aims to give you an idea of access management in a Shardman cluster.

2.5.1. Cluster Initialization Settings Related to Access Management

When a Shardman cluster is initialized, security-related settings are taken from the initialization file. You can change them later, but do this with care and remember that in most cases, the change will require a DBMS restart.

A Shardman cluster has two special users: *administrative* and *replication*. stolon and Shardman manage controlled DBMS instances with administrative users. stolon needs replication users for replications between controlled DBMS instances.

Security-related settings from the initialization file specify:

- Authentication methods for administrative and replication users — `PgSuAuthMethod`, `PgReplAuthMethod`
- Usernames for administrative and replication users — `PgSuUsername`, `PgReplUsername`
- Passwords for administrative and replication users — `PgSuPassword`, `PgReplPassword`
- `pg_hba.conf` rules used by DBMS instances — `StolonSpec.pgHBA`

See sdmspec.json for detailed descriptions of these settings.

To change security-related user settings, perform these steps:

1. Check that the user that you want to specify in `PgReplUsername` /`PgSuUsername` exists with `REPLICATION`/ `SUPERUSER` privileges on all replication groups in the cluster and his password matches the new `PgReplPassword`/`PgSuPassword` setting.
2. If this is true, create dump of the `shardman/cluster0/data/cluster` etcd key (here and further the name of the Shardman cluster is assumed to be `cluster0`). For example:

   ```
   $ etcdctl --endpoints etcdserver:2379 get --print-value-only  shardman/cluster0/data/cluster |jq . > clusterdata.json
   ```

   This example creates the dump of the `data/cluster` key for the Shardman cluster with the `cluster0` name from the etcd server `etcdserver` listening on port `2379`, formats the dump with jq and saves to the `clusterdata.json` file.
3. Edit the dump as necessary and store it back in etcd:

   ```
   $ cat clusterdata.json | etcdctl --endpoints etcdserver:2379 put shardman/cluster0/data/cluster
   ```

Modifying these settings will lead to a DBMS restart.

Unlike the above settings, the `StolonSpec.pgHBA` setting can be changed online. To do this, perform these steps:

1. Extract the `StolonSpec` definition from `shardman/cluster0/data/cluster`, save to some file, modify as necessary and update cluster settings with the `shardmanctl config update` command:

```
$ etcdctl --endpoints etcdserver:2379 get --print-value-only  shardman/cluster0/
data/cluster | jq .Spec.StolonSpec . > stolonspec.json
```

2. Edit `stolonspec.json` and replace the `StolonSpec.pgHBA` definition with the appropriate one, for example:

```
"pgHBA": [
        "host all postgres 0.0.0.0/0 scram-sha-256",
        "host replication postgres 0.0.0.0/0 scram-sha-256",
        "host replication postgres ::0/0 scram-sha-256",
        "host all someuser 0.0.0.0/0 scram-sha-256"
      ],
```

3. Apply the edited `stolonspec.json` file:

```
$ shardmanctl --store-endpoints etcdserver:2379 --cluster-name cluster0 config
 update -f stolonspec.json
```

2.5.2. Managing Users and Roles

Users and roles in a Shardman cluster are usual PostgreSQL users and roles. You can manage them separately on each server or globally, using broadcast DDL. Shardman also uses concepts of *global users* and *global roles*. And only the *global users* (or roles) can create and own other Shardman cluster-wide objects, such as sharded or global tables. Operations on such users are always performed on all replication groups simultaneously. For example, when you include a global role in some other role or drop it, this operation will be performed on all replication groups.

You can create a global user with a `CREATE USER ... IN ROLE global` statement, for example:

```
CREATE USER someuser ENCRYPTED PASSWORD 'somepass' IN ROLE global;
```

When a global user is created, Shardman automatically creates user mappings on all replication groups and grants this user with access to all foreign servers corresponding to existing replication groups. Therefore, when you create a global user, you need to specify either a cleartext password, so that it can be saved in a user mapping, or no password at all. A passwordless global user or role is unable to access foreign servers, but you can use such a role to accumulate some permissions and grant it to different users. You can also set a password for a passwordless global user later.

Global users can be created only by user with `CREATEROLE` permission on all cluster nodes.

`ALTER` and `DROP` statements for global users are broadcasted to all replication groups. When a role is granted to a global user, this operation is also broadcasted. Renaming a global user is not supported since this invalidates md5/scram-sha-256 passwords stored in user mappings.

The list of global users is stored in the `shardman.users` table.

The role specified in `PgSuUsername` (usually, `postgres`) is also created as global user during cluster initialization. However, the role specified in `PgReplUsername` is created as local user on each replication group.

The role `global` is reserved and cannot be used directly in a Shardman cluster. Note that 'global' is not a really defined role but just a reserved word.

2.5.3. Managing Permissions on Sharded Tables

In Shardman, a sharded table is basically a partitioned table where partitions are either local shards or foreign tables referencing shards in other replication groups.

Permissions granted on a sharded table are broadcasted to all replication groups and to all partitions of the table.

When a new replication group is added to a cluster, shardmanctl copies the schema from a random existing replication group to the new one. It also creates a foreign server for the new replication group on all existing replication groups and recreates foreign servers on new replication groups. Permissions for the created foreign servers and user mappings are copied from a random foreign server in an existing replication group. In the new replication group, for each partition of the sharded table shardmanctl creates a foreign table referencing the existing shard and replaces the partition with this foreign table. Later some of these foreign tables can be replaced by real tables. This happens during the `shardmanctl nodes add` rebalance stage when rebalance is enabled. Data

for these partitions is transferred from existing nodes using logical replication. When shardmanctl creates tables (or foreign tables), it copies permissions from the parent table. The parent table must already have correct permissions since they were copied from an existing replication group.

2.5.3.1. Examples

These examples assume administrator privileges.

If you want to create a sharded table and a global user, as well as grant him read-only access to the table, you can use the following statements:

```
CREATE USER someuser ENCRYPTED PASSWORD 'somepass' IN ROLE global;
CREATE TABLE pgbench_branches (
       bid integer NOT NULL PRIMARY KEY,
       bbalance integer,
       filler character(88)
)
WITH (distributed_by = 'bid', num_parts = 8);
GRANT SELECT ON pgbench_branches TO someuser;
```

To allow `someuser` to access a Shardman cluster, you should also provide proper settings in `pg_hba.conf` (as this is done earlier).

Now assume that a new clover is added to the cluster with the `shardmanctl nodes add` command, like this:

```
$ shardmanctl --store-endpoints http://etcdserver:2379 --cluster-name cluster0 nodes
 add -n newnode1,newnode2
```

In this example, some shards of the `pgbench_branches` table are transferred to new replication groups and `someuser` is granted the `SELECT` privilege on this table. Later you can drop `someuser` from all replication groups in the cluster in one command:

```
DROP USER someuser;
```

2.6. Backup and Recovery

This section describes basics of backup and recovery in Shardman.

You can use the `backup` command of the shardmanctl tool to perform a full binary consistent backup of a Shardman cluster to a shared directory or local directory (if `--use-ssh` is specified) and the `recover` command to perform a recovery from this backup.

Also you can use the `probackup backup` command of the shardmanctl tool to perform a full binary consistent backup of a Shardman cluster to the backup repository on the local host or S3-compatible object storage and the `probackup restore` command to perform a recovery from any backup from the repository.

The PostgreSQL pg_probackup utility for creating consistent full and incremental backups was integrated into shardman-utils. shardman-utils uses the pg_probackup approach to store backups in a pre-created repository. In addition, the pg_probackup commands `archive-get` and `archive-push` are used to deliver WAL logs into the backup repository. Backup and restore modes use a passwordless ssh connection between the cluster nodes and the backup node.

Shardman cluster configuration parameter enable_csn_snapshot must be set to `on`. This parameter is necessary for the cluster backup to be consistent. If this option is disabled, a consistent backup is not possible.

For consistent visibility of distributed transactions, the technique of global snapshots based on physical clocks is used. Similarly, it is possible to get a consistent snapshot for backups, only the time corresponding to the global snapshot must be mapped to the set of LSNs for each node. Such a set of consistent LSNs in a cluster is called a syncpoint. By getting the syncpoint and taking the LSN for each node in the cluster from it, we can make a backup of each node, which must necessarily contain that LSN. We can also recover to this LSN using the point in time recovery (PITR) mechanism.

The `backup` and `probackup` commands use different mechanisms to create backups. The `backup` command is based on the standard utilities pg_basebackup and pg_receivewal. The `probackup` command uses the pg_probackup utility and its options to create a cluster backup. In any case of using `backup` or `probackup` commands for restoration, the node names, defined by hostname or IP-address, must correspond to those that were in place at the time of the backup.

2.6.1. Cluster Backup with pg_basebackup

This section describes basics of backup and recovery in Shardman with the `basebackup` command.

2.6.1.1. Requirements

To backup and restore a Shardman cluster via the `basebackup` command, the following requirements must be met:

- Shardman cluster configuration parameter enable_csn_snapshot must be `on`. This parameter is necessary for the cluster backup to be consistent. If this parameter is disabled, a consistent backup is not possible.
- On each Shardman cluster node, Shardman utilities must be installed into `/opt/pgpro/sdm-14/bin`.
- On each Shardman cluster node, pg_basebackup must be installed into `/opt/pgpro/sdm-14/bin`.
- On each Shardman cluster node, `postgres` Linux user and group must be created.
- Passwordless SSH connection between Shardman cluster nodes for the `postgres` Linux user must be configured.
- If the `--use-ssh` flag isn't specified, all Shardman cluster nodes must be connected to a shared network storage and backup folder must be created on that shared network storage.
- If the `--use-ssh` flag is specified, the backup directory can be created on the local storage on the node where `recover` will be called.
- Access for the `postgres` Linux user to the backup folder must be granted.
- shardmanctl utility must be run as `postgres` Linux user.

2.6.1.2. basebackup Backup Process

shardmanctl conducts a backup task in several steps. The tool:

1. Takes necessary locks in etcd to prevent concurrent cluster-wide operations.
2. Connects to a random replication group and locks Shardman metadata tables to prevent modification of foreign servers during the backup.
3. Creates replication slots on each replication group to ensure that WAL records are not lost.
4. Dumps Shardman metadata stored in etcd to a JSON file in the backup directory.
5. To get backups from each replication group, concurrently runs pg_basebackup using replication slots created.
6. Creates the syncpoint and uses pg_receivewal to fetch WAL logs generated after finishing each basebackup until LSNs extracted from syncpoint are reached.
7. Fixes partial WAL files generated by pg_receivewal and creates the backup description file.

2.6.2. Cluster Recovery from a Backup Using pg_basebackup

You can restore a backup on the same or compatible cluster. By *compatible* clusters, those that use the same Shardman version and have the same number of replication groups are meant.

shardmanctl can perform either full restore, metadata-only or schema-only restore. Metadata-only restore is useful if issues are encountered with the etcd instance, but DBMS data is not corrupted.

During metadata-only restore, shardmanctl restores etcd data from the dump created during the backup.

> **Important**
>
> Restoring metadata to an incompatible cluster can lead to catastrophic consequences, including data loss, since the metadata state can differ from the actual configuration layout. Do not perform metadata-only restore if there were cluster reconfigurations after the backup, such as addition or deletion of nodes, even if the same nodes were added back again.

Schema-only recovery restore only schema information without data. It can be useful if the scale of the data is large and the schema is needed for testing or checking.

During a full restore, shardmanctl checks whether the number of replication groups in the target cluster matches the number of replication groups in the backup. This means that you cannot restore on an empty cluster, but need to add as many replication groups as necessary for the total number of them to match that of the cluster from which the backup was taken.

`shardmanctl probackup restore` can restore a working or partially working cluster from a backup that was created on a working or partially working cluster.

Also you can perform restoring only on a single shard using `--shard` parameter.

shardmanctl conducts full restore in several steps. The tool:

1. Takes the necessary locks in etcd to prevent concurrent cluster-wide operations and tries to assign replication groups in the backup to existing replication groups. If it cannot do this (for example, due to cluster incompatibility), the recovery fails.
2. Restores part of the etcd metadata: the cluster specification and parts of replication group definitions.
3. When the correct metadata is in place, runs stolon `init` in PITR initialization mode with `RecoveryTargetName` set to the value of the syncpoint LSN from the backup info file. `DataRestoreCommand` and `RestoreCommand` are also taken from the backup info file.
4. Waits for each replication group to recover.

2.6.3. Cluster Backup with pg_probackup

This section describes basics of backup and recovery in Shardman with the `probackup` command.

You can use the `probackup backup` command of the shardmanctl tool to perform binary backups of a Shardman cluster into the backup repository on the local (backup) host and the `probackup restore` command to perform a recovery from the selected backup. Full and partial (delta) backups are supported.

2.6.3.1. Requirements

To backup and restore a Shardman cluster via the `probackup` command, the following requirements must be met:

- Shardman cluster configuration parameter enable_csn_snapshot must be `on`. This parameter is necessary for the cluster backup to be consistent. If this parameter is disabled, a consistent backup is not possible.
- On the backup host, Shardman utilities must be installed into `/opt/pgpro/sdm-14/bin`.
- On the backup host and on each cluster node, pg_probackup must be installed into `/opt/pgpro/sdm-14/bin`.
- On the backup host, `postgres` Linux user and group must be created.
- Passwordless SSH connection between the backup host and each Shardman cluster node for the `postgres` Linux user must be configured. To do this, on each node:
  - The `postgres` user must create the `.ssh` subdirectory in the `/var/lib/postgresql` directory and place there the keys required for the passwordless SSH connection.
  - To perform a backup/restore in a pretty large number of threads, such as 50 (`-j`=50, see the section called “ `backup` ” for details), `MaxSessions` and `MaxStartups` must be set to 100 for the backup host in the `/etc/ssh/sshd_config` file.

> **Note**
>
> Setting the number of threads (`-j` option) to a value greater than 10 for `shardmanctl probackup` may result in the actual number of SSH connections exceeding the maximum allowed number of simultaneous SSH connections on the backup host and consequently lead to an “ERROR: Agent error: kex_exchange_identification: Connection closed by remote host” error. To correct the error, either reduce the number of `probackup` threads or adjust the value of `MaxStartups` configuration parameter of the backup host. If SSH is set up as a xinetd service on the backup host, adjust the value of the xinetd `per_source` configuration parameter rather than `MaxStartups`.

You can disable SSH for data copying by setting the `--storage-type` option to the `mount` or `S3` value (but SSH will be required to execute remote commands). Also this value will be automatically used in the restore process.

- A backup folder or bucket in the S3-compatible object storage must be created.
- Access for the `postgres` Linux user to the backup folder must be granted.
- shardmanctl utility must be run as `postgres` Linux user.
- `init` subcommand for the backup repository initialization must be successfully executed on the backup host.
- `archive-command add` subcommand for enabling `archive_command` for each replication group to stream WALs into the initialized repository must be successfully executed on the backup host.

2.6.3.2. pg_probackup Backup Process

shardmanctl conducts a backup task in several steps. The tool:

1. Takes necessary locks in etcd to prevent concurrent cluster-wide operations.
2. Connects to a random replication group and locks Shardman metadata tables to prevent modification of foreign servers during the backup.
3. Dumps Shardman metadata, stored in etcd, to a JSON file in the backup directory or bucket in the S3-compatible object storage.
4. To get backups from each replication group, concurrently runs pg_probackup using the configured `archive_command`.
5. Creates the syncpoint and gets LSNs for each replication group from the syncpoint data structure. Then uses the `pg_probackup archive-push` command to push WAL logs generated after finishing backup and the WAL file where syncpoint LSNs are present for each replication group.

2.6.4. Cluster Restore from a Backup with pg_probackup

You can restore a backup on the same or compatible cluster. By *compatible* clusters, those that use the same Shardman version and have the same number of replication groups are meant here.

Also, you can restore other clusters from the same backup if these clusters have the same topology.

shardmanctl can perform either full restore, metadata-only or schema-only restore. Metadata-only restore is useful if issues are encountered with the etcd instance, but DBMS data is not corrupted.

During metadata-only restore, shardmanctl restores etcd data from the dump created during the backup.

> **Important**
>
> Restoring metadata to an incompatible cluster can lead to catastrophic consequences, including data loss, since the metadata state can differ from the actual configuration layout. Do not perform metadata-only restore if there were cluster reconfigurations after the backup, such as addition or deletion of nodes, even if the same nodes were added back again.

Schema-only recovery restore only schema information without data. It can be useful if the scale of the data is large and the schema is needed for testing or checking.

During a full restore, shardmanctl checks whether the number of replication groups in the target cluster matches the number of replication groups in the backup. This means that you cannot restore on an empty cluster, but need to add as many replication groups as necessary for the total number of them to match that of the cluster from which the backup was taken.

Also you can perform restoring only on the single shard using `--shard` parameter.

Also you can perform Point-in-Time Recovery using `--recovery-target-time` parameter. In this case Shardman finds closest syncpoint to specified timestamp and suggests to restore on found LSN. You can also specify a `--wal-limit` option to limit the number of WAL segments to be processed.

shardmanctl conducts full restore in several steps. The tool:

1. Takes the necessary locks in etcd to prevent concurrent cluster-wide operations and tries to assign replication groups in the backup to existing replication groups. If it cannot do this (for example, due to cluster incompatibility), the recovery fails.
2. Restores part of the etcd metadata: the cluster specification and parts of replication group definitions.
3. When the correct metadata is in place, runs stolon `init` in PITR initialization mode with `RecoveryTargetName` set to the value of the syncpoint LSN from the backup info file. `DataRestoreCommand` and `RestoreCommand` are also taken from the backup info file. These commands are generated automatically during the backup phase, it is not recommended to make any corrections to the file containing the Shardman cluster backup description. When restoring a cluster for each replication group, the WAL files containing the final LSN to restore will be requested automatically from the backup repository from the remote backup node via the pg_probackup `archive-get` command.
4. Waits for each replication group to recover.
5. Finally we need to enable `archive_command` back.

When performing a sequential restoration in PostgreSQL, be cautious of potential timeline conflicts within WAL (Write-Ahead Logging) segments. This issue commonly arises when restoring a database from a backup that was created at a certain point in time. If the database continues to operate and generate WAL segments after this backup, these new WAL segments are associated with a different timeline. During restoration, if the system tries to replay WAL segments from a different timeline - one that diverged from the point of backup - it can lead to inconsistencies and conflicts. Additionally, after completing a restoration in PostgreSQL, it is strongly advised not to restore the database onto the same timeline or onto any timeline that precedes the one from which the backup was made.

2.6.5. Merging Backups with pg_probackup

The more incremental backups are created, the bigger the total size of the backup catalog grows. To save the disk space, it is possible to merge the incremental backups to their parent full backup by running the merge command, specifying the backup ID of the most recent incremental backup to merge:

```
$ shardmanctl --store-endpoints http://etcd1:2379,http://etcd2:2379,http://etcd3:2379
 probackup merge --backup-path backup_dir --backup-id backup_id
```

This command merges the backups that belong to a common incremental backup chain. If a full backup is specified, it is merged with its first incremental backup. If an incremental backup is specified, it is merged to its parent full backup, along with all the incremental backups between them. Once the merge is complete, the full backup covers all the merged data, and the incremental backups are removed as redundant. Thus, the merge operation virtually equals to removing all the outdated backups from a full backup, but a lot faster, especially for the large data volumes. It also saves I/O and network traffic when using pg_probackup in the remote mode.

Before merging, pg_probackup validates all the affected backups to ensure that they are valid. The current backup status can be seen by running the `show` command:

```
$ shardmanctl --store-endpoints http://etcd1:2379,http://etcd2:2379,http://etcd3:2379
 probackup show --backup-path backup_dir
```

For more information, see reference.

2.6.6. Deleting Backups with pg_probackup

To delete a backup that is no longer needed, run the following command:

```
$ shardmanctl --store-endpoints http://etcd1:2379,http://etcd2:2379,http://etcd3:2379
 probackup delete --backup-path backup_dir --backup-id backup_id
```

This command deletes a backup with a specified `backup_id`, along with all the incremental backups that descend from this `backup_id`, if any. It allows to delete some of the recent incremental backups, without affecting the underlying full backup and other incremental backups that follow it.

To delete the obsolete WAL files that are not needed for recovery, use the `--delete-wal` flag:

```
$ shardmanctl --store-endpoints http://etcd1:2379,http://etcd2:2379,http://etcd3:2379
 probackup delete --backup-path backup_dir --backup-id backup_id --delete-wal
```

For more information, see reference.

2.7. Configuring Secure Communications with etcd

This section describes how to configure secure communications between the etcd store and Shardman services and tools.

etcd is a critical component for a Shardman cluster. If an intruder gets access to the etcd store, it gains full control over the whole cluster, including DBMS access with DBA privileges. To protect your cluster, it is recommended that you configure TLS authentication between etcd daemons and Shardman services.

To this end, you can use HTTPS transport with certificates signed by your local certification authority (CA) to encrypt traffic between the etcd cluster and Shardman services and restrict etcd access. To do this, perform the steps described in the next sections.

2.7.1. Generating SSL Certificates

To generate SSL certificates, perform the following steps:

1. If the CA does not exist, generate a self-signed root certificate. Generate all certificates on one trusted host. Here certificates that expire in 10000 days are generated (you can choose a more suitable interval):

   ```
   # openssl genrsa -out rootCA.key 4096
   # openssl req -x509 -new -key rootCA.key -days 10000 -out rootCA.crt
   ```

2. Prepare the following openssl configuration file for each etcd host:

   ```
   [ req ]
   default_bits       = 4096
   distinguished_name = req_distinguished_name
   req_extensions     = req_ext
   [ req_distinguished_name ]
   countryName                 = Country Name (2 letter code)
   stateOrProvinceName         = State or Province Name (full name)
   localityName               = Locality Name (eg, city)
   organizationName           = Organization Name (eg, company)
   commonName                 = Common Name (e.g. server FQDN or YOUR name)
   [ req_ext ]
   subjectAltName = @alt_names
   [alt_names]
   DNS.1   = n1
   IP.1   = 192.168.1.1
   IP.2   = 127.0.0.1
   ```

   Under `[alt_names]`, specify alternative subject names for the etcd host. These names must include the etcd server hostname, IP address and local IP. Including the local IP is convenient rather than required.

   Save the file. For example, the names of configuration files for nodes `n1` — `n3` can be `n1.san.conf` — `n3.san.conf`.

3. Using the configuration files prepared, generate private keys and certificate requests for etcd hosts:

   ```
   # openssl genrsa -out n1.etcd.key 4096
   # openssl req -config n1.san.conf -new -key n1.etcd.key -out n1.etcd.csr -subj "/
   C=RU/ST=Moscow Region/L=Moscow/O=Test/CN=n1"
   ```

   Here " `/C=RU/ST=Moscow Region/L=Moscow/O=Test/CN=n1` " means that the certificate request is generated with the country name `RU`, state `Moscow Region`, locality `Moscow`, organization `Test` and common name `n1`. The common name must match the DNS name of your etcd server.

4. Sign the certification request:

   ```
   # openssl x509 -extfile n1.san.conf -extensions req_ext -req -in n1.etcd.csr  -CA
    rootCA.crt -CAkey rootCA.key -CAcreateserial -out n1.etcd.crt -days 10000
   ```

5. Check the certificates to ensure they contain correct `X509v3 Subject Alternative Name` fields. The fields must contain the list of DNS names and IP addresses that you added to the openssl configuration file:

```
# openssl x509 -in n1.etcd.crt -noout -text
```

6. Generate client certificates for Shardman services and client tools. These certificates do not need to contain the `subjectAltName` header, and `CN` is not important in the example below. It generates one common certificate-key pair for services and one — for tools:

```
# openssl x509 genrsa -out shardman_services.key 4096
# openssl req -new -key shardman_services.key -out shardman_services.csr -subj "/
C=RU/ST=Moscow Region/L=Moscow/O=Test/CN=shardman_services"
# openssl x509 -req -in shardman_services.csr  -CA rootCA.crt -CAkey rootCA.key -
CAcreateserial -out shardman_services.crt -days 10000
# openssl x509 genrsa -out shardman_tools.key 4096
# openssl req -new -key shardman_tools.key -out shardman_tools.csr -subj "/C=RU/
ST=Moscow Region/L=Moscow/O=Test/CN=shardman_tools"
# openssl x509 -req -in shardman_tools.csr  -CA rootCA.crt -CAkey rootCA.key -
CAcreateserial -out shardman_tools.crt -days 10000
```

2.7.2. Configuring etcd and shardmand Services

Now configure services (etcd and shardmand) to use the generated certificates. To do this, perform the following steps:

1. On each etcd node, put `rootCA.crt`, `nX.etcd.crt` and `nX.etcd.key` in the location accessible to the etcd daemon (for example, create `/etc/etcd` directory and put files there). Ensure that the `nX.etcd.key` file is only accessible to the etcd daemon user.

2. Specify the following configuration for etcd daemons in `/etc/default/etcd`:

```
# unqualified first name
ETCD_NAME=n1
# where we actually listen for peers
ETCD_LISTEN_PEER_URLS=https://0.0.0.0:2380
# where we actually listen for clients
ETCD_LISTEN_CLIENT_URLS=https://0.0.0.0:2379
# advertise where this machine is listening for clients
ETCD_ADVERTISE_CLIENT_URLS=https://n1:2379

# --initial flags are used during bootstrapping and ignored afterwards, so it is
# ok to specify them always
# advertise where this machine is listening for peer
ETCD_INITIAL_ADVERTISE_PEER_URLS=https://n1:2380
ETCD_INITIAL_CLUSTER_TOKEN=etcd-cluster
# ansible_nodename is fqdn
ETCD_INITIAL_CLUSTER=n1=https://n1:2380,n2=https://n2:2380,n3=https://n3:2380
ETCD_INITIAL_CLUSTER_STATE=new

ETCD_DATA_DIR=/var/lib/etcd/default/member
ETCD_AUTO_COMPACTION_RETENTION=1

ETCD_KEY_FILE=/etc/etcd/n1.etcd.key
ETCD_CERT_FILE=/etc/etcd/n1.etcd.crt
ETCD_TRUSTED_CA_FILE=/etc/etcd/rootCA.crt
ETCD_CLIENT_CERT_AUTH=true

ETCD_PEER_CERT_FILE=/etc/etcd/n1.etcd.crt
ETCD_PEER_KEY_FILE=/etc/etcd/n1.etcd.key
ETCD_PEER_TRUSTED_CA_FILE=/etc/etcd/rootCA.crt
ETCD_PEER_CLIENT_CERT_AUTH=true
```

   Replace `n1` here with the appropriate node name.

3. Restart etcd services on all etcd cluster nodes:

```
# systemctl restart etcd
```

4. To check the new configuration, use the following command:

```
# etcdctl  --endpoints=https://n1:2379,https://n2:2379,https://n3:2379 --cacert /
etc/etcd/rootCA.crt --cert /etc/etcd/n1.etcd.crt --key /etc/etcd/n1.etcd.key member
 list -w table

+------------------+---------+------+-----------------+-----------------
+------------+
|        ID        | STATUS  | NAME |    PEER ADDRS   |   CLIENT ADDRS  | IS LEARNER
 |
+------------------+---------+------+-----------------+-----------------
+------------+
| 66ebe06d7302c3f0 | started |   n2 | https://n2:2380 | https://n2:2379 |      false
 |
| b1080bf5ff059980 | started |   n1 | https://n1:2380 | https://n1:2379 |      false
 |
| d98323257249fefb | started |   n3 | https://n3:2380 | https://n3:2379 |      false
 |
+------------------+---------+------+-----------------+-----------------
+------------+
```

5. On each Shardman cluster node, put `rootCA.crt`, `shardman_services.crt` and `shardman_services.key` in a location accessible to the `postgres` user (for example, create the `/etc/shardman` directory and put files there). Ensure that the `shardman_services.key` file is only accessible to the `postgres` user.
6. Edit the shardmand configuration file `/etc/shardman/shardmand-cluster0.env` as follows:

```
SDM_STORE_ENDPOINTS=https://n1:2379,https://n2:2379,https://n3:2379
SDM_STORE_CA_FILE=/etc/shardman/rootCA.crt
SDM_STORE_CERT_FILE=/etc/shardman/shardman_services.crt
SDM_STORE_KEY=/etc/shardman/shardman_services.key
```

7. Restart `shardmand@cluster0` services on all Shardman nodes:

```
# systemctl restart shardmand@cluster0
```

2.7.3. Using Shardman Tools

Before using Shardman tools, copy `rootCA.crt`, `shardman_tools.crt` and `shardman_tools.key` to some location on the Shardman management node where they are accessible to the management user. Here, any node with installed Shardman utilities that is used to manage the Shardman cluster is meant by *management node*. This can also be one of the Shardman cluster nodes (or all of them). By *management user*, a user is meant who runs shardmanctl tool. It is assumed that the certificates and key are located in the `/etc/shardman` directory.

When using Shardman tools, be sure to add `--store-ca-file`, `--store-cert-file` and `--store-key` options to `shardmanctl` command. For example, the following command gets the cluster status:

```
# shardmanctl --store-ca-file /etc/shardman/rootCA.crt --store-cert-file /etc/shardman/
shardman_tools.crt --store-key /etc/shardman/shardman_tools.key --store-endpoints
 https://n1:2379,https://n2:2379,https://n3:2379 status
```

2.8. Upgrading a Cluster

This section discusses how to upgrade your database from one Shardman release to a newer one. It is best to review the Release Notes before an upgrade and look for any changes that may cause issues for your application. You can proceed to upgrade if there are no potential issues.

The process of updating a Shardman consists of several steps that must be performed sequentially:

1. Upgrade Shardman packages.

2. Restart all Shardman services and database instances.
3. Upgrade database shardman extension.

2.8.1. Upgrade Packages

2.8.1.1. APT-based Systems

To upgrade packages, typically run the following command:

```
$ apt update && apt --only-upgrade install shardman-tools shardman-services
 postgrespro-sdm-14-contrib postgrespro-sdm-14-server
```

or upgrade all packages:

```
$ apt update && apt upgrade
```

Check that all packages have been updated on each node:

```
$ dpkg -l | grep -E  '(postgrespro|shardman)'
```

2.8.1.2. RPM-based systems

To upgrade packages, typically run the following command:

```
$ yum update shardman-tools shardman-services postgrespro-sdm-14-contrib postgrespro-
sdm-14-server
```

or upgrade all packages:

```
$ yum update
```

Check that all packages have been updated on each node:

```
$ yum list --installed | grep -E  '(postgrespro|shardman)'
```

2.8.2. Restart Shardman Services and Database Instances

After updating the packages, you need to restart all cluster services. It can be done with a single `shardmanctl restart` command:

```
$ shardmanctl --cluster-name cluster0 --store-endpoints http://etcd1:2379,http://
etcd2:2379,http://etcd3:2379 restart
```

You can skip the `--cluster-name` and `--store-endpoints` options by setting the `SDM_CLUSTER_NAME` and `SDM_STORE_ENDPOINTS` environment variables as in the example below:

```
                    export SDM_STORE_ENDPOINTS=http://etcd1:2379,http://
etcd2:2379,http://etcd3:2379
                    export SDM_CLUSTER_NAME=cluster0
```

2.8.3. Upgrade the Extension

After restarting services of the cluster, you should update the server extensions by running the following command:

```
$ shardmanctl --cluster-name cluster0 --store-endpoints http://etcd1:2379,http://
etcd2:2379,http://etcd3:2379 upgrade
```

In the case when the shardman extension version and server library version are different, distributed queries and Shardman DDL will not work.

Shardman extensions try to ensure that they do not communicate with incompatible software. Incompatibilities can arise for several reasons: the shardman shared library version does not match the extension version or the remote server version does not match the local server version. In case when the extension and library versions mismatch, Shardman cannot modify its metadata and will refuse to perform operations on global objects until the extension is updated. In case when the remote server version does not match the local server version or when they belong to different clusters, Shardman will refuse to communicate with the server.

2.9. Fault Tolerance and High Availability

Shardman provides out-of-the-box fault tolerance. The shardmand daemon monitors the cluster configuration and manages stolon clusters, which are used to guarantee high availability of all shards and fault tolerance. The common Shardman configuration (shardmand, stolon clusters) is stored in an etcd cluster.

To ensure fault tolerance for each stolon cluster, you must set `Repfactor > 0` in the cross-replication mode (`PlacementPolicy=cross`) or add at least one replica in the manual-topology mode (`PlacementPolicy=manual`).

stolon `sentinels` have the responsibility of observing the `keepers` and carrying out elections to choose one of the `keepers` as the master. `Sentinels` hold elections when the cluster starts and every time the current master `keeper` goes down.

One of the `keepers` is elected as the master. All write operations take place at the master, and the other instances are used as follower instances.

In the case of automatic failover, stolon will take care of automatically changing slave to master and failed master to standby. Only one additional thing you need is etcd to store the master/slave instant information by stolon.

If necessary, you can switch to a new master manually by running the `shardmanctl shard switch` command.

#utomatic failover is based on the use of timeouts, which can be overridden in `sdmspec.json`, as in the example:

```
{
"ShardSpec":{
    "failInterval": "20s",
    "sleepInterval": "5s",
    "convergenceTimeout": "30s",
    "deadKeeperRemovalInterval": "48h",
    "requestTimeout": "10s",
    ...
},
...
},
...
```

You can specify some high-availability options to define cluster behavior in a fault state: masterDemotionEnabled, masterDemotionTimeout, minSyncMonitorEnabled and minSyncMonitorUnhealthyTimeout.

2.9.1. Timeouts

`convergenceTimeout`

Interval to wait for a database to be converged to the required state when no long operations are expected.

Default: `30s`.

`deadKeeperRemovalInterval`

Interval after which a dead `keeper` will be removed from the cluster data.

Default: `48h`.

`failInterval`

Interval after the first failure to declare a `keeper` or a database as not healthy.

Default: `20s`.

`requestTimeout`

Time after which any request (`keeper` checks from `sentinel` etc...) will fail.

Default: `10s`.

`sleepInterval`

Interval to wait before the next check.

Default: `5s`.

2.10. Logging

Shardman is a critical point in your infrastructure as it stores all of your data. This makes logging mandatory. So you should understand how logging works in Shardman. Due to the complexity of Shardman, it supports logging from several components: logs from the shardmand daemon that manages the cluster configuration and logs from PostgreSQL database instances.

2.10.1. PostgreSQL Logs

Shardman uses standard PostgreSQL logging settings, described *here*. Logging settings should be placed to `sdmspec.json` in the `pgParameters` section, as shown in the example below:

```
{
  "ShardSpec": {
  "pgParameters": {
      "log_line_prefix": "%m [%r][%p]",
      "log_min_messages": "INFO",
      "log_statement": "none",
      "log_destination": "stderr",
      "log_filename": "pg.log",
      "logging_collector": "on",
      "log_checkpoints": "false",
      ...
  },
  ...
  },
  ...
}
```

By default, logs are placed in the directory like this: `/var/lib/pgpro/sdm-14/data/keeper-cluster0-clover-1-shrn1-0/postgres/log`. In this example, `cluster0` is the current cluster, `clover-1-shrn1` is the name of the current shard, `0` is the identifier of the integrated `keeper` process. To change the log directory, set the `log_directory` parameter.

2.10.2. shardmand Logs

shardmand is a systemd unit, its logs are written to journald. You can use `journalctl` to examine it. For example, you can use the following command:

```
$ journalctl -u shardmand@cluster0.service
```

You can filter logs by arbitrary time limits using the `--since` and `--until` options, which restrict the entries displayed to those after or before the given time, respectively. The time values can come in a variety of formats. For absolute time values, you should use `YYYY-MM-DD HH:MM:SS`. For instance, we can see all of the entries since January 10th, 2023 at 5:15 PM by typing:

```
$ journalctl -u shardmand@cluster0.service --since "2023-01-10
 17:15:00"
```

If components of the above format are left off, some defaults will be applied. For instance, if the date is omitted, the current date will be assumed. If the time component is missing, “00:00:00” (midnight) will be substituted. The seconds field can be left off as well to default to “00”:

```
$ journalctl -u shardmand@cluster0.service --since "2023-01-10" --
until "2023-01-11 03:00"
```

To control the log verbosity for all Shardman services, set `SDM_LOG_LEVEL` in the shardmand configuration file.

2.10.3. Getting Information on Backend Crashes

Some crashes are caused by the hardware failure or the DBMS issues. To understand the root causes of the crash, use `crash_info`. To set it up, follow these steps:

- Create a directory on each cluster node that the Shardman operating system user has access to (usually, it is `postgres`). Error reports will be sent to this directory.

  ```
  install -d -o postgres -g postgres -m 700  /var/lib/postgresql/crashinfo
  ```

- Set the `crash_info_location` value.

  > **Note**
  >
  > This will cause the DBMS to restart.

  ```
  shardmanctl --store-endpoints http://etcdserver:2379 set -y  crash_info_location=/
  var/lib/postgresql/crashinfo
  ```

- To make sure the changes are applied, send a signal that will cause the backend failure and a core dump creation, along with the instance restart.

  > **Note**
  >
  > Do it in your test environment only.

Connect to your DBMS and find out PID of the backend associated with the current session:

```
postgres=# select pg_backend_pid();
pg_backend_pid
----------------
    23770
```

Then send the SIGSEGV signal to the process with the received PID:

```
kill -11 23770
```

This will result in this backend crash, and a log file with the time, backtrace and cause of an error will be written to `/var/lib/postgresql/crashinfo`:

```
 # Signal
Program received signal: 11 (SIGSEGV)
Signal    UTC date time: 25.10.2024 08:37:02


# Program
                        pid: 23770
                        ppid: 17506
    program_invocation_name: postgres: postgres postgres 10.42.42.10(34202) idle
program_invocation_short_name: tgres 10.42.42.10(34202) idle
                    exe_path: /opt/pgpro/sdm-14/bin/postgres
                        exe: postgres

# Backtrace
1   postgres + 0x5b55c0             0x55c5ba8459b7  0x00007ffcbef19070
 bt_crash_handler + 0x3f7
```

```
2   libc.so.6 + 0x4251f             0x7f01c2caa520  0x00007ffcbef19140  __sigaction +
 0x50
unknown  ./signal/../sysdeps/unix/sysv/linux/x86_64/libc_sigaction.c:0
3   libc.so.6 + 0x125f80            0x7f01c2d8df9a  0x00007ffcbef195b8  epoll_wait +
 0x1a
epoll_wait  ../sysdeps/unix/sysv/linux/epoll_wait.c:30
4   postgres + 0x433870             0x55c5ba6c39bb  0x00007ffcbef195c0
 WaitEventSetWait + 0x14b
5   postgres + 0x320de0             0x55c5ba5b0e74  0x00007ffcbef19650  secure_read +
 0x94
6   postgres + 0x327d20             0x55c5ba5b7dae  0x00007ffcbef196a0  pq_recvbuf +
 0x8e
7   postgres + 0x328980             0x55c5ba5b8995  0x00007ffcbef196c0  pq_getbyte +
 0x15
8   postgres + 0x457da0             0x55c5ba6e909c  0x00007ffcbef196d0  PostgresMain +
 0x12fc
9   postgres + 0x3ce210             0x55c5ba65ef86  0x00007ffcbef19a60  ServerLoop +
 0xd76
10  postgres + 0x3cf240             0x55c5ba65fe18  0x00007ffcbef1a040  PostmasterMain
 + 0xbd8
11  postgres + 0x14ecc0             0x55c5ba3df182  0x00007ffcbef1a0c0  main + 0x4c2
12  libc.so.6 + 0x29d10             0x7f01c2c91d90  0x00007ffcbef1a0f0
 __libc_init_first + 0x90
__libc_start_call_main  ../sysdeps/nptl/libc_start_call_main.h:58
13  libc.so.6 + 0x29dc0             0x7f01c2c91e40  0x00007ffcbef1a190
 __libc_start_main + 0x80
call_init  ../csu/libc-start.c:128
__libc_start_main_impl  ../csu/libc-start.c:379
14  postgres + 0x14f200             0x55c5ba3df225  0x00007ffcbef1a1e0  _start + 0x25
```

Chapter 3. Develop

A Shardman cluster uses two main ways to store data: sharded tables is the main way, designed for big data, and global tables, designed for small dictionaries. A sharded table contains different parts of the data in each shard, while a global table contains the same data in all shards. Efficient query execution on a Shardman cluster requires that the data is properly distributed across cluster shards and primarily, a sharding key is properly selected.

First of all, when transitioning from a regular database schema to the distributed one, it makes sense to start the design with deciding how the data will be distributed in the Shardman cluster. Shardman distributes table rows across shards according to the hash value of the column to use for the table partitioning. In other words, the desired distribution must be even, and it aims to distribute equal parts of the data across cluster nodes and evenly distribute the workload.

When a database architect chooses the column to use for the table partitioning, the majority of typical queries executed must be taken into account to ensure the maximum performance.

In general, for most queries, especially, for those that use joins, the sharding key must be included in the query text. Otherwise, Shardman will not push down queries to cluster nodes for execution, which will cause essential performance degradation as compared to usage of a single instance.

Secondly, when choosing a sharding key, it is important that it does not change. A resharding operation, that is, a change of the sharding key, is pretty time-consuming and resource-intensive. At present, Shardman lacks techniques that automate this procedure. In general, if resharding is required, the data in all the sharded tables should be either moved to local tables or to sharded tables with another sharding key. Then you will have to create new sharded tables with a new sharding key and move the data back. This operation is very expensive and resource-intensive. Such operations often cannot be performed without the system outage during the migration.

Another point is that distributed transactions, that is, those that update data on several cluster shards at the same point in time, cannot be performed for free. So the better data is located and computations are performed inside one shard, the faster queries are executed. In general, the proportion of distributed and non-distributed transactions must be shifted towards non-distributed ones. Only apply distributed transactions if you have a compelling need to do it.

And finally, Shardman is a distributed system, which has both advantages and disadvantages inherent to such systems. Besides, Shardman is primarily designed for OLTP load. OLAP queries to Shardman are also possible, but only pretty simple of them (for details, see limitations). If you want to load an OLTP system with OLAP functionality, bear in mind that the lists of analytic and aggregate SQL functions to be sent to other shards for execution are highly limited.

Also special attention should be paid to type casts in queries because inclusion of a type casting function in a query condition can make it impossible to be pushed down to a remote server.

Taking into account the above features and limitations of the RDBMS, we will provide two simple examples of the transition from a regular to a distributed database schema.

3.1. Migration of a Database Schema

Let's use the demo database “Airlines” as an example for development. The detailed description of the database schema is available at *https://postgrespro.ru/education/demodb*. This schema is used as a demo in *training courses of Postgres Professional*, for example, in “QPT. Query Optimization”.

The schema authors characterized it like this: “We tried to make the database schema as simple as possible, without overloading it with unnecessary details, but not too simple to allow writing interesting and meaningful queries.”

The database schema contains several tables with meaningful contents. For example, let's take the demo database version of 13.10.2016. You can find a link to downloading the database and schema dump (in Russian) following the link *https://postgrespro.ru/education/courses/QPT*. In addition to query examples provided below, you can find more examples from the above course and in the *“Postgres. The First Experience”* book.

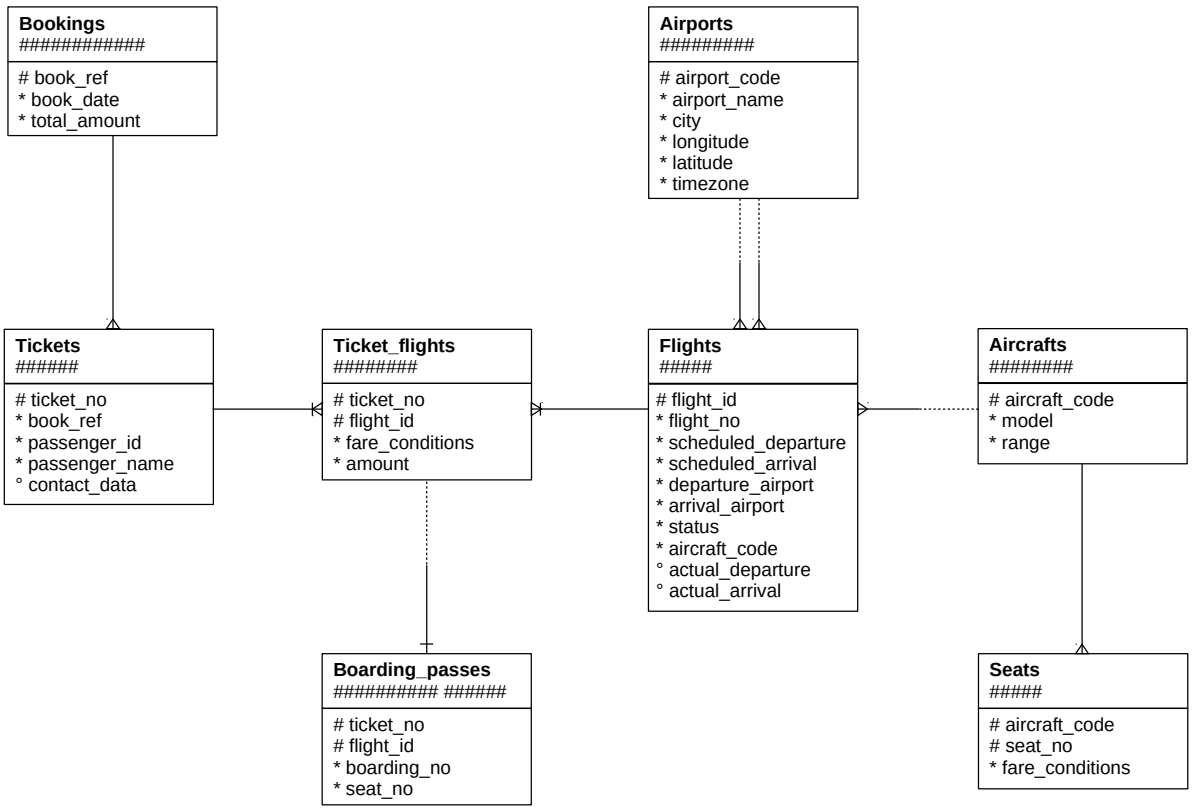
This section shows two examples of schema modification and query adaptation:

- *Naive approach*. It is simple, with minimal transformations to the schema, and it aims to add clarity to how queries work in a distributed schema.

- *Complex approach* It is more complex, provided for better understanding of problems and processes that a developer may confront when migrating to a distributed schema and adapting applications to such a schema.

3.1.1. Database Source Schema

Figure 3.1. Database Source Schema



The authors describe the “Airlines” database as follows:

> The main entity is a booking (`bookings`).
>
> One booking can include several passengers, with a separate ticket (`tickets`) issued to each passenger. A ticket has a unique number and includes information about the passenger. As such, the passenger is not a separate entity. Both the passenger's name and identity document number can change over time, so it is impossible to uniquely identify all the tickets of a particular person; for simplicity, we can assume that all passengers are unique.
>
> The ticket includes one or more flight segments (`ticket_flights`). Several flight segments can be included into a single ticket if there are no non-stop flights between the points of departure and destination (connecting flights), or if it is a round-trip ticket. Although there is no constraint in the schema, it is assumed that all tickets in the booking have the same flight segments.
>
> Each flight (`flights`) goes from one airport (`airports`) to another. Flights with the same flight number have the same points of departure and destination, but differ in departure date.
>
> At flight check-in, the passenger is issued a boarding pass (`boarding_passes`), where the seat number is specified. The passenger can check in for the flight only if this flight is included into the ticket. The flight-seat combination must be unique to avoid issuing two boarding passes for the same seat.

> The number of seats (`seats`) in the aircraft and their distribution between different travel classes depends on the model of the aircraft (`aircrafts`) performing the flight. It is assumed that every aircraft model has only one cabin configuration. Database schema does not check that seat numbers in boarding passes have the corresponding seats in the aircraft (such verification can be done using table triggers, or at the application level).

Let's look at the common entities and sizes of tables in the above schema. It is clear that `ticket_flights`, `boarding_passes` and `tickets` tables are linked by the `ticket_no` field. Additionally, the data size in these tables is 95% the total DB size.

Let's look at the `bookings` table. Although it seems to have a pretty compact structure, it can reach a considerable size over time.

Migration examples are provided for a Shardman cluster that contains four shards. Sharded tables are divided into four parts, so that one part of a sharded table is only located in one shard. This is done on purpose, to more clearly display query plans. In real life, the number of partitions should be determined by the maximum number of cluster nodes.

When migrating a real-life DB schema, you should think over in advance the number of partitions to partition data in distributed tables. Also bear in mind that the best migration approach is to use SQL transformations that impose minimal limitations on database objects.

3.1.2. Shardman Cluster Configuration

The Shardman cluster consists of four nodes — `node1`, `node2`, `node3` and `node4`. Each cluster node is a shard.

The examples assume that the tables are divided into four partitions by the sharding key (`num_parts` = 4) and distributed across cluster nodes. Each table part with the data is located in the corresponding shard:

- `shard-1` is located on the cluster node `node1`
- `shard-2` is located on the cluster node `node2`
- `shard-3` is located on the cluster node `node3`
- `shard-4` is located on the cluster node `node4`

The cluster is intentionally presented in a simplified configuration. Cluster nodes have no replicas, and the configuration is not fault-tolerant.

3.1.3. Selecting the Sharding Key

3.1.3.1. Naive[1] Approach — `ticket_no` Sharding Key

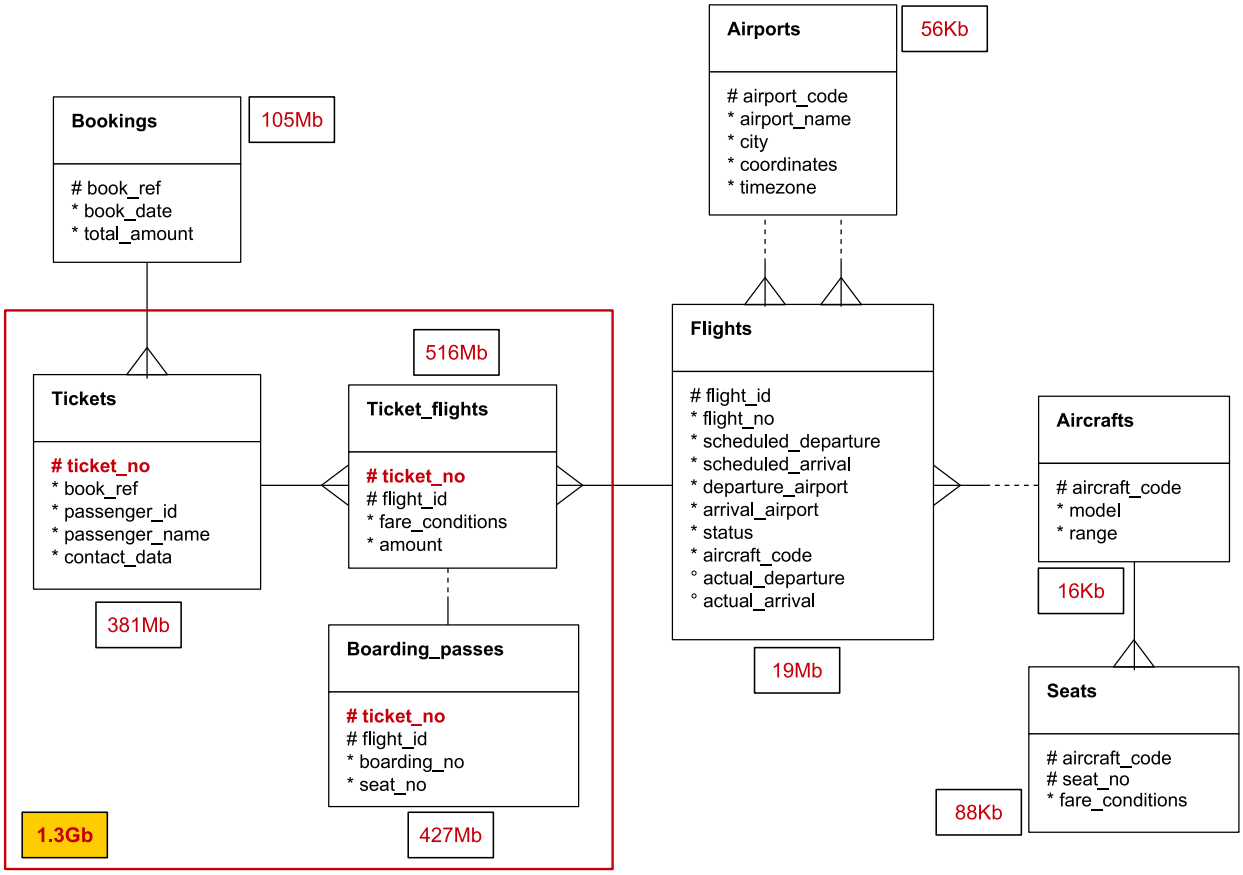
With this approach, the choice of the sharding key is pretty evident. It is the ticket number `ticket_no`. The ticket number is the primary key of the `tickets` table, and it is a foreign key of the `ticket_flights` and `boarding_passes` tables.

The primary key of the `ticket_flights` and `boarding_passes` tables is composite. It is a unique index composed of `ticket_no` and `flight_id`.

So if `ticket_no` is chosen to be a sharding key, the data of the three tables is distributed across cluster shards and partitions that contain linked data are located in the same shards.

The rest of the tables — `airports`, `flights`, `aircrafts` and `seats` are small enough and rarely change. This allows making them global tables, or dictionary tables.

Figure 3.2. Naive Approach Schema



The main advantage of this approach from the point of view of creating the schema and queries to the DB is that no changes are needed except those that are inherent to working with distributed systems, that is, explicitly declaring tables, sequences etc. as distributed when creating them.

Once the sharding key is selected, we can proceed to creation of the distributed schema.

1

3.1.3.1.1. Creating the Schema Distributed by `ticket_no`

First, turn on broadcasting DDL statements to all cluster shards:

```
SET shardman.broadcast_ddl TO on;
```

Let's create the `bookings` schema on all shards:

```
CREATE SCHEMA bookings;
```

As tables in the schema are linked with one another by a foreign key, the order of creating them, as well as auxiliary objects, matters.

The demo database contains “snapshots” of data, similar to a backup copy of a real system captured at some point in time. For example, if a flight has the `Departed` status, it means that the aircraft had already departed and was airborne at the time of the backup copy. The “snapshot” time is saved in the `bookings.now()` function. You can use this function in demo queries for cases where you would use the `now()` function in a real database. In addition, the return value of this function determines the version of the demo database. The latest version available is of 13.10.2016:

```
SELECT bookings.now() as now;
          now
```

[1] In the context of computer science, the expression “naïve approach” (verbatim: naive method, naive approach) means something very similar to “brute-force approach” and means the first basic idea that occurs in one's mind and often takes no account of the complexity, corner cases and of some requirements. On one hand, this is a coarse and direct method that only aims to get a working solution. On the other hand, such solutions are easy to understand and implement, but system resources may be used inefficiently.

```
-----------------------
2016-10-13 17:00:00+03
```

In relation to this moment, all flights are classified as past and future flights.

Let's create the utility function `bookings.now()`:

```
    CREATE FUNCTION bookings.now() RETURNS timestamp with time zone
    LANGUAGE sql IMMUTABLE COST 0.00999999978
    AS
    $sql$
    SELECT $qq$2016-10-13 17:00:00$qq$::TIMESTAMP AT TIME ZONE
$zz$Europe/Moscow$zz$;
    $sql$;
```

In addition to tables, a global sequence is needed for generating IDs for data insertion in the `flights` table. In this example, we create the sequence explicitly and link it with a column of this table by assigning the generated values by default.

Let's create the sequence using the following DDL statement:

```
CREATE SEQUENCE bookings.flights_flight_id_seq
    INCREMENT BY 1
    NO MINVALUE
    NO MAXVALUE
    CACHE 1 with(global);
```

`with(global)` creates a single distributed sequence available on all cluster nodes, which assigns values in a certain range for each shard, and the ranges for different shards do not intersect. See Section 7.6 and Section 6.5 for more details of global sequences.

Under the hood of global sequences, there are regular sequences on each shard, and they are allocated by sequential blocks (of 65536 numbers by default). When all the numbers in a block are over, the next block is allocated to the local sequence of the shard. I.e., numbers from the global sequences are unique, but there is no strict monotony, and there may be "holes" in the values given by the sequencer[2].

The sequences can have the `bigserial`, `smallserial`, or `serial` type. Sequences are applicable both for sharded and global tables.

You should not create local sequences in each shard as their values may be duplicated.

2

Now, we create global tables. As explained above, they are small-size, their data changes rarely, so they are actually dictionary tables, which must contain the same data in all cluster shards. It is required that each global table has a primary key.

Let's create global tables using the following DDL statements:

```
CREATE TABLE bookings.aircrafts (
    aircraft_code character(3) NOT NULL primary key,
    model text NOT NULL,
    range integer NOT NULL,
    CONSTRAINT aircrafts_range_check CHECK ((range > 0))
) with (global);

CREATE TABLE bookings.seats (
    aircraft_code character(3) references bookings.aircrafts(aircraft_code),
    seat_no character varying(4) NOT NULL,
    fare_conditions character varying(10) NOT NULL,
    CONSTRAINT seats_fare_conditions_check CHECK (((fare_conditions)::text = ANY
 (ARRAY[('Economy'::character varying)::text, ('Comfort'::character varying)::text,
 ('Business'::character varying)::text]))),
    PRIMARY KEY (aircraft_code, seat_no)
) with (global);
```

[2] As values from different ranges can be assigned, the value can leap. For example, the value of 5 may be assigned in the first shard, the value of 140003 — in the second one, 70003 — in the third one etc.

```
CREATE TABLE bookings.airports (
    airport_code character(3) NOT NULL primary key,
    airport_name text NOT NULL,
    city text NOT NULL,
    longitude double precision NOT NULL,
    latitude double precision NOT NULL,
    timezone text NOT NULL
)  with (global);

CREATE TABLE bookings.bookings (
    book_ref character(6) NOT NULL,
    book_date timestamp with time zone NOT NULL,
    total_amount numeric(10,2) NOT NULL,
    PRIMARY KEY (book_ref)
) with (global);

CREATE TABLE bookings.flights (
    flight_id bigint NOT NULL PRIMARY KEY,-- <= a sequence will be assigned
    flight_no character(6) NOT NULL,
    scheduled_departure timestamp with time zone NOT NULL,
    scheduled_arrival timestamp with time zone NOT NULL,
    departure_airport character(3) REFERENCES bookings.airports(airport_code),
    arrival_airport character(3) REFERENCES bookings.airports(airport_code),
    status character varying(20) NOT NULL,
    aircraft_code character(3) references bookings.aircrafts(aircraft_code),
    actual_departure timestamp with time zone,
    actual_arrival timestamp with time zone,
    CONSTRAINT flights_check CHECK ((scheduled_arrival > scheduled_departure)),
    CONSTRAINT flights_check1 CHECK (((actual_arrival IS NULL) OR ((actual_departure IS
 NOT NULL) AND (actual_arrival IS NOT NULL) AND (actual_arrival > actual_departure)))),
    CONSTRAINT flights_status_check CHECK (((status)::text = ANY (ARRAY[('On
 Time'::character varying)::text, ('Delayed'::character varying)::text,
 ('Departed'::character varying)::text, ('Arrived'::character varying)::text,
 ('Scheduled'::character varying)::text, ('Cancelled'::character varying)::text])))
) with (global);

-- associate the sequence with table column
ALTER SEQUENCE bookings.flights_flight_id_seq OWNED BY bookings.flights.flight_id;


-- assign the default value to the column
ALTER TABLE bookings.flights ALTER COLUMN flight_id SET DEFAULT
 nextval('bookings.flights_flight_id_seq');

ALTER TABLE bookings.flights ADD CONSTRAINT flights_flight_no_scheduled_departure_key
 UNIQUE (flight_no, scheduled_departure);
```

Next, we create sharded tables `tickets`, `ticket_flights` and `boarding_passes` in the `bookings` schema:

```
CREATE TABLE bookings.tickets (
    ticket_no character(13) PRIMARY KEY,
    book_ref character(6) REFERENCES bookings.bookings(book_ref),
    passenger_id character varying(20) NOT NULL,
    passenger_name text NOT NULL,
    contact_data jsonb
) with (distributed_by='ticket_no', num_parts=4);

CREATE TABLE bookings.ticket_flights (
```

```
    ticket_no character(13) NOT NULL,
    flight_id bigint references bookings.flights(flight_id),
    fare_conditions character varying(10) NOT NULL,
    amount numeric(10,2) NOT NULL,
    CONSTRAINT ticket_flights_amount_check CHECK ((amount >= (0)::numeric)),
    CONSTRAINT ticket_flights_fare_conditions_check CHECK (((fare_conditions)::text =
 ANY (ARRAY[('Economy'::character varying)::text, ('Comfort'::character varying)::text,
 ('Business'::character varying)::text]))),
    PRIMARY KEY (ticket_no, flight_id)
) with (distributed_by='ticket_no', colocate_with='bookings.tickets');

CREATE TABLE bookings.boarding_passes (
    ticket_no character(13) NOT NULL,
    flight_id bigint NOT NULL,
    boarding_no integer NOT NULL,
    seat_no character varying(4) NOT NULL,
    FOREIGN KEY (ticket_no, flight_id) REFERENCES bookings.ticket_flights(ticket_no,
 flight_id),
    PRIMARY KEY (ticket_no, flight_id)
) with (distributed_by='ticket_no', colocate_with='bookings.tickets');

-- constraints must contain sharding key
ALTER TABLE bookings.boarding_passes ADD CONSTRAINT
 boarding_passes_flight_id_boarding_no_key UNIQUE (ticket_no, flight_id, boarding_no);

ALTER TABLE bookings.boarding_passes ADD CONSTRAINT
 boarding_passes_flight_id_seat_no_key UNIQUE (ticket_no, flight_id, seat_no);
```

Additionally, when creating sharded tables, the `num_parts` parameter can be specified, which defines the number of sharded table partitions. In this example, it equals 4 to minimize the output of query plans. The default value is 20. This parameter may be important if in future you are going to add shards to a cluster and scale horizontally.

Based on the assumed future load and data size, `num_parts` should be sufficient for data rebalancing when new shards are added (`num_parts` must be greater than or equal to the number of cluster nodes). On the other hand, too many partitions cause a considerable increase of the query planning time. Therefore, an optimal balance should be achieved between the number of partitions and number of cluster nodes.

The last thing to do is to create a view that is needed to execute some queries:

```
CREATE VIEW bookings.flights_v AS
 SELECT f.flight_id,
    f.flight_no,
    f.scheduled_departure,
    timezone(dep.timezone, f.scheduled_departure) AS scheduled_departure_local,
    f.scheduled_arrival,
    timezone(arr.timezone, f.scheduled_arrival) AS scheduled_arrival_local,
    (f.scheduled_arrival - f.scheduled_departure) AS scheduled_duration,
    f.departure_airport,
    dep.airport_name AS departure_airport_name,
    dep.city AS departure_city,
    f.arrival_airport,
    arr.airport_name AS arrival_airport_name,
    arr.city AS arrival_city,
    f.status,
    f.aircraft_code,
    f.actual_departure,
    timezone(dep.timezone, f.actual_departure) AS actual_departure_local,
    f.actual_arrival,
    timezone(arr.timezone, f.actual_arrival) AS actual_arrival_local,
```

```
    (f.actual_arrival - f.actual_departure) AS actual_duration
   FROM bookings.flights f,
    bookings.airports dep,
    bookings.airports arr
  WHERE ((f.departure_airport = dep.airport_code) AND (f.arrival_airport =
 arr.airport_code));
```

Now creation of the distributed schema is complete. Let's turn off broadcasting of DDL statements:

```
SET shardman.broadcast_ddl TO off;
```

3.1.3.2. Complex Approach — book_ref Sharding Key

A more complex approach to the sharding key choice involves the source schema modification, inclusion of new parameters in queries and other important changes.

What if an airline is in the market for over 10 years and the `bookings` table reaches the size that does not allow you to continue having it global anymore? But distributing its data is impossible either as it does not contain fields contained in other tables that it can be distributed among (as in variant 1).

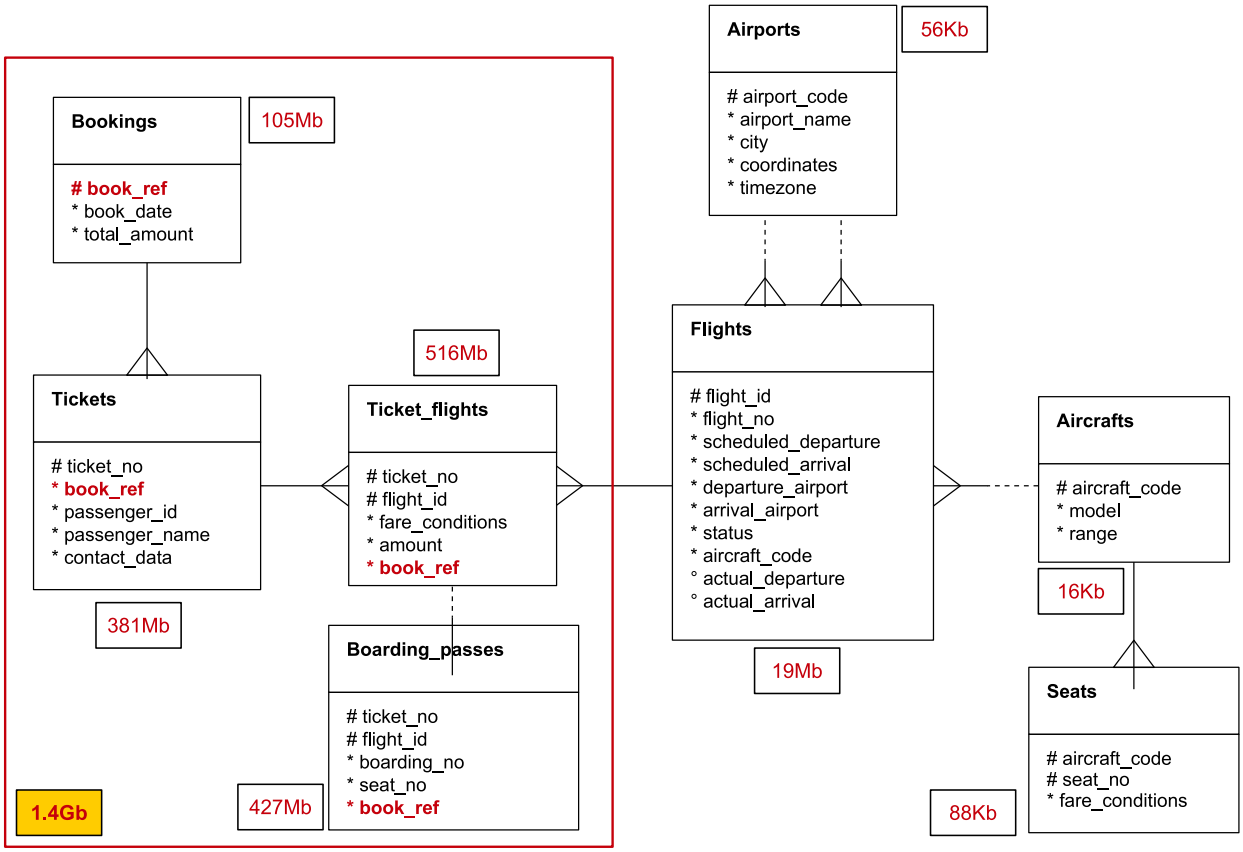
When modifying the source schema, another field can be appropriate for use as a sharding key.

Looking at the `bookings` table, we can notice that values of the `book_ref` field are unique and this field is a primary key. `book_ref` is also a foreign key to the `tickets` table. So this field seems suitable for being the sharding key. However, `book_ref` is missing from the `ticket_flights` and `boarding_passes` tables.

If we add `book_ref` to the `ticket_flights` and `boarding_passes` tables, distributing of all the tables `bookings`, `tickets`, `ticket_flights` and `boarding_passes` with the `book_ref` sharding key becomes possible.

`book_ref` should be added to `ticket_flights` and `boarding_passes` in the source schema, and `book_ref` must be filled with data from the `bookings` table.

Figure 3.3. Source Schema Modification



3.1.3.2.1. Modifying the Source Schema

To properly transfer data from the source schema to the distributed one, the schema should be modified as follows:

1. Add the `book_ref` field to the `ticket_flights` and `boarding_passes` tables:

```
ALTER TABLE ticket_flights
   ADD COLUMN book_ref char(6);

ALTER TABLE boarding_passes
   ADD COLUMN book_ref char(6);
```

2. In these tables, fill the added `book_ref` field with data:

```
WITH batch AS (SELECT book_ref,
                      ticket_no
                 FROM tickets)
UPDATE ticket_flights
   SET book_ref = batch.book_ref
  FROM batch
 WHERE ticket_flights.ticket_no = batch.ticket_no
   AND ticket_flights.book_ref IS NULL;


WITH batch AS (SELECT book_ref,
                      ticket_no
                 FROM tickets)
UPDATE boarding_passes
   SET book_ref = batch.book_ref
```

```
     FROM batch
    WHERE boarding_passes.ticket_no = batch.ticket_no
      AND boarding_passes.book_ref IS NULL;
```

Avoid using this example in a loaded production system as this approach blocks entire tables, that is, all rows in the tables. In production systems, data should be transferred incrementally, by parts.

Now the database schema is ready for data transferring.

3.1.3.2.2. Creating a Schema Distributed by `book_ref`

Here the Shardman shardman.broadcast_all_sql() function is used to broadcast DDL statements on all cluster nodes. Let's create the `bookings` schema on all shards:

```
SELECT shardman.broadcast_all_sql('CREATE SCHEMA bookings');
```

As tables in the schema are linked with an external key, the order of creating tables matters.

First we create a utility function `bookings.now()`:

```
SELECT shardman.broadcast_all_sql(
    $sql$
    CREATE FUNCTION bookings.now() RETURNS timestamp with time zone
    LANGUAGE sql IMMUTABLE COST 0.00999999978
    AS
    $q$
    SELECT $qq$2016-10-13 17:00:00$qq$::TIMESTAMP
                   AT TIME ZONE $zz$Europe/Moscow$zz$;
    $q$;
    $sql$
);
```

Tables, users and sequences are created with the regular SQL. This function is not needed for that.

In this example, the global sequence is not explicitly created as for the `bigserial` type, Shardman creates a global sequence automatically.

Now let's create global tables using the following DDL statements:

```
CREATE TABLE bookings.aircrafts (
    aircraft_code character(3) NOT NULL PRIMARY KEY,
    model text NOT NULL,
    range integer NOT NULL,
    CONSTRAINT aircrafts_range_check CHECK ((range > 0))
) WITH (global);

CREATE TABLE bookings.seats (
    aircraft_code character(3) REFERENCES bookings.aircrafts(aircraft_code),
    seat_no character varying(4) NOT NULL,
    fare_conditions character varying(10) NOT NULL,
    CONSTRAINT seats_fare_conditions_check CHECK ((
          (fare_conditions)::text = ANY (ARRAY[
             ('Economy'::character varying)::text,
             ('Comfort'::character varying)::text,
             ('Business'::character varying)::text])
           )),
    PRIMARY KEY (aircraft_code, seat_no)
) WITH (global);

CREATE TABLE bookings.airports (
    airport_code character(3) NOT NULL PRIMARY KEY,
    airport_name text NOT NULL,
    city text NOT NULL,
```

```
    longitude double precision NOT NULL,
    latitude double precision NOT NULL,
    timezone text NOT NULL
)  WITH (global);

CREATE TABLE bookings.flights (
-- the global sequence will be created automatically
-- the default value will be assigned
    flight_id bigserial NOT NULL PRIMARY KEY,
    flight_no character(6) NOT NULL,
    scheduled_departure timestamp with time zone NOT NULL,
    scheduled_arrival timestamp with time zone NOT NULL,
    departure_airport character(3) REFERENCES bookings.airports(airport_code),
    arrival_airport character(3) REFERENCES bookings.airports(airport_code),
    status character varying(20) NOT NULL,
    aircraft_code character(3) REFERENCES bookings.aircrafts(aircraft_code),
    actual_departure timestamp with time zone,
    actual_arrival timestamp with time zone,
    CONSTRAINT flights_check CHECK ((scheduled_arrival > scheduled_departure)),
    CONSTRAINT flights_check1 CHECK ((
                (actual_arrival IS NULL)
             OR ((actual_departure IS NOT NULL)
            AND (actual_arrival IS NOT NULL)
            AND (actual_arrival > actual_departure)))),
    CONSTRAINT flights_status_check CHECK (
           ((status)::text = ANY (
      ARRAY[('On Time'::character varying)::text,
            ('Delayed'::character varying)::text,
            ('Departed'::character varying)::text,
            ('Arrived'::character varying)::text,
            ('Scheduled'::character varying)::text,
            ('Cancelled'::character varying)::text])))
) WITH (global);

ALTER TABLE bookings.flights
   ADD CONSTRAINT flights_flight_no_scheduled_departure_key
   UNIQUE (flight_no, scheduled_departure);
```

Now let's create sharded tables `bookings`, `tickets`, `ticket_flights` and `boarding_passes` in the `bookings` schema, as in the previous example:

```
-- no modifications to these tables are done except distributing them
CREATE TABLE bookings.bookings (
   book_ref character(6) NOT NULL PRIMARY KEY,
   book_date timestamp with time zone NOT NULL,
   total_amount numeric(10,2) NOT NULL
) WITH (distributed_by='book_ref', num_parts=4);


CREATE TABLE bookings.tickets (
   ticket_no character(13) NOT NULL,
   book_ref character(6) REFERENCES bookings.bookings(book_ref),
   passenger_id character varying(20) NOT NULL,
   passenger_name text NOT NULL,
   contact_data jsonb,
   PRIMARY KEY (book_ref, ticket_no)
) WITH (distributed_by='book_ref', colocate_with='bookings.bookings');

-- adding the book_ref foreign key to these tables
```

```
CREATE TABLE bookings.ticket_flights (
   ticket_no character(13) NOT NULL,
   flight_id bigint NOT NULL,
   fare_conditions character varying(10) NOT NULL,
   amount numeric(10,2) NOT NULL,
   book_ref character(6) NOT NULL, -- <= added book_ref
   CONSTRAINT ticket_flights_amount_check
        CHECK ((amount >= (0)::numeric)),
   CONSTRAINT ticket_flights_fare_conditions_check
        CHECK (((fare_conditions)::text = ANY (
        ARRAY[('Economy'::character varying)::text,
              ('Comfort'::character varying)::text,
              ('Business'::character varying)::text]))),
   FOREIGN KEY (book_ref, ticket_no)
        REFERENCES bookings.tickets(book_ref, ticket_no),
   PRIMARY KEY (book_ref, ticket_no, flight_id) -- <= changed the primary key
) with (distributed_by='book_ref', colocate_with='bookings.bookings');


CREATE TABLE bookings.boarding_passes (
  ticket_no character(13) NOT NULL,
  flight_id bigint NOT NULL,
  boarding_no integer NOT NULL,
  seat_no character varying(4) NOT NULL,
  book_ref character(6) NOT NULL, – <= added book_ref
FOREIGN KEY (book_ref, ticket_no, flight_id)
REFERENCES bookings.ticket_flights(book_ref, ticket_no, flight_id),
PRIMARY KEY (book_ref, ticket_no, flight_id)
) WITH (distributed_by='book_ref', colocate_with='bookings.bookings');

-- constraints must contain the sharding key
ALTER TABLE bookings.boarding_passes
  ADD CONSTRAINT boarding_passes_flight_id_boarding_no_key
  UNIQUE (book_ref, ticket_no, flight_id, boarding_no);

ALTER TABLE bookings.boarding_passes
  ADD CONSTRAINT boarding_passes_flight_id_seat_no_key
  UNIQUE (book_ref, ticket_no, flight_id, seat_no);
```

Let's create the `bookings.flights` view:

```
SELECT shardman.broadcast_all_sql($$
CREATE VIEW bookings.flights_v AS
SELECT f.flight_id,
      f.flight_no,
      f.scheduled_departure,
      timezone(dep.timezone, f.scheduled_departure) AS scheduled_departure_local,
      f.scheduled_arrival,
      timezone(arr.timezone, f.scheduled_arrival)   AS scheduled_arrival_local,
      (f.scheduled_arrival - f.scheduled_departure) AS scheduled_duration,
      f.departure_airport,
      dep.airport_name                              AS departure_airport_name,
      dep.city                                      AS departure_city,
      f.arrival_airport,
      arr.airport_name                              AS arrival_airport_name,
      arr.city                                      AS arrival_city,
      f.status,
      f.aircraft_code,
      f.actual_departure,
```

```
      timezone(dep.timezone, f.actual_departure)    AS actual_departure_local,
      f.actual_arrival,
      timezone(arr.timezone, f.actual_arrival)      AS actual_arrival_local,
      (f.actual_arrival - f.actual_departure)       AS actual_duration
FROM bookings.flights f,
    bookings.airports dep,
    bookings.airports arr
WHERE ((f.departure_airport = dep.airport_code) AND (f.arrival_airport =
 arr.airport_code));
$$);
```

The schema creation is now complete. Let's proceed to data migration.

3.2. Data Migration

When migrating data, the order of fields in the source and target schema is important. The order and types of fields in the non-distributed and distributed databases must be the same.

The migration utility does exactly what is requested by the user, who does not interfere with data migration processes except, maybe, distributing the data directly to the shard where it must be stored.

Shardman provides convenient migration tools. Once the distributed schema is created and the sharding key chosen, it is now needed to define the data migration rules. The data source can be either export CSV data files or a single DBMS server.

It is not always convenient to use CSV files as they can reach a pretty large size and require additional resources for storage and transfer.

Migrating data directly from DB to DB without an intermediate storage phase is much more convenient.

The order of loading data during migration must be taken into account. Tables can be linked with a foreign key, so the data in tables that other tables will reference must be loaded first. To follow such an order, in the migration file, you should establish the priority that defines tables whose data must be loaded first. The higher the value of the `priority` parameter, the higher the priority. For example, if the priorities 1, 2 and 3 are defined, tables with the priority 3 will be loaded first, then those with the priority 2, and last with the priority 1.

The `shardmanctl load` command lets you define the order of migrating tables, which can be specified in the configuration YML file.

3.2.1. Naive Approach

The following is an example of the `migrate.yml` file:

```
version: "1.0"
migrate:
 connstr: "dbname=demo host=single-pg-instance port=5432 user=postgres password=******"
 jobs: 8
 batch: 2000
 options:
 schemas:
   - name: bookings
     # the all parameter set to false turns off automatic creation of pages
     # tables are already created, at the Schema Migration phase
     all: false
     tables:
       - name: airports
         # defining a global table
         type: global
         # as tables are linked, data migration priority must be defined
         # setting highest priority to tables whose data
         # must be copied first
```

```
          priority: 3
        - name: aircrafts
          type: global
          priority: 3
        - name: seats
          type: global
          priority: 3
        - name: bookings
          type: global
          priority: 3
        - name: flights
          type: global
          priority: 3
        - name: tickets
          type: sharded
          # defining a sharded table
          # specifying the sharding key
          distributedby: ticket_no
          partitions: 4
          priority: 2
        - name: ticket_flights
          type: sharded
          distributedby: ticket_no
          # defining a sharded and colocated table
          # specifying the name of the table that ticket_flights table will be colocated
 with
          colocatewith: tickets
          partitions: 4
          priority: 2
        - name: boarding_passes
          type: sharded
          distributedby: ticket_no
          colocatewith: tickets
          partitions: 4
          priority: 1
```

This file defines the data source, that is, the `single-pg-instance` node, its connection port, user name and password, and data source DB name. Some parameters of the migration utility operation are also defined (there can be quite a few of them, as explained in the section called “Loading Data with a Schema from PostgreSQL”). The file also defines the number of threads — 8, batch size, that is, the number of rows organized into batches for processing during migration, as well as table processing priorities. The data for the global tables is migrated first, then the data for the sharded tables `tickets` and `ticket_flights`, and migration of the `boarding_passes` table completes the migration. The value of `priority` defines the priority of data loading, data for tables with higher value will be loaded earlier than with the lower value. The following command performs the migration:

```
shardmanctl load --schema migrate.yml
```

If the utility completes with the message “data loading completed successfully”, it means that the migration was a success.

3.2.2. Complex Approach

With this approach, the launch and operation of the shardmanctl utility in the `load` mode is the same as with the naive approach. However, the file that defines the order of loading tables will slightly differ as the sharding key has changed:

```
 ---
version: "1.0"
migrate:
 connstr: "dbname=demo host=single-pg-instance port=5432 user=postgres
 password=postgres"
 jobs: 8
 batch: 2000
```

```
options:
schemas:
  - name: bookings
    all: false
    tables:
      - name: airports
        type: global
        priority: 5
      - name: aircrafts
        type: global
        priority: 5
      - name: seats
        type: global
        priority: 5
      - name: flights
        type: global
        priority: 5
      - name: bookings
        type: sharded
        priority: 4
        partitions: 4
        distributedby: book_ref
      - name: tickets
        type: sharded
        distributedby: book_ref
        colocatewith: bookings
        partitions: 4
        priority: 3
      - name: ticket_flights
        type: sharded
        distributedby: book_ref
        colocatewith: bookings
        partitions: 4
        priority: 2
      - name: boarding_passes
        type: sharded
        distributedby: book_ref
        colocatewith: bookings
        partitions: 4
        priority: 1
```

3.3. Queries

When all the migration operations were performed successfully, it's time to check how queries are executed in the distributed schema.

3.3.1. `q1` Query

The `q1` query is pretty simple, it selects the booking with the specified number:

```
SELECT *
  FROM bookings.bookings b
WHERE b.book_ref = '0824C5';
```

For the regular PostgreSQL and for the `ticket_no` sharding key, this query runs comparably fast. How fast the query is for the `book_ref` sharding key, depends on the shard where it is executed. If it is executed in a shard where there is physically no data, Shardman sends the query to another shard, which causes a time delay due to network communication.

3.3.2. `q2` Query

This `q2` query selects all the tickets from the specified booking:

```
SELECT t.*
FROM bookings.bookings b
JOIN bookings.tickets t
  ON t.book_ref = b.book_ref
WHERE b.book_ref = '0824C5';
```

With the `book_ref` sharding key, the query is pushed down to shards and the global table is joined with partitions of a sharded table:

```
Foreign Scan (actual rows=2 loops=1)
  Relations: (bookings_2_fdw b) INNER JOIN (tickets_2_fdw t)
  Network: FDW bytes sent=433 received=237
```

Let's look at the query plan for the `ticket_no` sharding key:

```
Append (actual rows=2 loops=1)
  Network: FDW bytes sent=1263 received=205
  ->  Nested Loop (actual rows=1 loops=1)
        ->  Seq Scan on tickets_0 t_1 (actual rows=1 loops=1)
              Filter: (book_ref = '0824C5'::bpchar)
              Rows Removed by Filter: 207092
        ->  Index Only Scan using bookings_pkey on bookings b (actual rows=1 loops=1)
              Index Cond: (book_ref = '0824C5'::bpchar)
              Heap Fetches: 0
  ->  Async Foreign Scan (actual rows=1 loops=1)
        Relations: (tickets_1_fdw t_2) INNER JOIN (bookings b)
        Network: FDW bytes sent=421 received=205
  ->  Async Foreign Scan (actual rows=0 loops=1)
        Relations: (tickets_2_fdw t_3) INNER JOIN (bookings b)
        Network: FDW bytes sent=421
  ->  Async Foreign Scan (actual rows=0 loops=1)
        Relations: (tickets_3_fdw t_4) INNER JOIN (bookings b)
        Network: FDW bytes sent=421
```

The plan contains `Async Foreign Scan` nodes, which mean network data exchange between the query source node and shards, that is, data is received from shards and final processing is done on the query source node.

Look at the `Network` line. A good criterion of whether query execution on shards is optimal is the value of `received`. The lower its value, the better shards execute distributed queries. Most processing is done remotely, and the query source node gets the result that is ready for further processing.

The case where the sharding key is `book_ref` looks much better as the table with ticket numbers already contains `book_ref`.

The plan of the query to be executed on an arbitrary node is as follows:

```
Foreign Scan (actual rows=2 loops=1)
  Relations: (bookings_2_fdw b) INNER JOIN (tickets_2_fdw t)
  Network: FDW bytes sent=433 received=237
```

The network data exchange is only done with one shard, in which the query is executed. It is `shard-3`, and the `tickets_2` partition of the `tickets` table is on the fourth node.

If this query is executed in the shard where the data is physically located, the query will be executed yet faster.

Let's look at the plan:

```
Nested Loop (actual rows=2 loops=1)
    ->  Index Only Scan using bookings_2_pkey on bookings_2
    ->  Bitmap Heap Scan on tickets_2
          ->  Bitmap Index Scan on tickets_2_book_ref_idx
```

Network data exchange is not needed here as the requested data is located within the shard in which the query is executed.

In some cases, the choice of the shard for query execution matters. Being aware of the distribution logic, you can implement it at the application level and send some queries immediately to the shard where the needed data is located based on the sharding key.

3.3.3. q3 Query

The `q3` query finds all the flights for one of the tickets in the booking selected earlier:

```
SELECT tf.*, t.*
FROM bookings.tickets t
JOIN bookings.ticket_flights tf
  ON tf.ticket_no = t.ticket_no
WHERE t.ticket_no = '0005435126781';
```

To choose a specific shard for query execution, as discussed in Section 3.3.2, note that with the `ticket_no` sharding key, the query execution will be more optimal in the shard that contains the partition with the data. The planner knows that the shard contains all the data needed for joining tables, so no network communication between shards will occur.

For the `book_ref` sharding key, note that from the booking number you can compute the ticket number and request it right from the "proper" shard.

So the query is as follows:

```
SELECT tf.*, t.*
FROM bookings.tickets t
JOIN bookings.ticket_flights tf
  ON tf.ticket_no = t.ticket_no
  AND t.book_ref = tf.book_ref
WHERE t.ticket_no = '0005435126781'
AND tf.book_ref = '0824C5';
```

The query is executed more slowly in the shard that does not contain the partition with the data sought:

```
Foreign Scan (actual rows=6 loops=1)
  Relations: (tickets_1_fdw t) INNER JOIN (ticket_flights_1_fdw tf)
  Network: FDW bytes sent=434 received=369
```

Network communication between shards is present in the plan, as it contains the `Foreign Scan` node.

The importance of including the sharding key in a query can be illustrated with the following query for the `book_ref` sharding key:

```
SELECT tf.*, t.*
FROM bookings.tickets t
JOIN bookings.ticket_flights tf
  ON tf.ticket_no = t.ticket_no
WHERE t.ticket_no = '0005435126781'
AND tf.book_ref = '0824C5';
```

Here the sharding key is not included in `join` on purpose. Let's look at the plan:

```
Nested Loop (actual rows=6 loops=1)
  Network: FDW bytes sent=1419 received=600
  ->  Foreign Scan on ticket_flights_2_fdw tf (actual rows=6 loops=1)
        Network: FDW bytes sent=381 received=395
  ->  Append (actual rows=1 loops=6)
        Network: FDW bytes sent=1038 received=205
        ->  Seq Scan on tickets_0 t_1 (actual rows=0 loops=6)
              Filter: (ticket_no = '0005435126781'::bpchar)
              Rows Removed by Filter: 207273
        ->  Async Foreign Scan on tickets_1_fdw t_2 (actual rows=0 loops=6)
              Network: FDW bytes sent=346 received=205
        ->  Async Foreign Scan on tickets_2_fdw t_3 (actual rows=1 loops=6)
              Network: FDW bytes sent=346
        ->  Async Foreign Scan on tickets_3_fdw t_4 (actual rows=0 loops=6)
              Network: FDW bytes sent=346
```

We can notice differences from previous examples. Here the query was executed on all nodes and index was not used, so to return as few as 6 rows, Shardman had to sequentially scan whole partitions of the `tickets` table, return the result to the query source

node and after that perform `join` with the `ticket_flights` table. `Async Foreign Scan` nodes indicate the sequential scan of the `tickets` table on shards.

3.3.4. q4 Query

This query returns all the flights for all the tickets included in a booking. There are several ways to do this: include a subquery in a `WHERE` clause with the booking number, in the `IN` clause, explicitly list ticket numbers or use the `WHERE...OR` clause. Let's check execution of the query for all these variants.

```
SELECT tf.*, t.*
FROM bookings.tickets t
JOIN bookings.ticket_flights tf
  ON tf.ticket_no = t.ticket_no
WHERE t.ticket_no IN (
   SELECT t.ticket_no
     FROM bookings.bookings b
     JOIN bookings.tickets  t
       ON t.book_ref = b.book_ref
    WHERE b.book_ref = '0824C5'
);
```

This is just the query from the non-distributed database that we tried to execute. But its execution is equally poor for both sharding keys.

The query plan is like this:

```
Hash Join (actual rows=12 loops=1)
  Hash Cond: (tf.ticket_no = t.ticket_no)
  ->  Append (actual rows=2360335 loops=1)
        ->  Async Foreign Scan on ticket_flights_0_fdw tf_1 (actual rows=589983
 loops=1)
        ->  Async Foreign Scan on ticket_flights_1_fdw tf_2 (actual rows=590175
 loops=1)
        ->  Seq Scan on ticket_flights_2 tf_3 (actual rows=590174 loops=1)
        ->  Async Foreign Scan on ticket_flights_3_fdw tf_4 (actual rows=590003
 loops=1)
  ->  Hash (actual rows=2 loops=1)
        Buckets: 1024  Batches: 1  Memory Usage: 9kB
        ->  Hash Semi Join (actual rows=2 loops=1)
              Hash Cond: (t.ticket_no = t_5.ticket_no)
              ->  Append (actual rows=829071 loops=1)
                    ->  Async Foreign Scan on tickets_0_fdw t_1 (actual rows=207273
 loops=1)
                    ->  Async Foreign Scan on tickets_1_fdw t_2 (actual rows=207058
 loops=1)
                    ->  Seq Scan on tickets_2 t_3 (actual rows=207431 loops=1)
                    ->  Async Foreign Scan on tickets_3_fdw t_4 (actual rows=207309
 loops=1)
              ->  Hash (actual rows=2 loops=1)
                    Buckets: 1024  Batches: 1  Memory Usage: 9kB
                    ->  Nested Loop (actual rows=2 loops=1)
                          ->  Index Only Scan using tickets_2_pkey on tickets_2 t_5
                          ->  Materialize (actual rows=1 loops=2)
                                ->  Index Only Scan using bookings_2_pkey on bookings_2
 b
```

This plan shows that Shardman coped with the `WHERE` subquery, then had to request all the rows of the `tickets` and `ticket_flights` tables and then process them on the query source node. This is a really poor performance. Let's try other variants:

For the `ticket_no` sharding key, the query is:

```
SELECT tf.*, t.*
FROM bookings.tickets t
JOIN bookings.ticket_flights tf
  ON tf.ticket_no = t.ticket_no
WHERE t.ticket_no IN ('0005435126781','0005435126782');
```

and the plan is:

```
Append (actual rows=12 loops=1)
  Network: FDW bytes sent=1098 received=1656
  ->  Async Foreign Scan (actual rows=6 loops=1)
        Relations: (tickets_0_fdw t_1) INNER JOIN (ticket_flights_0_fdw tf_1)
        Network: FDW bytes sent=549 received=1656
  ->  Async Foreign Scan (actual rows=6 loops=1)
        Relations: (tickets_1_fdw t_2) INNER JOIN (ticket_flights_1_fdw tf_2)
        Network: FDW bytes sent=549
```

Everything is pretty good here: the query was executed on two shards of four, and `Append` of the results received only had to be done.

Let's recall that `book_ref` is contained in both `tickets` and `ticket_flights` tables. So for the `book_ref` sharding key, the query is:

```
SELECT tf.*, t.*
FROM bookings.tickets t
JOIN bookings.ticket_flights tf
ON tf.ticket_no = t.ticket_no
AND tf.book_ref = t.book_ref
WHERE t.book_ref = '0824C5';
```

and the plan is:

```
Foreign Scan (actual rows=12 loops=1)
  Relations: (tickets_2_fdw t) INNER JOIN (ticket_flights_2_fdw tf)
  Network: FDW bytes sent=547 received=1717
```

This is an excellent result — the query was modified to execute well in the distributed schema.

3.3.5. q5 Query

This is a small analytical query, which returns the names and ticket numbers of the passengers who got registered first.

```
SELECT t.passenger_name, t.ticket_no
FROM bookings.tickets t
JOIN bookings.boarding_passes bp
  ON bp.ticket_no = t.ticket_no
GROUP BY t.passenger_name, t.ticket_no
HAVING max(bp.boarding_no) = 1
AND count(*) > 1;
```

This query is executed pretty slowly for both sharding keys. Below is the plan for `book_ref`:

```
HashAggregate (actual rows=424 loops=1)
  Group Key: t.ticket_no
  Filter: ((max(bp.boarding_no) = 1) AND (count(*) > 1))
  Batches: 85  Memory Usage: 4265kB  Disk Usage: 112008kB
  Rows Removed by Filter: 700748
  Network: FDW bytes sent=1215 received=77111136
  ->  Append (actual rows=1894295 loops=1)
        Network: FDW bytes sent=1215 received=77111136
        ->  Async Foreign Scan (actual rows=473327 loops=1)
              Relations: (tickets_0_fdw t_1) INNER JOIN (boarding_passes_0_fdw bp_1)
              Network: FDW bytes sent=404 received=813128
```

```
        ->  Async Foreign Scan (actual rows=472632 loops=1)
              Relations: (tickets_1_fdw t_2) INNER JOIN (boarding_passes_1_fdw bp_2)
              Network: FDW bytes sent=404
        ->  Async Foreign Scan (actual rows=475755 loops=1)
              Relations: (tickets_2_fdw t_3) INNER JOIN (boarding_passes_2_fdw bp_3)
              Network: FDW bytes sent=407
        ->  Hash Join (actual rows=472581 loops=1)
              Hash Cond: (bp_4.ticket_no = t_4.ticket_no)
              Network: FDW bytes received=28841344
              ->  Seq Scan on boarding_passes_3 bp_4 (actual rows=472581 loops=1)
              ->  Hash (actual rows=207118 loops=1)
                    Buckets: 65536  Batches: 4  Memory Usage: 3654kB
                    Network: FDW bytes received=9176680
                    ->  Seq Scan on tickets_3 t_4 (actual rows=207118 loops=1)
                          Network: FDW bytes received=9176680
```

Note a pretty large amount of network data transfer between shards. Let's improve the query by adding `book_ref` as one more condition for joining tables:

```
SELECT t.passenger_name, t.ticket_no
FROM bookings.tickets t
JOIN bookings.boarding_passes bp
  ON bp.ticket_no = t.ticket_no
  AND bp.book_ref=t.book_ref -- <= added book_ref
GROUP BY t.passenger_name, t.ticket_no
HAVING max(bp.boarding_no) = 1
AND count(*) > 1;
```

Let's look at the query plan:

```
GroupAggregate (actual rows=424 loops=1)
  Group Key: t.passenger_name, t.ticket_no
  Filter: ((max(bp.boarding_no) = 1) AND (count(*) > 1))
  Rows Removed by Filter: 700748
  Network: FDW bytes sent=1424 received=77092816
  ->  Merge Append (actual rows=1894295 loops=1)
        Sort Key: t.passenger_name, t.ticket_no
        Network: FDW bytes sent=1424 received=77092816
        ->  Foreign Scan (actual rows=472757 loops=1)
              Relations: (tickets_0_fdw t_1) INNER JOIN (boarding_passes_0_fdw bp_1)
              Network: FDW bytes sent=472 received=2884064
        ->  Sort (actual rows=472843 loops=1)
              Sort Key: t_2.passenger_name, t_2.ticket_no
              Sort Method: external merge  Disk: 21152kB
              Network: FDW bytes received=22753536
              ->  Hash Join (actual rows=472843 loops=1)
                    Hash Cond: ((bp_2.ticket_no = t_2.ticket_no) AND (bp_2.book_ref =
 t_2.book_ref))
                    Network: FDW bytes received=22753536
                    ->  Seq Scan on boarding_passes_1 bp_2 (actual rows=472843 loops=1)
                    ->  Hash (actual rows=207058 loops=1)
                          Buckets: 65536  Batches: 8  Memory Usage: 2264kB
                          Network: FDW bytes received=22753536
                          ->  Seq Scan on tickets_1 t_2 (actual rows=207058 loops=1)
                                Network: FDW bytes received=22753536
        ->  Foreign Scan (actual rows=474715 loops=1)
              Relations: (tickets_2_fdw t_3) INNER JOIN (boarding_passes_2_fdw bp_3)
              Network: FDW bytes sent=476 received=2884120
        ->  Foreign Scan (actual rows=473980 loops=1)
              Relations: (tickets_3_fdw t_4) INNER JOIN (boarding_passes_3_fdw bp_4)
```

```
            Network: FDW bytes sent=476 received=25745384
```

The situation considerably improved, the result was received on the query source node, and then final filtering, grouping and joining data were done.

For the `ticket_no` sharding key, the source query plan looks like this:

```
HashAggregate (actual rows=424 loops=1)
  Group Key: t.ticket_no
  Filter: ((max(bp.boarding_no) = 1) AND (count(*) > 1))
  Batches: 85  Memory Usage: 4265kB  Disk Usage: 111824kB
  Rows Removed by Filter: 700748
  Network: FDW bytes sent=1188 received=77103620
  ->  Append (actual rows=1894295 loops=1)
        Network: FDW bytes sent=1188 received=77103620
        ->  Async Foreign Scan (actual rows=473327 loops=1)
              Relations: (tickets_0_fdw t_1) INNER JOIN (boarding_passes_0_fdw bp_1)
              Network: FDW bytes sent=394
        ->  Hash Join (actual rows=472632 loops=1)
              Hash Cond: (bp_2.ticket_no = t_2.ticket_no)
              Network: FDW bytes received=77103620
              ->  Seq Scan on boarding_passes_1 bp_2 (actual rows=472632 loops=1)
              ->  Hash (actual rows=206712 loops=1)
                    Buckets: 65536  Batches: 4  Memory Usage: 3654kB
                    Network: FDW bytes received=23859576
                    ->  Seq Scan on tickets_1 t_2 (actual rows=206712 loops=1)
                          Network: FDW bytes received=23859576
        ->  Async Foreign Scan (actual rows=475755 loops=1)
              Relations: (tickets_2_fdw t_3) INNER JOIN (boarding_passes_2_fdw bp_3)
              Network: FDW bytes sent=397
        ->  Async Foreign Scan (actual rows=472581 loops=1)
              Relations: (tickets_3_fdw t_4) INNER JOIN (boarding_passes_3_fdw bp_4)
              Network: FDW bytes sent=397
```

We can see that table joining is done on shards, while data filtering, grouping and aggregation are done on the query source node. The source query does not need to be modified in this case.

3.3.6. q6 Query

For each ticket booked a week ago from now, this query displays all the included flight segments, together with connection time.

```
SELECT tf.ticket_no,f.departure_airport,
      f.arrival_airport,f.scheduled_arrival,
      lead(f.scheduled_departure) OVER w AS next_departure,
      lead(f.scheduled_departure) OVER w - f.scheduled_arrival AS gap
FROM bookings.bookings b
JOIN bookings.tickets t
 ON t.book_ref = b.book_ref
JOIN bookings.ticket_flights tf
 ON tf.ticket_no = t.ticket_no
JOIN bookings.flights f
 ON tf.flight_id = f.flight_id
WHERE b.book_date = bookings.now()::date - INTERVAL '7 day'

WINDOW w AS (
PARTITION BY tf.ticket_no
ORDER BY f.scheduled_departure);
```

For this query, the type of the `book_date` column must be cast from the `timestamptz` to `date`. When casting types, PostgreSQL casts the column data type to the data type specified in the filtering condition, but not vice versa. Therefore, Shardman must first get all the data from other shards, cast the type and apply filtering only after that. The query plan looks like this:

```
WindowAgg (actual rows=26 loops=1)
  Network: FDW bytes sent=1750 received=113339240
  ->  Sort (actual rows=26 loops=1)
        Sort Key: tf.ticket_no, f.scheduled_departure
        Sort Method: quicksort  Memory: 27kB
        Network: FDW bytes sent=1750 received=113339240
        ->  Append (actual rows=26 loops=1)
              Network: FDW bytes sent=1750 received=113339240
              ->  Hash Join (actual rows=10 loops=1)
                    Hash Cond: (t_1.book_ref = b.book_ref)
                    Network: FDW bytes sent=582 received=37717376
              ->  Hash Join (actual rows=6 loops=1)
                    Hash Cond: (t_2.book_ref = b.book_ref)
                    Network: FDW bytes sent=582 received=37700608
              ->  Hash Join (actual rows=2 loops=1)
                    Hash Cond: (t_3.book_ref = b.book_ref)
                    Network: FDW bytes sent=586 received=37921256
              ->  Nested Loop (actual rows=8 loops=1)
                    ->  Nested Loop (actual rows=8 loops=1)
                          ->  Hash Join (actual rows=2 loops=1)
                                Hash Cond: (t_4.book_ref = b.book_ref)
                                ->  Seq Scan on tickets_3 t_4 (actual rows=207118
 loops=1)
                    ->  Index Scan using flights_pkey on flights f (actual rows=1
 loops=8)
                          Index Cond: (flight_id = tf_4.flight_id)
```

Pay attention to the number of bytes received from other cluster shards and to the sequential scan of the `tickets` table. Let's try to rewrite the query to avoid the type cast.

The idea is pretty simple: the interval will be computed at the application level rather than at the database level, and the data of the `timestamptz` type will be readily passed to the query. Besides, creation of an additional index can help:

```
CREATE INDEX if not exists bookings_date_idx ON bookings.bookings(book_date);
```

For the `book_ref` sharding key, the query looks like this:

```
SELECT tf.ticket_no,f.departure_airport,
      f.arrival_airport,f.scheduled_arrival,
      lead(f.scheduled_departure) OVER w AS next_departure,
      lead(f.scheduled_departure) OVER w - f.scheduled_arrival AS gap
FROM bookings.bookings b
JOIN bookings.tickets t
 ON t.book_ref = b.book_ref
JOIN bookings.ticket_flights tf
 ON tf.ticket_no = t.ticket_no
AND tf.book_ref = t.book_ref -- <= added book_ref
JOIN bookings.flights f
 ON tf.flight_id = f.flight_id
WHERE b.book_date = '2016-10-06 14:00:00+00'
WINDOW w AS (
PARTITION BY tf.ticket_no
ORDER BY f.scheduled_departure);
```

This query has a different plan:

```
WindowAgg (actual rows=18 loops=1)
  Network: FDW bytes sent=2268 received=892
  ->  Sort (actual rows=18 loops=1)
        Sort Key: tf.ticket_no, f.scheduled_departure
        Sort Method: quicksort  Memory: 26kB
        Network: FDW bytes sent=2268 received=892
```

```
        ->  Append (actual rows=18 loops=1)
              Network: FDW bytes sent=2268 received=892
              ->  Nested Loop (actual rows=4 loops=1)
                    ->  Nested Loop (actual rows=4 loops=1)
                          ->  Nested Loop (actual rows=1 loops=1)
                                ->  Bitmap Heap Scan on bookings_0 b_1
                                      Heap Blocks: exact=1
                                      ->  Bitmap Index Scan on bookings_0_book_date_idx
                                ->  Index Only Scan using tickets_0_pkey on tickets_0
t_1
                                      Index Cond: (book_ref = b_1.book_ref)
                                      Heap Fetches: 0
                          ->  Index Only Scan using ticket_flights_0_pkey on
ticket_flights_0 tf_1
                                Heap Fetches: 0
                    ->  Index Scan using flights_pkey on flights f (actual rows=1
loops=4)
                          Index Cond: (flight_id = tf_1.flight_id)
              ->  Async Foreign Scan (actual rows=14 loops=1)
                    Network: FDW bytes sent=754 received=892
              ->  Async Foreign Scan (actual rows=0 loops=1)
                    Network: FDW bytes sent=757 -- received=0!
              ->  Async Foreign Scan (actual rows=0 loops=1)
                    Network: FDW bytes sent=757 -- received=0!
```

This is much better. First, the whole table is not scanned, `Index Only Scan` is only included. Second, it is clear how much the amount of network data transfer between nodes is reduced.

3.3.7. q7 Query

Assume that statistics is needed showing how many passengers there are per booking. To find this out, let's first compute the number of passengers in each booking and then the number of bookings with each number of passengers.

```
SELECT tt.cnt, count(*)
FROM (
   SELECT count(*) cnt
   FROM bookings.tickets t
   GROUP BY t.book_ref
   ) tt
GROUP BY tt.cnt
ORDER BY tt.cnt;
```

This query processes all the data in the `tickets` and `bookings` tables. So intensive network data exchange between shards cannot be avoided. Also note that the value of the `work_mem` parameter must be pretty high to avoid the use of disk when joining tables. So let's change the value of `work_mem` in the cluster:

```
shardmanctl set work_mem='256MB';
```

The query plan for the `ticket_no` sharding key is as follows:

```
GroupAggregate (actual rows=5 loops=1)
  Group Key: tt.cnt
  Network: FDW bytes sent=798 received=18338112
  ->  Sort (actual rows=593433 loops=1)
        Sort Key: tt.cnt
        Sort Method: quicksort  Memory: 57030kB
        Network: FDW bytes sent=798 received=18338112
        ->  Subquery Scan on tt (actual rows=593433 loops=1)
              Network: FDW bytes sent=798 received=18338112
              ->  Finalize HashAggregate (actual rows=593433 loops=1)
                    Group Key: t.book_ref
```

```
                    Batches: 1  Memory Usage: 81953kB
                    Network: FDW bytes sent=798 received=18338112
                    ->  Append (actual rows=763806 loops=1)
                          Network: FDW bytes sent=798 received=18338112
                          ->  Async Foreign Scan (actual rows=190886 loops=1)
                                Relations: Aggregate on (tickets_0_fdw t)
                                Network: FDW bytes sent=266 received=1558336
                          ->  Async Foreign Scan (actual rows=190501 loops=1)
                                Relations: Aggregate on (tickets_1_fdw t_1)
                                Network: FDW bytes sent=266
                          ->  Async Foreign Scan (actual rows=191589 loops=1)
                                Relations: Aggregate on (tickets_2_fdw t_2)
                                Network: FDW bytes sent=266
                          ->  Partial HashAggregate (actual rows=190830 loops=1)
                                Group Key: t_3.book_ref
                                Batches: 1  Memory Usage: 36881kB
                                Network: FDW bytes received=4981496
                                ->  Seq Scan on tickets_3 t_3 (actual rows=207118
 loops=1)
                                      Network: FDW bytes received=4981496
```

The query plan for the `book_ref` sharding key is as follows:

```
Sort (actual rows=5 loops=1)
  Sort Key: (count(*))
  Sort Method: quicksort  Memory: 25kB
  Network: FDW bytes sent=798 received=14239951
  ->  HashAggregate (actual rows=5 loops=1)
        Group Key: (count(*))
        Batches: 1  Memory Usage: 40kB
        Network: FDW bytes sent=798 received=14239951
        ->  Append (actual rows=593433 loops=1)
              Network: FDW bytes sent=798 received=14239951
              ->  GroupAggregate (actual rows=148504 loops=1)
                    Group Key: t.book_ref
                    ->  Index Only Scan using tickets_0_book_ref_idx on tickets_0 t
 (rows=207273)
                          Heap Fetches: 0
              ->  Async Foreign Scan (actual rows=148256 loops=1)
                    Relations: Aggregate on (tickets_1_fdw t_1)
                    Network: FDW bytes sent=266 received=1917350
              ->  Async Foreign Scan (actual rows=148270 loops=1)
                    Relations: Aggregate on (tickets_2_fdw t_2)
                    Network: FDW bytes sent=266
              ->  Async Foreign Scan (actual rows=148403 loops=1)
                    Relations: Aggregate on (tickets_3_fdw t_3)
                    Network: FDW bytes sent=266
```

The query plans differ first by the order of joining tables and by the computation of aggregates.

For the `ticket_no` sharding key, all the partially aggregated data of the joined tables is received (17 Mb), and all the rest of processing is performed on the query source node.

For the `book_ref` sharding key, as it is included in the query, most of the computation of aggregates is performed on the nodes and only the result (13 Mb) is returned to the query source node, which is then finalized.

3.3.8. q8 Query

This query answers the question: which are the most frequent combinations of first and last names in bookings and what is the ratio of the passengers with such names to the total number of passengers. A window function is used to get the result:

```
SELECT passenger_name,
      round( 100.0 * cnt / sum(cnt) OVER (), 2)
   AS percent
FROM (
 SELECT passenger_name,
        count(*) cnt
 FROM bookings.tickets
 GROUP BY passenger_name
) t
ORDER BY percent DESC;
```

For both sharding keys, the query plan looks like this:

```
Sort (actual rows=27909 loops=1)
  Sort Key: (round(((100.0 * ((count(*)))::numeric) / sum((count(*))) OVER (?)), 2))
 DESC
  Sort Method: quicksort  Memory: 3076kB
  Network: FDW bytes sent=816 received=2376448
  ->  WindowAgg (actual rows=27909 loops=1)
        Network: FDW bytes sent=816 received=2376448
        ->  Finalize HashAggregate (actual rows=27909 loops=1)
              Group Key: tickets.passenger_name
              Batches: 1  Memory Usage: 5649kB
              Network: FDW bytes sent=816 received=2376448
              ->  Append (actual rows=74104 loops=1)
                    Network: FDW bytes sent=816 received=2376448
                    ->  Partial HashAggregate (actual rows=18589 loops=1)
                          Group Key: tickets.passenger_name
                          Batches: 1  Memory Usage: 2833kB
                          ->  Seq Scan on tickets_0 tickets (actual rows=207273
 loops=1)
                    ->  Async Foreign Scan (actual rows=18435 loops=1)
                          Relations: Aggregate on (tickets_1_fdw tickets_1)
                          Network: FDW bytes sent=272 received=2376448
                    ->  Async Foreign Scan (actual rows=18567 loops=1)
                          Relations: Aggregate on (tickets_2_fdw tickets_2)
                          Network: FDW bytes sent=272
                    ->  Async Foreign Scan (actual rows=18513 loops=1)
                          Relations: Aggregate on (tickets_3_fdw tickets_3)
                          Network: FDW bytes sent=272
```

The plan shows that the data preprocessing, table joins and partial aggregation are performed on shards, while the final processing is performed on the query source node.

3.3.9. q9 Query

This query answers the question: who traveled from Moscow (SVO) to Novosibirsk (OVB) on seat 1A the day before yesterday, and when was the ticket booked. The day before yesterday is computed from the function `booking.now` rather than from the current date. The query in the non-distributed schema is as follows:

```
SELECT
   t.passenger_name,
   b.book_date v
FROM bookings b
JOIN tickets t ON
   t.book_ref = b.book_ref
JOIN boarding_passes bp
   ON bp.ticket_no = t.ticket_no
JOIN flights f ON
   f.flight_id = bp.flight_id
WHERE f.departure_airport = 'SVO'
```

```
AND f.arrival_airport = 'OVB'
AND f.scheduled_departure::date = bookings.now()::date - INTERVAL '2 day'
AND bp.seat_no = '1A';
```

As explained for the q6 Query, `INTERVAL` causes the type cast. Let's get rid of it and rewrite the query for the `book_ref` sharding key as follows:

```
SELECT
   t.passenger_name,
   b.book_date v
FROM bookings b
JOIN tickets t ON
   t.book_ref = b.book_ref
JOIN boarding_passes bp
   ON bp.ticket_no = t.ticket_no
   AND bp.book_ref = b.book_ref -- <= added book_ref
JOIN flights f ON
   f.flight_id = bp.flight_id
WHERE f.departure_airport = 'SVO'
AND f.arrival_airport = 'OVB'
AND f.scheduled_departure
  BETWEEN '2016-10-11 14:00:00+00' AND '2016-10-13 14:00:00+00'
AND bp.seat_no = '1A';
```

Let's also create a couple of additional indexes:

```
CREATE INDEX idx_boarding_passes_seats
     ON boarding_passes((seat_no::text));
CREATE INDEX idx_flights_sched_dep
     ON flights(departure_airport,arrival_airport,scheduled_departure);
```

As a result, the query plan appears pretty good:

```
Append (actual rows=1 loops=1)
  Network: FDW bytes sent=2484 received=102
  ->  Nested Loop (actual rows=1 loops=1)
        Join Filter: (bp_1.ticket_no = t_1.ticket_no)
        Rows Removed by Join Filter: 1
        ->  Nested Loop (actual rows=1 loops=1)
              ->  Hash Join (actual rows=1 loops=1)
                    Hash Cond: (bp_1.flight_id = f.flight_id)
                    ->  Bitmap Heap Scan on boarding_passes_0 bp_1 (actual rows=4919
 loops=1)
                          Recheck Cond: ((seat_no)::text = '1A'::text)
                          Heap Blocks: exact=2632
                          ->  Bitmap Index Scan on boarding_passes_0_seat_no_idx
 (actual rows=4919)
                                Index Cond: ((seat_no)::text = '1A'::text)
                    ->  Hash (actual rows=2 loops=1)
                          Buckets: 1024  Batches: 1  Memory Usage: 9kB
                          ->  Bitmap Heap Scan on flights f (actual rows=2 loops=1)
                                Recheck Cond:
                     ((departure_airport = 'SVO'::bpchar) AND (arrival_airport =
 'OVB'::bpchar) AND
                     (scheduled_departure >= '2016-10-11 14:00:00+00'::timestamp with
 time zone) AND
                     (scheduled_departure < '2016-10-13 14:00:00+00'::timestamp with
 time zone))
                                Heap Blocks: exact=2
                                ->  Bitmap Index Scan on idx_flights_sched_dep (actual
 rows=2 loops=1)
```

```
                                        Index Cond:
                                    ((departure_airport = 'SVO'::bpchar) AND
                                    (arrival_airport = 'OVB'::bpchar) AND
                          (scheduled_departure >= '2016-10-11 14:00:00+00'::timestamp with
time zone) AND
                          (scheduled_departure <= '2016-10-13 14:00:00+00'::timestamp with
time zone))
                   ->  Index Scan using bookings_0_pkey on bookings_0 b_1 (actual rows=1
loops=1)
                         Index Cond: (book_ref = bp_1.book_ref)
             ->  Index Scan using tickets_0_book_ref_idx on tickets_0 t_1 (actual rows=2
loops=1)
                   Index Cond: (book_ref = b_1.book_ref)
       ->  Async Foreign Scan (actual rows=0 loops=1)
             Relations: (((boarding_passes_1_fdw bp_2) INNER JOIN (flights f)) INNER JOIN
(tickets_1_fdw t_2)) INNER JOIN (bookings_1_fdw b_2)
             Network: FDW bytes sent=826 received=68
       ->  Async Foreign Scan (actual rows=0 loops=1)
             Relations: (((boarding_passes_2_fdw bp_3) INNER JOIN (flights f)) INNER JOIN
(tickets_2_fdw t_3)) INNER JOIN (bookings_2_fdw b_3)
             Network: FDW bytes sent=829 received=34
       ->  Async Foreign Scan (actual rows=0 loops=1)
             Relations: (((boarding_passes_3_fdw bp_4) INNER JOIN (flights f)) INNER JOIN
(tickets_3_fdw t_4)) INNER JOIN (bookings_3_fdw b_4)
             Network: FDW bytes sent=829
```

It is clear from this plan that all the table joining was done on shards and the query source node received the result that did not contain rows as the data was located on one shard where the query was executed.

If this query were executed on a different shard, the plan would be the same, but the data for finalization would be received from the shard with the data.

3.4. Connecting and Working with a Shardman Cluster

As explained in Section 3.1.2, the cluster considered consists of four shards. This is how the data partitions of the main sharded table are distributed across shards.

For the `ticket_no` sharding key:

- `tickets_0` — shard-1 (cluster node `node1`)
- `tickets_1` — shard-2 (cluster node `node2`)
- `tickets_2` — shard-3 (cluster node `node3`)
- `tickets_3` — shard-4 (cluster node `node4`)

For the `book_ref` sharding key:

- `bookings_0` — shard-1 (cluster node `node1`)
- `bookings_1` — shard-2 (cluster node `node2`)
- `bookings_2` — shard-3 (cluster node `node3`)
- `bookings_3` — shard-4 (cluster node `node4`)

The examples below are provided for the `book_ref` sharding key, but the code in the subsections is suitable for the `ticket_no` sharding key.

Do not treat this code as optimal or use it in a production environment. It only shows how to implement creation of a connection pull to work with a Shardman cluster.

What is common for all the examples is the cluster connection string, which must contain node names, TCP port numbers, user name and password, database name for connection and a set of session parameters.

You can get this string using the shardmanctl utility. In the simplest case, the string looks like this:

```
$ shardmanctl getconnstr

dbname=postgres host=node1,node2,node3,node4 port=5432,5432,5432,5432
```

You can get this string to connect to cluster nodes or to create the connection pool in applications.

3.4.1. SQL

A few convenient functions and views are implemented in Shardman that add cluster observability by:

- Listing global tables
- Listing sharded tables
- Listing global sequences
- Finding the shard number from the value of the sharding key
- Perfroming ANALYZE for all the global and sharded tables in the cluster

3.4.1.1. Listing Global Tables

To display all global tables in the cluster, use the `shardman.global_tables` view:

```
postgres=# select
    relname as table_name,
    nspname as schema
from shardman.global_tables;

 table_name |  schema
------------+----------
 aircrafts  | bookings
 seats      | bookings
 airports   | bookings
 flights    | bookings
(4 rows)
```

3.4.1.2. Listing Sharded Tables

To display information on all the sharded tables in the cluster, query the `shardman.sharded_tables` view as follows:

```
postgres=# select
    relname as table_name,
    nparts  as partitions,
    colocated_with::oid::regclass::text as colocated_with,
    nspname as schema
from shardman.sharded_tables;

   table_name    | partitions | colocated_with |  schema
-----------------+------------+----------------+----------
 bookings        |          4 |                | bookings
 ticket_flights  |          4 | bookings       | bookings
 tickets         |          4 | bookings       | bookings
 boarding_passes |          4 | bookings       | bookings
(4 rows)
```

3.4.1.3. Listing Global Sequences

To display all the global sequences in the cluster, use the `shardman.sequence` view:

```
postgres=# select
    seqns   as schema,
    seqname as sequence_name,
    seqmin  as min_value,
```

```
    seqmax  as max_value,
    seqblk  as bulk_size
from shardman.sequence;

  schema  |      sequence_name      | min_value |      max_value      | bulk_size
----------+-------------------------+-----------+---------------------+-----------
 bookings | flights_flight_id_seq   |    262145 | 9223372036854775807 |     65536
(1 rows)
```

3.4.1.4. Finding the Shard Number from the Sharding Key Value

To display the name of the partition that contains data and the replication group name, call the `shardman.get_partition_for_value()` function. For example, for `book_ref` = 0369E5:

```
postgres=# select * from shardman.get_partition_for_value(
            'bookings'::regclass,
            '0369E5'::character(6));

 rgid | local_nspname | local_relname | remote_nspname | remote_relname
------+---------------+---------------+----------------+----------------
    1 | bookings      | bookings_0    | bookings       | bookings_0
```

This output shows that the data is in the `bookings_0` partition of the `bookings` table and is located on the node where the query was executed.

Let's create a query to display the name of the server where the partition with data is located. If we connect to the server that contains the partition, the server name is displayed as “current server”. If the data is on a different server, the hostname of the shard master is displayed:

```
SELECT p.rgid,
      local_relname AS partition_name,
      CASE
          WHEN r.srvid IS NULL THEN 'current server'
          ELSE (SELECT (SELECT split_part(kv, '=', 2)
                        FROM (SELECT unnest(fs.srvoptions) as kv) x
                        WHERE split_part(kv, '=', 1) = 'host')
                FROM shardman.repgroups rg
                         JOIN pg_catalog.pg_foreign_server AS fs ON fs.oid = rg.srvid
                WHERE rg.id = p.rgid)
          END       AS server_name
FROM shardman.get_partition_for_value('bookings'::regclass, '0369E5'::character(6)) p
        JOIN shardman.repgroups AS r ON
   r.id = p.rgid;


 rgid | partition_name |  server_name
------+----------------+----------------
    1 | bookings_0     | current server
(1 row)
```

Execution of this query with another value of the sharding key, 0369E6, produces the output:

```
 rgid | partition_name | server_name
------+----------------+-------------
    4 | bookings_3_fdw | node4
(1 row)
```

It is clear that the partition is on the `node4` node.

Also note that the `shardman.rgid` parameter allows you to find the number of the node with the connection session. To do this, execute the query:

```
SELECT pg_catalog.current_setting('shardman.rgid');
```

You can use this value to determine the location of connection sessions for queries like discussed in this section.

The `shardman.get_partition_for_value()` is mainly designed for administration purposes, to better understand the data topology.

As a rule, do not use administration functions when writing SQL code for data access.

3.4.1.5. Understanding How Partitions of Sharded Tables Are Distributed Across Shards

You can get the list of all sharded tables in the `bookings` schema, together with the number of partitions and their distribution across servers (shards) from Shardman metadata on any cluster node.

Consider the following query:

```
SELECT p.rel::regclass::text AS table_name,
      p.pnum,
      p.rgid,
      r.srvid,
      fs.srvname
FROM shardman.parts p
JOIN shardman.repgroups r
  ON p.rgid = r.id
LEFT OUTER JOIN pg_foreign_server fs
             ON r.srvid = fs.oid;
```

To learn how the data is distributed, let's combine this query with a subquery from Section 3.4.1.4:

```
SELECT p.rel::regclass AS table_name,
      st.nparts AS total_parts,
      p.pnum AS num_part,
      CASE
         WHEN r.srvid IS NULL THEN 'connected server'
      ELSE
       (SELECT split_part(kv, '=', 2)
       FROM (SELECT unnest(fs.srvoptions) AS kv) x
       WHERE split_part(kv, '=', 1) = 'host')
       END AS server_name
FROM shardman.parts p
       JOIN shardman.repgroups r
         ON p.rgid = r.id
       LEFT JOIN shardman.sharded_tables st
              ON p.rel = st.rel
       LEFT JOIN pg_foreign_server fs
              ON r.srvid = fs.oid
WHERE st.nspname = 'bookings'
ORDER BY table_name, num_part, server_name;
```

The output format is the table name, number of table partitions, partition number and server name:

```
        table_name        | total_parts | num_part |   server_name
--------------------------+-------------+----------+------------------
 bookings.bookings        |           4 |        0 | connected server
 bookings.bookings        |           4 |        1 | node2
 bookings.bookings        |           4 |        2 | node3
 bookings.bookings        |           4 |        3 | node4
 bookings.ticket_flights  |           4 |        0 | connected server
 bookings.ticket_flights  |           4 |        1 | node2
 bookings.ticket_flights  |           4 |        2 | node3
 bookings.ticket_flights  |           4 |        3 | node4
 bookings.tickets         |           4 |        0 | connected server
 bookings.tickets         |           4 |        1 | node2
 bookings.tickets         |           4 |        2 | node3
```

```
 bookings.tickets          |           4 |        3 | node4
 bookings.boarding_passes  |           4 |        0 | connected server
 bookings.boarding_passes  |           4 |        1 | node2
 bookings.boarding_passes  |           4 |        2 | node3
 bookings.boarding_passes  |           4 |        3 | node4
```

3.4.1.6. Collecting Statistics

To collect statistics for sharded and global tables, call the `shardman.global_analyze()` function. This function first collects statistics for all local partitions of sharded tables on each node and then broadcasts this statistics to other nodes. For a global table, the function first collects statistics on a certain node and then the statistics is broadcast to all the other nodes.

3.4.2. psql/libpq

To connect to a Shardman cluster and successfully work with it, it is sufficient to connect to one cluster node. To do this, first get the connection string.

The PostgreSQL documentation contains the description of the *cluster connection string*. The string can be specified using two formats: a keyword/value string and URI. Any of them can be used to connect to a Shardman cluster.

Some parameters must also be specified. The *list of parameters* is also available in the PostgreSQL documentation.

The value of `target_session_attrs` must be set to `read-write`. Only connections that allow read/write transactions are acceptable. If the connection to a cluster node is a success, the request “SHOW transaction_read_only;” is sent. If it returns `on`, the connection is closed. If several servers are specified in the connection string, other servers will be iterated through, the same way as with the failed connection attempt. The `target_session_attrs` parameter allows you to specify both masters and replicas of the Shardman cluster.

The following examples illustrate the connection:

```
psql -d  "dbname=postgres host=node3,node4,node2,node1 port=5432,5432,5432,5432
 user=username password=password target_session_attrs=read-write"
```

```
psql postgres://username:password@node1:5432,node2:5432,node3:5432,node4:5432/postgres?
target_session_attrs=read-write
```

3.4.3. Python

Connection to a Shardman cluster using the psycopg2 library looks like this:

```
import psycopg2
from psycopg2 import pool

pool = psycopg2.pool.SimpleConnectionPool(
                              min_size=1,
                              max_size=5,
                              user="pguser",
                              password="*****",
                              host="node1,node2,node3,node4",
                              port="5432,5432,5432,5432",
                              database="postgres",
                              target_session_attrs="read-write")

connection = pool.getconn()
```

A connection pool with the following parameters is created: the minimum and maximum number of connections `min_size`=1 and `max_size`=5. Then a specific connection to the cluster is selected, the user login and password are specified, as well as the list of nodes and TCP ports, database and connection parameters (see Section 3.4.2 for more information).

3.4.4. Java

Connection to a Shardman cluster using JDBC looks like this:

```
String url = "jdbc:postgresql://node1:5432,node2:5432,node3:5432,node4:5432/postgres?
loadBalanceHosts=true&targetServerType=primary";
Properties props = new Properties();


props.setProperty("user","postgres");
props.setProperty("password","********");

Connection conn = DriverManager.getConnection(url, props);
```

`url` contains the connection string, where all the available shard masters are listed. If no additional connection parameters of the JDBC driver are specified, connection to the cluster is performed through the first node available for connection. This is not always convenient. Therefore, connection string settings are added that allow using different cluster shards for different connections.

`loadBalanceHosts=true` allows iterating through nodes connecting to one of them, and `targetServerType=primary` indicates a need to only choose masters, then replicas can be added to the connection string.

3.4.5. Go

Ways to connect to a Shardman cluster for Go are pretty much the same as those accepted in Java or Python. You need to specify lists of nodes, their TCP ports, as well as connection parameters and choose a suitable driver.

One of these drivers for Go is *pgx* version 4 or 5.

The following is an example of a connection string and creation of a pool for connecting to a cluster:

```
dbURL := "postgres://username:password@node1:5432,node2:5432,node3:5432,node4:5432/
postgres?target_session_attrs=read-write")
dbPool, err := pgxpool.New(context.Background(), dbURL)
```

Also pay attention to the description of the `target_session_attrs` parameter.

Chapter 4. Additional Features

Shardman includes some additional features and modules imported from Postgres Pro Enterprise, namely AQO (Adaptive Query Optimization), CFS (Compressed File System) support, as well as pgpro_stats, pgpro_pwr, and pg_query_state modules.

4.1. AQO (Adaptive Query Optimization)

AQO is a Shardman extension that uses query execution statistics for improving cardinality estimation, which can optimize execution plans and, consequently, speed up query execution.

To turn on AQO:

1. Add `aqo` to the `shared_preload_libraries` parameter in sdmspec.json.
2. Create extension `aqo` on all nodes.

   ```
   SET shardman.broadcast_ddl TO ON;
   CREATE EXTENSION aqo;
   RESET shardman.broadcast_ddl;
   ```

3. Set `aqo.mode` for `learn` and run queries that you want to optimize with `EXPLAIN ANALYZE` until the plan stops changing.

   ```
   BEGIN;
   SET aqo.mode = 'learn';
   EXPLAIN ANALYZE <query>
   RESET aqo.mode;
   COMMIT;
   ```

   Note that aqo statistics is collected separately on all nodes in a Shardman cluster. So you need to repeat this process on each node in the cluster. Alternatively, you can set `aqo.mode` to `learn` and run your application for some time and later turn it back to the default mode (`controlled`).

> **Note**
>
> AQO will not be activated if you join less than `aqo.join_threshold` relations (3 by default).

Complete aqo documentation can be found *here*.

4.2. CFS (Compressed File System)

CFS enables page-level compression in Shardman. Compression can only be enabled for separate tablespaces. To compress a tablespace, you need to enable the compression option when creating this tablespace. For example:

```
CREATE TABLESPACE data LOCATION '/mnt/data-{rgid}' WITH (global, compression='zlib');
```

Now you can create tables and indexes in this tablespace or move existing table or index to it.

```
CREATE TABLE pgbench_branches (
       bid integer NOT NULL PRIMARY KEY USING INDEX TABLESPACE data,
       bbalance integer,
       filler character(88)
)
WITH (distributed_by = 'bid') TABLESPACE data;
```

> **Note**
>
> The `cfs_compression_ratio()` function returns the actual compression ratio for all segments of the compressed relation. However, it returns `NaN` for partitioned and foreign tables, so it works only for local partitions of a sharded table.

Complete CFS documentation can be found *here*.

4.3. pgpro_stats (Planning and Execution Statistics)

The pgpro_stats extension provides a means for tracking planning and execution statistics of all SQL statements executed by a server. In addition to tracking local statements, the pgpro_stats extension collects the aggregated statistics for distributed queries that involve multiple nodes in a cluster. This allows users to get a better understanding of how system resources are being used for distributed queries.

The architecture of Shardman additions to thepgpro_stats extension is described in Section 7.8.

Complete pgpro_stats documentation can be found *here*.

4.4. pgpro_pwr (Workload Reporting)

pgpro_pwr is designed to discover most resource-intensive activities in your database. This extension is based on *Postgres Pro's Statistics Collector* views and the *pgpro_stats* or *pg_stat_statements* extension.

To build workload reports using pgpro_pwr on a Shardman cluster, perform the following installation:

- Install the *dblink* module and the *pgpro_stats* extension on each Shardman cluster node.
- Install pgpro_pwr compatible with Shardman on each Shardman cluster node as follows:

  ```
  sudo apt install pgpro-pwr-sdm-14
  ```

Complete pgpro_pwr documentation can be found *here*.

4.5. pg_query_state

The `pg_query_state` module provides facility to know the current state of query execution on working backend and `silkworm` multiplexer workers.

Complete pg_query_state documentation can be found *here*.

Chapter 5. Performance Tuning

Performan#e tuning should be done during application development and include an accurate choice of hardware (for example, estimating the number of CPUs and memory per Shardman cluster node or tuning your storage), OS tuning (for example, tuning the `swappiness` parameter or network-related behavior) and DBMS tuning (choosing efficient configuration). But first of all, an application should be tested and tuned for distributed DBMS. This includes designing a distributed schema (or converting an existing schema to a distributed one), tuning queries, using connection poolers, caching and even checking performance issues related to possible serialization errors or Shardman node outage. The design of the schema should include accurate selection of a sharding key and a decision which tables should become global. Usually you select a sharding key so that:

1. Most of the queries filter out most of sharded table partitions.
2. Sharded tables are colocated and all joins of sharded tables are equi-joins on the sharding key.

These rules allow Shardman to efficiently exclude unused shards from queries and to push down joins to shards where the required data resides.

Each Shardman node operates as a usual DBMS server, so all standard recommendations for tuning PostgreSQL for production load remain in place. You should select `shared_buffers`, `work_mem`, `effective_cache_size` depending on resources available to DBMS. Keep in mind that if the cluster topology is set to `cross`, `Repfactor` instances run on a single node `w`. When all cluster nodes are online, replicas should not utilize a lot of CPUs. However, in case of node failure, masters for `Repfactor` replication groups can become running on one server, which can create significant load on it. While tuning the `max_connections` parameter, note that each transaction can initiate `n`-1 connections, where `n` is the number of replication groups in the cluster. When Silk is enabled, it is still true for transactions containing DML operations. When Silk is disabled, it is also true for read-only transactions.

Other parameters, which you perhaps would like to tune, are foreign server options. They can be set in `FDWOptions` section of Shardman configuration file. Parameters that significantly affect Shardman performance are `fetch_size`, `batch_size` and `async_capable`. When Silk transport is not enabled, `fetch_size` determines the number of records that are fetched from a remote server at once. When Silk transport is enabled, `fetch_size` currently does not have significant impact on the query execution. `batch_size` specifies how many rows can be combined in a single remote `INSERT` operation for a sharded table. `async_capable` allows asynchronous execution and should always be turned on (which is the default).

The shardman.gt_batch_size configuration parameter allows you to optimize the size of an intermediate buffer for `INSERT` and `DELETE` operations on global tables.

5.1. Examining Plans

Tuning query execution is better on a subset of production data that represents actual data distribution. Let's look at some sample plans.

```
EXPLAIN VERBOSE
SELECT bid,avg(abalance) FROM pgbench_accounts
WHERE bid IN (10,20,30,40)
GROUP BY bid;
                                                                QUERY PLAN

-------------------------------------------------------------------------------------------------
 Append  (cost=0.29..21.98 rows=4 width=36)
   ->  GroupAggregate  (cost=0.29..18.98 rows=1 width=36)
         Output: pgbench_accounts.bid, avg(pgbench_accounts.abalance)
         Group Key: pgbench_accounts.bid
         ->  Index Scan using pgbench_accounts_15_pkey on public.pgbench_accounts_15
 pgbench_accounts  (cost=0.29..18.96 rows=1 width=8)
               Output: pgbench_accounts.bid, pgbench_accounts.abalance
               Index Cond: (pgbench_accounts.bid = ANY ('{10,20,30,40}'::integer[]))
   ->  Async Foreign Scan  (cost=0.99..0.99 rows=1 width=36)
         Output: pgbench_accounts_1.bid, (avg(pgbench_accounts_1.abalance))
         Relations: Aggregate on (public.pgbench_accounts_16_fdw pgbench_accounts_1)
         Remote SQL: SELECT bid, avg(abalance) FROM public.pgbench_accounts_16 WHERE
 ((bid = ANY ('{10,20,30,40}'::integer[]))) GROUP BY 1
```

```
        Transport: Silk
   ->  Async Foreign Scan  (cost=0.99..0.99 rows=1 width=36)
         Output: pgbench_accounts_2.bid, (avg(pgbench_accounts_2.abalance))
         Relations: Aggregate on (public.pgbench_accounts_17_fdw pgbench_accounts_2)
         Remote SQL: SELECT bid, avg(abalance) FROM public.pgbench_accounts_17 WHERE
 ((bid = ANY ('{10,20,30,40}'::integer[]))) GROUP BY 1
         Transport: Silk
   ->  Async Foreign Scan  (cost=1.00..1.00 rows=1 width=36)
         Output: pgbench_accounts_3.bid, (avg(pgbench_accounts_3.abalance))
         Relations: Aggregate on (public.pgbench_accounts_19_fdw pgbench_accounts_3)
         Remote SQL: SELECT bid, avg(abalance) FROM public.pgbench_accounts_19 WHERE
 ((bid = ANY ('{10,20,30,40}'::integer[]))) GROUP BY 1
         Transport: Silk
 Query Identifier: -1714706980364121548
```

We see here that queries scanning three partitions are going to be sent to other nodes, coordinator data is also going to be scanned using `Index Scan`. We do not know what plan will be used on the remote side, but we see which queries will be sent (marked with `Remote SQL`). Note that `Transport: Silk` section is present in the foreign scan description. This indicates that Silk transport will be used to transfer results. We see that `Async` foreign scan is going to be used, which is fine. To discover which servers are used in the query, we should look at foreign tables definitions. For example, we can find out that `public.pgbench_accounts_19_fdw` is located on the `shardman_rg_2` server listening on `127.0.0.2:65432`:

```
SELECT srvname,srvoptions FROM pg_foreign_server s JOIN pg_foreign_table ON ftserver =
 s.oid
WHERE ftrelid = 'public.pgbench_accounts_19_fdw'::regclass;
-[ RECORD
 1 ]----------------------------------------------------------------------------------
srvname    | shardman_rg_2
srvoptions |
 {async_capable=on,batch_size=100,binary_format=on,connect_timeout=5,dbname=postgres,extende
```

Now we can connect to `shardman_rg_2` server and find out which plan is used for the local query which was shown by the above `EXPLAIN`:

```
EXPLAIN SELECT bid, avg(abalance)
FROM public.pgbench_accounts_19
WHERE ((bid = ANY ('{10,20,30,40}'::integer[]))) GROUP BY 1;

                                    QUERY PLAN
-------------------------------------------------------------------------------
 HashAggregate  (cost=3641.00..3641.01 rows=1 width=36)
   Group Key: bid
   ->  Seq Scan on pgbench_accounts_19  (cost=0.00..3141.00 rows=100000 width=8)
         Filter: (bid = ANY ('{10,20,30,40}'::integer[]))
```

While looking at distributed query plans, we can see that sometimes aggregates are not pushed down:

```
EXPLAIN VERBOSE
SELECT avg(abalance) FROM pgbench_accounts;

 QUERY PLAN

-----------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=156209.38..156209.39 rows=1 width=32) (actual
 time=590.359..590.371 rows=1 loops=1)
   Output: avg(pgbench_accounts.abalance)
   ->  Append  (cost=2891.00..156209.33 rows=20 width=32) (actual time=56.815..590.341
 rows=20 loops=1)
         ->  Partial Aggregate  (cost=2891.00..2891.01 rows=1 width=32) (actual
 time=56.812..56.813 rows=1 loops=1)
               Output: PARTIAL avg(pgbench_accounts.abalance)
```

```
               ->  Seq Scan on public.pgbench_accounts_0 pgbench_accounts
 (cost=0.00..2641.00 rows=100000 width=4) (actual time=0.018..38.478 rows=100000
 loops=1)
                     Output: pgbench_accounts.abalance
         ->  Partial Aggregate  (cost=23991.00..23991.01 rows=1 width=32) (actual
 time=75.133..75.134 rows=1 loops=1)
               Output: PARTIAL avg(pgbench_accounts_1.abalance)
               ->  Foreign Scan on public.pgbench_accounts_1_fdw pgbench_accounts_1
 (cost=100.00..23741.00 rows=100000 width=4) (actual time=41.281..67.293 rows=100000
 loops=1)
                     Output: pgbench_accounts_1.abalance
                     Remote SQL: SELECT abalance FROM public.pgbench_accounts_1
                     Transport: Silk
.....
```

Here `avg()` is calculated on the coordinator side. This can lead to a significant growth of data transfer between nodes. The actual data transfer can be monitored with the `NETWORK` parameter of `EXPLAIN ANALYZE` (look at the `Network received` field of the topmost plan node):

```
EXPLAIN (ANALYZE, VERBOSE, NETWORK)
SELECT avg(abalance) FROM pgbench_accounts

 QUERY PLAN

-------------------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=156209.38..156209.39 rows=1 width=32) (actual
 time=589.014..589.027 rows=1 loops=1)
   Output: avg(pgbench_accounts.abalance)
   Network: FDW bytes sent=3218 received=14402396
   ->  Append  (cost=2891.00..156209.33 rows=20 width=32) (actual time=52.111..588.999
 rows=20 loops=1)
         Network: FDW bytes sent=3218 received=14402396
         ->  Partial Aggregate  (cost=2891.00..2891.01 rows=1 width=32) (actual
 time=52.109..52.109 rows=1 loops=1)
               Output: PARTIAL avg(pgbench_accounts.abalance)
               ->  Seq Scan on public.pgbench_accounts_0 pgbench_accounts
 (cost=0.00..2641.00 rows=100000 width=4) (actual time=0.020..34.472 rows=100000
 loops=1)
                     Output: pgbench_accounts.abalance
         ->  Partial Aggregate  (cost=23991.00..23991.01 rows=1 width=32) (actual
 time=78.616..78.617 rows=1 loops=1)
               Output: PARTIAL avg(pgbench_accounts_1.abalance)
               Network: FDW bytes sent=247 received=2400360
               ->  Foreign Scan on public.pgbench_accounts_1_fdw pgbench_accounts_1
 (cost=100.00..23741.00 rows=100000 width=4) (actual time=42.359..69.984 rows=100000
 loops=1)
                     Output: pgbench_accounts_1.abalance
                     Remote SQL: SELECT abalance FROM public.pgbench_accounts_1
                     Transport: Silk
                     Network: FDW bytes sent=247 received=2400360
.....
```

In such cases, we sometimes can rewrite the query:

```
EXPLAIN  (ANALYZE, NETWORK, VERBOSE)
SELECT sum(abalance)::float/count(abalance) FROM pgbench_accounts where abalance is not
 null;

                                                                                QUERY PLAN
```

```
-------------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=12577.20..12577.22 rows=1 width=8) (actual
 time=151.632..151.639 rows=1 loops=1)
   Output: ((sum(pgbench_accounts.abalance))::double precision /
 (count(pgbench_accounts.abalance))::double precision)
   Network: FDW bytes sent=3907 received=872
   ->  Append  (cost=3141.00..12577.10 rows=20 width=16) (actual time=55.589..151.621
 rows=20 loops=1)
         Network: FDW bytes sent=3907 received=872
         ->  Partial Aggregate  (cost=3141.00..3141.01 rows=1 width=16) (actual
 time=55.423..55.424 rows=1 loops=1)
               Output: PARTIAL sum(pgbench_accounts.abalance), PARTIAL
 count(pgbench_accounts.abalance)
               ->  Seq Scan on public.pgbench_accounts_0 pgbench_accounts
 (cost=0.00..2641.00 rows=100000 width=4) (actual time=0.023..37.212 rows=100000
 loops=1)
                     Output: pgbench_accounts.abalance
                     Filter: (pgbench_accounts.abalance IS NOT NULL)
         ->  Async Foreign Scan  (cost=1.00..1.00 rows=1 width=16) (actual
 time=0.055..0.089 rows=1 loops=1)
               Output: (PARTIAL sum(pgbench_accounts_1.abalance)), (PARTIAL
 count(pgbench_accounts_1.abalance))
               Relations: Aggregate on (public.pgbench_accounts_1_fdw
 pgbench_accounts_1)
               Remote SQL: SELECT sum(abalance), count(abalance) FROM
 public.pgbench_accounts_1 WHERE ((abalance IS NOT NULL))
               Transport: Silk
               Network: FDW bytes sent=300 received=800
....
```

Rewriting the query here, we could decrease incoming network traffic generated by the query from 13 MB to 872 bytes.

Now let's look at two nearly identical joins.

```
EXPLAIN ANALYZE SELECT count(*) FROM pgbench_branches b
JOIN pgbench_history h ON b.bid = h.bid
WHERE mtime > '2023-03-14 10:00:00'::timestamptz AND b.bbalance > 0;

                                                                                QUERY PLAN

-------------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=8125.68..8125.69 rows=1 width=8) (actual time=27.464..27.543
 rows=1 loops=1)
   ->  Append  (cost=3.85..8125.63 rows=20 width=8) (actual time=0.036..27.475 rows=20
 loops=1)
         ->  Partial Aggregate  (cost=3.85..3.86 rows=1 width=8) (actual
 time=0.033..0.036 rows=1 loops=1)
               ->  Nested Loop  (cost=0.00..3.69 rows=67 width=0) (actual
 time=0.025..0.027 rows=0 loops=1)
                     Join Filter: (b.bid = h.bid)
                     ->  Seq Scan on pgbench_branches_0 b  (cost=0.00..1.01 rows=1
 width=4) (actual time=0.023..0.024 rows=0 loops=1)
                           Filter: (bbalance > 0)
                           Rows Removed by Filter: 1
                     ->  Seq Scan on pgbench_history_0 h  (cost=0.00..1.84 rows=67
 width=4) (never executed)
                           Filter: (mtime > '2023-03-14 10:00:00+03'::timestamp with
 time zone)
```

```
        ->  Partial Aggregate  (cost=222.65..222.66 rows=1 width=8) (actual
 time=3.969..3.973 rows=1 loops=1)
              ->  Nested Loop  (cost=200.00..222.43 rows=86 width=0) (actual
 time=3.736..3.920 rows=86 loops=1)
                    Join Filter: (b_1.bid = h_1.bid)
                    ->  Foreign Scan on pgbench_branches_1_fdw b_1
 (cost=100.00..101.22 rows=1 width=4) (actual time=1.929..1.932 rows=1 loops=1)
                    ->  Foreign Scan on pgbench_history_1_fdw h_1
 (cost=100.00..120.14 rows=86 width=4) (actual time=1.795..1.916 rows=86 loops=1)
                          Filter: (mtime > '2023-03-14 10:00:00+03'::timestamp with
 time zone)
        ->  Partial Aggregate  (cost=864.54..864.55 rows=1 width=8) (actual
 time=1.780..1.786 rows=1 loops=1)
              ->  Hash Join  (cost=200.01..864.53 rows=5 width=0) (actual
 time=1.769..1.773 rows=0 loops=1)
                    Hash Cond: (h_2.bid = b_2.bid)
                    ->  Foreign Scan on pgbench_history_2_fdw h_2
 (cost=100.00..760.81 rows=975 width=4) (never executed)
                          Filter: (mtime > '2023-03-14 10:00:00+03'::timestamp with
 time zone)
                    ->  Hash  (cost=100.00..100.00 rows=1 width=4) (actual
 time=1.740..1.742 rows=0 loops=1)
                          Buckets: 1024  Batches: 1  Memory Usage: 8kB
                          ->  Foreign Scan on pgbench_branches_2_fdw b_2
 (cost=100.00..100.00 rows=1 width=4) (actual time=1.738..1.738 rows=0 loops=1)
....
 Planning Time: 6.066 ms
 Execution Time: 33.851 ms
```

An interesting thing to note is that joining of `pgbench_branches` and `pgbench_history` partitions happens locally. It is a fetch-all plan — you can discover this by joins being located above foreign scans. It is not always evident why join pushdown does not happen. But if we look at the `pgbench_history` definition, we can see that `mtime` has the `timestamp without time zone` type.

```
\d pgbench_history
              Partitioned table "public.pgbench_history"
 Column |            Type             | Collation | Nullable | Default
--------+-----------------------------+-----------+----------+---------
 tid    | integer                     |           |          |
 bid    | integer                     |           |          |
 aid    | integer                     |           |          |
 delta  | integer                     |           |          |
 mtime  | timestamp without time zone |           |          |
 filler | character(22)               |           |          |
Partition key: HASH (bid)
Number of partitions: 20 (Use \d+ to list them.)
```

And in the above query, the string describing time is converted to `timestamp with timezone`. This requires comparison of `mtime` column (of `timestamp` type) and `timestamptz` value. The comparison is implicitly performed using the stable function `timestamp_gt_timestamptz`. A filter containing a non-immutable function cannot be pushed down to the foreign server, so join is executed locally. If we rewrite the query, converting the string to a timestamp, we can see not only that joins are pushed down, but also that remote queries can be executed asynchronously because foreign scans in a plan tree are located immediately below `Append`:

```
EXPLAIN ANALYZE SELECT count(*) FROM pgbench_branches b
JOIN pgbench_history h ON b.bid = h.bid
WHERE mtime > '2023-03-14 10:00:00'::timestamp AND b.bbalance > 0;
                                                                    QUERY PLAN

-------------------------------------------------------------------------------------------
```

```
 Finalize Aggregate  (cost=84.30..84.31 rows=1 width=8) (actual time=22.962..22.990
 rows=1 loops=1)
   ->  Append  (cost=3.85..84.25 rows=20 width=8) (actual time=0.196..22.927 rows=20
 loops=1)
         ->  Partial Aggregate  (cost=3.85..3.86 rows=1 width=8) (actual
 time=0.032..0.034 rows=1 loops=1)
               ->  Nested Loop  (cost=0.00..3.69 rows=67 width=0) (actual
 time=0.024..0.026 rows=0 loops=1)
                     Join Filter: (b.bid = h.bid)
                     ->  Seq Scan on pgbench_branches_0 b  (cost=0.00..1.01 rows=1
 width=4) (actual time=0.023..0.023 rows=0 loops=1)
                           Filter: (bbalance > 0)
                           Rows Removed by Filter: 1
                     ->  Seq Scan on pgbench_history_0 h  (cost=0.00..1.84 rows=67
 width=4) (never executed)
                           Filter: (mtime > '2023-03-14 10:00:00'::timestamp without
 time zone)
         ->  Async Foreign Scan  (cost=0.99..0.99 rows=1 width=8) (actual
 time=10.870..10.871 rows=1 loops=1)
               Relations: Aggregate on ((pgbench_branches_1_fdw b_1) INNER JOIN
 (pgbench_history_1_fdw h_1))
         ->  Async Foreign Scan  (cost=0.99..0.99 rows=1 width=8) (actual
 time=0.016..0.017 rows=1 loops=1)
               Relations: Aggregate on ((pgbench_branches_2_fdw b_2) INNER JOIN
 (pgbench_history_2_fdw h_2))
...
 Planning Time: 7.729 ms
 Execution Time: 14.603 ms
```

Note that foreign scans here include a list of joined relations. The expected cost of a foreign join is below 1.0. This is due to an optimistic technique of foreign join cost estimation, turned on by the `postgres_fdw.enforce_foreign_join` setting. Compare the total execution time (planning time + execution time) of the original and modified query — we could decrease it from about 40 to 22 ms.

Overall, while examining query plans, pay attention to what queries are actually pushed down. Some of the common reasons why joins cannot be pushed down is the absence of equi-joins on the sharding key and filters that contain non-immutable functions (possibly implicitly). If data is fetched from multiple replication groups, check that execution is mostly asynchronous.

5.1.1. `EXPLAIN` Parameters

This section lists Shardman-specific `EXPLAIN` parameters.

`NETWORK` (`boolean`)

Include the actual data transfer between nodes in the `EXPLAIN ANALYZE` output. If this parameter is not specified, `off` is assumed. If the parameter is specified without a value, `on` is assumed.

`REMOTE` (`boolean`)

Include plans for queries executed on foreign servers. If this parameter or its value is not specified, `on` is assumed.

5.2. DML Optimizations

While evaluating performance of DML statements, it is important to understand how they are processed in Shardman.

First of all, the execution of `INSERT` significantly differs from the execution of `UPDATE` and `DELETE` statements. The behavior of `INSERT` for sharded tables is controlled by the `batch_size` foreign server option, which can be set in `FDWOptions` section of Shardman configuration file. If `batch_size` is greater than 0, an `INSERT` in the same statement of several values that fall into the same foreign partition leads to the values being grouped together in batches of the specified size. Remote `INSERT` statements are prepared with the necessary number of parameters and then are executed with the given values. If the number of values does not match the number of prepared arguments, the modified statement with the necessary number of parameters is prepared again. A batch

insert optimization can fail if a transaction inserts records one by one or records routed to different foreign tables are intermixed in one `INSERT` statement. A batch is formed for a single foreign modify operation. It is sent to the remote server when the batch is filled or when the modify operation is over. The modify operation is over when we start routing tuples to another sharded table partition. So, for bulk load, inserting multiple values in a single `INSERT` command or using `COPY` is recommended (as `COPY` is optimized in a similar way). Large `batch_size` values allow issuing less `INSERT` statements on remote side and so significantly reduce communication costs. However, during construction of parameters for prepared `INSERT` statements, all inserted values should be copied to libpq-allocated memory. This can lead to unrestricted memory usage on the query coordinator side when several large `text` or `bytea` objects are loaded.

`UPDATE` and `DELETE` statements can be executed in a direct or indirect mode. A direct mode is used when a statement can be directly sent to a foreign server. In this mode, to modify a table on a remote server, a new statement is created based on the original `ModifyTable` plan node. Using a direct update is not always possible. In particular, it is impossible when some conditions should be evaluated locally. In this case, a much less efficient indirect modification is used. An indirect modification includes several statements. The first one is `SELECT FOR UPDATE` to lock remote rows. The second one is an actual `UPDATE` or `DELETE`, which is prepared once and then executed with different parameters for each row of the `SELECT FOR UPDATE` statement result after local filters are applied to the result. Evidently, direct modifications are much more efficient.

You can easily identify whether a DML statement is going to be executed in a direct or indirect mode looking at the query plan. A typical example of an indirect modification is:

```
EXPLAIN VERBOSE DELETE FROM pgbench_history
WHERE bid = 20 AND mtime > '2023-03-14 10:00:00'::timestamptz;
                                                 QUERY PLAN

-------------------------------------------------------------------------------------------------
 Delete on public.pgbench_history  (cost=100.00..142.66 rows=0 width=0)
   Foreign Delete on public.pgbench_history_17_fdw pgbench_history_1
     Remote SQL: DELETE FROM public.pgbench_history_17 WHERE ctid = $1
   ->  Foreign Scan on public.pgbench_history_17_fdw pgbench_history_1
 (cost=100.00..142.66 rows=4 width=10)
         Output: pgbench_history_1.tableoid, pgbench_history_1.ctid
         Filter: (pgbench_history_1.mtime > '2023-03-14 10:00:00+03'::timestamp with
 time zone)
         Remote SQL: SELECT mtime, ctid FROM public.pgbench_history_17 WHERE ((bid =
 20)) FOR UPDATE
```

If we had chosen another type for the string constant, this would become a direct update.

```
EXPLAIN VERBOSE DELETE FROM pgbench_history
WHERE bid = 20 AND mtime > '2023-03-14 10:00:00'::timestamp;
explain verbose delete from pgbench_history where bid = 20 and mtime > '2023-03-14
 10:00:00'::timestamp;
                                                          QUERY PLAN

-------------------------------------------------------------------------------------------------
 Delete on public.pgbench_history  (cost=100.00..146.97 rows=0 width=0)
   Foreign Delete on public.pgbench_history_17_fdw pgbench_history_1
   ->  Foreign Delete on public.pgbench_history_17_fdw pgbench_history_1
 (cost=100.00..146.97 rows=4 width=10)
         Remote SQL: DELETE FROM public.pgbench_history_17 WHERE ((mtime > '2023-03-14
 10:00:00'::timestamp without time zone)) AND ((bid = 20))
```

We see that in a direct update mode, only one statement is executed on the remote server.

5.2.1. DML Optimizations of Global Tables

The shardman.gt_batch_size configuration parameter, which you can tune, defines the size of an intermediate buffer used before sending data to a remote server.

`INSERT` uses the binary protocol and creates batches of the `shardman.gt_batch_size` size. Large values of the buffer size enable sending fewer network requests on the remote side and thus substantially reduce the connection costs. On the other hand,

large values of this parameter can increase memory consumption on the query coordinator side. Therefore, when specifying the buffer size, it is important to achieve a compromise between the connection costs and the allocated memory size.

For `UPDATE`, a query for each column and each row is created on the coordinator and sent to remote nodes.

For `DELETE`, a query for a batch of data of the `shardman.gt_batch_size` size is created on the coordinator and sent to remote nodes.

5.3. Time Synchronization

The algorithm that provides data consistency on all the cluster nodes uses the system clock installed on the hosts. Therefore, the transaction commit latency depends on clock drift on different hosts, as the coordinator always waits for the most lagging host to catch up. This makes it crucial that the time on all the connected nodes of a Shardman cluster are synchronized, as lack of synchronization may have a negative impact on Shardman performance by increasing the query latency.

First, to ensure time synchronization on all cluster nodes, install chrony daemon when deploying a new cluster.

```
sudo apt update
sudo apt install -y chrony
sudo systemctl enable --now chrony
```

Check that chrony is working properly.

```
chronyc tracking
```

Expected output:

```
Reference ID    : C0248F82 (Time100.Stupi.SE)
Stratum         : 2
Ref time (UTC)  : Tue Apr 18 11:50:44 2023
System time     : 0.000019457 seconds slow of NTP time
Last offset     : -0.000005579 seconds
RMS offset      : 0.000089375 seconds
Frequency       : 30.777 ppm fast
Residual freq   : -0.000 ppm
Skew            : 0.003 ppm
Root delay      : 0.018349268 seconds
Root dispersion : 0.000334640 seconds
Update interval : 1039.1 seconds
Leap status     : Normal
```

Note that managing the clock drift should be performed using the OS tools. Shardman diagnostic tools cannot be considered as the only and defining measurement utility.

To see if any major drift already exists, use the shardman.pg_stat_csn view that shows statistics on delays that take place during import of CSN snapshots. Its values are calculated when any related action is performed, or if any of the `shardman.trim_csnxid_map()` or `shardman.pg_oldest_csn_snapshot()` functions are called. These functions are called from the `csn trimmer routine` worker, therefore disabling this worker will result in these statistics not being collected.

The `csn_max_shift` field of the `shardman.pg_stat_csn` view shows the maximum registered snapshot CSN shift that caused a delay. This value defines the clock drift between the nodes in the cluster. A consecutive increase of this value means at least one's cluster system clock is out of sync. If this value exceeds 1000 (microseconds), it is recommended to check the time synchronization settings.

The same can be discovered if the `csn_total_import_delay` value increases while `csn_max_shift` remains unchanged. However, one-time increase may be due to single failures, non-related to the time issues.

Also, if the difference between `CSNXidMap_head_csn` and `shardman.oldest_csn` exceeds the `csn_snapshot_defer_time` parameter value and stays the same for a long time, it means that the `CSNSnapshotXidMap` map is full. It can result in a global transaction failure.

There are two main reasons for this issue.

- There is a transaction that runs for more than `csn_snapshot_defer_time` seconds and holds the entire cluster, holding the `VACUUM` process. In this case, `xid` field of the `shardman.oldest_csn` view is used to determine the transaction ID of this transaction, and the `rgid` field is used to determine the cluster node where this transaction is located.
- The `CSNSnapshotXidMap` map lacks capacity. During the normal operation the system might have transactions that exceed the `csn_snapshot_defer_time` value. To fix it, increase the `csn_snapshot_defer_time` time so that these transactions stay below this value.

If the `shardman.silk_tracepoints` configuration parameter is enabled, executing the `EXPLAIN` command for the distributed queries outputs the rows with information about how much time was spent on the query execution and what result it ended with, depending on the system components. These rows show metric values for the time spent on each component. The `net (qry)`, `net (1st tup)`, `net (last tup)` metrics calculate the difference between timestamps on different servers. This difference includes both time spent on a message transfer and the clock drift (positive or negative) between these servers. Therefore, these metrics can also help to determine whether there is any clock drift.

5.4. Distributed Query Diagnostics

Shardman enhances the `EXPLAIN` command so that it can provide additional information about a query if it is distributed. The work with the distributed tables is based on the plan nodes with the `ForeignScan` type. A query to each remote partition is determined by a single plan node of this type, with Shardman submitting additional information to the `EXPLAIN` blocks with the node description.

When executing a distributed query, the part of the plan (a subtree) that relates to a specific remote partition is serialized into an SQL statement. This process is known as deparsing. Then, this statement is sent to a remote server. The result of this query is the output of a `ForeignScan` node. It is used to gather the final results of the distributed query execution.

When the `VERBOSE` option of the `EXPLAIN` command is set to `on`, the `Remote SQL` field of the `ForeignScan` node block shows the statement sent to the remote server. Also, the `Server` field indicates the name of the server as it was specified during the cluster configuration and as it is displayed in `pg_foreign_server`, along with the transport method used to send this statement. The `transport` field can take two values: `silk` for the the enhanced interconnect Shardman mechanism, or `libpq` for sending via the standard PostgreSQL protocol.

5.4.1. Displaying Plans from the Remote Server

To see the execution plan that will be used on the remote server under the `EXPLAIN` block of the `ForeignScan` node, use the `postgres_fdw.foreign_explain` configuration parameter. The possible values are: `none` to exclude the `EXPLAIN` output from the remote servers, `full` to include the `EXPLAIN` output from the remote servers, `collapsed` to include the `EXPLAIN` output only for the first `ForeignScan` node under its `Append`/`MergeAppend`.

In production, it is recommended to disable this parameter (set it to `none`) or set it to `collapsed`, because obtaining any `EXPLAIN` information results in an additional implicit request to the server. Moreover, this request is executed in a synchronous mode, meaning the overall `EXPLAIN` output is built only once all the servers are sequentially queried. It can be a costly operationin case of a table with a large number of partitions.

Note that in case of the internal request for obtaining the `EXPLAIN` blocks for a remote plan, certain parameters are forcibly disabled, regardless of the parameters specified by a user when requesting `EXPLAIN` from the coordinator: `ANALYZE OFF`, `TIMING OFF`, `SUMMARY OFF`, `SETTINGS OFF`, `NETWORK OFF`. In this case, the `EXPLAIN` block of a remote plan will lack the corresponding metrics. Other `EXPLAIN` parameters (`FORMAT`, `VERBOSE`, `COSTS`, `BUFFERS`, `WAL`) are inherited from the coordinator.

If the subplan deparsing forms a statement that includes parameters (in the statement using symbols `$1`, `$2`, etc.), such a statement generally cannot be sent to the remote server to obtain `EXPLAIN` results. Therefore, the `ForeignExplain` blocks are not formed for the SQL statements with parameters.

5.4.2. Network Metrics and Latency

Setting the `NETWORK` option of the `EXPLAIN` command to `on` shows the network operation metrics for the plan nodes, including individual `ForeignScan` nodes and general nodes `Append` or `MergeAppend`.

For each plan node, the `FDW bytes`, `sent`, and `received` parameters are displayed for the outgoing and incoming traffic when the node is executed (regardless of the transport type). Note that these metrics are only output when the `ANALYZE` option of the `EXPLAIN` command is set to `on`.

When the `track_fdw_wait_timing` configuration parameter is enabled, the `wait_time` metric is also output. This metric summarizes all stages of the plan node execution, starting from the time the request is sent to the remote server, including the time spent on the execution itself and all the time until the complete set of results for that plan node is received.

Note that the `ForeignScan` node can operate in both synchronous and asynchronous modes. For the asynchronous execution, the node's execution function sends a request to the remote server and completes its execution without waiting for the result. The result is considered and processed later, upon receipt. In this scenario, the `wait_time` metric may not accurately reflect the actual execution time.

5.4.3. Query Tracing for Silk Transport

For the Silk transport, there is an option to output the extended debug information about tracing of a query passing from the coordinator to the remote server and back, including the results from the remote server. This information is only available if the `ANALYZE` option of the `EXPLAIN` command is set to `on`, and the `shardman.silk_tracepoints` configuration parameter is enabled.

When these parameters are enabled, each message transferred through the Silk transport (sending the SQL query, delivering it to the recipient, executing the query, and returning the execution result) is accompanied by an array of the timestamps measured at certain points in the pipeline. Once the query is executed, this information is displayed in the `EXPLAIN` block as rows starting with the word `Trace`. Each metric represents the difference between the timestamps at different points, in milliseconds:

Table 5.1. Query Tracing for Silk Transport Metrics

Interval	Description
bk shm->mp1 (qry)	The time taken to transfer an SQL query from the coordinator to its multiplexer via the shared memory.
mp1 shm->net (qry)	The time between receiving a query within the multiplexer from the shared memory and transferring it over the network.
net (qry)	The time spent by an SQL query to transfer over the network between the multiplexers.
mp2 recv->shm (qry)	The time between receiving an SQL query from the network and placing it in the queue in the shared memory on a remote multiplexer.
wk exec (1st tup)	The time spent to execute a query in `Silkworm` until the first row of the result is received.
wk exec (all tups)	The time spent to execute a query on `Silkworm` until the complete result is received.
wk->shm (1st tup)	The time taken to place the first row of the result into the `Silkworm` queue.
wk->shm (last tup)	The time taken to place the last row of the result into the `Silkworm` queue.
mp2 shm->net (1st tup)	The time between reading the first row of the result from the queue by the remote multiplexer and transferring it over the network.
net (1st tup)	The time spent to transfer the first row of the result over the network between the multiplexers.
mp1 recv->shm (1st tup)	The time between receiving the first row of the result from the network and placing it in the queue by the local multiplexer.
mp1 shm->bk (1st tup)	The time spent to retrieve the first row of the result from the queue by the coordinator.
mp2 shm->net (last tup)	The time between reading of the last row of the result from the queue by the remote multiplexer and transferring it over the network.
net (last tup)	The time spent to transfer the last row of the result over the network between the multiplexers.

Interval	Description
mp1 recv->shm (last tup)	The time between receiving the last row of the result from the network and placing it in the queue by the local multiplexer.
mp1 shm->bk (last tup)	The time taken by the coordinator to retrieve the last row o the result from the queue.
END-TO-END	The total time from sending the query to receiving the last row of the result. This approximately corresponds to the `wait_time`.

For the metrics `net (qry)`, `net (1st tup)`, and `net (last tup)`, the interval value is calculated as the difference between timestamps on different servers. Therefore, negative values may appear in these lines. This difference includes both time spent on a message transfer and the clock drift (positive or negative) between these servers. Thus, even with a slight drift, the values will be negative if its absolute value exceeds the duration of network transfer. Although it is not a bug, you should pay close attention to whether the cluster clocks are synchronized. For more information, see Section 5.3.

Chapter 6. Shardman Reference

The entries in this Reference are meant to provide in reasonable length an authoritative, complete, and formal summary about their respective subjects. More information about the use of Shardman, in narrative, tutorial, or example form, can be found in other parts of this book. See the cross-references listed on each reference page.

6.1. Functions

`shardman.broadcast_all_sql(`*`statement`* `text )`

Executes *`statement`* on every replication group.

> **Warning**
>
> The `shardman.broadcast_all_sql` function cannot be executed recursively. An attempt to do so results in an error “Command execution must be initiated by coordinator”.

`shardman.broadcast_query(`*`statement`* `text)`

Functions as shardman.broadcast_all_sql and returns an executed SQL *`statement`* results.

You may optionally set *`include_rgid`* to `true`, then the resulting tuples will have a number of the node the tuple originated from.

Example with *`include_rgid`* set to `false`:

```
select shardman.broadcast_query('SELECT relname from pg_class where relkind=''f''');
broadcast_query
-----------------
(t_1_fdw)
(t_2_fdw)
(t_0_fdw)
(t_2_fdw)
(t_0_fdw)
(t_1_fdw)
(6 rows)
```

Example with *`include_rgid`* set to `true`:

```
select shardman.broadcast_query('SELECT relname from pg_class where relkind=''f''',
 include_rgid => true);
broadcast_query
-----------------
(1,t_1_fdw)
(1,t_2_fdw)
(2,t_0_fdw)
(2,t_2_fdw)
(3,t_0_fdw)
(3,t_1_fdw)
(6 rows)
```

`shardman.broadcast_sql(`*`statement`* `text)`

Executes *`statement`* on every replication group but the current one.

> **Warning**
>
> The `shardman.broadcast_sql` function cannot be executed recursively. An attempt to do so results in an error “Command execution must be initiated by coordinator”.

`shardman.get_partition_for_value(relid oid, val variadic "any") → shardman.get_partition_for_value_type ( rgid int, local_nspname text, local_relname text, remote_nspname text, remote_relname text)`

Finds out which partition of a sharded table with oid `relid` the `val` belongs to. Returns NULL if the sharded table with oid `relid` does not exist. Returns the local schema name and relation names. If the value belongs to a partition stored in another replication group, also returns the remote schema and relation name. Returns only rgid if second-level partitioning is used.

Example:

```
select * from  shardman.get_partition_for_value('pgbench_branches'::regclass, 20);
 rgid | local_nspname |      local_relname      | remote_nspname |   remote_relname

------+---------------+-------------------------+----------------
+--------------------
    3 | public        | pgbench_branches_17_fdw | public         |
 pgbench_branches_17
```

`shardman.global_analyze()`

Performs cluster-wide analysis of sharded and global tables. First, this function executes ANALYZE on all local partitions of sharded tables on each node, then sends this statistics to other nodes. Next, it selects one node per global table and runs ANALYZE of this table on the selected node. Gathered statistics is broadcast to all other nodes in the cluster.

Example:

```
select shardman.global_analyze();
```

`shardman.attach_subpart(relid regclass,  snum int, partition_bound  text[])`

Attaches a previously detached subpartition number *snum* to a locally-partitioned table *relid* as a partition for the values within *partition_bound*. All subpartition tables and foreign tables should already exist. The *partition_bound* parameter is a pair of lower and upper bounds for the partition. If lower and upper bounds are both NULL, the subpartition is attached as the default one.

The operation is performed cluster-wide.

Example:

```
select shardman.attach_subpart('pgbench_history'::regclass, 1,$${'2021-01-01 00:00',
 '2022-01-01 00:00'}$$);
```

`shardman.create_subpart(relid regclass,  snum int, partition_bound  text[])`

Creates a subpartition number *snum* for a locally-partitioned table *relid* as a partition for the values within *partition_bound*. The *partition_bound* parameter is a pair of lower and upper bounds for the partition. If lower and upper bounds are both NULL, the subpartition is created as the default one. If the subpartition number is not specified, it will be selected as the next available partition number.

The operation is performed cluster-wide.

Examples:

```
select shardman.create_subpart('pgbench_history'::regclass, 1, $${'2021-01-01
 00:00', '2022-01-01 00:00'}$$);
select shardman.create_subpart('pgbench_history'::regclass, partition_bound:=$
$${'2022-01-01 00:00', '2023-01-01 00:00'}$$);
```

`shardman.detach_subpart(relid regclass,  snum int)`

Detaches a subpartition number *snum* from a locally-partitioned table *relid*. The partition number can be determined from the `shardman.subparts` view.

The operation is performed cluster-wide.

Example:

```
select shardman.detach_subpart('pgbench_history'::regclass, 1);
```

`shardman.drop_subpart(`*`relid`* `regclass,` *`snum`* `int)`

Drops subpartition number *`snum`* from locally-partitioned table *`relid`*. Partition number can be determined from the `shardman.subparts` view.

The operation is performed cluster-wide.

Example:

```
select shardman.drop_subpart('pgbench_history'::regclass, 1);
```

`shardman.am_coordinator()`

Returns whether the current session is the query coordinator. This check allows avoiding cases where global and sharded table triggers fire twice, first on the query coordinator, then on the remote nodes when data is modified.

```
SELECT shardman.am_coordinator();
am_coordinator
----------------
t
(1 row)
```

Example of the trigger function checking the query coordinator:

```
CREATE OR REPLACE FUNCTION trg_func() RETURNS TRIGGER
AS $$
BEGIN
IF NOT shardman.am_coordinator() THEN
        -- exit on non coordinator
        RETURN NEW;
END IF;
-- execute only by coordinator
RAISE WARNING 'Trigger fired!';
END
$$ LANGUAGE plpgsql;
```

`shardman.silk_statinfo_reset()`

Resets the values of the metrics with prefix `transferred_` and time-based metrics (with prefixes `read_efd_`, `write_efd_`, and `sort_time_`) in the shardman.silk_statinfo view.

`shardman.silk_routing`

Retrieves the results of the multiplexer `silk_connects`, `silk_backends`, and `silk_routes` functions.

`shardman.silk_rbc_snap`

Retrieves a consistent snapshot of all the connects, backends and routes that can be used by `silk_connects`, `silk_backends`, and `silk_routes` functions.

6.2. pgpro_stats Functions

pgpro_stats_sdm_stats_updated

returns a number of statistics entries received from each shard node and the timestamp of the last received statistics.

pgpro_stats_sdm_stats_updated_reset

resets the information specified above.

6.3. Advisory Lock Functions

Advisory locks are cluster-wide locks with no enforced use. Here is a list of functions to work with these locks.

Table 6.1. Advisory Lock Functions

Function	Returns
`shardman.advisory_xact_lock(key64 BIGINT);`	void
`shardman.advisory_xact_lock_shared(key64 BIGINT);`	void
`shardman.try_advisory_xact_lock(key64 BIGINT);`	bool
`shardman.try_advisory_xact_lock_shared( key64 BIGINT);`	bool
`shardman.advisory_xact_lock(key1 INT, key2 INT);`	void
`shardman.advisory_xact_lock_shared(key1 INT, key2 INT);`	void
`shardman.try_advisory_xact_lock(key1 INT, key2 INT);`	bool
`shardman.try_advisory_xact_lock_shared( key1 INT, key2 INT);`	bool

6.4. Views

6.4.1. Shardman-specific Views

6.4.1.1. `shardman.pg_stat_csn`

The `shardman.pg_stat_csn` view has one row showing statistics on delays that take place during import of CSN snapshots. These delays occur because system clocks on Shardman cluster nodes may be out of sync. The delays negatively impact the performance by increasing the query latency. The `shardman.pg_stat_csn` view allows tracking these delays. The view data is based on *The Statistics Collector*. The columns of the view are shown in Table 6.2.

Table 6.2. `shardman.pg_stat_csn` Columns

Name	Type	Description
`csn_snapshots_imported`	`bigint`	Total number of imported CSN snapshots
`csn_total_import_delay`	`interval`	Total duration of all delays in importing CSN snapshots, in microseconds
`csn_max_shift`	`bigint`	Maximum registered snapshot CSN shift that caused a delay
`local_oldest_csn`	`bigint`	CSN of the oldest transaction on the current node
`local_oldest_xid`	`xid`	XID of the oldest transaction on the current node
`indoubt_threshold_incidents`	`bigint`	Total number of transactions that exceeded the 10 seconds limit in the `inDoubt` state.
`CSNXidMap_head_csn`	`bigint`	Most recent CSN in the `CSNSnapshotXidMap`
`CSNXidMap_head_xid`	`xid`	XID corresponding to the most recent CSN in the `CSNSnapshotXidMap`

Name	Type	Description
CSNXidMap_tail_csn	bigint	Oldest CSN in the CSNSnapshotXidMap
CSNXidMap_tail_xid	xid	XID corresponding to the oldest CSN in the CSNSnapshotXidMap
stats_reset	timestamp with time zone	Time at which these statistics were last reset
CSNXidMap_last_trim	timestamp with time zone	Shows the last time when the shardman.trim_csnxid_map() function was called.

To reset CSN-related statistics, call the pg_stat_reset_shared function with the only text argument equal to csn.

> **Note**
>
> Shardman functionality related to CSN snapshots is work in progress. So anticipate changes to the corresponding views in future releases.

6.4.1.2. shardman.pg_indoubt_xacts

The view shardman.pg_indoubt_xacts displays information about transactions that are currently in the InDoubt state. An entry is removed when the transaction state changes.

Table 6.3. shardman.pg_indoubt_xacts Columns

Name	Type	Description
xid	xid	Transaction ID of a transaction in the InDoubt state
duration_msec	bigint	Time the transaction was in the InDoubt state, in milliseconds

When the shardman.pg_indoubt_xacts view is accessed, the internal transaction manager data structures are momentarily locked, and a copy is made for the view to display. This ensures that the view produces a consistent set of results, while not blocking normal operations longer than necessary. Nonetheless there could be some impact on database performance if this view is frequently accessed.

6.4.1.3. shardman.pg_stat_xact_time

The shardman.pg_stat_xact_time view shows statistics for the time spent on a transaction. The columns of the view are shown in Table 6.4.

Table 6.4. shardman.pg_stat_xact_time Columns

Name	Type	Description
overall_committed_xact_time	bigint	Overall time spent for the committed transactions
overall_aborted_xact_time	bigint	Overall time spent for the aborted transactions
overall_commit_time	bigint	Overall time spent for the committing transactions
local_commit_time	bigint	Overall time spent for writing to WAL for all the committed transactions
global_commit_time	bigint	Overall time spent for the distributed queries sending messages about transaction statuses for all the committed transactions

Name	Type	Description
overall_abort_time	bigint	Overall time spent for aborting transactions
local_abort_time	bigint	Overall time spent for writing to WAL for all the aborted transactions
global_abort_time	bigint	Overall time spent for the distributed queries sending messages about transaction statuses for all the aborted transactions
stats_reset	timestamp with time zone	Time at which these statistics were last reset

6.4.1.4. shardman.oldest_csn

The shardman.oldest_csn view has one row showing tuple csn, xid, and rgid containing CSN and XID of the oldest transaction in the cluster along with transaction's replication group number.

6.4.1.5. shardman.pg_stat_monitor

The shardman.pg_stat_monitor view has one row showing metrics of the Shardman monitor. The view data is based on *the Statistics Collector*. The columns of the view are shown in Table 6.5.

Table 6.5. shardman.pg_stat_monitor Columns

Name	Type	Description
resolved_deadlocks	bigint	Number of resolved distributed deadlocks
aborted_xacts	bigint	Number of aborted outdated prepared transactions
committed_xacts	bigint	Number of committed outdated prepared transactions
errors	bigint	Number of Shardman monitor errors
stats_reset	timestamp with time zone	Time at which these statistics were last reset

6.4.1.6. shardman.pg_stat_netusage

The shardman.pg_stat_netusage view has one row showing the cumulative network traffic between Shardman cluster nodes. The view data is based on *the Statistics Collector*. The columns of the view are shown in Table 6.6.

Table 6.6. shardman.pg_stat_netusage Columns

Name	Type	Description
netusage_recv_bytes	numeric	Total number of bytes received from other nodes through the network by each Shardman cluster node
netusage_sent_bytes	numeric	Total number of bytes sent to other nodes through the network by each Shardman cluster node
stats_reset	timestamp with time zone	Time at which these statistics were last reset

6.4.1.7. shardman.pg_stat_foreign_stat_bytes

The shardman.pg_stat_foreign_stat_bytes view shows the amount of statistics for foreign relations transferred over the network between Shardman cluster nodes. The view data is based on *The Statistics Collector*. The columns of the view are shown in Table 6.7.

Table 6.7. `shardman.pg_stat_foreign_stat_bytes` Columns

Name	Type	Description
`foreign_stat_recv_bytes`	`bigint`	Total number of bytes of the statistics for the foreign relations received from other nodes through the network by this node
`stats_reset`	`timestamp with time zone`	Time at which these statistics were last reset

6.4.1.8. Shardman-specific Global Views

6.4.1.8.1. `shardman.gv_sharded_tables`

This view displays information on all the sharded tables in the cluster.

6.4.1.8.2. `shardman.gv_global_tables`

This view displays information on all the global tables in the cluster.

6.4.2. Multiplexor Diagnostics Views

Views in this section provide various information related to Silk multiplexing. See Section 7.4 for details of `silkroad` multiplexing process.

6.4.2.1. `shardman.silk_routes`

The `shardman.silk_routes` view displays the current snapshot of the multiplexer routing table. The columns of the view are shown in Table 6.8.

Table 6.8. `shardman.silk_routes` Columns

Name	Type	Description
`hashvalue`	`integer`	Internal unique route identifier. Can be used to join with other Silk diagnostics views.
`origin_ip`	`inet`	IP address of the source node, which generated this route
`origin_port`	`int2`	External TCP connection port of the source node, which generated this route
`channel_id`	`integer`	Route sequential number within the node that generated this route. `channel_id` is unique for the pair `origin_ip` + `origin_port`. This pair is a unique node identifier within the Shardman cluster and hence the `origin_ip` + `origin_port` + `channel_id` tuple is a unique route identifier within the Shardman cluster.
`from_cn`	`integer`	Connect index in the `shardman.silk_connects` view for incoming routes, that is, not generated by this node, and -1 for routes generated by this node.
`backend_id`	`integer`	ID of the local process that is currently using this route: either the ID of the backend that generated this route or the ID of the `silkworm` worker assigned to this route. Equals -1 for queued incoming routes that have not been assigned a worker yet.

Name	Type	Description
pending_queue_bytes	bigint	Size of the queue of delayed messages (awaiting a free worker) for this route, in bytes. This value is only meaningful for incoming routes of each node that are not assigned to a worker yet.
pending_queue_messages	bigint	Number of messages in the queue of delayed messages (awaiting a free worker) for this route. This value is only meaningful for incoming routes of each node that are not assigned to a worker yet.
connects	integer[]	List of indexes of connects that are currently using this route.

6.4.2.2. shardman.silk_connects

The shardman.silk_connects view displays the current list of multiplexer connects. The columns of the view are shown in Table 6.9.

Table 6.9. shardman.silk_connects Columns

Name	Type	Description
cn_index	integer	Unique connect index
reg_ip	inet	“Registration” IP address of the node with which the connection is established. See Notes for details.
reg_port	int2	“Registration” TCP port of the node with which the connection is established. See Notes for details.
read_ev_active	boolean	true if the multiplexer is ready to receive data to the incoming queue. See Notes for details.
write_ev_active	boolean	true if the multiplexer filled the queue of non-sent messages and is waiting for it to get free. See Notes for details.
is_outgoing	boolean	true if the connection is outgoing, that is, created by connect, and false for incoming connects, that is, created by accept. Only used during the handshaking.
state	text	Current state of the connect: connected — if the connection is established, in progress — if the client has already connected, but handshaking has not happened yet, free — if the client has already disconnected, but the connect structure for the disconnected client has not been destroyed yet.
pending_queue_bytes	bigint	Size of the queue of non-sent messages for this connect, in bytes
pending_queue_messages	bigint	Number of messages in the queue of non-sent messages for this connect
blocked_by_backend	integer	ID of the backend that blocked this connect

Name	Type	Description
blocks_backends	integer[]	List of IDs of backends that are blocked by this connect
routes	integer[]	List of unique IDs of routes that use this connect
elapsed_time_write	bigint	Time from the last writing event of a connect
elapsed_time_read	bigint	Time from the last reading event of a connect

6.4.2.3. shardman.silk_backends

The shardman.silk_backends view displays the current list of processes of two kinds: backends that serve client connections and silkworm multiplexer workers, which interact with the multiplexer. The columns of the view are shown in Table 6.10.

Table 6.10. shardman.silk_backends Columns

Name	Type	Description
backend_id	integer	Unique backend/worker identifier
pid	integer	OS process ID
attached	boolean	Value is true if backend is attached to multiplexer, false otherwis
read_ev_active	boolean	true if the backend/worker is ready to receive data to the incoming queue. See Notes for details.
write_ev_active	boolean	true if the backend/worker filled the queue of non-sent messages and is waiting for it to get free. See Notes for details.
is_worker	boolean	true if this process is a silkworm multiplexer worker and false otherwise
pending_queue_bytes	bigint	Size of the queue of messages being sent to this backend/worker, in bytes
pending_queue_messages	bigint	Number of messages in the queue of messages being sent to this backend/worker
blocked_by_connect	integer	Index of the connect that blocks this backend/worker
blocks_connects	integer[]	List of indexes of connects that are blocked by this backend/worker
routes	integer[]	List of unique IDs of routes that are used by this backend/worker
in_queue_used	bigint	Number of queued data bytes in the incoming queue in the shared memory between the backend and multiplexer
out_queue_used	bigint	Number of queued data bytes in the outgoing queue in the shared memory between the backend and multiplexer
elapsed_time_write	bigint	Time from the last writing event of a backend
elapsed_time_read	bigint	Time from the last reading event of backend

6.4.2.4. shardman.silk_routing

The `shardman.silk_routing` view displays the results of the `shardman.silk_routing` function. Table 6.11.

Table 6.11. `shardman.silk_routing` Columns

Name	Type	Description
`hashvalue`	`integer`	Internal unique route identifier
`origin_ip`	`inet`	IP address of the node that generated this route
`origin_port`	`int2`	External TCP connection port of the source node that generated this route
`channel_id`	`integer`	Route sequential number within the node that generated this route
`is_reply`	`bool`	Index of the connect from which a message was received that caused generation of this route
`pending_queue_bytes`	`bigint`	Pending queue size, in bytes
`pending_queue_messages`	`bigint`	Number of pending queue messages
`backend_id`	`integer`	ID of the local process that is currently using this route: either the ID of the backend that generated this route or the ID of the `silkworm` worker assigned to this route. Equals -1 for queued incoming routes that have not been assigned a worker yet.
`backend_pid`	`integer`	Returns the process ID of the server process attached to the current session
`attached`	`boolean`	Value is true if backend is attached to multiplexer, false otherwis
`backend_rd_active`	`boolean`	`true` if the backend/worker is ready to receive data to the incoming queue. See Notes for details.
`backend_wr_active`	`boolean`	`true` if the backend/worker filled the queue of non-sent messages and is waiting for it to get free. See Notes for details.
`is_worker`	`boolean`	`true` if this process is a `silkworm` multiplexer worker and `false` otherwise
`backend_blocked_by_cn`	`integer`	Index of the connect that blocks this backend/worker
`blocks_connects`	`integer[]`	List of indexes of connects that are blocked by this backend/worker
`in_queue_used`	`bigint`	Number of queued data bytes in the incoming queue in the shared memory between the backend and multiplexer
`out_queue_used`	`bigint`	Number of queued data bytes in the outgoing queue in the shared memory between the backend and multiplexer
`connect_id`	`integer`	Unique connect index
`reg_ip`	`inet`	“Registration” IP address of the node with which the connection is established
`reg_port`	`int2`	“Registration” TCP port of the node with which the connection is established

Name	Type	Description
connect_rd_active	boolean	true if the multiplexer is ready to receive data to the incoming queue
connect_wr_active	boolean	true if the multiplexer filled the queue of non-sent messages and is waiting for it to get free
connect_is_outgoing	boolean	true if the connection is outgoing, that is, created by connect, and false for incoming connects, that is, created by accept. Only used during the handshaking.
connect_state	text	Current state of the connect: connected — if the connection is established, in progress — if the client has already connected, but handshaking has not happened yet, free — if the client has already disconnected, but the connect structure for the disconnected client has not been destroyed yet
connect_outgoing_queue_bytes	bigint	Size of the queue of non-sent messages for this connect, in bytes
connect_outgoing_queue_messages	bigint	Number of messages in the queue of non-sent messages for this connect
connect_blocked_by_bk	integer	ID of the backend that blocked this connect
blocks_backends	integer[]	List of IDs of backends that are blocked by this connect
connect_elapsed_time_write	bigint	Time from the last writing event of a connect
connect_elapsed_time_read	bigint	Time from the last reading event of a connect
backend_elapsed_time_write	bigint	Time from the last writing event of a backend
backend_elapsed_time_read	bigint	Time from the last reading event of a backend

6.4.2.5. shardman.silk_pending_jobs

The shardman.silk_pending_jobs view displays the current list of routes in the queue of delayed multiplexer jobs, that is, jobs that are not assigned to workers yet. The columns of the view are shown in Table 6.12.

Table 6.12. shardman.silk_pending_jobs Columns

Name	Type	Description
hashvalue	integer	Internal unique route identifier
origin_ip	inet	IP address of the node that generated this route
origin_port	int2	TCP connection port of the node that generated this route
channel_id	integer	Route sequential number within the node that generated this route
query	text	The first queued message

Name	Type	Description
pending_queue_bytes	bigint	Pending queue size, in bytes
pending_queue_messages	bigint	Number of pending queue messages

6.4.2.6. shardman.silk_statinfo

The shardman.silk_statinfo view displays the current multiplexer state information. The columns of the view are shown in Table 6.13.

Table 6.13. shardman.silk_statinfo Columns

Name	Type	Description
pid	integer	silkroad process ID
started_at	timestamp with time zone	Time when the silkroad backend was started.
transferred_bytes	json	JSON object of key value pairs, where the key is the name of the message type, and the value is total number of bytes sent for the message types with at least one message sent
transferred_pkts	json	JSON object of key value pairs, where the key is the name of the message type, and the value is the total number of sent messages for the message types with at least one message sent
transferred_max	json	JSON object of key value pairs, where the key is the name of the message type, and the value is the maximum size of a message for the message types with at least one message sent
memcxt_dpg_allocated	bigint	The mem_allocated value of the process in DPGMemoryContext
memcxt_top_allocated	bigint	The mem_allocated value of the process in TopMemoryContext
read_efd_max	bigint	Maximum reading time of the eventfd since reset
write_efd_max	bigint	Maximum writing time of the eventfd since reset
read_efd_total	bigint	Total reading time of the eventfd since reset
write_efd_total	bigint	Total writing time of the eventfd since reset
read_efd_count	bigint	Total number of reading events of the eventfd since reset
write_efd_count	bigint	Total number of writing events of the eventfd since reset
sort_time_max	bigint	Maximum time of sorting operations with the silk_flow_control enabled (any value other than none)
sort_time_total	bigint	Total time of sorting operations with the silk_flow_control enabled (any value other than none)

Name	Type	Description
sort_time_count	bigint	Total number of the sorting operations with the silk_flow_control enabled (any value other than none)

Note that read_efd_max, write_efd_max, read_efd_total, write_efd_total, read_efd_count, write_efd_count, sort_time_max, sort_time_total, and sort_time_count are only calculated if the shardman.silk_track_time configuration parameter is enabled.

6.4.2.7. shardman.silk_state

The shardman.silk_state view displays the current silkroad process state. The columns of the view are shown in Table 6.14.

Table 6.14. shardman.silk_state Columns

Name	Type	Description
state	text	State of the silkroad process

6.4.2.8. Notes

reg_ip and reg_port values are not actual network addresses, but the addresses by which the multiplexer accesses the node. They are determined during a handshake between multiplexer nodes and are equal to the corresponding parameters of an appropriate server in the *pg_foreign_server* table.

All the read_ev_active values are true and all the write_ev_active values are false when the multiplexer is in the idle state.

6.4.3. Global Views

Shardman has a list of global views based on the PostgeSQL local views. The definition of global view columns is the same as in its corresponding local view. Fetching from a global view returns a union of rows from the corresponding local views. The rows are fetched from each of their cluster nodes. Another difference is that the global views have an added column rgid. The rgid value shows the replication group ID of the cluster node from which a row is fetched.

6.4.3.1. Global Views for Statistics

Below is the list of the statistics-related global views with links to their corresponding local views:

Table 6.15. Statistics-related global and local views

Global view	Local view	Description
shardman.gv_stats	*pg_stats*	One row per planner statistics.
shardman.gv_stats_ext	*pg_stats_ext*	Provides access to information about each extended statistics object in the database.
shardman.gv_stats_ext_exprs	*pg_stats_ext_exprs*	Provides access to information about all expressions included in extended statistics objects.
shardman.gv_stat_activity	*pg_stat_activity*	One row per server process, showing information related to he current activity of that process.
shardman.gv_stat_replication	*pg_stat_replication*	One row per WAL sender process, showing statistics about replication to that sender's connected standby server.
shardman.gv_stat_replication_slots	*pg_stat_replication_slots*	One row per replication slot, showing statistics about the replication slot's usage.
shardman.gv_stat_subscription	*pg_stat_subscription*	One row per subscription for main worker (with null PID if the worker is not running), and additional rows for workers

Global view	Local view	Description
		handling the initial data copy of the subscribed tables.
shardman.gv_stat_ssl	*pg_stat_ssl*	One row per backend or WAL sender process, showing statistics about SSL usage on this connection.
shardman.gv_stat_gssapi	*pg_stat_gssapi*	One row per backend, showing information about GSSAPI usage on this connection.
shardman.gv_stat_archiver	*pg_stat_archiver*	One row only, showing statistics about the WAL archiver process's activity.
shardman.gv_stat_bgwriter	*pg_stat_bgwriter*	One row only, showing statistics about the background writer process's activity.
shardman.gv_stat_progress_analyze	*pg_stat_progress_analyze*	One row for each backend (including autovacuum worker processes) running ANALYZE, showing current progress.
shardman.gv_stat_progress_basebackup	*pg_stat_progress_basebackup*	One row for each WAL sender process streaming a base backup, showing current progress.
shardman.gv_stat_progress_cluster	*pg_stat_progress_cluster*	One row for each backend running `CLUSTER` or `VACUUM FULL`, showing current progress.
shardman.gv_stat_checkpointer	*pg_stat_checkpointer*	One row only, containing data about the checkpointer process of the cluster.
shardman.gv_statistic_ext	*pg_statistic_ext*	Extended planner statistics (definition)
shardman.gv_stat_progress_create_index	*pg_stat_progress_create_index*	One row for each backend running `CREATE INDEX` or `REINDEX`, showing current progress.
shardman.gv_stat_progress_vacuum	*pg_stat_progress_vacuum*	One row for each backend (including autovacuum worker processes) that is currently vacuuming
shardman.gv_stat_progress_copy	*pg_stat_progress_copy*	One row for each backend running COPY, showing current progress.
shardman.gv_stat_wal	*pg_stat_wal*	One row only, showing statistics about WAL activity.
shardman.gv_stat_database	*pg_stat_database*	One row per database, showing database-wide statistics about query cancels due to conflict with recovery on standby servers.
shardman.gv_stat_database_conflicts	*pg_stat_database_conflicts*	One row per database, showing database-wide statistics about query cancels occurring due to conflicts with recovery on standby servers. This view will only contain information on standby servers, since conflicts do not occur on primary servers.
shardman.gv_stat_all_tables	*pg_stat_all_tables*	One row for each table in the current database, showing statistics about accesses to that specific table.
shardman.gv_stat_sys_tables	*pg_stat_sys_tables*	Same as `pg_stat_all_tables`, except that only system tables are shown.

Global view	Local view	Description
shardman.gv_stat_user_tables	*pg_stat_user_tables*	Same as pg_stat_all_tables, except that only user tables are shown.
shardman.gv_stat_all_indexes	*pg_stat_all_indexes*	One row for each index in the current database, showing statistics about accesses to that specific index.
shardman.gv_stat_user_indexes	*pg_stat_user_indexes*	Same as pg_stat_all_indexes, except that only indexes on user tables are shown.
shardman.gv_stat_sys_indexes	*pg_stat_sys_indexes*	Same as pg_stat_all_indexes, except that only indexes on system tables are shown.
shardman.gv_stat_user_indexes	*pg_stat_user_indexes*	Same as pg_stat_all_indexes, except that only indexes on user tables are shown.
shardman.gv_statio_user_indexes	*pg_statio_user_indexes*	Same as pg_statio_all_indexes, except that only indexes on user tables are shown.
shardman.gv_statio_all_tables	*pg_statio_all_tables*	One row for each table in the current database, showing statistics about I/O on that specific table.
shardman.gv_statio_all_indexes	*pg_statio_all_indexes*	One row for each index in the current database, showing statistics about I/O on that specific index.
shardman.gv_statio_sys_indexes	*pg_statio_sys_indexes*	Same as pg_statio_all_indexes, except that only indexes on system tables are shown.
shardman.gv_statio_all_sequences	*pg_statio_all_sequences*	One row for each sequence in the current database, showing statistics about I/O on that specific sequence.
shardman.gv_statio_user_sequences	*pg_statio_user_sequences*	Same as pg_statio_all_sequences, except that only user sequences are shown.
shardman.gv_statio_sys_sequences	*pg_statio_sys_sequences*	Same as pg_statio_all_sequences, except that only system sequences are shown.
shardman.gv_statio_sys_tables	*pg_statio_sys_tables*	Same as pg_statio_all_tables, except that only system tables are shown.
shardman.gv_statio_user_tables	*pg_statio_user_tables*	Same as pg_statio_all_tables, except that only user tables are shown.
shardman.gv_stat_user_functions	*pg_stat_user_functions*	One row for each tracked function, showing statistics about executions of that function.
shardman.gv_stat_slru	*pg_stat_slru*	One row per SLRU, showing statistics of operations.
shardman.gv_stat_csn	shardman.pg_stat_csn	One row showing statistics on delays that take place during import of CSN snapshots.
shardman.gv_stat_monitor	shardman.pg_stat_monitor	One row showing metrics of the Shardman monitor.

Global view	Local view	Description
shardman.gv_stat_netusage	shardman.pg_stat_net_usage	One row showing the cumulative network traffic between Shardman cluster nodes.
shardman.gv_stat_xact_time	shardman.pg_stat_xact_time	One row showing statistics for the time spent on a transaction.
shardman.gv_silk_routes	shardman.silk_routes	One row showing the current snapshot of the multiplexer routing table.
shardman.gv_silk_connects	shardman.silk_connects	One row showing the current list of multiplexer connects.
shardman.gv_silk_backends	shardman.silk_backends	One row showing the current list of processes of two kinds: backends that serve client connections and `silkworm` multiplexer workers, which interact with the multiplexer.
shardman.gv_silk_pending_jobs	shardman.silk_pending_jobs	One row showing the current list of routes in the queue of multiplexer jobs that are not assigned to workers yet.
shardman.gv_silk_routing	shardman.silk_routing	One row showing the results of the `shardman.silk_routing` function.
shardman.gv_stats_sdm_statements	pgpro_stats_sdm_statements	This view allows accessing the aggregated statistics for the distributed queries. This view can only be created if Shardman is installed for the database that has pgpro_stats. The pgpro_stats must be created on all the cluster nodes for the global view to work.
shardman.gv_lock_graph	shardman.lock_graph	One row showing a graph of locks between processes on Shardman cluster nodes including external locks. This view is based on the `pg_locks` and `pg_prepared_xacts` *system views* and on the *`pg_stat_activity`* view of the Statistics Collector.
shardman.gv_stat_foreign_bytes	shardman.pg_stat_foreign_stat_bytes	One row showing the amount of statistics for foreign relations transferred over the network between Shardman cluster nodes.
shardman.gv_stat_wal_receiver	*pg_stat_wal_receiver*	One row, showing statistics about the WAL receiver from that receiver's connected server.
shardman.gv_stat_xact_all_tables	*pg_stat_xact_all_tables*	Similar to `pg_stat_all_tables`, but counts actions taken so far within the current transaction (which are *not* yet included in `pg_stat_all_tables` and related views). The columns for numbers of live and dead rows and vacuum and analyze actions are not present in this view.
shardman.gv_stat_xact_sys_tables	*pg_stat_xact_sys_tables*	Same as `pg_stat_xact_all_tables`, except that only system tables are shown.
shardman.gv_stat_xact_user_functions	*pg_stat_xact_user_functions*	Similar to `pg_stat_user_functions`, but counts only calls during the current transaction (which are *not* yet

Global view	Local view	Description
		included in `pg_stat_user_functions`).
`shardman.gv_stat_xact_user_tables`	*pg_stat_xact_user_tables*	Same as `pg_stat_xact_all_tables`, except that only user tables are shown.

6.4.3.2. Global Views for System Calalog

Below is the list of the global views that relate to the system catalog, and links to their corresponding local views:

Table 6.16. Global and local views for system catalog

Global view	Local view	Description
`shardman.gv_aggregate`	*pg_aggregate*	Stores information about aggregate functions
`shardman.gv_am`	*pg_am*	Relation access methods
`shardman.gv_amop`	*pg_amop*	Access method operators
`shardman.gv_amproc`	*pg_amproc*	Access method support functions
`shardman.gv_attrdef`	*pg_attrdef*	Column default values
`shardman.gv_attribute`	*pg_attribute*	Table columns (“attributes”)
`shardman.gv_auth_members`	*pg_auth_members*	Authorization identifier membership relationships
`shardman.gv_available_extension_versions`	*pg_available_extension_versions*	Specific extension versions that are available for installation
`shardman.gv_available_extensions`	*pg_available_extensions*	Extensions that are available for installation
`shardman.gv_cast`	*pg_cast*	Casts (data type conversions)
`shardman.gv_class`	*pg_class*	Tables, indexes, sequences, views (“relations”)
`shardman.gv_collation`	*pg_collation*	Collations (locale information)
`shardman.gv_config`	*pg_config*	Compile-time configuration parameters of the currently installed version of Postgres Pro
`shardman.gv_constraint`	*pg_constraint*	Check constraints, unique constraints, primary key constraints, foreign key constraints
`shardman.gv_conversion`	*pg_conversion*	Encoding conversion information
`shardman.gv_database`	*pg_database*	Databases within this database cluster
`shardman.gv_db_role_setting`	*pg_db_role_setting*	Per-role and per-database settings
`shardman.gv_efault_acl`	*pg_default_acl*	Default privileges for object types
`shardman.gv_depend`	*pg_depend*	Dependencies between database objects
`shardman.gv_description`	*pg_description*	Descriptions or comments on database objects
`shardman.gv_enum`	*pg_enum*	Enum label and value definitions
`shardman.gv_event_trigger`	*pg_event_trigger*	Event triggers
`shardman.gv_extension`	*pg_extension*	Installed extensions
`shardman.gv_file_setting`	*pg_file_settings*	Installed extensions

Global view	Local view	Description
shardman.gv_foreign_data_wrapper	*pg_foreign_data_wrapper*	Foreign-data wrapper definitions
shardman.gv_foreign_server	*pg_foreign_server*	Foreign server definitions
shardman.gv_foreign_table	*pg_foreign_table*	Additional foreign table information
shardman.gv_group	*pg_group*	Exists for backwards compatibility: it emulates a catalog that existed in Postgres Pro before version 8.1
shardman.gv_hba_file_rules	*pg_hba_file_rules*	Summary of the contents of the client authentication configuration file
shardman.gv_index	*pg_index*	Additional index information
shardman.gv_indexes	*pg_indexes*	Provides access to useful information about each index in the database
shardman.gv_inherits	*pg_inherits*	Table inheritance hierarchy
shardman.gv_init_privs	*pg_init_privs*	Object initial privileges
shardman.gv_language	*pg_language*	Languages for writing functions
shardman.gv_largeobject	*pg_largeobject*	Data pages for large objects
shardman.gv_largeobject_metadata	*pg_largeobject_metadata*	Metadata associated with large objects
shardman.gv_matviews	*pg_matviews*	Provides access to useful information about each materialized view in the database
shardman.gv_namespace	*pg_namespace*	Schemas
shardman.gv_opclass	*pg_opclass*	Access method operator classes
shardman.gv_operator	*pg_operator*	Operators
shardman.gv_opfamily	*pg_opfamily*	Access method operator families
shardman.gv_partitioned_table	*pg_partitioned_table*	Information about partition key of tables
shardman.gv_proc	*pg_proc*	Functions and procedures
shardman.gv_profile	*pg_profile*	Profiles, a set of authentication restrictions
shardman.gv_publication	*pg_publication*	Publications for logical replication
shardman.gv_publication_rel	*pg_publication_rel*	Relation to publication mapping
shardman.gv_publication_tables	*pg_publication_tables*	Information about the mapping between publications and information of tables they contain
shardman.gv_range	*pg_range*	Information about range types
shardman.gv_replication_origin	*pg_replication_origin*	Registered replication origins
shardman.gv_replication_origin_status	*pg_replication_origin_status*	Information about how far replay for a certain origin has progressed
shardman.gv_replication_slots	*pg_replication_slots*	Provides a listing of all replication slots that currently exist on the database cluster, along with their current state
shardman.gv_rewrite	*pg_rewrite*	Query rewrite rules

Global view	Local view	Description
shardman.gv_rules	*pg_rules*	Provides access to useful information about query rewrite rules
shardman.gv_seclabel	*pg_seclabel*	Security labels on database objects
shardman.gv_seclabels	*pg_seclabels*	Provides information about security labels
shardman.gv_sequence	*pg_sequence*	Information about sequences
shardman.gv_sequences	*pg_sequences*	Provides access to useful information about each sequence in the database
shardman.gv_settings	*pg_settings*	Provides access to run-time parameters of the server
shardman.gv_shdepend	*pg_shdepend*	Dependencies on shared objects
shardman.gv_shdescription	*pg_shdescription*	Comments on shared objects
shardman.gv_shseclabel	*pg_shseclabel*	Security labels on shared database objects
shardman.gv_subscription	*pg_subscription*	Logical replication subscriptions
shardman.gv_subscription_rel	*pg_subscription_rel*	Relation state for subscriptions
shardman.gv_tablespace	*pg_tablespace*	Tablespaces within this database cluster
shardman.gv_tables	*pg_tables*	Provides access to useful information about each table in the database
shardman.gv_prepared_xacts	*pg_prepared_xacts*	Provides information about transactions that are currently prepared for two-phase commit
shardman.gv_timezone_names	*pg_timezone_names*	List of time zone names that are recognized by `SET TIMEZONE`, along with their associated abbreviations, UTC offsets, and daylight-savings status
shardman.gv_timezone_abbrevs	*pg_timezone_abbrevs*	List of time zone abbreviations that are currently recognized by the datetime input routines
shardman.gv_transform	*pg_transform*	Transforms (data type to procedural language conversions)
shardman.gv_trigger	*pg_trigger*	Triggers
shardman.gv_ts_config	*pg_ts_config*	Text search configurations
shardman.gv_ts_config_map	*pg_ts_config_map*	Text search configurations' token mappings
shardman.gv_ts_dict	*pg_ts_dict*	Text search dictionaries
shardman.gv_ts_parser	*pg_ts_parser*	Text search parsers
shardman.gv_ts_template	*pg_ts_template*	Text search templates
shardman.gv_type	*pg_type*	Data types
shardman.gv_user_mapping	*pg_user_mapping*	Mappings of users to foreign servers
shardman.gv_user_mappings	*pg_user_mappings*	Provides access to information about user mappings
shardman.gv_views	*pg_views*	Provides access to useful information about each view in the database
shardman.gv_locks	*pg_locks*	Provides access to information about the locks held by active processes within the database server.

Global view	Local view	Description
`shardman.gv_shmem_allocations`	*pg_shmem_allocations*	Shows allocations made from the server's main shared memory segment.

6.5. SQL Commands

Shardman extends some DDL SQL commands supported by PostgreSQL to enable distributed DDL processing. This reference only describes Shardman-specific command syntax. See *PostgreSQL documentation* for a description of standard DDL SQL commands.

ALTER SEQUENCE

ALTER SEQUENCE — change the definition of a sequence generator

Synopsis

```
ALTER SEQUENCE [ IF EXISTS ] name
    [ AS data_type ]
    [ INCREMENT [ BY ] increment ]
    [ MINVALUE minvalue | NO MINVALUE ] [ MAXVALUE maxvalue | NO MAXVALUE ]
    [ RESTART [ [ WITH ] restart ] ]
    [ CACHE cache ] [ [ NO ] CYCLE ]
    [ OWNED BY { table_name.column_name | NONE } ]
ALTER SEQUENCE [ IF EXISTS ] name OWNER TO { new_owner | CURRENT_ROLE | CURRENT_USER |
 SESSION_USER }
ALTER SEQUENCE [ IF EXISTS ] name RENAME TO new_name
ALTER SEQUENCE [ IF EXISTS ] name SET SCHEMA new_schema
```

Description

`ALTER SEQUENCE` changes the parameters of an existing sequence generator. The extended forms of `ALTER SEQUENCE` are mostly the same as in PostgreSQL (see *`ALTER SEQUENCE`*) except for the following differences:

- The minimum sequence value parameter in Shardman works more like a lower boundary on the global interval of available values, so it can only be increased to make sure no duplicate numbers are generated.
- The `RESTART WITH` clause allows restarting a sequence at any arbitrary lower bound, but in this case, there is no guarantee that previously generated numbers will not repeat.
- Using both `RESTART WITH` and `MINVALUE` in a single statement is not permitted to avoid confusion.

Examples

Alter the block size parameter of a sequence called `serial`:

```
ALTER SEQUENCE serial SET (block_size = 8192);
```

See Also

CREATE SEQUENCE, Section 7.6

ALTER TABLE

ALTER TABLE — change the definition of a table

Synopsis

```
ALTER TABLE [ IF EXISTS ] [ ONLY ] name [ * ]
    action [, ... ]
ALTER TABLE [ IF EXISTS ] [ ONLY ] name [ * ]
    RENAME [ COLUMN ] column_name TO new_column_name
ALTER TABLE [ IF EXISTS ] [ ONLY ] name [ * ]
    RENAME CONSTRAINT constraint_name TO new_constraint_name
ALTER TABLE [ IF EXISTS ] name
    RENAME TO new_name
ALTER TABLE [ IF EXISTS ] name
    SET SCHEMA new_schema
ALTER TABLE ALL IN TABLESPACE name [ OWNED BY role_name [, ... ] ]
    SET TABLESPACE new_tablespace [ NOWAIT ]
ALTER TABLE [ IF EXISTS ] name
    ATTACH PARTITION partition_name { FOR VALUES partition_bound_spec | DEFAULT }
ALTER TABLE [ IF EXISTS ] name
    DETACH PARTITION partition_name [ CONCURRENTLY | FINALIZE ]

where action is one of:

    ADD [ COLUMN ] [ IF NOT EXISTS ] column_name data_type [ COLLATE collation ]
 [ column_constraint [ ... ] ]
    DROP [ COLUMN ] [ IF EXISTS ] column_name [ RESTRICT | CASCADE ]
    ALTER [ COLUMN ] column_name [ SET DATA ] TYPE data_type [ COLLATE collation ]
 [ USING expression ]
    ALTER [ COLUMN ] column_name SET DEFAULT expression
    ALTER [ COLUMN ] column_name DROP DEFAULT
    ALTER [ COLUMN ] column_name { SET | DROP } NOT NULL
    ALTER [ COLUMN ] column_name DROP EXPRESSION [ IF EXISTS ]
    ALTER [ COLUMN ] column_name ADD GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY
 [ ( sequence_options ) ]
    ALTER [ COLUMN ] column_name { SET GENERATED { ALWAYS | BY DEFAULT } |
 SET sequence_option | RESTART [ [ WITH ] restart ] } [...]
    ALTER [ COLUMN ] column_name DROP IDENTITY [ IF EXISTS ]
    ALTER [ COLUMN ] column_name SET STATISTICS integer
    ALTER [ COLUMN ] column_name SET ( attribute_option = value [, ... ] )
    ALTER [ COLUMN ] column_name RESET ( attribute_option [, ... ] )
    ALTER [ COLUMN ] column_name SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN }
    ALTER [ COLUMN ] column_name SET COMPRESSION compression_method
    ADD table_constraint [ NOT VALID ]
    ADD table_constraint_using_index
    ALTER CONSTRAINT constraint_name [ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY
 DEFERRED | INITIALLY IMMEDIATE ]
    VALIDATE CONSTRAINT constraint_name
    DROP CONSTRAINT [ IF EXISTS ]  constraint_name [ RESTRICT | CASCADE ]
    DISABLE TRIGGER [ trigger_name | ALL | USER ]
    ENABLE TRIGGER [ trigger_name | ALL | USER ]
    ENABLE REPLICA TRIGGER trigger_name
    ENABLE ALWAYS TRIGGER trigger_name
    DISABLE RULE rewrite_rule_name
    ENABLE RULE rewrite_rule_name
    ENABLE REPLICA RULE rewrite_rule_name
```

```
    ENABLE ALWAYS RULE rewrite_rule_name
    DISABLE ROW LEVEL SECURITY
    ENABLE ROW LEVEL SECURITY
    FORCE ROW LEVEL SECURITY
    NO FORCE ROW LEVEL SECURITY
    CLUSTER ON index_name
    SET WITHOUT CLUSTER
    SET WITHOUT OIDS
    SET TABLESPACE new_tablespace
    SET { LOGGED | UNLOGGED }
    SET ( storage_parameter [= value] [, ... ] )
    RESET ( storage_parameter [, ... ] )
    INHERIT parent_table
    NO INHERIT parent_table
    OF type_name
    NOT OF
    OWNER TO { new_owner | CURRENT_ROLE | CURRENT_USER | SESSION_USER }
    REPLICA IDENTITY { DEFAULT | USING INDEX index_name | FULL | NOTHING }

and partition_bound_spec is:

IN ( partition_bound_expr [, ...] ) |
FROM ( { partition_bound_expr | MINVALUE | MAXVALUE } [, ...] )
  TO ( { partition_bound_expr | MINVALUE | MAXVALUE } [, ...] ) |
WITH ( MODULUS numeric_literal, REMAINDER numeric_literal )

and column_constraint is:

[ CONSTRAINT constraint_name ]
{ NOT NULL |
  NULL |
  CHECK ( expression ) [ NO INHERIT ] |
  DEFAULT default_expr |
  GENERATED ALWAYS AS ( generation_expr ) STORED |
  GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY [ ( sequence_options ) ] |
  UNIQUE index_parameters |
  PRIMARY KEY index_parameters |
  REFERENCES reftable [ ( refcolumn ) ] [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ]
    [ ON DELETE referential_action ] [ ON UPDATE referential_action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]

and table_constraint is:

[ CONSTRAINT constraint_name ]
{ CHECK ( expression ) [ NO INHERIT ] |
  UNIQUE ( column_name [, ... ] ) index_parameters |
  PRIMARY KEY ( column_name [, ... ] ) index_parameters |
  EXCLUDE [ USING index_method ] ( exclude_element WITH operator
 [, ... ] ) index_parameters [ WHERE ( predicate ) ] |
  FOREIGN KEY ( column_name [, ... ] ) REFERENCES reftable [ ( refcolumn [, ... ] ) ]
    [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ] [ ON DELETE referential_action ] [ ON
 UPDATE referential_action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]

and table_constraint_using_index is:

    [ CONSTRAINT constraint_name ]
    { UNIQUE | PRIMARY KEY } USING INDEX index_name
```

```
    [ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]

index_parameters in UNIQUE, PRIMARY KEY, and EXCLUDE constraints are:

[ INCLUDE ( column_name [, ... ] ) ]
[ WITH ( storage_parameter [= value] [, ... ] ) ]
[ USING INDEX TABLESPACE tablespace_name ]

exclude_element in an EXCLUDE constraint is:

{ column_name | ( expression ) } [ opclass ] [ ASC | DESC ] [ NULLS { FIRST | LAST } ]
```

Description

Shardman extension of the `ALTER TABLE` syntax allows coherently changing definitions of sharded and global tables.

The set of `ALTER` operations supported for global and sharded tables is restricted. For details, see `ALTER TABLE` Limitations.

Parameters

Storage Parameters

Shardman extends *storage parameters* of tables with its own *storage metaparameters*. They are not stored in the corresponding catalog entry, but are used to tell the Shardman extension to perform some additional actions.

`global`

This parameter can be specified only for global tables. If set to 0, the global table will be converted to a regular one on the replication group where the command is executed. The global table will not exist on other nodes after completion of this statement. No other storage parameter can be set when `global` parameter is specified.

Examples

Create a global table `pgbench_tellers` and then convert it to local.

```
CREATE TABLE pgbench_tellers (
       tid       integer PRIMARY KEY,
       bid       integer,
       tbalance  integer,
       filler    character(84)
)
WITH (global);
ALTER TABLE pgbench_tellers SET (global=0);
```

See Also

`ALTER TABLE` Limitations , *PostgreSQL `ALTER TABLE`*

CREATE SEQUENCE

CREATE SEQUENCE — define a new sequence generator

Synopsis

```
CREATE SEQUENCE [ IF NOT EXISTS ] name
    [ AS data_type ]
    [ INCREMENT [ BY ] increment ]
    [ MINVALUE minvalue | NO MINVALUE ] [ MAXVALUE maxvalue | NO MAXVALUE ]
    [ START [ WITH ] start ] [ CACHE cache ] [ [ NO ] CYCLE ]
    [ OWNED BY { table_name.column_name | NONE } ]
    WITH ( [ global ],
           [ block_size = block_size ]
         )
```

Description

Shardman extensions to the `CREATE SEQUENCE` command enable creation of global sequence number generators. This command creates an ordinary PostgreSQL sequence on all nodes in a cluster and records sequence parameters in the global sequence state dictionary. (See Section 7.6 for details.)

After a global sequence is created, usual `nextval` function can be used to generate next sequence values that are guaranteed be unique across the entire cluster. Other standard sequence manipulation functions (e.g. `setval`) must not be used on global sequences as this may lead to unexpected results.

Parameters

In addition to the parameters recognized by PostgreSQL, the following parameters are supported by Shardman.

`global`

If specified, the sequence object is created as a Shardman-managed global sequence.

`block_size`

The number of elements allocated for a local sequence. The default value is 65536.

Notes

Global sequences are meant to behave similarly to ordinary PostgreSQL sequences (see *`CREATE SEQUENCE`*) with some limitations, the most important one being that a global sequence is always increasing. There's no support for negative increment values or wraparound (as in `CYCLE`), which also means there's practically no difference between the minimum sequence value and its starting value, so both parameters cannot be provided at the same time to avoid confusion.

Just like with regular sequence objects, the `DROP SEQUENCE` command removes a global sequence and the `ALTER SEQUENCE` command allows changing some of the global sequence parameters.

Examples

Create a global sequence called `serial`.

```
CREATE SEQUENCE serial MINVALUE 100 WITH (global);
```

Select the next number from this sequence:

```
SELECT nextval('serial');
 nextval
---------
     100
(1 row)
```

See Also

ALTER SEQUENCE, Section 7.6

CREATE TABLE

CREATE TABLE — define a new table

Synopsis

```
CREATE [ UNLOGGED ] TABLE [ IF NOT EXISTS ] table_name ( [
  { column_name data_type [ COLLATE collation ] [ column_constraint [ ... ] ]
    | table_constraint
    | LIKE source_table [ like_option ... ]}
    [, ... ]
] )
[ USING method ]
[ WITH ( storage_parameter [= value] [, ... ] ) ]
[ TABLESPACE tablespace_name ]

CREATE TABLE table_name ( [
  { column_name data_type }
    [, ... ]
] )
WITH ( { distributed_by = 'column_name'
         [, num_parts = number_of_partitions ]
         [, colocate_with = 'colocation_table_name' ]
         [, partition_by = 'column_name',
            partition_bounds = 'array_of_partition_bound_exprs' ] |
         global }
)

where column_constraint is:

[ CONSTRAINT constraint_name ]
{ NOT NULL |
  NULL |
  CHECK ( expression ) [ NO INHERIT ] |
  DEFAULT default_expr |
  UNIQUE index_parameters |
  PRIMARY KEY index_parameters }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]

and table_constraint is:

[ CONSTRAINT constraint_name ]
{ CHECK ( expression ) [ NO INHERIT ] |
  UNIQUE ( column_name [, ... ] ) index_parameters |
  PRIMARY KEY ( column_name [, ... ] ) index_parameters |
  EXCLUDE [ USING index_method ] ( exclude_element WITH operator
 [, ... ] ) index_parameters [ WHERE ( predicate ) ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

Description

Shardman extension of the `CREATE TABLE` syntax enables creation of sharded tables distributed across all replication groups with a single DDL statement.

The extended `CREATE TABLE` syntax imposes limitations on the general syntax of the command. For example, there is currently no support for:

- Generated columns.
- `REFERENCES` and `FOREIGN KEY` constraints between non-colocated sharded tables.
- `PARTITION BY` and `PARTITION OF` clauses.

When creating a colocated table, have in mind the related limitations. Specifically, from these limitations, it follows that a foreign key on a global table can reference only another global table and a foreign key on a sharded table can reference a colocated sharded table or a global table. Note that when a foreign key on a sharded or a global table references a global table, only `NO ACTION` or `RESTRICT` referential actions are supported for the `ON UPDATE` action and only `NO ACTION`, `RESTRICT` or `CASCADE` are supported for the `ON DELETE` action.

Columns of the `SERIAL8` type are implemented using an automatically created global sequence, so all global sequence properties also apply here. (See Section 7.6 for details.)

Parameters

`IF NOT EXISTS`

Do not throw an error if a relation with the same name already exists. A notice is issued in this case. `IF NOT EXISTS` does not lead to an error if an existing table with the same name is global or sharded, or if it is local yet is located on a node from which the query is run. Otherwise, such a query fails with error.

Storage Parameters

Shardman extends *storage parameters* of tables with its own *storage metaparameters*. They are not stored in the corresponding catalog entry, but are used to tell the Shardman extension to perform some additional actions. Regular storage parameters are transparently passed to table partitions.

`distributed_by` (`text`)

This specifies the name of the column to use for the table partitioning. Only hash partitioning is currently supported, so this is effectively an equivalent of `PARTITION BY HASH`, but all the leaf partitions will be created immediately on all replication groups and the table will be registered in the Shardman metadata.

`num_parts` (`integer`)

This sets the number of partitions that will be created for this table. This parameter is optional. If it is not specified, for a sharded table, the value of the global setting of shardman.num_parts will be used, for a colocated table, the value will be taken from the corresponding colocating table.

`colocate_with` (`text`)

This specifies the name of the table to colocate with. If set, Shardman will try to place partitions of the created table with the same partition key on the same nodes as *`colocation_table_name`*. This parameter is optional.

`partition_by` (`text`)

This specifies the name of the column to use for the second-level table partitioning. Only range partitioning is currently supported. When this parameter is used, each table partition is created as a partitioned table. Subpartitions can be created immediately if *`partition_bounds`* parameter is set. This parameter is optional.

`partition_bounds` (`text`)

This sets bounds of second-level table partitions. Bounds should be a string representation of a two-dimensional array. Each array member is a pair of a lower and upper bound for partitions. If lower and upper bounds are both NULL, the default partition is created. Number of partitions is determined by the first array dimension. This parameter is optional.

`global` (`boolean`)

This defines that the table is global. If set, the table will be distributed on all replication groups and will be synchronized by triggers. This parameter is optional.

Examples

In this example, the table `pgbench_branches` is created, as well as colocated tables `pgbench_accounts` and `pgbench_history`. Each partition of the `pgbench_history` table is additionally subpartitioned by range.

```
CREATE TABLE pgbench_branches (
       bid integer NOT NULL PRIMARY KEY,
       bbalance integer,
       filler character(88)
)
WITH (distributed_by = 'bid',
      num_parts = 8);
CREATE TABLE pgbench_accounts (
       aid integer NOT NULL,
       bid integer,
       abalance integer,
       filler character(84),
       PRIMARY KEY (bid, aid)
)
WITH (distributed_by = 'bid',
      num_parts = 8,
      colocate_with = 'pgbench_branches');
CREATE TABLE public.pgbench_history (
            tid integer,
            bid integer,
            aid integer,
            delta integer,
            mtime timestamp without time zone,
            filler character(22)
        )
WITH (distributed_by = 'bid',
      colocate_with = 'pgbench_branches',
      partition_by = 'mtime',
      partition_bounds =
          $${{minvalue, '2021-01-01 00:00'},{'2021-01-01 00:00', '2022-01-01 00:00'},
{'2022-01-01 00:00', maxvalue}}$$
);
```

These simple examples of `CREATE TABLE` illustrate limitations related to creation of colocated tables:

This command creates a table to colocate with:

```
CREATE TABLE teams_players (
       team_id integer NOT NULL,
       player_id integer,
       scores int,
       PRIMARY KEY (team_id, player_id)
) WITH (distributed_by='team_id, player_id');
```

This command correctly creates a colocated table:

```
CREATE TABLE players_scores (
       player_id integer NOT NULL,
       team_id integer,
       interval tstzrange,
       scores integer,
       foreign key (team_id, player_id) references teams_players(team_id, player_id)
) WITH (distributed_by='team_id, player_id', colocate_with='teams_players');
```

And this command contains an error in the definition of a foreign key:

```
CREATE TABLE players_scores (
       player_id integer NOT NULL,
       team_id integer,
       interval tstzrange,
       scores integer,
```

```
       foreign key (team_id, player_id) references teams_players(team_id, player_id)
) WITH (distributed_by='player_id, team_id', colocate_with='teams_players');
ERROR:  foreign key should start with distributed_by columns
```

Consider another example:

```
CREATE TABLE teams (team_id integer primary key, team_name text) with
 (distributed_by='team_id');
CREATE TABLE players_teams (
       player_id integer,
       team_id integer references teams(team_id),
       scores integer
) WITH (distributed_by='player_id', colocate_with='teams');
ERROR:  foreign key should start with distributed_by columns
```

See Also

`CREATE TABLE` Limitations , *PostgreSQL `CREATE TABLE`*

CREATE TABLESPACE

CREATE TABLESPACE — define a new tablespace

Synopsis

```
CREATE TABLESPACE tablespace_name
    [ OWNER { new_owner | CURRENT_ROLE | CURRENT_USER | SESSION_USER } ]
    LOCATION 'template'
    [ WITH ( tablespace_option = value [, ... ] ) ]
```

Description

Shardman extension of the `CREATE TABLESPACE` syntax enables creation of a new cluster-wide tablespace. All tablespaces in a Shardman cluster must be cluster-wide. The cluster-wide tablespace is created on each cluster node with the location derived from the *`template`* parameter.

Parameters

`tablespace_name`

The name of a tablespace to be created. The name cannot begin with `pg_` as such names are reserved for system tablespaces. Also the name cannot contain *new line* characters.

`template`

The directory name template that will be used for the tablespace. The template must include "{rgid}" substring, which will be translated into the actual replication group ID on each instance where a statement is executed. The directory name template must allow conversion to an absolute path. The path cannot contain *new line* characters. `CREATE TABLESPACE` will create the corresponding directory if it is missing. If the directory exists, it must be empty and must be owned by the PostgreSQL system user.

`tablespace_option`

A tablespace parameter to be set or reset. The list of parameters must include `global` boolean parameter. Creation of non-global tablespaces is not allowed by default.

Examples

To create a tablespace `dbspace` under the file system location `/data/dbs`, first create the directory using operating system facilities on all nodes and set the correct ownership (or ensure that `postgres` user has permissions to create it):

```
mkdir /data/dbs
chown postgres:postgres /data/dbs
```

Then issue the tablespace creation command inside PostgreSQL:

```
CREATE TABLESPACE dbspace LOCATION '/data/dbs/ts-{rgid}' WITH (global);
```

See Also

PostgreSQL `CREATE TABLESPACE`

6.6. SQL Limitations

To ensure consistency of a sharded database, Shardman imposes some restrictions on SQL commands executed.

6.6.1. `ALTER SYSTEM` Limitations

- *`ALTER SYSTEM`* is prohibited (configuration changes should be performed via `shardmanctl config update`).

6.6.2. `ALTER TABLE` Limitations

- *`ALTER TABLE`* is prohibited for partitions of sharded tables.

- All forms of `ALTER TABLE` are prohibited for sharded or global tables except these:
  - `ALTER TABLE OWNER` is allowed. For sharded table it also changes the owner of table partitions. Only the global user can be an owner of sharded or global table.
  - `ALTER TABLE COLUMN TYPE` is allowed with limitations. You cannot alter type of sharded table column participating in sharding or partitioning key. You cannot alter type of sharded table column with `USING` clause (but for global tables it is allowed). Also, it is a user's duty for now to create and keep new type exactly equal on every cluster node.
  - `ALTER TABLE COLUMN RENAME` is allowed.
  - Adding or dropping table-wide unique constraints and checks is allowed. For global tables dropping primary key constraint or dropping columns, participating in primary key, is forbidden.
  - Adding foreign keys between sharded tables is possible only when they are colocated and a foreign key references tuples that are stored in the same replication group. A foreign key between sharded tables must begin with the columns used for table partitioning in both tables. A foreign key on a global table can reference only another global table. A foreign key on a sharded table can reference a colocated sharded table or a global table.
  - `SET/DROP NOT NULL` is allowed.
  - Setting storage options is allowed for global tables.
  - Global tables cannot inherit other tables.
  - `ALTER COLUMN SET STATISTICS` is allowed for global and sharded tables.

6.6.3. CREATE TABLE Limitations

- For *`CREATE TABLE`*, all limitations for `ALTER TABLE` apply.
- Using of non-builtin types (types with OIDs >= 10000) or non-base types or arrays are not allowed in 'distributed_by' columns.
- Only the global user can create sharded or global table.
- In a colocated table, the number and types of columns used for table partitioning must be the same as for the table to colocate with.
- A temporary table cannot be created as sharded or global.
- Self-referencing sharded tables are allowed only if a foreign key is referencing the same partition of the sharded table.
- For tables created using `LIKE` *`source_table`* where *`source_table`* is a local table, the following limitations apply:
  - Copying without the *`like_option`* clause or with `INCLUDING INDEXES` is only supported.
  - With `INCLUDING INDEXES`, only unique indexes and indexes supporting the primary relation key are copied.
  - Copying indexes for columns is not supported.
  - `EXCLUDE` constraints are not supported.
  - Local tables used in `CREATE TABLE LIKE` statement must only have columns of base types.
  - Partial indexes are not supported.
  - Standard collations are only supported.
  - `NULLS NOT DISTINCT` constraint is not supported.

6.6.4. DROP TABLE Limitations

- Sharded or global tables and local tables cannot be dropped in the same statement with *`DROP TABLE`*.
- Partitions of a sharded table cannot be dropped.

6.6.5. CREATE INDEX CONCURRENTLY Limitations

- *`CREATE INDEX CONCURRENTLY`* is a non-transactional command. If a problem arises while building index on sharded or global tabe, such as network failure, deadlock or a uniqueness violation in a unique index, the `CREATE INDEX CON-`

CURRENTLY will partially fail, but can leave behind valid or invalid indexes on Shardman cluster nodes. Also an index can be completely missing on some nodes. In the later case DROP INDEX will fail to drop the index. The recommended way to remove such index cluster-wide is to use DROP INDEX IF EXISTS command. Note that DROP INDEX CONCURRENTLY is not supported on sharded tables, so this operation should be better performed in a maintenance window.

6.6.6. UPDATE Limitations

- *UPDATE* of a sharded table is executed as a series of usual UPDATEs if it doesn't move data between partitions or subpartitions. Otherwise it is executed internally as DELETE from one partition and INSERT into another (so called target partition). If a partition where UPDATE INSERTs data, is going to be UPDATED in the same statement, an error will be raised. In practice this means that if UPDATE moves data between partitions, you should explicitly exclude target partition from updating in WHERE clause of the statement.

6.6.7. INSERT ON CONFLICT DO UPDATE Limitations

- *INSERT ON CONFLICT DO UPDATE* *table_name*... ON CONFLICT [conflict_target] conflict_action [WHERE condition] command is not supported on foreign tables when conflict_target is DO UPDATE. For sharded tables it is supported if expressions in SET and WHERE clause can be safely deparsed (currently deparsing of sqlvalue-functions, parameters and subqueries inside these clauses is not supported) and a non-partial unique index, containing only table columns (not coulmn-based expressions), corresponds to conflict_target expression. This is usually the case with table's primary key.

6.6.8. Limitations of Managing Global Roles

- Global users can be created only by user with CREATEROLE permission on all cluster nodes.
- Global roles cannot be renamed.
- Global and local roles cannot be dropped in the same statement.
- *GRANT* to a local and global role in the same statement is prohibited.
- *REVOKE* from a local and global role in the same statement is prohibited.

6.6.9. Limitations of User Mappings

- The CREATE USER MAPPING, ALTER USER MAPPING, and DROP USER MAPPING commands are prohibited when applied to mappings for foreign servers from the Shardman cluster. Use Shardman mechanisms of Managing Users and Roles instead.

6.6.10. ALTER SCHEMA Limitations

- Schemas containing global or sharded tables cannot be renamed with *ALTER SCHEMA* . Shardman service schemas (shardman) cannot be renamed or dropped.

6.6.11. DROP SERVER Limitations

- Shardman cluster servers cannot be dropped with *DROP SERVER* . Use Shardman tools to remove servers from the cluster.

6.6.12. Limitations of Using Custom Databases

- Custom databases are not supported. All the local custom databases can be corrupted or lost during the shardmanctl operations.

6.6.13. CREATE COLLATION Limitations

- If you use custom collation with *CREATE COLLATION*, all servers must have same version of icu. Otherwise results of queries on sharded tables may be incorrect.

6.6.14. Logical Replication Limitations

- If you attempt to *publish* a table containing foreign partitions with the publish_via_partition_root option enabled, the operation will fail. Without this option, only the local partitions will be included in the publication.

- When using `FOR TABLES IN SCHEMA` or `FOR ALL TABLES`, only local partitions will be published.
- If you publish using `FOR TABLES IN SCHEMA WITH` or `FOR ALL TABLES` along with the `publish_via_partition_root` option, any tables with foreign partitions will be excluded from the publication.
- When executing *`ALTER SUBSCRIPTION ... REFRESH PUBLICATION`*, depending on changes to table partitions, tables may be added to or removed from the publication.
- When using `FOR ALL TABLES`, tables from the `shardman` schema are excluded from the publication. However, you can still create a publication specifically for tables in this schema or for individual tables within it.

6.6.15. Other Limitations

- *`DROP TYPE CASCADE`* is prohibited if it affects types used in global or sharded tables.
- Access privileges management per columns is not supported for global tables.

6.7. Shardman CLI Reference

shardmanctl

shardmanctl — Shardman auxiliary command-line client and deployment tool

Synopsis

shardmanctl [*common_options*] backup --datadir *directory* [--maxtasks *number_of_tasks*] --use-ssh

shardmanctl [*common_options*] daemon check -n | --nodes *node_names:port*

shardmanctl [*common_options*] cleanup [-p | --processrepgroups] --after-node-operation --after-rebalance

shardmanctl [*common_options*] config generate [-f | --file *filename*]

shardmanctl [*common_options*] config verify [-f | --file *filename*]

shardmanctl [*common_options*] config get [-f | --file] [-c | --choose-revision] [-r | --revision]

shardmanctl [*common_options*] config revisions rm [-r | --revision] [-y | --yes]

shardmanctl [*common_options*] config update [[-f | --file *stolon_spec_file*|*shardman_spec_file*] | *spec_text* [--force] [-p | --patch] [-w | --wait *time_duration*]]

shardmanctl [*common_options*] config rollback [-r | --revision] [-w | --wait *time_duration*] [--force]

shardmanctl [*common_options*] config update credentials [-u | --user] [-p | --password] [-k | --ssl-key] [-c | --ssl-cert] [-w | --wait *time_duration*] [-f | --force] [-y | --yes]

shardmanctl [*common_options*] config revisions [-f | --format json|text]

shardmanctl [*common_options*] config revisions set --keep-config-revisions

shardmanctl [*common_options*] config update ip [-u | ip_1=ip_2,hostname_1=hostname_2] [-y | --yes]

shardmanctl [*common_options*] config update fdw [-y | --yes]

shardmanctl [*common_options*] cluster repfactor set --value *value*

shardmanctl [*common_options*] cluster start

shardmanctl [*common_options*] cluster stop [-y | --yes]

shardmanctl [*common_options*] cluster topology [-f | --format table|json|text]

shardmanctl [*common_options*] forall --sql *query* [--twophase]

shardmanctl [*common_options*] getconnstr --all

shardmanctl [*common_options*] init [-y | --yes] [-f | --spec-file *spec_file_name*] | *spec_text*

shardmanctl [*common_options*] intcheck [-s | --system] [-c | --catalog] [-u | --user] [-o | --output] [-n | --node *node*]

shardmanctl [*common_options*] load [-b | --batch-size *lines_limit*] [--destination-fields *fields_list*] [--distributed-keys *key_type_list*] [-D | --delimiter *character*] [--null_marker *string*] [-e | --escape *character*] [-f | --file *input_file*] [-F | --format text | csv] [-j | --jobs *task_total*] [-q | --quote *character*] [--reject-file *filename*] [--schema *filename*] [--source file | postgres]

[--source-connstr *connect_string*][--source-fields *fields_list*][--source-table *table*|*view*|*func*][-t|--table *destination_table*][-h|--help]

shardmanctl[*common_options*]nodes add -n|--nodes *node_names*[--no-rebalance]

shardmanctl[*common_options*]nodes start -n|--nodes *node_names*[--no-wait]

shardmanctl[*common_options*]nodes restart -n|--nodes *node_names*[--no-wait]

shardmanctl[*common_options*]nodes stop -n|--nodes *node_names*[--no-wait]

shardmanctl[*common_options*]nodes replace --old *old_node* --new *new_node*

shardmanctl[*common_options*]nodes rm -n|--nodes *node_names*

shardmanctl[*common_options*]probackup[init|archive-command|backup|checkdb|delete|merge|restore|set-config|show|validate|show-config][*subcommand_options*]

shardmanctl[*common_options*]rebalance[-f|--force]

shardmanctl[*common_options*]recover[--info *file*][--dumpfile *file*][--shard *shard*][--meta-data-only][--schema-only][--timeout *seconds*]

shardmanctl[*common_options*]restart[-y|--yes][--no-wait]

shardmanctl[*common_options*]set pgParam1=value1 [pgParam2=value2 [...]][-y|--yes][-w|--wait *time_duration*][-f|--force]

shardmanctl[*common_options*]shard -s|--shard *shard_name* add -n|--node *node_names*

shardmanctl[*common_options*]shard -s|--shard *shard_name* master set -n|--node *node_names*

shardmanctl[*common_options*]shard -s|--shard *shard_name* master reset

shardmanctl[*common_options*]shard -s|--shard *shard_name* reset[-y|--yes][--new-primary|-p]

shardmanctl[*common_options*]shard -s|--shard *shard_name* rm -n|--node *node_names*[-f|--force]

shardmanctl[*common_options*]shard -s|--shard *shard_name* switch[--new-primary *node_names*]

shardmanctl [*common_options*] shard -s | --shard *shard_name* start [--no-wait] [-n | --node *node_name*]

shardmanctl[*common_options*]shard -s|--shard *shard_name* stop[-n|--node *node_name*]

shardmanctl[*common_options*] shard -s|--shard *shard_name* replicas reinit[--no-wait][-y|--yes][-n|--node *node_names*]

shardmanctl[*common_options*]status[--filter all | dictionary | primary | metadata | rg | shardmand | store | topology | restart_required_params][-f|--format *text* | *json*][-s|--sort *node* | *rg* | *status*]

shardmanctl[*common_options*]status transactions[-r|--repgroup *replication_group_name*]

shardmanctl[*common_options*]store dump[-f|--file *filename*]

shardmanctl[*common_options*]store restore[--delete-old-keys][-f|--file *filename*][-y|--yes]

shardmanctl[*common_options*]store get[-a|--alias *cluster* | *ladle* | *repgroups* | *stolonspec* | *spec*][-k|--key *keyname*][-f|--file *filename*]

shardmanctl[*common_options*]store keys

shardmanctl[*common_options*]store set[-a|--alias *cluster* | *ladle* | *repgroups* | *stolonspec* | *spec*][-k|--key *keyname*][-f|--file *filename*]

```
shardmanctl [common_options] store lock [ -f | --format  text  |  json  ]

shardmanctl [common_options] tables sharded info [ -t | --table  table  ]

shardmanctl [common_options] tables sharded list

shardmanctl [common_options] tables sharded norebalance

shardmanctl [common_options] tables  sharded  partmove [ -t | --table  table  ] [ -s | --shard
shard_name  ] [ -p | --partnum  partition_number  ]

shardmanctl [common_options] tables  sharded  rebalance [ -t | --table  table  ] [ --skip-run-
rebalance ]

shardmanctl [common_options] upgrade

shardmanctl [common_options] bench  init [ --schema-type  single|simple|shardman|custom ] [ -S |
--schema-file file_name ] [ -s | --scale scale_value ] [ --partitions partitions_value ] [ -n | --no-
vacuum ] [ -F | --fillfactor fillfactor_value ]

shardmanctl [common_options] bench  run [ --schema-type  single|simple|shardman|custom ] [ -f |
--file file_name ] [ -c | --client client_value ] [ -C | --connect ] [ --full-output ] [ -j | --jobs job-
s_value ] [ -s | --scale scale_factor ] [ -T | --time seconds ] [ -t | --transactions transactions_value ]
[ -P | --progress seconds ] [ -R | --rate rate ] [ -M | --protocol querymode ]

shardmanctl [common_options] bench cleanup

shardmanctl [common_options] bench  generate [ -c | --config config_file ] [ -o | --output-file
file_name ]

shardmanctl [common_options] script [ -s | --shard shard_name ] [[ -f | --file file_name | --sql query ]]

shardmanctl [common_options] psql -s | --shard shard_name

shardmanctl [common_options] daemon  set [ --session-log-level | debug | info | warn | error ] [ --
session-log-format | text | json ] [ --session-log-nodes ]

shardmanctl [common_options] history [ -r | --reverse ] [ -f | --format  text  |  json  ] [ -l | --limit
number_of_commands ]
```

Here *common_options* are:

```
[ --cluster-name cluster_name ] [ --log-level error | warn | info | debug ] [ --monitor-port  port ] [ --
retries retries_number ] [ --session-timeout seconds ] [ --store-endpoints store_endpoints ] [ --
store-ca-file store_ca_file ] [ --store-cert-file store_cert_file ] [ --store-key client_pri-
vate_key ] [ --store-timeout duration ] [ --version ] [ -h | --help ]
```

Description

shardmanctl is an utility for managing a Shardman cluster.

For any command that uses the node name as an argument, the node name can be specified either by its hostname or IP address.

The `backup` command is used to backup a Shardman cluster. A backup consists of a directory with base backups of all replication groups and WAL files needed for recovery. etcd metadata is saved to the `etcd_dump` file. The `backup_info` file is created during a backup and contains the backup description. For details of the `backup` command logic, see Cluster backup with pg_basebackup. For usage details of the command, see the section called “Backing up a Shardman Cluster”.

The `cleanup` command is used for cleanup after failure of the `nodes add` command or of the shardmanctl `rebalance` command. Final changes to the etcd store are done at the end of the command execution. This simplifies the `cleanup` process. During cleanup, incomplete clover definitions and definitions of the corresponding replication groups are removed from the etcd metadata. Definitions of the corresponding foreign servers are removed from the DBMS metadata of the remaining replication groups. Since the `cleanup` process can be destructive, by default, the tool operates in the report-only mode: it only shows actions

to be done during the actual cleanup. To perform the actual cleanup, add the `-p` flag. For usage details of the command, see the section called “Performing Cleanup”.

The `daemon check` command is used to verify that shardmand daemon is running on the nodes specified by `--nodes` option and is configured for the same cluster as shardmanctl . For usage details of the command, see the section called “Checking shardmand Service on Nodes”.

The `init` command is used to register a new Shardman cluster in the etcd store or to reinitialize the existing cluster defining a new cluster configuration and removing all data and nodes. In the init mode, shardmanctl reads the cluster specification, processes it and saves to the etcd store as parts of two JSON documents: `ClusterSpec` — as part of `shardman/cluster0/data/cluster` and `LadleSpec` — as part of `shardman/cluster0/data/ladle` (`cluster0` is the default cluster name used by Shardman utilities). Common options related to the etcd store, such as `--store-endpoints`, are also saved to the etcd store and pushed down to all Shardman services started by shardmand. For the description of the Shardman initialization file format, see sdmspec.json. For usage details of the command, see the section called “Registering a Shardman Cluster”.

The `config generate` command is used to create a default `sdmspec.json` template. By default, data is returned to the standard output. To write the result to a file, use flag `-f filename`. For the description of the Shardman initialization file format, see sdmspec.json.

The `config verify` command is used to check a correctness of the input Shardman initialization file. By default, the configuration is read from standard input. To read the configuration from a file, use flag `-f filename`. For the description of the Shardman initialization file format, see sdmspec.json.

The `config get` command is used to output the current full cluster specification or a configuration of the specified revision. The command takes the current cluster configuration from the cluster store. For the description of the Shardman initialization file format, see sdmspec.json.

The `config update` command is used to update the stolon or full Shardman configuration. The new configuration is applied to all replication groups and is saved in `shardman/cluster0/data/cluster` etcd key. Note that `config update` can cause a DBMS restart.

The `forall` command is used to execute an SQL statement on all replication groups in a Shardman cluster.

The `getconnstr` command is used to get the libpq connection string for connecting to a cluster as administrator.

The `load` command is used to upload data from a text file to a distributed table or to upload a database schema from a PostgreSQL database to Shardman. When loading data from a file, `text` and `csv` formats are supported. If a file is compressed with gzip, it will be automatically decoded while reading. To read data from stdin, specify `--file=-`. The data loading process can be optimized by specifying the number of parallel workers (key `-j`).

The `nodes add` command is used to add new nodes to a Shardman cluster. With the default `cross` placement policy, nodes are added to a cluster by *clovers*. Each node in a clover runs the primary DBMS instance and perhaps several replicas of other nodes in the clover. The number of replicas is determined by the Repfactor configuration parameter. So, each clover consists of `Repfactor + 1` nodes and can stand loss of `Repfactor` nodes.

With *manual* placement policy, each new node is added as a replication group consisting of one primary server. After adding primary nodes, you can add replicas to the new replication group by calling the `shard add` command.

shardmanctl performs the `nodes add` operation in several steps:

1. Acquires a global metadata lock.
2. For each specified node, checks that shardmand is running on it and that it sees the current cluster configuration.
3. Calculates the services to be present on each node and saves this information in etcd as part of the `shardman/cluster0/data/ladle` `Layout` object.
4. Generates the configuration for new stolon clusters (also called *replication groups*) and initializes them.
5. Registers the added replication groups in the `shardman/cluster0/data/ladle` etcd key.
6. Waits for shardmand to start all the necessary services, checks that new replication groups are accessible and have correct configuration.

7. Creates an auxiliary broadcaster that holds locks on each existing replication group in the cluster.
8. For each new replication group, copies all schemas and `shardman` schema data from a randomly selected existing replication group to the new one, ensures that the Shardman extension is installed on the new replication group, and recalculates OIDs used in the extension configuration tables.
9. On each existing replication group, defines foreign servers referencing the new replication group and recreates definitions of foreign servers on the new replication group.
10. Recreates all partitions of sharded tables as foreign tables referencing data from old replication groups and has the changes registered in the etcd storage.
11. For each new replication group, copies the global table data from existing replication groups to the new one.
12. Rebalances partitions of sharded tables. The rebalancing process for each sharded table iteratively determines the replication group with the maximum and minimum number of partitions and creates a task to move one partition to the replication group with the minimum number of partitions. This process is repeated while `max - min > 1`. To move partitions, we use logical replication. Partitions of colocated tables are moved together with partitions of the distributed tables to which they refer. You can skip this step using the `--no-rebalance`.

For usage details of the command, see the section called “Adding Nodes to a Shardman Cluster”.

The `nodes rm` command is used to remove nodes from a Shardman cluster. In the manual-topology mode, this command only removes the specified nodes from the cluster and if a node is the last in the replication group, the entire group gets removed. In the cross-replication mode, this command removes clovers containing the specified nodes from the cluster. The last clover in the cluster cannot be removed. Any data (such as partitions of sharded relations) on removed replication groups is migrated to the remaining replication groups using logical replication, and all references to the removed replication groups (including definitions of foreign servers) are removed from the metadata of the remaining replication groups. Finally, the metadata in etcd is updated. For usage details of the command, see the section called “Removing Nodes from a Shardman cluster”.

The `probackup` command is used to backup and restore the Shardman cluster using pg_probackup backup utility. For details of the `probackup` command logic, see Backup anf Recovery Shardman Backups using pg_probackup. For usage details of the command, see the section called “ `probackup` ”.

The `rebalance` command is used to evenly rebalance sharded tables in a cluster. This can be useful, for example, if you did not perform rebalance when adding nodes to the cluster. If the `--force` option is not provided, then tables with manually moved partitions will be skipped.

The `cleanup` command with flag --after-rebalance is used to perform cleanup after failure of a `rebalance` command. On each node, it cleans up subscriptions and publications left from the `rebalance` command and drops tables that store data of partially-transferred partitions of sharded tables.

The `cluster repfactor set` command is used to set the value of the replication factor for the Shardman cluster. This command can only be used in `manual` topology cluster mode. The value of the new replication factor is passed through the command line flag `--value` *`repfactor`*.

The `cluster start` command is used to start all stopped PostgreSQL instances with the `cluster stop` command. For the command to work, shardmand must be running.

The `cluster stop` command is used to stop all PostgreSQL instances for the Shardman cluster. At the same time, the shardmand daemons continue to work.

The `cluster topology` command is used visualize the topology of a cluster. By default, the topology is returned in a table view. If you want to get a JSON or text representation, then use the flag `--format json|text`.

The `recover` command is used to restore a Shardman cluster from a backup created by the `backup` command. For details of the `recover` command logic, see Cluster recovery from a backup using pg_basebackup. For usage details of the command, see the section called “Restoring a Shardman Cluster”.

The `restart` command is used to restart a Shardman cluster, including all shardmand instances. If PostgreSQL instances were previously stopped using the `cluster stop` command, they will be started. The command returns control after all primary nodes in the cluster have been restarted.

The `set` command is used to set one or more parameters for DBMS instances of the Shardman cluster. Parameters are passed as arguments to the command line, each of them looks like `param=value`. The command is actually an alternative to `shardmanctl config update -p` to update database settings.

The `status` command is used to display health status of Shardman cluster subsystems. It can show status of several components: store, metadata, shardmand, replication groups, primary nodes, dictionary, and restart of the required parameters. If only some subsystems are of interest, option `--filter` may be used. Also `status` supports sorting its messages by `status`, `node` or `replication group` and printing the result to stdout as a table (`table`), text (`text`) or `JSON` (`json`) with `table` as the default. For usage details of the command, see the section called “Getting the Status of Cluster Subsystems”.

The `store dump` command gets all the keys and their values from the etcd store and outputs them into the `--file`, where `-` value is used for outputting to stdout (default). It is intended to be used for debugging, so some harmless errors may be produced during execution, yet all the available information will be dumped. Only keys for the current cluster (with current cluster prefix like `shardman/cluster0/`) will be dumped. For usage details of the command, see the section called “Dumping All Keys from the Store to Debug Error Configuration”.

The `store get` command gets a particular value from the store by its key name. It is expected to be a JSON value, so if it is not (which is not prohibited), some harmless errors may be produced. The key to retrieve from store can be specified with `--key` option; several keys have *aliases* — short names for easy use. To get a key by its alias, use `--alias` option with one of the available aliases (use `--help` or examples below for reference). Also aliases *stolonspec* and *spec* can be used to manipulate initial cluster and stolon configuration explicitly, without retrieving it from the full cluster specification. It is recommended to use existing aliases instead of full key names since there are some additional checks in alias processing, which help to achieve safer results. By default, a key is printed to stdout (explicitly — with `--file=-` option), but can be output to any desired file. For usage details of the command, see the section called “Getting the Current stolon Specification”.

The `store keys` command shows all the keys in the store for the current cluster (with cluster prefix) and its aliases. Aliases `stolonspec` and `spec` are not shown since they are parts of other keys. For usage details of the command, see the section called “Getting the Cluster and Ladle Key Names For the Current Cluster”.

The `store set` command creates or rewrites one particular key in the store. It is not expected to be a JSON value for a random key, but if it is one of the keys that have aliases with a known mapping (like `ladle` or `cluster`), the command will not accept incorrect JSON structures. Just like `store get` command, `store set` accepts a key name via `--key` or `--alias` option and the input source file as `--file` (stdin is specified with `-` value). For usage details of the command, see the section called “Setting a New Spec for the Cluster”.

The `store lock` command show the current cluster meta lock information. In case lock does not exist returns `Lock not found`. Displays cluster id, command that acquired locks, host name and lock time. You can specify `--format` to output in `json` format or in `text` format (by default). For usage details of the command, see the section called “Output Current Cluster Meta Lock Information”.

The `upgrade` command is used to update the version of Postgresql shardman extension on all cluster nodes. Before upgrading extensions, you need to install new packages and run the `restart` command. As a result of `upgrade`, utilities will upgrade shardman and all the other extensions on the server.

Sometimes after running the `upgrade` command or some user's manual manipulations, dictionary errors may appear in the output of the `status` command. One of the reasons for these errors is that the value of the `srvoptions` field of the `pg_foreign_server` table differs from what the system expects. To solve this specific issue, use the `config update fdw` command, which will return `srvoptions` to the expected state.

> **Note**
>
> Most of the described shardmanctl commands take a global metadata lock.

Command-line Reference

This section describes `shardmanctl` commands. For Shardman common options used by the commands, see the section called “Common Options”.

backup

Syntax:

```
    shardmanctl [common_options] backup --datadir directory [--
maxtasks number_of_tasks] [--use-ssh]
```

Backs up a Shardman cluster.

`--datadir` *`directory`*

Required.

Specifies the directory to write the output to. If the directory exists, it must be empty. If it does not exist, shardmanctl creates it (but not parent directories).

`--maxtasks` *`number_of_tasks`*

Specifies the maximum number of concurrent tasks (`pg_probackup` commands) to run.

Default: number of logical CPUs of the system.

`--use-ssh`

If specified shardmanctl `recover` command will use `scp` command to restore data. It allows to use backup repository on the local host.

For more details, see the section called “Backing up a Shardman Cluster”

cleanup

Syntax:

```
shardmanctl [common_options] cleanup [-p|--processrepgroups] --after-node-operation|--
after-rebalance
```

Performs cleanup after the `nodes add` or `rebalance` command.

`-p` *`node_names`*
`--processrepgroups=`*`node_names`*

Perform an actual cleanup. By default, the tool only shows actions to be done during the actual cleanup. For more details, see the section called “Performing Cleanup”.

`--after-node-operation`

Perform cleanup after a failure of a `nodes add` command.

`--after-rebalance`

Perform cleanup after a failure of a `rebalance` command.

config update credentials

Syntax:

```
shardmanctl [common_options] config update credentials [-u | --user] [-p | --password]
 [-k | --ssl-key] [-c | --ssl-cert] [-w|--wait time_duration] [--force] [-y | --yes]
```

Updates password or certificate/key of a user to connect to a Shardman cluster. It only updates the authentication type that was specified by the user (`scram-sha-256`, `ssl`) and not the type itself.

`-u`
`--user`

User that requires an update of the authentication parameters.

`-p`
`--password`

New password.

`-k`
`--ssl-key`

New SSL key.

`-c`
`--ssl-cert`

New SSL certificate.

`-w`
`--wait`

Sets shardmanctl to wait for configuration changes to take effect. If a new configuration cannot be loaded by all replication groups, shardmanctl will wait forever.

`--force`

Perform forced update if a cluster operation is in progress.

`-y`
`--yes`

Confirm the operation instead of asking approval from the standard input.

`cluster repfactor set`

Syntax:

`shardmanctl [`*`common_options`*`] cluster repfactor set --value `*`new_repfactor`*

Sets the replication factor for the manual-topology mode.

`--value=`*`new_repfactor`*

New replication factor value

`cluster start`

Syntax:

`shardmanctl [`*`common_options`*`] cluster start`

Starts all PostgreSQL server instances.

`cluster stop`

Syntax:

`shardmanctl [`*`common_options`*`] cluster stop [-y|--yes]`

Stops all PostgreSQL server instances.

`-y`
`--yes`

confirm the operation instead of asking approval from the standard input.

`cluster topology`

Syntax:

`shardmanctl [`*`common_options`*`] cluster topology -f|--format table|json|text`

Displays the cluster topology.

`-f table|json|text`
`--format=table|json|text`

Output format. For more details, see the section called “Displaying the Cluster Topology”.

`daemon check`

Syntax:

`shardmanctl [common_options] daemon check -n|--nodes node_name:port`

Checks shardmand on nodes.

`-n node_name:port`
`--nodes=node_name:port`

List of nodes to check shardmand on. For more details, see the section called “Checking shardmand Service on Nodes”.

`forall`

Syntax:

`shardmanctl [common_options] forall --sql query[ --sql query[ --sql query ...]] [--twophase]`

Executes an SQL statement on all replication groups in a Shardman cluster.

`--sql query`

Specifies the statement to be executed.

`--twophase`

Use the two-phase-commit protocol to execute the statement.

`getconnstr`

Syntax:

`shardmanctl [common_options] getconnstr --all`

Gets the libpq connection string for connecting to a cluster as administrator.

`--all`

Adds replicas to `getconnstr`.

`init`

Syntax:

```
shardmanctl [common_options] init [-y|--yes] [-f|--spec-
file spec_file_name]|spec_text
```

Registers a new Shardman cluster in the etcd store or reinitializes the existing cluster defining a new cluster configuration and removing all data and nodes.

`-f spec_file_name`
`--specfile=spec_file_name`

Specifies the file with the cluster specification string. The value of `-` means the standard input. By default, the string is passed in *`spec_text`*. For usage details, see the section called “Registering a Shardman Cluster”.

`-y`
`--yes`

Confirm the operation instead of asking approval from the standard input.

`intcheck`

Syntax:

```
    shardmanctl [common_options] intcheck [-s|--system] [-u|--user] [-c|--catalog] [-
o|--output]  [-n|--node node]
```

Runs pg_integrity_check on all nodes of a Shardman cluster or on a selected one node.

`-s`
`--system`

Validate checksums for read-only files. Checksums for read-only files control both file contents and file attributes.

`-u`
`--user`

Validate checksums for additional files. Checksums for additional files control both file contents and file attributes.

`-c`
`--catalog`

Validate checksums for system catalog tables. For the `-c` option to work correctly, the database server must be started and accept connections.

`-o`
`--output`

Recalculate checksums and write them into a file

`-n` *`node_names`*
`--node=`*`node_names`*

Only execute the pg_integrity_check command on the selected node

`load`

Syntax:

```
     shardmanctl [common_options] load [ -b | --batch-size lines_limit] [ --
destination-fields fields_list]
     [ --distributed-keys key_type_list] [ -D | --delimiter character]
     [--null_marker string] [ -e | --escape character] [ -f | --file input_file]
     [ -F | --format text|csv ] [ -j | --jobs task_total] [ -q | --quote character]
     [ --reject-file filename] [ --schema filename] [ --source file|postgres]
     [ --source-connstr connect_string] [ --source-fields fields_list] [ --source-
table source_table]
     [ -t | --table destination_table]
```

Loads data to a Shardman cluster.

`-b` *`lines_limit`*
`--batch-size=`*`lines_limit`*

Number of rows per batch to write to the Shardman cluster.

Default: `1000`.

`--destination-fields=`*`fields_list`*

Comma-separated list of target table fields. If the value is not set, then all fields of the table are used in the order they are declared.

`--distributed-keys=`*`key_type_list`*

Comma-separated list of pairs. Each pair consists of a field number (starting with zero) and a type, which are separated by a colon. The following types are supported: `bool`, `char`, `float4`, `float8`, `int2`, `int4`, `int8`, `name`, `text`, `varchar` and `uuid`.

`-D` *character*
`--delimiter=`*character*

Specifies the character that separates columns within each row (line) of the file. This must be a single one-byte character.

Default: tab for text format, comma for CSV format

`--null_marker=`*string*

Specifies the string that represents a null value.

Default: \N for text format, unquoted empty string for CSV format.

`-e` *character*
`--escape=`*character*

Specifies the character that should appear before a data character that matches the QUOTE value. The default is the same as the QUOTE value (so that the quoting character is doubled if it appears in the data). This must be a single one-byte character. This option is allowed only when using CSV format.

`-f` *filename*
`--file=`*filename*

Input data filename (or `-` for stdin)

`-F` *text*|*csv*
`--format=`*text*|*csv*

Input data format. Possible values are `text` and `csv`.

Default: `text`.

`-j` *number*
`--jobs=`*number*

Number of parallel processes to load data.

Default: number of replication groups.

`-q` *character*
`--quote=`*character*

Specifies the quoting character to be used when a data value is quoted. The default is double-quote. This must be a single one-byte character. This option is allowed only when using CSV format.

`--reject-file=`*filename*

All data batches with errors during upload will be written to this file. If the value is not set, then such batches will be skipped.

`--schema=`*filename*

The schema that defines the rules for transferring data from PostgreSQL to Shardman. If this option is set, then all other options are not used.

`--source=`*file*|*postgres*

Data source type — `file` or `postgres`.

Default: `file`.

`--source-connstr=`*string*

Data source database connection string

`--source-fields=`*fields_list*

Comma-separated list of source table fields. If the value is not set, then all fields of the table are used in the order they are declared.

`--source-table=table`

Source table, view or function (`funcname(param1,...,paramN)`).

`-t table`
`--table=table`

Destination table.

`nodes add`

Syntax:

```
shardmanctl [common_options] nodes add -n|--nodes node_names [--no-rebalance]
```

Adds nodes to a Shardman cluster.

`-n node_names`
`--nodes=node_names`

Required.

Specifies the comma-separated list of nodes to be added.

`--no-rebalance`

Skip the step of rebalancing partitions of sharded tables. For more details, see the section called “Adding Nodes to a Shardman Cluster”.

`nodes rm`

Syntax:

```
shardmanctl [common_options] nodes rm -n|--nodes node_names
```

Removes nodes from a Shardman cluster.

`-n node_names`
`--nodes=node_names`

Specifies the comma-separated list of nodes to be removed. For usage details, see the section called “Removing Nodes from a Shardman cluster”.

`probackup`

Syntax:

```
shardmanctl [common_options] probackup
       [init|archive-command|backup|checkdb|delete|merge|restore|set-config|show|
validate|show-config]
       [--log-to-console][--help]
       [subcommand_options]
```

Creates a backup of a Shardman cluster and restores the Shardman cluster from a backup using pg_probackup.

List of subcommands:

`init`

Initializes a new repository folder for the Shardman cluster backup and creates a configuration file on all nodes for connection to the backup storage if `--storage-type` is `S3`.

`archive-command`

Adds `archive_command` to each replication group (or to a single one if the `--shard` option is specified) and enables or disables it in the Shardman cluster.

`backup`

Creates a backup of the Shardman cluster.

`checkdb`

Verifies the Shardman cluster correctness by detecting physical and logical corruption.

`delete`

Deletes a backup of the Shardman cluster with the specified `backup_id`.

`merge`

Merges the backups that belong to a common incremental backup chain. The full backup merges the backups with their first incremental backup. The incremental backup merges the backups with their parent full backup, along with all the incremental backups between them. Once the merge is complete, the full backup covers all the merged data, and the incremental backups are removed as redundant. In this version, you cannot run the `merge` command using the S3 interface.

`restore`

Restores the Shardman cluster from the selected backup.

`show`

Shows the list of backups of the Shardman cluster.

`validate`

Checks the selected Shardman cluster backup for integrity.

`show-config`

Displays all the current pg_probackup configuration settings, including those that are specified in the `pg_probackup.conf` configuration file located in the *`backup_dir`*`/backups/`*`shard_name`* directory and those that were provided on a command line.

`set-config`

Adds the specified settings to the `pg_probackup.conf` or modifies those previously added.

The following options can be used with all `probackup` subcommands:

`--log-to-console`

Outputs a full `probackup` log to the console. By default, for each replication group the `probackup` log file is written to the backup directory (see `--backup-path` below) as the `<backup-directory>/backup/log/pg_probackup-<repgroup-name>.log` file. The log rotation file size is 20MB. If this value is reached, the log file is rotated once a `shardmanctl probackup validate` or `shardmanctl probackup backup` command is launched.

`--help`

Shows subcommand help.

`init`

Syntax:

```
shardmanctl [common_options] probackup init
-B|--backup-path path
-E|--etcd-path path
[--remote-port port]
[--remote-user username]
[--ssh-key path]
[-t|--timeout seconds]
[-m|--maxtasks number_of_tasks]
[--storage-type mount|remote|S3]
[--s3-config-only]
[--s3-config-path path]
```

```
[--s3-host S3_host]
[--s3-port S3_port]
[--s3-access-key S3_access_key]
[--s3-secret-key S3_secret_key]
[--s3-bucket S3_bucket]
[--s3-region S3_region]
[--s3-buffer-size size]
[--s3-retries number_of_retries]
[--s3-timeout time]
[--s3-https]
[-y|--yes]
```

Initializes a new repository folder for the Shardman cluster backup.

`-B path`
`--backup-path path`

Required if `--s3-config-only` is not used. Specifies the path to the backup catalog where Shardman cluster backups should be stored.

`-E path`
`--etcd-path path`

Required if `--s3-config-only` is not used. Specifies the path to the catalog where the etcd dumps should be stored.

`--remote-port port`

Specifies the remote ssh port for replication group instances.

Default: `22`.

`--remote-user username`

Specifies the remote ssh user for replication group instances.

Default: `postgres`.

`--ssh-key path`

Specifies the ssh private key for execution of remote ssh commands.

Default: `$HOME/.ssh/id_rsa`.

`--storage-type mount|remote|S3`

Type of the backup storage. If the value is `remote`, SSH is used to copy data files to the remote backup directory. But this behavior is different if a directory mounted to all nodes or an S3-compatible object storage is used to store backups. To specify these kinds of storage, the value of the `--storage-type` option is set to `mount` or `S3`, respectively.

Default: `remote`.

`--s3-config-path path`

Specifies the path where the S3 configuration file will be created on all Shardman nodes.

Default: `<shardman-data-dir>/s3.config`.

`--s3-config-only`

Create only S3 configuration files on all nodes and skip backup repository initialization. This flag is useful if the value of `--storage-type` is `S3`.

`--s3-host host`

Specifies the S3 host to connect to S3-compatible storage.

`--s3-port` *port*

Specifies the S3 port to connect to S3-compatible storage.

`--s3-access-key` *access-key*

Specifies the S3 access key to connect to S3-compatible storage.

`--s3-secret-key` *access-key*

Specifies the S3 secret key to connect to the S3-compatible storage.

`--s3-bucket` *bucket*

Specifies the bucket in the S3-compatible object storage for storing backups.

`--s3-region` *bucket*

Specifies the region in the S3-compatible object storage.

`--s3-buffer-size` *size*

Size of the read/write buffer for pg_probackup to communicate with the S3-compatible object storage, in MiB.

Default: 16.

`--s3-retries` *number_of_retries*

Maximum number of attempts for pg_probackup to execute an S3 request in case of failures.

Default: 5.

`--s3-timeout` *time*

Maximum allowable amount of time for pg_probackup to transfer data of size `--s3-buffer-size` to/from the S3-compatible object storage, in seconds.

Default: 300.

`--s3-https`

Specifies the HTTPS URL to connect to the S3-compatible object storage.

`-y|--yes`

Approve the operation regardless of whether the file specified in `--s3-config-path` exists.

`archive-command`

Syntax:

```
shardmanctl [common_options] probackup archive-command [add|rm]
        -B|--backup-path path
        [-j|--jobs count]
        [--compress]
        [--compress-algorithm algorithm]
        [--compress-level level]
        [--batch-size batch_size]
        [--storage-type mount|remote|S3]
        [--remote-port port]
        [--remote-user username]
        [-s|--shard shard-name]
        [--s3-config-path path]
        [-y|--yes]
```

Adds/removes and enables/disables the archive command for every replication group in the Shardman cluster to put WAL logs into the initialized backup repository.

`add`

Adds and enables the `archive` command for every replication group in the Shardman cluster.

`rm`

Disables the `archive` command in every replication group in the Shardman cluster. No additional options are required.

`-B` *`path`*
`--backup-path` *`path`*

Required when adding `archive_command`. Specifies the path to the backup catalog where the Shardman cluster backups should be stored.

`--batch-size` *`batch_size`*

To speed up the archiving, specify the `--batch-size` option to copy the WAL segments in batches of a specified size. If the `--batch-size` option is used, it is also possible to specify the `-j` option to copy a batch of the WAL segments on multiple threads.

`--jobs` *`count`*
`-j` *`count`*

The number of parallel threads that pg_probackup uses when creating a backup. Default: 1.

`--compress`

Enables backup compression. If this flag is not specified, compression will be disabled. If the flag is specified, the default `zstd` algorithm is used with the compression level set to 1, while other compression options are ignored even if they are specified.

`--compress-algorithm` *`algorithm`*

Defines the compression algorithm: `zlib`, `lz4`, `zstd`, `pglz`, or `none`. Once defined, it checks if the values are valid within the scale of the defined algorithm.

The supported compression algorithms depend on the version of Postgres Pro Enterprise that includes the pg_probackup used, as explained in *Compression Options*.

Default: `none`.

`--compress-level` *`level`*

Defines the compression level — 0-9 for `zlib`, 1 for `pglz`, 0-22 for `zstd`, and 0-12 for `lz4`.

Default: 1.

`--storage-type mount|remote|S3`

Type of the backup storage. If the value is `remote`, SSH is used to copy data files to the remote backup directory. But this behavior is different if a directory mounted to all nodes or an S3-compatible object storage is used to store backups. To specify these kinds of storage, the value of the `--storage-type` option is set to `mount` or `S3`, respectively.

Default: `remote`.

`--remote-port` *`port`*

Specifies the remote ssh port for replication group instances.

Default: `22`.

`--remote-user` *`username`*

Specifies the remote ssh user for replication group instances.

Default: `postgres`.

`-s|--shard` *shard-name*

Specifies the name of the shard where the `archive` command must be added, enabled or disabled. If not specified, the `archive` command is enabled or disabled for every shard.

`--s3-config-path` *path*

Specifies the path to the S3 configuration file.

Default: `<shardman-data-dir>/s3.config`.

`-y`
`--yes`

Confirm the restart instead of asking approval from the standard input. Only applies for the `add` command.

backup

Syntax:

```
shardmanctl [common_options] probackup backup -B|--backup-path path
        -E|--etcd-path path
        -b|--backup-mode MODE
        [-j|--jobs count]
        [--compress]
        [--compress-algorithm algorithm]
        [--compress-level level]
        [--batch-size batch_size]
        [--storage-type mount|remote|S3]
        [--remote-port port]
        [--remote-user username]
        [--ssh-key path]
        [-t|--timeout seconds]
        [-m|--maxtasks number_of_tasks]
        [--log-directory path]
        [--s3-config-path path]
        [--no-validate]
        [--skip-block-validation]
        [--log-to-console]
        [--retention-redundancy]
        [--retention-window]
        [--wal-depth]
        [--delete-wal]
        [--delete-expired]
        [--merge-expired]
        [-y | --yes]
        [--lock-lifetime]
```

Creates a backup of the Shardman cluster.

`-B` *path*
`--backup-path` *path*

Required. Specifies the path to the backup catalog where Shardman cluster backups should be stored.

`-E` *path*
`--etcd-path` *path*

Required. Specifies the path to the catalog where the etcd dumps should be stored.

`-b` *MODE*
`--backup-mode` *MODE*

Required. Defines the backup mode: `FULL`, `PAGE`, `DELTA`, `PTRACK`.

`--batch-size` *batch_size*

To speed up the archiving, specify the `--batch-size` option to copy the WAL segments in batches of a specified size. If the `--batch-size` option is used, it is also possible to specify the `-j` option to copy a batch of the WAL segments on multiple threads.

`--jobs` *count*
`-j` *count*

The number of parallel threads that pg_probackup uses when creating a backup. Default: 1.

`--compress`

Enables backup compression. If this flag is not specified, compression will be disabled. If the flag is specified, the default `zstd` algorithm is used with the compression level set to 1, while other compression options are ignored even if they are specified.

`--compress-algorithm` *algorithm*

Defines the compression algorithm: `zlib`, `lz4`, `zstd`, `pglz`, or `none`.

The supported compression algorithms depend on the version of Postgres Pro Enterprise that includes the pg_probackup used, as explained in *Compression Options*.

Default: `none`.

`--compress-level` *level*

Defines the compression level — 0-9 for `zlib`, 1 for `pglz`, 0-22 for `zstd`, and 0-12 for `lz4`.

Default: 1.

`--remote-port` *port*

Specifies the remote ssh port for replication group instances.

Default: `22`.

`--remote-user` *username*

Specifies the remote ssh user for replication group instances.

Default: `postgres`.

`--ssh-key` *path*

Specifies the ssh private key for execution of remote ssh commands.

Default: `$HOME/.ssh/id_rsa`.

`-t` *seconds*
`--timeout` *seconds*

Exit with error after waiting until the cluster is ready for the specified number of seconds.

`-m` *number_of_tasks*
`--maxtasks` *number_of_tasks*

Specifies the maximum number of concurrent tasks (`pg_probackup` commands) to run.

Default: number of logical CPUs of the system.

`--no-validate`

Skip automatic validation after the backup is taken. You can use this flag if you validate backups regularly and would like to save time when running backup operations.

Default: `false`.

`--skip-block-validation`

Disables block-level checksum verification to speed up the backup process.

Default: `false`.

`--storage-type mount|remote|S3`

Type of the backup storage. If the value is `remote`, SSH is used to copy data files to the remote backup directory. But this behavior is different if a directory mounted to all nodes or an S3-compatible object storage is used to store backups. To specify these kinds of storage, the value of the `--storage-type` option is set to `mount` or `S3`, respectively.

Default: `remote`.

`--log-to-console`

Enables output of the pg_probackup logs to the console.

Default: `false`.

`--log-directory` *`path`*

Specifies the directory for pg_probackup logs. Required if `--storage-type` is set to `S3` unless the `SDM_LOG_DIRECTORY` environment variable is set.

Default: `<backup-directory>/backup/log`.

`--s3-config-path` *`path`*

Specifies the path to the S3 configuration file.

Default: `<shardman-data-dir>/s3.config`.

`--retention-redundancy=`*`redundancy`*

Specifies the number of full backup copies to keep in the data directory. Must be a non-negative integer. The zero value disables this setting.

Default: current value of the `pg_probackup.conf` file, `0` if not specified.

`--retention-window=`*`window`*

Number of days of recoverability. Must be a non-negative integer. The zero value disables this setting.

Default: current value of the `pg_probackup.conf` file, `0` if not specified.

`--wal-depth=`*`wal_depth`*

Number of latest valid backups on every timeline that must retain the ability to perform PITR. Must be a non-negative integer. The zero value disables this setting.

Default: current value of the `pg_probackup.conf` file, `0` if not specified.

`--delete-wal`

Deletes WAL files that are no longer required to restore the cluster from any of the existing backups.

Default: `false`.

`--delete-expired`

Deletes backups that do not conform to the retention policy.

Default: `false`.

`--merge-expired`

Merges the oldest incremental backup that satisfies the requirements of retention policy with its parent backups that have already expired.

Default: `false`.

`-y`
`--yes`

Confirm the restart instead of asking approval from the standard input.

`--lock-lifetime`

Allows setting the maximum time that probackup can hold the lock, in seconds.

Default: 1800.

`checkdb`

Syntax:

```
shardmanctl [common_options] probackup checkdb
[--amcheck [--skip-block-validation] [--heapallindexed]] [--shard shard]
[-m|--maxtasks number_of_tasks]
```

Verifies the Shardman cluster correctness by detecting physical and logical corruption.

`--amcheck`

Performs logical verification of indexes if no corruption was found while checking data files. You must have the *amcheck* extension or the *amcheck_next* extension installed in the database to check its indexes. For databases without amcheck, index verification will be skipped. The amcheck extension is included with the Shardman package.

`--heapallindexed`

Checks that all heap tuples that should be indexed are actually indexed. You can use this flag only together with the `--amcheck` flag. This option is effective depending on the version of amcheck/amcheck_next installed. The amcheck extension included in the Shardman package supports this verification.

`--skip-block-validation`

Skip validation of data files. You can use this flag only together with the `--amcheck` flag, so that only logical verification of indexes is performed.

`--shard shard`

Perform the verification only on the specified shard. By default, the verification is performed on all shards.

`-m number_of_tasks`
`--maxtasks number_of_tasks`

Specifies the maximum number of concurrent tasks (`pg_probackup` commands) to run.

Default: number of logical CPUs of the system.

`delete`

Syntax:

```
shardmanctl [common_options] probackup delete -B|--backup-path path
        [-i|--backup-id backup_id]
        [-j|--jobs count]
        [-m|--maxtasks number_of_tasks]
        [--storage-type mount|remote|S3]
        [--s3-config-path path]
        [--delete-wal]
        [-y|--yes]
        [--retention-redundancy]
        [--retention-window]
        [--wal-depth]
        [--delete-expired]
        [--merge-expired]
```

Deletes a backup of the Shardman cluster with specified `backup_id` or launches the retention purge of backups and archived WAL that do not satisfy the current retention policies.

Note that `backup_id` cannot be used with `merge-expired` or `delete-expired`.

`-B` *`path`*
`--backup-path` *`path`*

Required. Specifies the path to the backup catalog (or key in the bucket of the S3-compatible storage) where Shardman cluster backups should be stored.

`-i` *`backup_id`*
`--backup-id` *`backup_id`*

Specifies the unique identifier of the backup.

`--jobs` *`count`*
`-j` *`count`*

The number of parallel threads that pg_probackup uses when creating a backup. Default: 1.

`-m` *`number_of_tasks`*
`--maxtasks` *`number_of_tasks`*

Specifies the maximum number of concurrent tasks (`pg_probackup` commands) to run.

Default: number of logical CPUs of the system.

`--storage-type mount|remote|S3`

Type of the backup storage. If the value is `remote`, SSH is used to copy data files to the remote backup directory. But this behavior is different if a directory mounted to all nodes or an S3-compatible object storage is used to store backups. To specify these kinds of storage, the value of the `--storage-type` option is set to `mount` or `S3`, respectively.

Default: `remote`.

To delete the backup that was created with a `--storage-type` option with a `S3` value, set a `--storage-type` option to a `S3` value in the `delete` command.

`--s3-config-path` *`path`*

Specifies the path to the S3 configuration file.

Default: `<shardman-data-dir>/s3.config`.

`--delete-wal`

Deletes WAL files that are no longer required to restore the cluster from any of the existing backups.

Default: `false`.

`-y`
`--yes`

Approve operation.

Default: `false`.

`--retention-redundancy=`*`redundancy`*

Specifies the number of full backup copies to keep in the data directory. Must be a non-negative integer. The zero value disables this setting.

Default: current value of the `pg_probackup.conf` file, `0` if not specified.

`--retention-window=`*`window`*

Number of days of recoverability. Must be a non-negative integer. The zero value disables this setting.

Default: current value of the `pg_probackup.conf` file, `0` if not specified.

`--wal-depth=`*`wal_depth`*

Number of latest valid backups on every timeline that must retain the ability to perform PITR. Must be a non-negative integer. The zero value disables this setting.

Default: current value of the `pg_probackup.conf` file, `0` if not specified.

`--delete-expired`

Deletes backups that do not conform to the retention policy.

Default: `false`.

`--merge-expired`

Merges the oldest incremental backup that satisfies the requirements of retention policy with its parent backups that have already expired.

Default: `false`.

`merge`

Syntax:

```
shardmanctl [common_options] probackup merge -B|--backup-path path
        -i|--backup-id backup_id
        [-j|--jobs count]
        [-m|--maxtasks number_of_tasks]
        [--no-validate]
        [--no-sync]
        [-y|--yes]
```

Merges the backups that belong to a common incremental backup chain. The full backup merges the backups with their first incremental backup. The incremental backup merges the backups with their parent full backup, along with all the incremental backups between them. Once the merge is complete, the full backup covers all the merged data, and the incremental backups are removed as redundant.

`-B` *`path`*
`--backup-path` *`path`*

Required. Specifies the path to the backup catalog where Shardman cluster backups should be stored.

`-i` *`backup_id`*
`--backup-id` *`backup_id`*

Required. Specifies the unique identifier of the backup.

`--jobs` *`count`*
`-j` *`count`*

The number of parallel threads that pg_probackup uses when creating a backup. Default: 1.

`-m` *`number_of_tasks`*
`--maxtasks` *`number_of_tasks`*

Specifies the maximum number of concurrent tasks (`pg_probackup` commands) to run.

Default: number of logical CPUs of the system.

`--no-sync`

Do not sync merged files to disk. You can use this flag to speed up the merge process. Using this flag can result in data corruption in case of operating system or hardware crash.

Default: `false`.

`--no-validate`

Skip automatic validation before and after merge.

Default: `false`.

`-y`
`--yes`

Approve the operation.

Default: `false`.

`restore`

Syntax:

```
shardmanctl [common_options] probackup restore
        -B|--backup-path path
        -i|--backup-id id
        -j|--jobs count
        [--recovery-target-time timestamp]
        [-I|--recovery-mode incremental_mode]
        [-t|--timeout seconds]
        [-m|--maxtasks number_of_tasks]
        [--metadata-only] [--schema-only] [--shard shard]
        [--no-validate]
        [--skip-block-validation]
        [--s3-config-path path]
        [--storage-type mount|remote|S3]
        [--wal-limit number_of_wal_segments]
        [--log-directory path]
        [--data-validate]
```

Restores a Shardman cluster from the selected backup.

`-B` *`path`*
`--backup-path` *`path`*

Required. Specifies the path to the backup catalog where Shardman cluster backups should be stored.

`-i` *`id`*
`--backup-id` *`id`*

Required. Specifies backup ID for restore.

`--jobs` *`count`*
`-j` *`count`*

The number of parallel threads that pg_probackup uses when restoring from a backup. Default: 1.

`--recovery-target-time` *`timestamp`*

Point-in-Time Recovery (PITR) option. Specifies the timestamp for restore. Example: '2024-01-25 15:30:36' in UTC.

`-I` *`incremental_mode`*
`--recovery-mode` *`incremental_mode`*

Specifies the incremental restore mode to be used. Possible values are:

- `checksum` — replace only pages with mismatched checksum and LSN.
- `lsn` — replace only pages with LSN greater than point of divergence.
- `none` — regular restore, default.

`-t` *seconds*
`--timeout` *seconds*

Exit with error after waiting until the cluster is ready or the recovery is complete for the specified number of seconds.

`--metadata-only`

Perform metadata-only restore. By default, full restore is performed.

`--schema-only`

Perform schema-only restore. By default, full restore is performed.

`--shard` *shard*

Perform restoring only on the specified shard. By default, restoring is performed on all shards.

`--no-validate`

Skip backup validation. You can use this flag if you validate backups regularly and would like to save time when running restore operations.

Default: `false`.

`--skip-block-validation`

Disable block-level checksum verification to speed up validation. During automatic validation before the restore only file-level checksums will be verified.

Default: `false`.

`--s3-config-path` *path*

Specifies the path to the S3 configuration file.

Default: `<shardman-data-dir>/s3.config`.

`--storage-type mount|remote|S3`

Type of the backup storage. If the value is `remote`, SSH is used to copy data files to the remote backup directory. But this behavior is different if a directory mounted to all nodes or an S3-compatible object storage is used to store backups. To specify these kinds of storage, the value of the `--storage-type` option is set to `mount` or `S3`, respectively. When creating backup with a `--storage-type` option with a `S3` value, set `--storage-type` option to a `S3` value in the `restore` command.

Default: `remote`.

`--wal-limit` *number_of_wal_segments*

Specifies the number of WAL segments in which the closest synchronization points will be searched in the case of PITR.

Default: 0 — no limit.

`--log-directory` *path*

Specifies the directory for pg_probackup logs. Required if `--storage-type` is set to `S3` unless the `SDM_LOG_DIRECTORY` environment variable is set.

Default: `<backup-directory>/backup/log`.

`--data-validate`

If enabled, verifies data with `probackup validate` before restoring.

Default: false.

show

Syntax:

```
shardmanctl [common_options] probackup show
        -B|--backup-path path
        [-f|--format table|json]
        [--archive ]
        [-i|--backup-id backup-id]
        [--instance instance]
        [--storage-type mount|remote|S3]
        [--s3-config-path path]
```

Shows the list of backups of the Shardman cluster.

`-B path`
`--backup-path path`

Required. Specifies the path to the backup catalog where Shardman cluster backups should be stored.

`-f table|json`
`--format table|json`

Specifies the output format.

Default: `table`.

`--archive`

Shows the WAL archive information.

`-i backup-id`
`--backup-idbackup-id`

Shows information about the specific backups.

`--instanceinstance`

Shows information about the specific instance.

`--s3-config-path path`

Specifies the path to the S3 configuration file.

Default: `<shardman-data-dir>/s3.config`.

`--storage-type mount|remote|S3`

Type of the backup storage. If the value is `remote`, SSH is used to copy data files to the remote backup directory. But this behavior is different if a directory mounted to all nodes or an S3-compatible object storage is used to store backups. To specify these kinds of storage, the value of the `--storage-type` option is set to `mount` or `S3`, respectively. To show a backup that was created with the `S3` value of `--storage-type`, set `--storage-type` to `S3` in the `show` command.

Default: `remote`.

show-config

Syntax:

```
shardmanctl [common_options] probackup show-config
    -B backup_path
    [--format=text|json]
    [--no-scale-units]
    -s|--shard shard_name
    [--s3-config-path path]
    [--storage-type mount|remote|S3]
```

Displays all the current pg_probackup configuration settings, including those that are specified in the `pg_probackup.conf` configuration file located in the *`backup_dir`*`/backups/`*`shard_name`* directory and those that were provided on a command line.

`-B` *`string`*
`--backup-path=`*`string`*

Required. Specifies the absolute path to the backup catalog.

`--format text|json`

Specifies the output format.

Default: `text`.

`--no-scale-units`

Output the configuration parameter values for the time and the amount of memory in the default units.

Default: `false`.

`-s` *`string`*
`--shard=`*`string`*

A name of the shard to execute the `show-config` command for.

`--s3-config-path` *`path`*

Specifies the path where the S3 configuration file will be created on all Shardman nodes.

Default: `<shardman-data-dir>/s3.config`.

`--storage-type mount|remote|S3`

Type of the backup storage. If the value is `remote`, SSH is used to copy data files to the remote backup directory. But this behavior is different if a directory mounted to all nodes or an S3-compatible object storage is used to store backups. To specify these kinds of storage, the value of the `--storage-type` option is set to `mount` or `S3`, respectively.

Default: `remote`.

`validate`

Syntax:

```
shardmanctl [common_options] probackup validate
        -B|--backup-path path
        -i|--backup-id id
        [-t|--timeout seconds]
        [-m|--maxtasks number_of_tasks]
        [--log-to-console]
        [--storage-type mount|remote|S3]
        [--s3-config-path path]
        [--log-directory path]
        [--remote-port port]
        [--remote-user username]
```

Checks the selected Shardman cluster backup for integrity.

`-B` *`path`*
`--backup-path` *`path`*

Required. Specifies the path to the backup catalog where Shardman cluster backups should be stored.

`-i` *`id`*
`--backup-id` *`id`*

Required. Specifies backup ID for validation.

`--log-to-console`

Enables output of pg_probackup logs to the console.

Default: `false`.

`-t` *`seconds`*
`--timeout` *`seconds`*

Exit with error after waiting until the cluster is ready for the specified number of seconds.

`-m` *`number_of_tasks`*
`--maxtasks` *`number_of_tasks`*

Specifies the maximum number of concurrent tasks (`pg_probackup` commands) to run.

Default: number of logical CPUs of the system.

`--s3-config-path` *`path`*

Specifies the path to the S3 configuration file.

Default: `<shardman-data-dir>/s3.config`.

`--storage-type mount|remote|S3`

Type of the backup storage. If the value is `remote`, SSH is used to copy data files to the remote backup directory. But this behavior is different if a directory mounted to all nodes or an S3-compatible object storage is used to store backups. To specify these kinds of storage, the value of the `--storage-type` option is set to `mount` or `S3`, respectively. To validate a backup that was created with the `S3` value of `--storage-type`, set `--storage-type` to `S3` in the `validate` command.

Default: `remote`.

`--log-directory` *`path`*

Specifies the directory for pg_probackup logs. Required if `--storage-type` is set to `S3` unless the `SDM_LOG_DIRECTORY` environment variable is set.

Default: `<backup-directory>/backup/log`.

`--remote-port` *`port`*

Specifies the remote ssh port for replication group instances.

Default: `22`.

`--remote-user` *`username`*

Specifies the remote ssh user for replication group instances.

Default: `postgres`.

`--ssh-key` *`path`*

Specifies the ssh private key for execution of remote ssh commands.

Default: `$HOME/.ssh/id_rsa`.

`set-config`

Syntax:

```
shardmanctl [common_options] probackup set-config
        [--archive-timeout int]
        [-B | --backup-path string]
        [-m |--maxtasks int]
        [--remote-port int]
        [--remote-user string]
        [--retention-redundancy int]
        [--retention-window int]
        [--wal-depth int]
```

```
        [--s3-config-path string]
        [-s |--shard string]
        [--storage-type string]
```

Adds the specified settings to the `pg_probackup.conf` or modifies those previously added.

`--archive-timeout` *`int`*

Sets a timeout for the WAL segment archiving and streaming, in seconds.

Default: pg_probackup waits for 300 seconds.

`-B` *`string`*
`--backup-path=`*`string`*

Specifies the absolute path to the backup catalog.

`-m` *`int`*
`--maxtasks=`*`int`*

Specifies the maximum number of concurrent tasks (`pg_probackup` commands) to run.

Default: number of logical CPUs of the system.

`--remote-port` *`int`*

An SSH remote backup port.

Default: `22`.

`--remote-user` *`string`*

An SSH remote backup user.

`--retention-redundancy` *`int`*

Specifies the number of the full backup copies to store in the data directory. It must be set to a non-negative integer. The zero value disables this setting.

Default: `0`.

`--retention-window` *`int`*

A number of days of recoverability. It must be set to a non-negative integer. The zero value disables this setting.

Default: `0`.

`--wal-depth` *`int`*

A number of the latest valid backups on every timeline that must retain the ability to perform PITR. Must be set to a non-negative integer. The zero value disables this setting.

`--s3-config-path` *`string`*

A path to the S3 configuration file.

Default: `/var/lib/pgpro/sdm-14/data/s3.config`

`-s` *`string`*
`--shard=`*`string`*

A name of the shard to make the `set-config` command for. If not specified, the command is run for all the shards.

Default: current value of the `pg_probackup.conf` file.

`--storage-type` *`string`*

A backup storage type, the possible values are `remote`, `mount`, `S3`.

Default: `remote`.

`rebalance`

Syntax:

```
shardmanctl [common_options] rebalance [-f|--force]
```

Rebalances sharded tables.

`-f`
`--force`

Perform forced rebalance of sharded tables whose partitions were manually moved.

`recover`

Syntax:

```
shardmanctl [common_options] recover [--info file] [--dumpfile file] [--shard shard]
 [--metadata-only][--schema-only] [--timeout seconds]
```

Restores a Shardman cluster from a backup created by the `backup` command.

`--dumpfile file`

Required for metadata-only restore.

Specifies the file to load the etcd metadata dump from.

`--info file`

Required for full restore.

Specifies the file to load information about the backup from.

`--shard shard`

Perform restoring only on the specified shard. By default, restoring is performed on all shards.

`--metadata-only`

Perform metadata-only restore. By default, full restore is performed.

`--schema-only`

Perform schema-only restore. By default, full restore is performed.

`--timeout seconds`

Exit with error after waiting until the cluster is ready or the recovery is complete for the specified number of seconds.

For more details, see the section called “Restoring a Shardman Cluster”

`restart`

Syntax:

```
shardmanctl [common_options] restart [-y|--yes] [--no-wait]
```

Restarts a Shardman cluster.

`-y`
`--yes`

Confirm the operation instead of asking approval from the standard input.

`--no-wait`

Do not wait for the replicas to start.

shard add

Syntax:

`shardmanctl [common_options] shard -s|--shard shard_name add -n|--nodes node_names [--no-wait]`

Adds a replica to a shard.

`-s shard_name`
`--shard=shard_name`

Shard name.

`-n node_names`
`--nodes=node_names`

Specifies the comma-separated list of replica nodes to be added.

`--no-wait`

Do not wait for the shard to start.

shard master set

Syntax:

`shardmanctl [common_options] shard -s|--shard shard_name master set -n| node node_names`

Sets the precedence for a certain primary server for a specified shard.

`-s shard_name`
`--shard=shard_name`

Shard name.

`master set`

Primary server with precedence.

`-n node_names`
`--nodes=node_names`

Specifies the comma-separated list of replica nodes.

shard master reset

Syntax:

`shardmanctl [common_options] shard -s|--shard shard_name master reset`

Resets the parameters of the master with precedence for the shard.

`-s shard_name`
`--shard=shard_name`

Shard name.

`master reset`

Resets the parameters of the master with precedence for the shard.

`-n node_names`
`--nodes=node_names`

Specifies the comma-separated list of replica nodes.

shard add

Syntax:

```
shardmanctl [common_options] shard -s|--shard shard_name reset [--yes | -y][--new-
primary | -p]
```

Resets nodes of a replication group if they are in a state of hanging.

`-s shard_name`
`--shard=shard_name`

Shard name.

`-y`
`--yes`

Confirm the operation instead of asking approval from the standard input.

`--new-primary`
`-p`

New primary node host.

shard rm

Syntax:

```
shardmanctl [common_options] shard -s|--shard shard_name rm -n|--nodes node_names
 [-f|--force]
```

Removes a replica from a shard.

`-s shard_name`
`--shard=shard_name`

Shard name

`-n node_names`
`--nodes=node_names`

Specifies the comma-separated list of replica nodes to be removed.

`-f`
`--force`

Perform forced removal of the node, even if it is dead.

shard switch

Syntax:

```
shardmanctl [common_options] shard -s|--shard shard_name switch [--new-
primary node_names]
```

Switches the primary node.

`-s shard_name`
`--shard=shard_name`

Shard name.

`--new-primary=node_names`

New primary node host.

shard start

Syntax:

```
shardmanctl [common_options] shard -s |--shard shard_name start [--no-wait] [-n|--
node node_name]
```

Starts the shard.

`-s` *`shard_name`*
`--shard=`*`shard_name`*

Shard name.

`--no-wait`

Do not wait for the shard to start.

`-n` *`node_name`*
`--node=`*`node_name`*

Specifies the node to start.

`shard stop`

Syntax:

```
shardmanctl [common_options] shard -s |--shard shard_name stop [-n|--
node node_name]
```

Stops the shard.

`-s` *`shard_name`*
`--shard=`*`shard_name`*

Shard name.

`-n` *`node_name`*
`--node=`*`node_name`*

Specifies the node to stop.

`shard replicas reinit`

Syntax:

```
shardmanctl [common_options] shard -s|--shard shard_name replicas reinit [-n|--
node node_names] [-y|--yes] [--no-wait]
```

Resets replicas of a specific shard.

`-s` *`shard_name`*
`--shard=`*`shard_name`*

Shard name.

`-n` *`node_names`*
`--node=`*`node_names`*

Specifies the node on which to reset replicas. If not specified, checks shard replicas on all nodes.

`-y`
`--yes`

Confirm the operation instead of asking approval from the standard input.

`--no-wait`

Do not wait wait for replicas to become ready.

For more details, see the section called “Reinitializing Replicas”

`nodes start`

Syntax:

```
shardmanctl [common_options] nodes start -n|--nodes node_names [--no-wait]
```

Starts the nodes.

`-n node_names`
`--nodes=node_names`

Node names.

`--no-wait`

Sets shardmanctl not to wait for the nodes to start.

`nodes restart`

Syntax:

```
shardmanctl [common_options] nodes restart -n|--nodes node_names [--no-wait]
```

Restarts the nodes.

`-n node_names`
`--nodes=node_names`

Node names.

`--no-wait`

Do not wait for the nodes to restart.

`nodes stop`

Syntax:

```
shardmanctl [common_options] nodes stop -n|--nodes node_names [--no-wait]
```

Stops the nodes.

`-n node_names`
`--nodes=node_names`

Node names.

`--no-wait`

Do not wait for the nodes to stop.

`status`

Syntax:

```
shardmanctl [common_options] status [-f|--format table|json] [--
filter store|metadata|shardmand|rg|master|dictionary|all|restart_required_params] [-
s|--sort node|rg|status]
```

Reports on the health status of Shardman cluster subsystems.

`-f table|json`
`--format=table|json`

Specifies the report format.

Default: `table`.

For more details, see the section called “Getting the Status of Cluster Subsystems”.

`--filter store|metadata|shardmand|rg|master|dictionary|allrestart_required_params`

Specifies subsystems whose status information should be included in the output.

Default: `all`.

For more details, see the section called “Getting the Status of Cluster Subsystems”.

`-s node|rg|status`
`--sort node|rg|status`

Sort messages inside one group (table) as specified.

Default: `node`.

For more details, see the section called “Getting the Status of Cluster Subsystems”.

`status transactions`

Syntax:

```
shardmanctl [common_options] status transactions [-r|--
repgroup replication_group_name
```

Shows distributed transactions that Shardman built-in monitoring tools failed to resolve.

`-r replication_group_name`
`--repgroup=replication_group_name`

Specifies the replication group for which to output transactions.

Default: all replication groups.

For more details, see the section called “Outputting the List of Unresolved Distributed Transactions”.

`store dump`

Syntax:

```
shardmanctl [common_options] store dump [-f|--file filename]
```

Dumps current cluster specifications from the store.

`-f filename`
`--file=filename`

Specifies the output file (`-` for stdout).

Default: `-`.

For more details, see the section called “Dumping All Keys from the Store to Debug Error Configuration”.

`store restore`

Syntax:

```
shardmanctl [common_options] store restore [--delete-old-keys][-f|--file filename]
[-y|--yes]
```

Allows to safely restore the `etcd` cluster from the dump. To do this, shardmand must be disabled on every shard. Also, it only works for the cold backup.

`--delete-old-keys`

Clean all the `etcd` keys before restoring.

`-f` *`filename`*
`--file=`*`filename`*

Specifies the name of the `etcd` keys dump.

`-y`
`--yes`

Perform automatic confirmation.

`store lock`

Syntax:

```
shardmanctl [common_options] store lock [-f|--format text|json]
```

Shows the current cluster meta lock information.

`-f=text|json`
`--format=text|json`

Specifies the output format.

Default: `text`.

For more details, see the section called “Output Current Cluster Meta Lock Information”.

`store get`

Syntax:

```
shardmanctl [common_options] store get [[-a|--alias aliasname]|[-k|--key keyname]
 [-f|--file filename]]
```

Gets the specified key from the store.

`-a` *`aliasname`*
`--alias=ladle|cluster |spec|stolonspec`

Specifies the use of alias instead of the full key name. Cannot be used with `--key`.

For more details, see the section called “Getting the Current stolon Specification”.

`-k` *`keyname`*
`--key=`*`keyname`*

Specifies the key to retrieve from the store. Cannot be used with `--alias`.

For more details, see the section called “Getting the Current stolon Specification”.

`-f` *`filename`*
`--file=`*`filename`*

Specifies the file to print the value to.

Default: `-` (stdout).

For more details, see the section called “Getting the Current stolon Specification”.

`store keys`

Syntax:

```
shardmanctl [common_options] store keys
```

Gets all keys with the current cluster prefix from the store.

For more details, see the section called “Getting the Cluster and Ladle Key Names For the Current Cluster”.

`store set`

Syntax:

```
shardmanctl [common_options] store set [[-a|--alias aliasname]|[-k|--key keyname]]
 [-f|--file filename]
```

Creates or rewrites a key in the store.

`-a ladle|cluster |spec|stolonspec`
`--alias=ladle|cluster |spec|stolonspec`

Specifies the use of alias instead of the full key name. Cannot be used with `--key`.

`-k keyname`
`--key=keyname`

Specifies the key name to set in the store. Cannot be used with `--alias`.

`-f filename`
`--file=filename`

Specifies the file with input data (`-` for stdin).

For more details, see the section called “Setting a New Spec for the Cluster”.

`tables sharded info`

Syntax:

```
shardmanctl [common_options] tables sharded info [-t|--table table_name]
```

Gets information about a sharded table.

`-t table`
`--table=table`

Specifies the name of the table in the format `schema.table`

`tables sharded list`

Syntax:

```
shardmanctl [common_options] tables sharded list
```

Gets the list of all sharded tables.

`tables sharded norebalance`

Syntax:

```
shardmanctl [common_options] tables sharded norebalance
```

Gets the list of sharded tables with automatic rebalancing disabled.

`tables sharded partmove`

Syntax:

```
shardmanctl [common_options] tables sharded partmove [-t|--table table_name] [-s|--
shard shard_name] [-p|--partnum number]
```

Moves the specified partition of a sharded table to a new shard.

`-t` *`table`*
`--table=`*`table`*

Specifies the name of the table in the format `schema.table`.

`-p` *`number`*
`--partnum=`*`number`*

Specifies the number of the partition to move.

`-s` *`shard_name`*
`--shard=`*`shard_name`*

Specifies the name of the new shard for the partition.

`tables sharded rebalance`

Syntax:

```
shardmanctl [common_options] tables sharded rebalance [-t|--table table_name]
```

Enables and runs automatic data rebalancing for the selected sharded table.

`-t` *`table`*
`--table=`*`table`*

Specifies the name of the table in the format `schema.table`.

`config get`

Syntax:

```
shardmanctl [common_options] config get [-c | --choose-revision] [-r | --revision ] [-f
 | --file]
```

Outputs the current full cluster specification or a configuration of the specified revision.

`-c`
`--choose-revision`

Enables an interactive mode of choosing a configuration of the specified revision.

`-r`
`--revision`

ID of a configuration revision.

`-f` *`file_name`*
`--file=`*`file_name`*

Name of a file for writing the configuration. If not specified, the value is stdout.

`config revisions rm`

Syntax:

```
shardmanctl [common_options] config revisions rm [-r | --revision ] [-y | --yes]
```

Deletes a specified configuration revision from history.

`-r`
`--revision`

ID of a configuration revision. If not specified, enables an interactive mode of choosing a configuration of the specified revision. This is a timestamp of an operation that resulted in Shardman configuration change.

`-y`
`--yes`

Perform automatic confirmation.

config update

Syntax:

```
shardmanctl [common_options] config update [[-f|--file stolon_spec_file|
shardman_spec_file]|spec_text [-p|--patch][-w|--wait]] [--force] [-y | --yes]
```

Updates the stolon or full Shardman configuration.

`-f stolon_spec_file|shardman_spec_file`
`--specfile=stolon_spec_file|shardman_spec_file`

Specifies the file with the stolon or full Shardman configuration. The configuration file type is determined automatically. The value of `-` means the standard input. By default, the configuration is passed in *`spec_text`*.

`-w`
`--wait`

Sets shardmanctl to wait for configuration changes to take effect. If a new configuration cannot be loaded by all replication groups, shardmanctl will wait forever.

`-p`
`--patch`

Merge the new configuration into the existing one. By default, the new configuration replaces the existing one.

`--force`

Perform forced update if a cluster operation is in progress.

`-y`
`--yes`

Confirm the restart necessary for the parameters to take effect. If this option is not specified, and the parameters update requires a restart, the manual confirmation will be requested. If not confirmed, the cluster will continue to work, yet the new parameter values will only take effect after the restart.

config rollback

Syntax:

```
shardmanctl [common_options] config rollback [-r | --revision] [-w|--
wait time_duration] [--force] [-y|--yes]
```

Makes a rollback of Shardman to one of the previous states. When rolling back to the config revision that has `max_connections`, `max_prepared_transactions`, or `max_worker_processes` parameters, the replicas are reinitialized.

`-r`
`--revision`

ID of a revision the rollback must be made to. It is a timestamp of an operation that resulted in Shardman configuration change.

If not specified, a user is presented with a list of revisions that he can choose from.

`-w`
`--wait`

Sets shardmanctl to wait for configuration changes to take effect. If a new configuration cannot be loaded by all replication groups, shardmanctl will wait forever.

Default: `1h`.

`-f`
`--force`

Perform forced setting of a parameter if a cluster operation is in progress.

`-y`
`--yes`

Perform automatic confirmation.

`config revisions`

Syntax:

`shardmanctl [common_options] config revisions [-f|--format text|json]`

Outputs the revision history of the Shardman cluster configuration. It has the following information for each revision:

- `revision_id` — timestamp of the command that resulted in the Shardman cluster configuration change
- `host` — name of the host from which this command was executed
- `user` — user who executed this command
- `command` — the command itself

`-f=text|json`
`--format=text|json`

Specifies the output format.

Default: `text`.

`config revisions set`

Syntax:

`shardmanctl [common_options] config revisions set [--keep-config-revisions]`

Allows setting the length of the configuration revision history. This length cannot be lower than 5, in which case it is automatically set to 5. For Shardman clusters where the configuration revision history was not collected yet, the length is automatically set to 20.

`--keep-config-revisions`

A limit on the number of revisions for one Shardman configuration. If the limit is lower than the current history length, the older versions out of this limit will be deleted. Also, if the number of operations resulting in configuration changes exceeds the limit, the oldest revision is deleted.

Default: `20`.

`config update ip`

Syntax:

`shardmanctl [common_options] config update ip [-u|ip_1=ip_2,hostname_1=hostname_2][-y|--yes]`

Updates the specified node IPs in the cluster.

`-u`
`ip_1=ip_2,hostname_1=hostname_2`

Specifies the node IPs to be updated.

`-y`
`--yes`

Perform automatic confirmation.

`set`

Syntax:

`shardmanctl [common_options] set pgParam1=value1 [pgParam2=value2 [...]] [-y|--yes] [-w|--wait time_duration] [-f|--force]`

Sets the values of the specified Shardman cluster database parameters.

`-w`
`--wait`

Sets shardmanctl to wait for configuration changes to take effect. Value examples: `2h45m`, `1m30s`, `5m`, `10s`.

Default: `1h`.

`-y`
`--yes`

Confirm the restart necessary for the parameters to take effect. If this option is not specified, and the parameters update requires a restart, the manual confirmation will be requested. If not confirmed, the cluster will continue to work, yet the new parameter values will only take effect after the restart.

`-f`
`--force`

Perform forced setting of a parameter if a cluster operation is in progress.

`upgrade`

Syntax:

`shardmanctl [`*`common_options`*`] upgrade`

Upgrades the shardman database extension and updates `pg_foreign_server` options.

`bench init`

Syntax:

`shardmanctl [`*`common_options`*`] bench init [--schema-type single|simple|shardman|custom] [--schema-file `*`file_name`*`] [-s|--scale `*`scale_value`*`] [-n|--no-vacuum] [-F|--fillfactor `*`fillfactor_value`*`]`

Initializes the benchmark schema via pgbench. Schema can be custom or predefined. Creates `tpc-b` schema tables and fills them.

`--schema-type=single|simple|shardman|custom`

Type of schema used by schema initialization. Possible values:

- `single` — schema for a single PostgreSQL benchmark test
- `simple` — simple sharded schema
- `shardman` — sharded schema optimized for Shardman
- `custom` — schema initialized by the user from the `--schema-file` file

Default schema: `shardman`.

`--schema-file=`*`file_name`*

File with DDL query for the custom schema type, to be used to create `tpc-b` tables for pgbench: `pgbench_accounts`, `pgbench_branches`, `pgbench_tellers`, `pgbench_history`.

`-s `*`scale_value`*
`--scale=`*`scale_value`*

Multiply the number of generated rows by the given scale factor.

`-n`
`--no-vacuum`

Perform no vacuuming during initialization.

`-F `*`fillfactor_value`*
`--fillfactor=`*`fillfactor_value`*

Fill pgbench tables with the given fillfactor value.

`bench run`

Syntax:

```
shardmanctl [common_options] bench run [--schema-type single|simple|shardman|custom]
[-f|--file file_name] [-c|--client client_value]  [-C|--connect] [--full-output]
[-j|--jobs jobs_value][-T|--time seconds][-t|--transactions transactions_value]
[-s|--scale scale_factor] [ -P | --progress seconds] [ -R | --rate rate] [ -M | --
protocol querymode]
```

Runs the initialized benchmark via pgbench. Can use the default pgbench script or a custom script from a file.

`--schema-type=single|simple|shardman|custom`

Type of schema used by schema initialization (`bench init`). Possible values:

- `single` — schema for single PostgreSQL benchmark
- `simple` — simple sharded schema
- `shardman` — sharded schema optimized for Shardman
- `custom` — schema initialized by the user from the `--schema-file` file.

Default schema: `shardman`.

`-f` *`file_name`*
`--file=`*`file_name`*

Add a transaction script read from *`filename`* to the list of scripts to be executed.

Optionally, write an integer weight after `@` to adjust the probability of selecting this script versus other ones. The default weight is `1`. (To use a script file name that includes an `@` character, append a weight so that there is no ambiguity, for example `filen@me@1`).

`-c` *`client_value`*
`--client=`*`client_value`*

Number of clients simulated, that is, number of concurrent database sessions.

`-C`
`--connect`

Establish a new connection for each transaction rather than doing it just once per client session.

`--full-output`

Print all pgbench output.

`-j` *`jobs_value`*
`--jobs=`*`jobs_value`*

Number of worker threads within pgbench.

`-s` *`scale_factor`*
`--scale=`*`scale_factor`*

Multiply the number of generated rows by + the given scale factor.

`-T` *`seconds`*
`--time=`*`seconds`*

Run the test for this many seconds instead of a fixed number of transactions per client.

`-t` *`transactions_value`*
`--transactions=`*`transactions_value`*

Number of transactions each client runs.

Default: 10.

`-P` *`seconds`*
`--progress=`*`seconds`*

Show progress report every *`sec`* seconds. The report includes the time since the beginning of the run, the TPS since the last report, and the transaction latency average, standard deviation, and the number of failed transactions since the last report. Under throttling (`-R`), the latency is computed with respect to the transaction scheduled start time, not the actual transaction beginning time, thus it also includes the average schedule lag time. When `--max-tries` is used to enable transaction retries after serialization/deadlock errors, the report includes the number of retried transactions and the sum of all retries.

`-R` *`rate`*
`--rate=`*`rate`*

Execute transactions targeting the specified rate instead of running as fast as possible (the default). The rate is given in transactions per second. If the targeted rate is above the maximum possible rate, the rate limit won't impact the results.

`-M` *`querymode`*
`--protocol=`*`querymode`*

Protocol to use for submitting queries to the server:

- `simple`: use simple query protocol.
- `extended`: use extended query protocol.
- `prepared`: use extended query protocol with prepared statements.

In the `prepared` mode, pgbench reuses the parse analysis result starting from the second query iteration, so pgbench runs faster than in other modes.

Default: `simple`.

`bench cleanup`

Syntax:

`shardmanctl [`*`common_options`*`] bench cleanup`

Cleans up schema database after benchmarks. Drops `tpc-b` tables.

`bench generate`

Syntax:

`shardmanctl [`*`common_options`*`] bench generate [-c|--config `*`file_name`*`] [-o|--output-file `*`file_name`*`]`

Gets the benchmark configuration from a file and generates a bash script to create a schema optimized for Shardman and run the benchmark using pgbench. The configuration file must be in `yaml` format.

`-f` *`file_name`*
`--file=`*`file_name`*

The configuration file path. The file contains a sequence of script confugurations. Each script must have a `schema_type`: `single|simple|shardman|custom`. For a custom schema it is necessary to specify the `schema_file` with the DDL script. Optional parameters: `init_flags` (default set: `-s 1000`), `run_flags` (default set: `-n -P 10 -c 10 -j 4 -T 60`), `partitions` (default value: `50`). It is highly recomended to use `-n` (`--no-vacuum`) parameter inside `run_flags`. Configuration file example:

```
benches:
- schema_type: single
  init_flags: "-s 3"
  run_flags: "-n -P 10 -c 10 -j 4 -T 10"
- schema_type: simple
  init_flags: "-s 4"
  run_flags: "-n -P 10 -c 20 -j 4 -T 10"
  partitions: 100
- schema_type: shardman
```

```
          init_flags: "-s 5"
          run_flags: "-n -P 10 -c 20 -j 4 -T 10"
        - schema_type: custom
          init_flags: "-s 6"
          schema_file: "schema.psql"
```

`-o` *`file_name`*
`--output-file=`*`file_name`*

Output file. Default: stdout.

`script`

Syntax:

`shardmanctl [`*`common_options`*`] script -s|--shard `*`shard_name`*`][[-f|--file `*`file_name`*`][--sql `*`query`*`]]`

Executes non-transactional commands from a file or from the command-line on the specified shards.

`-s` *`shard_name`*
`--shard=`*`shard_name`*

Shard name.

`-f` *`file_name`*
`--file=`*`file_name`*

Add a transaction script from the *`file_name`* file to the list of scripts to be executed.

`--sql` *`query`*

Specifies the statement to be executed and can only be used separately from `-f`.

`psql`

Syntax:

`shardmanctl [`*`common_options`*`] psql -s|--shard `*`shard_name`*

Connects to the first available primary node if no options are specified.

`-s` *`shard_name`*
`--shard=`*`shard_name`*

Name of the shard. If specified, the connection is installed with this shard current primary.

`daemon set`

Syntax:

`shardmanctl [`*`common_options`*`] daemon set [--session-log-level debug | info | warn | error] [--session-log-format json|text] [--session-log-nodes]`

Allows updating the log parameters “on the fly”.

`--session-log-level debug | info | warn | error`

Updates the log level to `debug`, `info`, `warn`, or `error`.

`--session-log-format json|text`

Updates the log output format to `text` or `json`.

`--session-log-nodes`

Specifies which cluster nodes must be updated. If not specified, the parameters are updated on every node.

Default: all nodes.

`history`

Syntax:

```
shardmanctl [common_options] history [--reverse | -r] [-f|--format json|text] [-l|--
limit number_of_commands]
```

Shows history of the commands that updated the cluster. By default, they are sorted from the most recent to the oldest ones.

`-r`
`--reverse`

Switches to the ascending sorting order.

`-f json|text`
`--format=json|text`

Output format.

Default: `text`.

`-l`
`--limit=number_of_commands`

Limit for the number of the most recent commands in the output. The maximum value is 200.

Default: 20.

Common Options

shardmanctl common options are optional parameters that are not specific to the utility. They specify etcd connection settings, cluster name and a few more settings. By default shardmanctl tries to connect to the etcd store `127.0.0.1:2379` and use the `cluster0` cluster name. The default log level is `info`.

`-h, --help`

Show brief usage information.

`--cluster-name` *`cluster_name`*

Specifies the name for a cluster to operate on. The default is `cluster0`.

`--log-level` *`level`*

Specifies the log verbosity. Possible values of *`level`* are (from minimum to maximum): `error`, `warn`, `info` and `debug`. The default is `info`.

`--retries` *`number`*

Specifies how many times shardmanctl retries a failing etcd request. If an etcd request fails, most likely, due to a connectivity issue, shardmanctl retries it the specified number of times before reporting an error. The default is 5.

`--session-timeout` *`seconds`*

Specifies the session timeout for shardmanctl locks. If there is no connectivity between shardmanctl and the etcd store for the specified number of seconds, the lock is released. The default is 30.

`--store-endpoints` *`string`*

Specifies the etcd address in the format: `http[s]://address[:port](,http[s]://address[:port])*`. The default is `http://127.0.0.1:2379`.

`--store-ca-file` *`string`*

Verify the certificate of the HTTPS-enabled etcd store server using this CA bundle.

`--store-cert-file` *`string`*

Specifies the certificate file for client identification by the etcd store.

`--store-key` *`string`*

Specifies the private key file for client identification by the etcd store.

`--store-timeout` *`duration`*

Specifies the timeout for a etcd request. The default is 5 seconds.

`--monitor-port` *`number`*

Specifies the port for the shardmand http server for metrics and probes. The default is 15432.

`--api-port` *`number`*

Specifies the port for the shardmand http api server. The default is 15432.

`--version`

Show shardman-utils version information.

Environment

`SDM_BACKUP_MODE`

An alternative to setting the `--backup-mode` option.

`SDM_BACKUP_PATH`

An alternative to setting the `--backup-path` option.

`SDM_CLUSTER_NAME`

An alternative to setting the `--cluster-name` option.

`SDM_ETCD_PATH`

An alternative to setting the `--etcd-path` option.

`SDM_FILE`

An alternative to setting the `--file` option for `config update`.

`SDM_LOG_LEVEL`

An alternative to setting the `--log-level` option.

`SDM_NODES`

An alternative to setting the `--nodes` option for `nodes add` and `nodes rm`.

`SDM_RETRIES`

An alternative to setting the `--retries` option.

`SDM_SPEC_FILE`

An alternative to setting the `--spec-file` option for `init`.

`SDM_STORE_ENDPOINTS`

An alternative to setting the `--store-endpoints` option.

`SDM_STORE_CA_FILE`

An alternative to setting the `--store-ca-file` option.

`SDM_STORE_CERT_FILE`

An alternative to setting the `--store-cert-file` option.

`SDM_STORE_KEY`

An alternative to setting the `--store-key` option.

`SDM_STORE_TIMEOUT`

An alternative to setting the `--store-timeout` option.

`SDM_SESSION_TIMEOUT`

An alternative to setting the `--session-timeout` option.

Usage

Adding Nodes to a Shardman Cluster

To add nodes to a Shardman cluster, run the following command:

```
shardmanctl [common_options] nodes add -n|--nodes node_names
```

You must specify the `-n` (`--nodes`) option to pass the comma-separated list of nodes to be added. Nodes can be referred by their hostname or IP address. Hostnames must be correctly resolved on all nodes.

If `nodes add` command fails during execution, use the `cleanup --after-node-operation` command to fix possible cluster configuration issues.

Performing Cleanup

By default, `cleanup` operates in the report-only mode, that is, the following command will only show actions to be done during actual cleanup:

```
shardmanctl [common_options] cleanup --after-node-operation|--after-rebalance
```

To perform the actual cleanup, run the following command:

```
shardmanctl [common_options] cleanup -p|--processrepgroups --after-node-
operation|--after-rebalance
```

Displaying the Cluster Topology

`cluster topology` displays the current cluster topology. The default is the table mode. All cluster nodes will be grouped by the replication groups they belong to. For each node, its status will be displayed.

```
shardmanctl [common_options] cluster topology -f|--format table|json|text
```

Checking shardmand Service on Nodes

`daemon check` not only checks that shardmand service is running on specified nodes, but also assures those services are configured for the same cluster as shardmanctl:

```
shardmanctl [common_options] daemon check -n|--nodes node_names
```

Removing Nodes from a Shardman cluster

To remove nodes from a Shardman cluster, run the following command:

```
shardmanctl [common_options] nodes rm -n|--nodes node_names
```

Specify the `-n` (`--nodes`) option to pass the comma-separated list of nodes to be removed.Recreates all partitions of sharded tables

> **Note**
>
> Do not use the `cleanup` command to fix possible cluster configuration issues after a failure of `nodes rm`. Redo the `nodes rm` command instead.

To remove all nodes in a cluster and not care about the data, just reinitialize the cluster. If a removed replication group contains local (non-sharded and non-global) tables, the data is silently lost after the replication group removal.

Getting the Status of Cluster Subsystems

To get a report on the health status of Shardman cluster in a table format for `metadata` and `store` subsystems sorted by replication group, run the following command:

```
shardmanctl [common_options] status --filter=metadata,store --sort=rg
```

To get the report in JSON format, use `-f|--format=json` option (omitted above since `table` format is used by default). Each detected issue is reported as an Unknown, Warning, Error or Fatal error status. The tool can also report an Operational error, which means there was an issue during the cluster health check. When the command encounters a Fatal or Operational error, it stops further diagnostics. For example, an inconsistency in the store metadata does not allow correct cluster operations and must be handled first.

Outputting the List of Unresolved Distributed Transactions

To view the list of distributed transactions that Shardman built-in monitoring tools failed to resolve, run the following command:

```
shardmanctl [common_options] status transactions -r|--
repgroup replication_group_name
```

Each output transaction consists of `tx_id` (transaction ID), `coordinator_id`, `creation_time` and `description` (error or transaction status). To display the list of transactions for a specific replication group, use the `-r|--repgroup` option (for all replication groups by default). In case there are no such transactions, returns `null` value in JSON.

Dumping All Keys from the Store to Debug Error Configuration

After facing an error while using Shardman cluster, to fill in an exhaustive report, it is convinient to dump all specifications that could produce such an error with the following command:

```
shardmanctl [common_options] store dump -f|--file filename
```

Some harmless errors may be shown, but they will not interrupt dumping. If you do not specify the filename, dump will be sent to stdout and may pollute your terminal.

Getting the Current stolon Specification

To get the current stolon specification, which is normally a part of cluster key in the store, use the following command:

```
shardmanctl [common_options] store get -a|--alias stolonspec -f|--file filename
```

If the cluster key is corrupted itself, stolon specification will not be shown either. Instead of using the alias, you may also find out the full cluster data key name (by listing all keys with `store keys` command), use `store get` to retrieve it and find the stolon part there. Mind that while using the last option, `shardman.config_uuid` parameter will not be deleted, which may result in a conflict in later use of this data; for manipulation with stolon specification, it is recommended to use `shardmanctl store get -a stolonspec` command.

Getting the Cluster and Ladle Key Names For the Current Cluster

To get all key names in the store at once, run the following command:

```
shardmanctl [common_options] store keys
```

It can only be shown in JSON format. It will also print alias names for keys that have them (excluding `stolonspec` and `spec`, since they are parts of other keys)

Output Current Cluster Meta Lock Information

You can view information about current cluster meta locks that acquired by any command:

```
shardmanctl [common_options] store lock -f|--format json
```

To get the report in JSON format, use `-f|--format=json` option (omitted above since `text` format is used by default). In case the lock does not exists returns `Lock not found`

Setting a New Spec for the Cluster

To set a new spec part of the cluster specification, run the following command:

```
shardmanctl [common_options] store set --alias=spec --file=spec.json
```

Since `spec` is a part of cluster data key, it cannot be set with `--key`. If the provided file is not a valid JSON, the new spec part will not be set.

Backing up a Shardman Cluster

Requirements for backing up and restoring a Shardman cluster using the `basebackup` command are listed in Section 2.6.1.1.

To backup a Shardman cluster, you can run the following command:

```
shardmanctl [common_options] backup --datadir directory [--use-ssh]
```

You must pass the directory to write the output to through the `--datadir` option. You can limit the number of running concurrent tasks (`pg_receivewal` or `pg_basebackup` commands) by passing the limit through the `--maxtasks` option.

If `--use-ssh` is specified shardmanctl `recover` command will use `scp` command to restore data. It allows to use backup repository on the local host.

Registering a Shardman Cluster

To register a Shardman cluster in the etcd store, run the following command:

```
shardmanctl [common_options] init [-y|--yes] [-f|--spec-
file spec_file_name]|spec_text
```

You must provide the string with the cluster specification. You can do it as follows:

- On the command line — do not specify the `-f` option and pass the string in *`spec_text`*.
- On the standard input — specify the `-f` option and pass `-` in *`spec_file_name`*.
- In a file — specify the `-f` option and pass the filename in *`spec_file_name`*.

Restoring a Shardman Cluster

shardmanctl can perform either full restore, metadata-only or schema-only restore of a Shardman cluster from a backup created by the `backup` command.

To perform full restore, you can run the following command:

```
shardmanctl [common_options] recover --info file
```

Pass the file to load information about the backup from through the `--info` option. In most cases, set this option to point to the `backup_info` file in the backup directory or to its modified copy.

If you encounter issues with an etcd instance, it makes sense to perform metadata-only restore. To do this, you can run the following command:

```
shardmanctl [common_options] recover --dumpfile file --metadata-only
```

You must pass the file to load the etcd metadata dump from through the `--dumpfile` option.

If you need to restore only schema information, like: tables, roles and etc. you should specify `--schema-only` option.

For all kinds of restore, you can specify `--timeout` for the tool to exit with error after waiting until the cluster is ready or the recovery is complete for the specified number of seconds.

You can specify `--shard` parameter for restoring only on the single shard.

Before running the `recover` command, specify `DataRestoreCommand` and `RestoreCommand` in the `backup_info` file. `DataRestoreCommand` fetches the base backup and restores it to the stolon data directory. `RestoreCommand` fetches the WAL file and saves it to stolon `pg_wal` directory. These commands can use the following substitutions:

`%p`

Destination path on the server.

`%s`

`SystemId` of the restored database (the same in the backup and in restored cluster).

`%f`

Name of the WAL file to restore.

stolon `keeper` thread runs both commands on each node in the cluster. Therefore:

- Make the backup accessible to these nodes (for example, by storing it in a shared filesystem or by using a remote copy protocol, such as SFTP).
- Commands to fetch the backup are executed as the operating system user under which stolon daemons work (usually `postgres`), so set the permissions for the backup files appropriately.

These examples show how to specify `RestoreCommand` and `DataRestoreCommand`:

- If a backup is available through a passwordless SCP, you can use:

```
"DataRestoreCommand": "scp -r user@host:/var/backup/shardman/%s/backup/* %p",
"RestoreCommand": "scp user@host:/var/backup/shardman/%s/wal/%f %p"
```

- If a backup is stored on NFS and available through `/var/backup/shardman` path, you can use:

```
"DataRestoreCommand": "cp -r /var/backup/shardman/%s/backup/* %p",
"RestoreCommand": "cp /var/backup/shardman/%s/wal/%f %p"
```

Backing up a Shardman Cluster Using `probackup` Command

Requirements for backing up and restoring a Shardman cluster using the `probackup` command are listed in Section 2.6.3.1.

For example, following these requirements, on the backup host:

```
groupadd postgres
useradd -m -N -g postgres -r -d /var/lib/postgresql -s /bin/bash
```

Then add SSH keys to provide passwordless SSH connection between the backup host and Shardman cluster hosts. Then on the backup host:

```
apt-get install pg-probackup shardman-utils
mkdir -p directory
chown postgres:postgres directory -R
shardmanctl [common_options] probackup init --backup-path=directory --etcd-
path=directory/etcd --remote-user=postgres --remote-port=22
shardmanctl [common_options] probackup archive-command --backup-path=directory --
remote-user=postgres --remote-port=22
```

If all the requirements are met, then run the `backup` subcommand for the cluster backup:

```
shardmanctl [common_options] probackup backup --backup-path=directory --etcd-
path=directory --backup-mode=MODE
```

You must pass the directories through the `--backup-path` and `--etcd-path` options and backup mode through `--backup-mode`. Full and delta backups are available with `FULL`, `DELTA`, `PTRACK` and `PAGE` values. Also it is possible to specify backup

compression options through `--compress`, `--compress-algorithm` and `--compress-level` flags, as well as specify `--remote-port` and `--remote-user` flags. You can limit the number of running concurrent tasks when doing backup by passing the limit through the `--maxtasks` flag.

By default, copying data via SSH is used to create a backup. To copy data to a mounted partition instead, use the `--storage-type` option with the `mount` value. This value will be automatically used in the restore process.

You can also copy data to an S3-compatible object storage. To do this, use the `--storage-type` option with the `S3` value. When this value is used, it is required to specify the directory for pg_probackup logs. You can do it either by specifying `--log-directory` for each command or set the environment variable `SDM_LOG_DIRECTORY`, for example:

```
export SDM_LOG_DIRECTORY=/backup/logs
```

If you are going to perform backup/restore only for an S3-compatible object storage, you can also set an environment variable instead of specifying `--storage-type` in each `probackup` command:

```
export SDM_STORAGE_TYPE=S3
```

Restoring a Shardman Cluster using probackup command

shardmanctl in probackup mode can perform either full restore, metadata-only or schema-only restore of a Shardman cluster from a backup created by the `probackup backup` command.

To perform full or partial restore, firstly you must select needed backup to restore from. To show list of available backups run the following command:

`shardmanctl [`*`common_options`*`] probackup show --backup-path=`*`path`*` --format=`*`format`*` [--archive ] [-i|--backup-id `*`backup-id`*`] [--instance `*`instance`*`]`

The output should be a list of backups with their IDs in a table or JSON format. Then pick the needed backup ID and run the probackup restore command.

`shardmanctl [`*`common_options`*`] probackup restore --backup-path=`*`path`*` --backup-id=`*`id`*

Pass the path to the repo through the `--backup-path` option and backup ID througt `--backup-id` flag.

If you encounter issues with an etcd instance, it makes sense to perform metadata-only restore. To do this, you can run the following command:

`shardmanctl [`*`common_options`*`] probackup restore --backup-path=`*`path`*` --backup-id=`*`id`*` --metadata-only`

If you need to restore only schema information, like: tables, roles and etc. you should specify `--schema-only` option.

For both kinds of restore, you can specify `--timeout` for the tool to exit with error after waiting until the cluster is ready or the recovery is complete for the specified number of seconds.

You can specify `--shard` parameter for restoring only on the single shard.

Also you can specify `--recovery-target-time` option for Point-in-Time Recovery. In this case Shardman finds the closest syncpoint to specified timestamp and suggests restoring on the found LSN. You can also specify `--wal-limit` to limit the number of WAL segments to be processed.

> **Important**
>
> Before restoring a Shardman cluster, make sure that the cluster is up by executing the `shardmanctl status` command. If the output shows errors, performing the restore can result in the cluster becoming unavailable. First, fix the errors by reinitializing the cluster and restoring the etcd metadata. Then you can proceed to restoring the cluster from backup.

Reinitializing Replicas

If replicas are in an incorrect state, you can reset them using the shardmanctl command:

```
shardmanctl [common_options] shard --shard=shard_name replicas reinit
```

This command determines the nodes on which replicas of the specified shard are running and sends a request to shardmand on these nodes. After receiving this request, shardmand clears the postgres data directory and restarts the `keeper` thread that is responsible for managing the replica. After that, the replicas are restarted and begin to receive data from the corresponding primary.

Examples

Initializing the Cluster

To initialize a Shardman cluster that has the `cluster0` name, uses an etcd cluster consisting of `n1`,`n2` and `n3` nodes listening on port 2379, ensure proper settings in the spec file `sdmspec.json` and run:

```
$ shardmanctl --store-endpoints http://n1:2379,http://n2:2379,http://n3:2379 init -f
 sdmspec.json
```

Getting the Cluster Connection String

To get the connection string for a Shardman cluster that has the `cluster0` name, uses an etcd cluster consisting of `n1`,`n2` and `n3` nodes listening on port 2379, run:

```
$ shardmanctl --store-endpoints http://n1:2379,http://n2:2379,http://n3:2379
getconnstr

dbname=postgres host=n1,n4,n2,n1,n1,n2,n4,n3 password=yourpasswordhere
port=5432,5433,5432,5433,5432,5433,5432,5433 user=postgres
```

To add replicas to `getconnstr`, use `--all`.

Getting the Cluster Status

Here is a sample status output from shardmanctl with OK and Error statuses:

```
        $ shardmanctl status --filter store,shardmand,rg --sort=node

##########################################################################################
#                                          == STORE STATUS ==
                  #
##########################################################################################
#   STATUS   #                  MESSAGE                  #   REPLICATION GROUP   #
       NODE         #
##########################################################################################
#     OK     # etcd store is OK                          #                       #
                  #
##########################################################################################
##########################################################################################
#                                       == SHARDMAND STATUS ==
                   #
##########################################################################################
#   STATUS   #                  MESSAGE                  #   REPLICATION GROUP   #
       NODE         #
##########################################################################################
#     OK     # shardmand on node 56d819b4e9e4 is OK      #                       #
   56d819b4e9e4     #
##########################################################################################
#     OK     # shardmand on node 6d0aabd50acc is OK      #                       #
   6d0aabd50acc     #
##########################################################################################
##########################################################################################
#                                     == REPLICATION GROUP STATUS ==
                   #
```

```
#########################################################################################################
#   STATUS   #                    MESSAGE                    #   REPLICATION GROUP   #
        NODE         #
#########################################################################################################
#     OK     # Replication group clover-1-56d819b4e9e4 is   # clover-1-56d819b4e9e4 #
                     #
#            # OK                                           #                       #
                     #
#########################################################################################################
#            # Replication connection is down for slave     #                       #
                     #
#    Error   # 6d0aabd50acc:5442 in replication group       # clover-1-6d0aabd50acc #
 6d0aabd50acc:5442   #
#            # clover-1-6d0aabd50acc                        #                       #
                     #
#########################################################################################################
#########################################################################################################

#                            == RESTART REQUIRED PARAMS STATUS ==
   #

#########################################################################################################

#  STATUS  #                MESSAGE               # REPLICATION GROUP #        NODE
   #

#########################################################################################################

#    OK    #     No pending restart parameters    #      shard-1      #       shrn1
   #

#########################################################################################################

#    OK    #     No pending restart parameters    #      shard-2      #       shrn4
   #

#########################################################################################################
```

Rewriting stolon Specification

First, get the list of available keys in the store using the following command:

```
$ shardmanctl store keys
```

```
{
 "Key": "shardman/cluster0/data/cluster",
 "Alias": "cluster"
}{
 "Key": "shardman/cluster0/data/shardmand/56d819b4e9e4"
}{
...
 "Key": "shardman/cluster0/stolon/remoteLogs/6d0aabd50acc/clover-1-6d0aabd50acc/
keeper_1/error"
}
```

Get stolon configuration from the store and save it in the `stolonspec.json` file with the command

```
$ shardmanctl store get -a stolonspec -f stolonspec.json
```

Apply the necessary changes to the file and upload the new specification using `shardmanctl config update`. Mind that `shardman.config_uuid` parameter is deleted with `shardmanctl store get -a stolonspec` and not with `shardmanctl store get -k full/path/to/clusterspec`; using spec with existing `shardman.config_uuid` will result in a conflict.

> **Important**
>
> Do not use `store set` command to update cluster configurations because it *does not* apply a new specification on all nodes, it only writes it to the store. For the above example with stolon specification, `shardmanctl config update` is acceptable.

To double-check, you can get the cluster key with new `StolonSpec` by the full key name (which was shown earlier with `store keys` command):

```
$ shardmanctl store get -k shardman/cluster0/data/cluster
```

```
{
  "FormatVersion": 1,
  "Spec": {
    "PgSuAuthMethod": "md5",
    "PgSuPassword": "12345",
    "PgSuUsername": "postgres",
    "PgReplAuthMethod": "md5",
    "PgReplPassword": "12345",
    "PgReplUsername": "repluser",
    "ShardSpec": {
      ...
}
```

Adding Nodes to the Cluster

To add `n1`,`n2`, `n3` and `n4` nodes to the cluster, run:

```
$ shardmanctl --store-endpoints http://n1:2379,http://n2:2379,http://n3:2379 nodes add
 -n n1,n2,n3,n4
```

> **Important**
>
> The number of nodes being added must be a multiple of `Repfactor + 1` if `cross` placement policy is used.

Removing Nodes from the Cluster

To remove `n1` and `n2` nodes from the `cluster0` cluster, run:

```
$ shardmanctl --store-endpoints http://n1:2379,http://n2:2379,http://n3:2379
nodes rm -n n1,n2
```

If `cross` placement policy is used, then the clovers that contain them will be deleted along with the nodes.

Executing a Query on All Replication Groups

To execute the `select version()` query on all replication groups, run:

```
 $ shardmanctl --store-endpoints http://n1:2379,http://n2:2379,http://n3:2379 forall --
sql 'select version()'

 Node 1 says:
 [PostgreSQL 13.1 on x86_64-pc-linux-gnu, compiled by gcc (Ubuntu
 9.3.0-17ubuntu1~20.04) 9.3.0, 64-bit]
 Node 4 says:
 [PostgreSQL 13.1 on x86_64-pc-linux-gnu, compiled by gcc (Ubuntu
 9.3.0-17ubuntu1~20.04) 9.3.0, 64-bit]
 Node 3 says:
 [PostgreSQL 13.1 on x86_64-pc-linux-gnu, compiled by gcc (Ubuntu
 9.3.0-17ubuntu1~20.04) 9.3.0, 64-bit]
 Node 2 says:
 [PostgreSQL 13.1 on x86_64-pc-linux-gnu, compiled by gcc (Ubuntu
 9.3.0-17ubuntu1~20.04) 9.3.0, 64-bit]
```

Performing Rebalance

To rebalance sharded tables in the `cluster0` cluster, run:

```
$ shardmanctl --store-endpoints http://n1:2379,http://n2:2379,http://n3:2379 rebalance
```

Updating PostgreSQL Configuration Settings

To set the `max_connections` parameter to `200` in the cluster, create the spec file (for instance, `~/stolon.json`) with the following contents:

```
{
  "pgParameters": {
    "max_connections": "200"
  }
}
```

Then run:

```
$ shardmanctl --store-endpoints http://n1:2379,http://n2:2379,http://n3:2379 config
update -p -f ~/stolon.json
```

Since changing `max_connections` requires a restart, DBMS instances are restarted by this command.

Performing Backup and Recovery

To create a backup of the `cluster0` cluster using etcd at `etcdserver` listening on port `2379` and store it in the local directory `/var/backup/shardman`, run:

```
$ shardmanctl --store-endpoints http://etcdserver:2379 backup --datadir=/var/backup/
shardman --use-ssh
```

Assume that you are performing a recovery from a backup to the `cluster0` cluster using etcd at `etcdserver` listening on port `2379` and you take the backup description from the `/var/backup/shardman/backup_info` file. Edit the `/var/backup/shardman/backup_info` file, set `DataRestoreCommand` , `RestoreCommand` as necessary and run:

```
$ shardmanctl --store-endpoints http://etcdserver:2379 recover --info /var/backup/
shardman/backup_info
```

For metadata-only restore, run:

```
$ shardmanctl --store-endpoints http://etcdserver:2379 recover --metadata-only --
dumpfile /var/backup/shardman/etcd_dump
```

For schema-only restore, run:

```
$ shardmanctl --store-endpoints http://etcdserver:2379 recover --schema-only --
dumpfile /var/backup/shardman/etcd_dump
```

For single shard restore, run:

```
$ shardmanctl --store-endpoints http://etcdserver:2379 recover  --info /var/backup/
shardman/backup_info --shard shard_1
```

Performing Backup and Recovery with probackup Command

To create a backup of the `cluster0` cluster using etcd at `etcdserver` listening on port `2379` and store it in the local directory `/var/backup/shardman`, first initialize the backups repository with the `init` subcommand:

```
$ shardmanctl --store-endpoints http://etcdserver:2379 probackup init --backup-path=/
var/backup/shardman --etcd-path=/var/backup/etcd_dump
```

Then add and enable `archive_command` with the `archive-command` subcommand:

```
$ shardmanctl --store-endpoints http://etcdserver:2379 probackup archive-command add --
backup-path=/var/backup/shardman
```

If the repository is successfully initialized and `archive-command` successfully added, create a `FULL` backup with the `backup` subcommand:

```
$ shardmanctl --store-endpoints http://etcdserver:2379 probackup backup --backup-path=/
var/backup/shardman --etcd-path=/var/backup/etcd_dump --backup-mode=FULL --compress --
compress-algorithm=zlib --compress-level=5
```

To create `DELTA`, `PTRACK` or `PAGE` backup, run the `backup` subcommand with `DELTA`, `PTRACK` or `PAGE` value of the `--backup-mode` option:

```
$ shardmanctl --store-endpoints http://etcdserver:2379 probackup backup --backup-path=/
var/backup/shardman --etcd-path=/var/backup/etcd_dump --backup-mode=DELTA --compress --
compress-algorithm=zlib --compress-level=5
```

To show the created backup ID, run `show` subcommand:

```
$ shardmanctl --store-endpoints http://etcdserver:2379 probackup show --backup-path=/
var/backup/shardman --format=table
```

```
########################################################################################
#
               == BACKUP ID 'S88FRO'
                                                #
#
                        ==
                                                #
########################################################################################
#              INSTANCE             #                HOST               #
 RECOVERY TIME           #    MODE    #  WAL MODE  #     TLI    #    DATA    #     WAL
   #   Z-RATIO  #  START LSN #  STOP LSN  #   STATUS   #
########################################################################################
#              shard-1              #               n1                  #
 2024-02-02 14:19:05+00      #    FULL    #   ARCHIVE  #     1/0    #  42.37MiB  #
 16MiB   #    1.00    #  0/C000028 #  0/D0018B0 #     OK     #
########################################################################################
#              shard-2              #               n2                  #
 2024-02-02 14:19:05+00      #    FULL    #   ARCHIVE  #     1/0    #  42.38MiB  #
 16MiB   #    1.00    #  0/C000028 #  0/D001E00 #     OK     #
########################################################################################
```

In PTRACK backup mode, Shardman tracks page changes on the fly. Continuous archiving is not necessary for it to operate. Each time a relation page is updated, this page is marked in a special PTRACK bitmap. Tracking implies some minor overhead on the database server operation, but speeds up incremental backups significantly.

If you are going to use PTRACK backups, complete the following additional steps:

- Preload the ptrack shared library on each node. This can be done by adding the `ptrack` value to the `shared_preload_libraries` parameter.
- #reate the PTRACK extension on each cluster node:

```
$ shardmanctl --store-endpoints http://etcdserver:2379 forall --sql "create extension ptrack"
```

- To enable tracking page updates, set the `ptrack.map_size` parameter as follows:

```
$ shardmanctl --store-endpoints http://etcdserver:2379 update '{"pgParameters":{"ptrack.map_size":"64"}}'
```

  For optimal performance, it is recommended to set `ptrack.map_size` to `N/1024`, where `N` is the maximum size of the cluster node, in MB. If you set this parameter to a lower value, PTRACK is more likely to map several blocks together, which leads to false-positive results when tracking changed blocks and increases the incremental backup size as unchanged blocks can also be copied into the incremental backup. Setting `ptrack.map_size` to a higher value does not affect PTRACK operation, but it is not recommended to set this parameter to a value higher than 1024.

To validate the created backup, run `validate` subcommand:

```
$ shardmanctl --store-endpoints http://etcdserver:2379 probackup validate --backup-path=/var/backup/shardman --backup-id=RFP1FI
```

Assume that you are performing a recovery from a backup to the `cluster0` cluster using etcd at `etcdserver` listening on port `2379` and you take the backup ID from the `show` command:

```
$ shardmanctl --store-endpoints http://etcdserver:2379 probackup restore --backup-path=/var/backup/shardman --backup-id=RFP1FI
```

Finally we need to enable `archive_command` back.

```
$ shardmanctl --store-endpoints http://etcdserver:2379 probackup archive-command add --backup-path=/var/backup/shardman
```

For metadata-only restore, run:

```
$ shardmanctl --store-endpoints http://etcdserver:2379 probackup restore --metadata-only --backup-path=/var/backup/shardman --backup-id=RFP1FI
```

For metadata-only restore, run:

```
$ shardmanctl --store-endpoints http://etcdserver:2379 probackup restore --schema-only --backup-path=/var/backup/shardman --backup-id=RFP1FI
```

For single shard restore, run:

```
$ shardmanctl --store-endpoints http://etcdserver:2379 probackup restore --backup-path=/var/backup/shardman --backup-id=RFP1FI --shard shard_1
```

For Point-in-Time Recovery, run:

```
$ shardmanctl --store-endpoints http://etcdserver:2379 probackup restore --metadata-only --backup-path=/var/backup/shardman --backup-id=RFP1FI --recovery-target-time='2006-01-02 15:04:05' -s
```

Loading Data from a Text File

To load data into a Shardman cluster, run the following command:

```
$ shardmanctl --store-endpoints http://etcdserver:2379 load --file=/var/load/data.tsv --table=mytable --source file --format text -j 8
```

In this example, data is loaded from the `/var/load/data.tsv` data file (tab-delimited) into the table `mytable` in 8 parallel threads. You can use `schema.table` as the table name.

Loading data from PostgreSQL table

To load data into a Shardman cluster from a PostgreSQL table, run the following command:

```
$ shardmanctl --store-endpoints http://etcdserver:2379 load -t desttable --
source postgres --source-connstr "dbname=db host=srchost port=srcport user=login
 password=passwd" --source-table sourcetable -j 8
```

In this example, data is loaded from the table `sourcetable` into the `desttable` table in 8 parallel threads. You can use `schema.table` as table names.

Loading Data with a Schema from PostgreSQL

To load data with a schema into Shardman cluster from PostgreSQL, run the following command:

```
$ shardmanctl --store-endpoints http://etcdserver:2379 load --schema load_schema.yaml
```

The file `load_schema.yaml` has the folowing format:

```
version: "1.0"
migrate:
  connstr: "dbname=workdb host=workhost port=workport user=workuser
 password=workpassword"
  jobs: 8
  batch: 1000
  options:
    - create_schema
    - create_table
    - create_index
    - create_sequence
    - create_foreign_key
    - create_role
    - copy_ownership
    - copy_grants
    - truncate_table
    - skip_no_pkey_tables
    - skip_create_index_error
    - skip_create_extension_error
    - skip_load_errors
    - skip_create_foreign_key_error
    - skip_create_role_error
    - skip_copy_grants_error
    - skip_copy_ownership_error
  schemas:
    - name: public
      all: false
      tables:
        - name: tab1
          type: sharded
          partitions: 6
          distributedby: id
          priority: 3
        - name: tab2
          type: global
        - name: tab3
          type: sharded
```

```
          partitions: 6
          distributedby: field_id
          colocatewith: tab1
        - name: table4
            type: global
            source: schema.view
            source_pk: field_id
        - name: table5
            type: global
            source: schema.func(arg)
            source_pk: field_id
    - name: schema2
      all: false
      default_type: sharded
      default_partitions: 6
      tables:
        - name: table1
          distributedby: field_id
          priority: 2
        - name: table2
          type: global
        - name: table3
          source: schema.view
          distributedby: field_id
          priority: 3
        - name: table4
          distributedby: field_id
          source: schema.func(arg)
        - name: table5
          source: schema."complex.""table.name"
          distributedby: field_id
    - name: schema3
      all: true
      skip_tables: [table1, table2, table3]
  roles:
    - name: test_user1
      password: test_password
    - name: test_user2
```

The `migrate.jobs` value defines the number of parallel data loader processes.

The `migrate.batch` value is the number of rows in one batch (recommended value is 1000).

The `migrate.schemas` section defines an array of source database schemas that you are working with. All other schemas will be skipped.

If the `all` value is set to `true`, then all tables from the current schema will be migrated (with `global` type by default). If a table is listed in the `migrate.schemas.tables` array, then the target table type must be explicitly specified for it. Two types of tables are currently supported: `global` and `sharded`. `Global` tables are loaded first, then `sharded` tables and at the end `sharded` tables with the `colocatedwith` parameter. The order of loading tables of the same type can be changed using `priority` option.

The `migrate.schemas.skip_tables` section defines an array of table names that will be skipped when the schema is loaded even if the `all` parameter is set to `true`.

For sharded tables, the following attributes must be set: `distributedby` (specifies the name of the column to use for the table partitioning) and `partitions` (number of partitions that will be created for this table). Optionally, for sharded tables `colocate-with` attribute can be set (name of the table to colocate with). Shardman will try to place partitions of the created table with the same partition key on the same nodes as the corresponding partitions of the table specified by `colocatewith`.

You can specify the table `default_type` option for a schema: `global` or `sharded` (default: `global`). For the `sharded` type you can also specify the `default_partitions` option (default: 20). If you set `default_type` to `sharded`, you need to specify the `distributedby` option for each table.

The `source` option for a table should include the schema and table source: `schema.source`. The source can be a table, view or function. For example: `public.table`, `public.view`, `public.func(arg)`. If you set the `source` view or function for a global table, you should specify `source_pk` to set the primary key for this table. If `source` is not specified or contains the name of a table, you can also specify `source_pk` to create a primary key or override the existing one.

The `priority` option for table determines the order in which the tables of the same type are loaded. Tables with higher `priority` are loaded earlier. Default `priority` value is 0.

The `migrate.roles` section defines an array of role names and passwords that will be copied from the source database if `create_role` is specified.

The schema supports the following options:

- `create_schema` — create database schemas if they do not exist.
- `create_table` — create tables if they do not exist.
- `create_index` — create indexes after creating tables.
- `create_sequence` — create sequences if they do not exist.
- `create_foreign_key` — create foreign keys after creating tables.
- `truncate_table` — truncate tables before data load.
- `create_role` — create global roles defined in `migrate.roles` and copy role parameters from the source database.
- `copy_grants` — copy access privileges from the source database.
- `copy_ownership` — change of table owners to the owner in the source database.
- `skip_no_pkey_tables` — skip tables without primary keys.
- `skip_create_index_error` — skip index creation errors.
- `skip_create_extension_error` — skip extension creation errors.
- `skip_load_errors` — continue loading if errors occur.
- `skip_create_foreign_key_error` — skip foreign key creation errors.
- `skip_create_role_error` — skip role creation errors.
- `skip_copy_ownership_error` — skip table owner changing errors.
- `skip_copy_grants_error` — skip errors when copying access privuleges from the source database.

Initialization and Running Benchmarks

To initialize a benchmark via shardmanctl using pgbench with the `shardman` schema, scale=`1000`, partitions=`40`, run:

```
$ shardmanctl bench init --schema-type=shardman --scale=1000 --partitions=40
```

To run an initialized benchmark for the same `shardman` schema, number of jobs=`4`, number of clients=`10`, duration in seconds=`60` and full pgbench output, use:

```
$ shardmanctl bench run --schema-type=shardman --jobs=4 --client=10 --time=60 --full-output
```

To initialize a benchmark with the `custom` schema from file `schema.psql` with scale=`1000` run:

```
$ shardmanctl bench init --schema-type=custom --schema-file=schema.psql --scale=1000
```

To run an initialized benchmark with the `custom` schema and custom transaction script from `script.psql` with the number of jobs=`4`, number of clients=`10`, duration in seconds=`60`, use:

```
$ shardmanctl bench run --schema-type=custom --file=script.psql --jobs=4 --client=10 --
time=60
```

To clean up a PostgreSQL database of `tpc-b` tables, use:

```
$ shardmanctl bench cleanup
```

Benchmark Generation Scripts

To generate a benchmark sequence via shardmanctl from the config file=`cfg.yaml` and output the result to file=`script.sh`, run:

```
$ shardmanctl bench generate --config=cfg.yaml --output-file=script.sh
```

Configuration file example:

```
benches:
- schema_type: single
  init_flags: "-s 3"
  run_flags: "-n -P 10 -c 10 -j 4 -T 10"
- schema_type: simple
  init_flags: "-s 4"
  run_flags: "-n -P 10 -c 20 -j 4 -T 10"
  partitions: 100
- schema_type: shardman
  init_flags: "-s 5"
  run_flags: "-n -P 10 -c 20 -j 4 -T 10"
- schema_type: custom
  init_flags: "-s 6"
  schema_file: "schema.psql"
```

See Also

sdmspec.json , shardmand

sdmspec.json

sdmspec.json — Shardman initialization file

Synopsis

```
sdmspec.json
```

Description

shardmanctl uses the `sdmspec.json` configuration file during Shardman cluster initialization. A `shardman-utils` package provides a sample configuration file.

`sdmspec.json` file contains basic filesystem paths used by Shardman, global settings of the cluster, database-related settings, i. e., administrative and replication user logins and authentication method, FDW parameters and shard configuration (`ShardSpec`).

Note that there is a number of the internal Shardman parameters that, if modified by user, can result in the total cluster failure. These parameters are:

- `shardman.cluster_uuid` defines the version of a running cluster that the node belongs to.
- `shardman.config_uuid` defines the config version. Ignored if set via shardmanctl `config update` or shardmanctl `init`.
- `shardman.manual_execution` controls the consistent work with the global objects.
- `shardman.silk_never_restart` prohibits the multiplexer workers restart in case of an error.
- `shardman.pre_promote_mode` applies the consistent promotion mechanism (from standby to primary).

List of Parameters

`Repfactor`

Integer determining how many replicas shardmanctl should configure for each DBMS. This setting can only be changed for a Shardman cluster with a manual-topology mode.

`PlacementPolicy`

String determining the policy of placing DBMS instances. Currently, `cross` and `manual` placement policy is only supported. The former value `clover` is used as an alias for `cross` policy.

With `cross` placement policy, nodes are grouped in *clovers*, where each node is running the master DBMS server and replicas for all other nodes in the clover. The number of nodes in a clover is determined by Repfactor and equals `Repfactor` + 1.

`manual` placement policy allows you to manually add/remove the required number of replicas to/from the specified replication groups. In this case, `R#pfactor` is only used for recommendation purposes and does not impose restrictions.

`DataDir`

Allows you to specify a directory other than the default one (`/var/lib/pgpro/sdm-14/data`) for storing data. This parameter cannot be changed after the cluster has been initialized.

`PGsInitialPort`

Ports starting with this integer are assigned to PostgeSQL instances. This parameter cannot be changed after the cluster has been initialized.

`SilkInitialPort`

Ports starting with this integer are assigned to Silk (Shardman InterLinK) instances. This parameter cannot be changed after the cluster has been initialized.

`AuthMethod`

Authentication method used by the administrative user to connect to the DBMS. Can be any authentication method supported by PostgreSQL. `scram-sha-256` is currently recommended. `md5` is currently allowed but not recommended. This parameter cannot be changed after the cluster has been initialized. Located under a separate `Users` block for each array element.

Default: `trust`.

`Groups`

An array that can have two possible values, `su` for superuser or `repl` for replication.

`HTTP`

Defines settings for the secure HTTP/HTTPS connection, with *`Port`* being an API port, and *`PortMetrics`* being a port for the metrics. If these ports are the same, then API and metrics listen to the same port.

Default: `15432`.

`Name`

Name of the user. Created on cluster initialization. Defaults to the name of the effective user running `shardmanctl init`. This parameter cannot be changed after the cluster has been initialized. Located under a separate `Users` block for each array element.

`Password`

Password for the user. Can be changed using shardmanctl `config update credentials`. Located under a separate `Users` block for each array element.

`PgSuSSLCert`

Client certificate for the administrative DBMS user.

`PgSSLRootCert`

Location of the root certificate file for the DBMS user connection.

`PgSuSSLKey`

Client private key for the administrative DBMS user.

`PgSSLMode`

SSL mode for the DBMS user. Allowed values: `verify-ca` and `verify-full`.

`PgReplSSLCert`

Client certificate for the replication DBMS user.

`PgReplSSLKey`

Client private key for the replication DBMS user.

`ShardSpec`

Shard cluster specification. For more details, see `ShardSpec` Parameters. Can be changed using shardmanctl `config update`.

`FDWOptions`

This object contains FDW settings.

These settings can be changed using shardmanctl `config update` (with the exception of settings related to authorization, server connection, SSL and Kerberos, as well as the `service`, `target_session_attrs` options).

Foreign servers corresponding to Shardman replication groups will also get `extended_features` setting automatically enabled. Never set this parameter for postgres_fdw foreign servers which you define for your own purposes (for example, to load data into Shardman cluster).

ShardSpec Parameters

The `ShardSpec` specification can include all usual stolon options described in *Stolon Cluster Specification*. However, the following options should be carefully tuned for a Shardman cluster.

`pgHBA`

JSON array of `pg_hba.conf` strings. The default value allows user from the `su` group access from anywhere with `AuthMethod` authentication method. If the value of `defaultSUReplAccessMode` is `strict`, `pg_hba.conf` strings must explicitly allow users from the groups `su` or `repl` access from all Shardman cluster nodes.

`forceSuUserLocalPeerAuth`

When enabled, it sets a peer authentication via unix socket for the `postgres` user, if `strictUserHBA` is not set to `true`.

Default: `false`.

`synchronousReplication`

Determines whether replicas should use synchronous replication. Should be `true` in a Shardman cluster.

Default: `true`.

`maxSynchronousStandbys`

Maximum number of required synchronous standbys when synchronous replication is enabled. Should be >= Repfactor in a Shardman cluster. Default: `Repfactor`.

`strictUserHBA`

Prohibits adding automatically generated lines to `pg_hba.conf` file. Default: `false`.

`automaticPgRestart`

Determines whether a DBMS instance should be automatically restarted after a change of the pgParameters hash table that requires a restart. Should be enabled in a Shardman cluster.

Default: `true`.

`masterDemotionEnabled`

Enable master demotion in case the replica group master has lost connectivity with etcd. The master attempts to connect to each of its standby nodes to determine if any of them has become the master. If it discovers another master, it shuts down its own DBMS instance until the connectivity with etcd is restored. If the master fails to connect to one of its standby nodes for a long time, a DBMS instance shutdown occurs.

Default: `false`.

`masterDemotionTimeout`

The timeout during which the master attempts to connect to its standbys in cases where connectivity with etcd is lost. Works only if the `masterDemotionEnabled` parameter is set to true.

Default: `30s`.

`minSyncMonitorEnabled`

Enable the monitor for the `MinSynchronousStandbys` value for every replica group. If a node loses connection with the cluster (all `keepers` are unhealthy: a `keeper` does not update its state longer than `minSyncMonitorUnhealthyTimeout`), the monitor decreases the `MinSynchronousStandbys` value for every replica group related to the disconnected node to the maximum available value. This allows preventing the read-only condition caused by the fake replica. The maximum available value is always less than or equal to the value specified in the cluster configuration. If all `keepers` related to the disconnected node become healthy, the monitor changes `MinSynchronousStandbys` value for the replica group to the value specified in the cluster configuration.

Default: `false`.

`minSyncMonitorUnhealthyTimeout`

Time interval after which the node (and all `keepers` related to this node) will be considered in an unhealthy condition. Works only if the `minSyncMonitorEnabled` parameter is set to true.

Default: `30s`.

`syncPointMonitorEnabled`

Enable the monitor that creates a syncpoint every minute, ensuring the Shardman can restore to a consistent LSN. At each syncpoint, the cluster's state is consistent, meaning that all transactions are complete. If this parameter is set to true, PITR will be guaranteed to work. If set to `true`, it saves the syncpoint history in etcd with the key `shardman/{cluster_name}/data/cluster/syncpoints`.

Default: `false`.

`dbWaitRewindTimeout`

Before full resync of a replica, the cluster software first tries to do pg_rewind. Because the rewind operation is significantly faster than other approaches when the database is large and only a small fraction of blocks differs between the clusters. The `dbWaitRewindTimeout` parameter specifies the maximum working time for pg_rewind (examples of values: `5m`, `30s`, `1m30s`).

Default: `7m`.

`additionalReplicationSlots`

Array of names of physical replication slots that are created on the master. Each slot name must begin with the `stolon_` prefix.

`createSlotsOnFollowers`

If `true`, physical replication slots are also created on standby nodes.

`additionalSlotsLagLimit`

The limit of the volume by which replication slots defined by the `additionalReplicationSlots` configuration parameter can lag behind. If this value is exceeded, the slot is recreated. Specify the value as a number followed by a unit of measurement. Possible units: `B`, `kB`, `kiB`, `MB`, `MiB`, `GB`, `GiB`, `TB`, `TiB`, `PB`, `PiB`, `EB`, `EiB`, `ZB`, `ZiB`, `YB`, and `YiB`. For example: `100MB`.

`pgParameters`

Hash table that determines PostgreSQL settings, including Shardman-specific settings. Supports the following placeholders for `postgres` parameters: `{{dataDir}}` for data directory, `{{keeperDir}}` for keeper data directory under `dataDir`, `{{keeperName}}` for `keeper` name, `{{keeperID}}` for keeper ID, `{{cluster}}` for cluster name, `{{shard}}` for shard name, `{{host}}` for host with the working `postgres` instance.

Shardman-specific PostgreSQL Settings

The following settings in `pgParameters` are Shardman-specific:

`enable_csn_snapshot` (`boolean`)

Enables or disables Commit Sequence Number (CSN) based tracking of the transaction visibility for a snapshot.

PostgreSQL uses the clock timestamp as a CSN, so enabling CSN-based snapshots can be useful for implementing global snapshots and global transaction visibility.

When this parameter is enabled, PostgreSQL creates the `pg_csn` directory under `PGDATA` to keep track of CSN and XID mappings.

Default: `off`.

`enable_custom_cache_costs` (`boolean`)

Enables estimation logic for plan costs. It helps the planner choose generic plans more often considering the runtime pruning.

Default: `off`.

`enable_sql_func_custom_plans` (`boolean`)

If enabled, custom plans can be created to execute statements inside SQL functions. These plans depend on the parameter values.

Query plans can be cached within one query. First, the plan is built five times with different parameter values, then a generic plan is created regardless of the values. If custom and generic plan price is slightly different, then the generic plan is cached and is set to be used in the future. However, custom plans allow a more effective way of excluding queries to the sharded table partitions if the choice of these partitions depends on the query parameter.

Default: `off`.

`enable_merge_append` (`boolean`)

Enables the use of `MergeAppend` plans by the query planner.

Default: `on`.

`enable_async_merge_append` (`boolean`)

Enables or disables the query planner's use of async-aware merge append plan types. The default is `on`.

`csn_snapshot_defer_time` (`integer`)

Specifies the minimal age of records that are allowed to be vacuumed, in seconds.

All global transactions must start on all participant nodes within `csn_snapshot_defer_time` seconds after start, otherwise, they are aborted with a "`csn snapshot too old`" error.

Default: `15`.

`csn_commit_delay` (`integer`)

Specifies the maximum possible clock skew (in nanoseconds) in the cluster. Adds a delay before every commit in the system to ensure external consistency. If set to 0, external consistency is not guaranteed. Value suffixes `ns`, `us`, `ms` and `s` are allowed.

Default: `0`.

`csn_lsn_map_size` (`integer`)

Size of CSNLSNMap.

The commit record of each completed transaction in Shardman contains the assigned `CSN` for this transaction. This value, together with the `LSN`of this record, forms a pair of values `(CSN, LSN)`. Each of the cluster nodes stores a certain number of such pairs in RAM in a special structure - the `CSNLSNMap`. This map is used to get the syncpoint. See the "Syncpoints and Consistent Backup" section of the Internals chapter for more information.

Default: `1024`.

`csn_max_shift_error` (`boolean`)

When checked against the `csn_max_shift` value, raises an error if the `csn_max_shift` value is exceeded.

Default: `off`.

`csn_max_shift` (`integer`)

Maximum CSN shift in seconds for distributed queries and imported snapshots. If the shift exceeds the `csn_max_shift` value, an error or warning will occur. If the value is set to 0, no check is run.

Default: `15` (seconds).

`foreign_analyze_interval` (`integer`)

Specifies how often foreign statistics should be gathered during autovacuum, in seconds. If the value of `foreign_analyze_interval` is less than *`autovacuum_naptime`*, foreign statistics will be gathered each `autovacuum_naptime` seconds.

Default: 60.

foreign_join_fast_path (boolean)

Turns on a fast path for foreign join planning. When it is on, foreign join paths for SELECT queries are searched before all other possible paths and the search stops for a join as soon as a foreign join path is found.

Default: off.

optimize_correlated_subqueries (boolean)

Enables or disables the query planner's logic of transforming correlated subqueries into semi-joins.

Default: on.

port (integer)

A TCP port the server listens on. For a Shardman cluster, the port is assigned automatically by the system and is based on the PGsInitialPort parameter. If changed manually, the value will be overwritten by the configuration parameter that is automatically assigned.

enable_partition_pruning_extra (boolean)

Enables the extended partition pruning for the prepared queries with a known partitioning key. If turned on, the partition-wise join plans can be pruned.

Default: off.

crash_info (boolean)

When set to on, Shardman will write diagnostic information about a backend crash into a file.

Default: on.

crash_info_dump (text)

Specifies a comma-separated list of character strings that contain data sources to provide data for a crash dump. Possible values of the strings are as follows:

- queries — query texts
- memory_context — memory context
- system — information on the OS
- module — information on modules loaded to the postgres process
- cpuinfo — information on the processor
- virtual_memory — information on virtual memory regions

Default: system,module,queries,memory_context

crash_info_location (string)

Specifies the directory where information about a backend crash is to be stored. The value of stderr sends information about the crash to stderr. If this parameter is set to the empty string '', the $PGDATA/crash_info directory is used. If you wish to keep the files elsewhere, create the target directory in advance and grant appropriate privileges.

Default: ''.

shardman.context_log (bool)

Logs the remote contexts. If enabled, in case of an error, displays a field Remote CONTEXT. Note that if the standart log level is set to log_verbosity=terse, the shardman.context_log will be disabled automatically.

Default: on.

`postgres_fdw.enforce_foreign_join` (`boolean`)

Turns on alternative estimations for foreign join costs, which highly increases chances for join of several foreign tables referring to the same server to be pushed down. The cost of original join is estimated as `(1 - 1/(cost + 1))`, where `cost` is an originally estimated cost for this remote join.

Default: `off`.

`postgres_fdw.foreign_explain` (`enum`)

Defines how to include the `EXPLAIN` command output from the remote servers if the query plan contains `ForeignScan` nodes. The possible values are: `none` to exclude the `EXPLAIN` output from the remote servers, `full` to include the `EXPLAIN` output from the remote servers, `collapsed` to include the `EXPLAIN` output only for the first `ForeignScan` node under its `Append`/`MergeAppend`.

Default: `collapsed`.

`postgres_fdw.optimize_cursors` (`boolean`)

Sets postgres_fdw to try fetching the first portion of cursor data immediately after declaration and delay the cursor closing.

This postgres_fdw parameter forces it to avoid closing cursors after the end of scan. Cursors are closed at the end of transaction.

Default: `off`.

`postgres_fdw.subplan_pushdown` (`boolean`)

Enables or disables postgres_fdw logic of pushing down subqueries referencing only foreign server tables to this foreign server.

Default: `off`.

`postgres_fdw.use_twophase` (`enum`)

Sets postgres_fdw to use the two-phase commit (2PC) protocol for distributed transactions.

This postgres_fdw parameter forces it to use a two-phase commit if the transaction touches several nodes. When set to `auto`, a two-phase commit is only used in transactions with `enable_csn_snapshot=true` and isolation level equal to or higher than `REPEATABLE READ`.

Temporary tables cannot be used in 2PC transactions.

Default: `auto`.

`postgres_fdw.estimate_as_hashjoin` (`boolean`)

When enabled, the planner estimates a foreign join cost in a way similar to a cost of a hash-join whenever possible. This cost is compared to the default cost (which is similar to nested loops) and the smaller cost is selected for the path.

Default: `off`.

`postgres_fdw.additional_ordered_paths` (`boolean`)

When enabled, sorting on the remote server is considered if it allows performing `MergeJoin` or `MergeAppend` operations. This parameter is enabled by default in new installations but must be explicitly enabled in upgraded clusters.

`shardman.broadcast_ddl` (`boolean`)

Sets Shardman extension to broadcast DDL statements to all replication groups.

When this parameter is on, Shardman extension broadcasts supported DDL statements to all replication groups if it does make sense for those statements. You can enable/disable this behavior anytime. This parameter is not honored when set in configuration file.

Default: `off`.

`shardman.enable_limit_pushdown` (`boolean`)

Enable pushing down limit clauses through the underlying appends. When on, Shardman optimizer will try to push down a limit clause to the subpaths of the underlying `Append`/`MergeAppend` plan node if they reference postgres_fdw foreign tables. This optimization works only for `SELECT` plans when limit option is represented as a constant or a parameter. It is also restricted for `Append` paths, corresponding to a partitioned table. The optimization does not work for `SELECT` with locking clauses (`SELECT FOR UPDATE`/`NO KEY UPDATE`/`FOR SHARE`/`KEY SHARE`).

Default: `on`.

`shardman.num_parts` (`integer`)

Specifies the default number of sharded table partitions.

A sharded table has this default number of partitions unless *`num_parts`* is specified in CREATE TABLE.

To allow scaling, `shardman.num_parts` should be larger than the expected maximum number of nodes in a Shardman cluster.

Possible values are from `1` to `1000`.

Default: `20`.

`shardman.rgid` (`integer`)

Specifies the replication group ID of a Shardman node.

This parameter is set by Shardman utilities when the node is added to the cluster and should never be changed manually.

Default: `-1`.

`shardman.sync_schema` (`boolean`)

Sets Shardman to propagate all DDL statements that touch sharded and global relations to all replication groups.

When this parameter is on, Shardman broadcasts all supported utility statements touching sharded and global relations to all replication groups. It is not recommended to turn this off. This parameter is not honored when set in configuration file.

Default: `on`.

`shardman.sync_cluster_settings` (`boolean`)

Enables cluster-wide synchronization of configuration parameters set by user. The configuration parameters are propagated with each remote query.

Default: `on`.

`shardman.sync_cluster_settings_blacklist` (`boolean`)

Excludes the options not to be propagated to a remote cluster.

Default: local system configuration parameters that are never synchronized.

`shardman.query_engine_mode` (`enum`)

Switches between modes of query planning/execution. Possible values are `none` and `text`.

`none` means that query planning/execution will not use the Silk transport.

`text` means that the text query representation is transferred via Silk transport for remote execution.

Default: `none`.

`shardman.silk_use_ip` (`string`)

Silk transport uses IP address specified by this parameter for node identification. If the host name is specified, it is resolved and the first IP address corresponding to this name, is used.

Default: node hostname.

`shardman.silk_listen_ip` (`string`)

The Silk routing daemon listens for incoming connections on this IP address. If the host name is specified, it is resolved and the first IP address corresponding to this name, is used.

Default: node hostname.

`shardman.silk_use_port` (`integer`)

The Silk routing daemon listens for incoming connections on this port. This setting should be the same for all nodes in the Shardman cluster.

Default: `8888`.

`shardman.silk_tracepoints` (`bool`)

Enables tracing of queries passing through the Silk pipeline. The tracing results can be accessed by running the `EXPLAIN` command with `ANALYZE` set to `ON`.

Default: `off`.

`shardman.silk_num_workers` (`integer`)

Number of background workers allocated for distributed execution. This setting must be less than `max_worker_processes` (including auxilary postgres worker processes).

Default: `2`.

`shardman.silk_stream_work_mem` (`integer`)

Sets the base maximum amount of memory to be used by a Silk stream (as a buffer size) before writing to the temporary disk files. If this value is specified without units, the default is kilobytes.

Note that most queries can perform multiple fetch operations at the same time, usually one for each remote partition of a sharded table, if any. Each fetch operation is generaly allowed to use as much memory as this value specifies before it starts to write data into temporary files. Also, several running sessions can execute such operations concurrently. Therefore, the total memory used by Silk for buffers could be many times the value of `shardman.silk_stream_work_mem` and is correlated with shardman.num_parts. Thus, mind this fact when choosing the value.

Default: `16MB`.

`shardman.silkworm_fetch_size` (`integer`)

Number of rows in a chunk that the `silkworm` worker extracts and sends to the multiplexer as a result, per one reading iteration.

Default: 100.

`shardman.silk_unassigned_job_queue_size` (`integer`)

Size of queue for jobs that have not yet been assigned to the `silkworm` multiplexer workers, in case all the workers are busy.

Default: 1024.

`shardman.silk_max_message` (`integer`)

Maximum message size that can be transfered with Silk, in bytes. Note that this parameter does not limit the maximum size of the result returned by the query. It only affects messages sent to workers. Increasing this parameter value will result in a proportional memory increase consumed by Shardman. It is strongly recommended to use the default value unless there is an urgent need.

Default: 524288.

`shardman.silk_hello_timeout` (`integer`)

Handshake timeout between multiplexers of different nodes, in seconds.

Default: 3.

`shardman.silk_scheduler_mode` (`enum`)

Enables additional CPU scheduling settings for multiplexer processes (`silkroad` and `silkworm`).

When this parameter is `fifo`, Shardman assigns scheduling policy SCHED_FIFO for processes `silkroad` and each of `silkworm`. It assigns the static schediling priority (sched_priority) to values `shardman.silkroad_sched_priority` and `shardman.silkworm_sched_priority` respectively.

This setting improves silk transport performance while it operates under heavy CPU load.

Note that postgres binary need to have CAP_SYS_NICE capability to use this option. If no appropriate capability was assigned to the process, enabling this setting will have no effect. The capability must be assigned to postgres binary before starting postgres. Postgres (i.e. processes silkroad and silkworm) will apply scheduling options once during service start. You need restart postgres service if you want to change scheduling options.

Default: `none`.

To set capability you need execute following command once after postgres installed:

```
$ sudo setcap cap_sys_nice+ep /opt/pgpro/sdm-14/bin/postgres
```

Replace `/opt/pgpro/sdm-14/bin/postgres` to the correct path to your postgres binary if needed. Also note that your filesystem should support extended file attributes. You need set this for each node in the cluster to take the full effect.

In the Linux kernel, there is a mechanism called real-time throttling, which is designed to prevent tasks with real-time scheduling policies (like `SCHED_FIFO`) from monopolizing CPU resources. This ensures that other tasks with lower priorities, typically scheduled under the `SCHED_OTHER` policy, still get some amount of the CPU time. This mechanism is controlled by two parameters, exported into the `proc` filesystem or the `sysctl` interface:

- `/proc/sys/kernel/sched_rt_period_us` sets the duration of a scheduling period in microseconds. During this period, both real-time and non-real-time tasks share CPU time.
- `/proc/sys/kernel/sched_rt_runtime_us` specifies how much of the scheduling period is allocated to real-time tasks (with `SCHED_FIFO`). The remainder of the time is left for non-real-time tasks (`SCHED_OTHER`).

A typical and acceptable configuration for Shardman might set these parameters as follows:

```
# cat /proc/sys/kernel/sched_rt_period_us
1000000
# cat /proc/sys/kernel/sched_rt_runtime_us
950000
```

This configuration allows real-time tasks to use up to 950 milliseconds of each second, leaving 50 milliseconds for non-real-time tasks.

However, in some Linux distributions, the default values for these parameters might be set so low (or even to zero) that real-time tasks receive very little or no CPU time. This can make real-time scheduling ineffective or prevent the configuration from being applied. For example, attempting to manually set a task to the `SCHED_FIFO` priority using `chrt` might result in an error like:

```
$ sudo chrt -f -p 2 $(pgrep -f silkroad)
chrt: failed to set pid 1897706's policy: Operation not permitted
```

This error indicates that the kernel parameters are not configured correctly. In such cases, run the following:

```
echo 1000000 > /proc/sys/kernel/sched_rt_period_us
echo 950000 > /proc/sys/kernel/sched_rt_runtime_us
```

Or add the corresponding values into `/etc/sysctl.conf` and reload the settings using `sysctl -p`:

```
kernel.sched_rt_period_us = 1000000
kernel.sched_rt_runtime_us = 950000
```

`shardman.silkroad_sched_priority` (`integer`)

Value of static scheduling priority (sched_priority) for `silkroad` process. It only makes sense if `shardman.silk_scheduler_mode` equals to 'fifo'.

Default: `2`.

`shardman.silkworm_sched_priority` (`integer`)

Value of static scheduling priority (sched_priority) for `silkworm` processes (the same value for each of them). It only makes sense if `shardman.silk_scheduler_mode` equals to 'fifo'.

Default: `1`.

`shardman.silk_set_affinity` (`bool`)

Enables pinning of multiplexer processes (`silkroad` and `silkworm`) to CPU cores to eliminate negative effects of thread's cross-cpu migration.

When this parameter is `true`, `silkroad` process will be pinned to the first available CPU core and `silkworm` processes (all of them) will pinned to all available CPU cores except the first one.

This setting improves silk transport performance while it operates under heavy CPU load.

Note that postgres binary need to have CAP_SYS_NICE capability to use this option. If no appropriate capability was assigned to the process, enabling this setting will have no effect. The capability must be assigned to postgres binary before starting postgres. Postgres (i.e. processes silkroad and silkworm) will apply affinity options once during service start. You need restart postgres service if you want to change affinity options.

To set capability you need execute following command once after postgres installed:

```
$ sudo setcap cap_sys_nice+ep /opt/pgpro/sdm-14/bin/postgres
```

Replace `/opt/pgpro/sdm-14/bin/postgres` to the correct path to your postgres binary if needed. Also note that your filesystem should support extended file attributes. You need set this for each node in the cluster to take the full effect.

Default: `false`.

`shardman.silk_flow_control` (`boolean`)

Controls the mode of handling read events. It has three possible values: `none`, `round_robin`, and `shortest_job_first`.

The `none` mode means no control nor additional overhead. Yet in this case, the channel may become occupied by just one distributed query.

The `round_robin` mode means the events created earlier are the first ones to be processed, for each event loop. If enabled, all the backends are grouped, and the client backends are prioritized over the other.

The `shortest_job_first` mode means full control over the traffic. If enabled, all the backends are grouped, and the client backends are prioritized over the others, along with the workers with the least session traffic.

Default: `round_robin`.

`shardman.silk_track_time` (`boolean`)

Enables or disables the metrics with prefix `transferred_` and time-based metrics (with prefixes `read_efd_`, `write_efd_`, and `sort_time_`). If disabled, these metrics have `0` values.

Default: `off`.

`shardman.silk_tracelog` (`bool`)

Enables or disables Silk logging.

Default: `off`.

`shardman.silk_tracelog_category` (`string`)

Defines the Silk message categories to be traced.

Default:`streams, routing, events`.

`shardman.database` (`string`)

Name of the database that all Silk workers connect to.

Default: `postgres`.

`shardman.monitor_interval` (`integer`)

`shardman.monitor_interval` is deprecated and acts as noop.

Use `shardman.monitor_dxact_interval` instead.

`shardman.monitor_dxact_interval` (`integer`)

Interval between checks for outdated prepared transactions.

The Shardman monitor background process wakes up every `shardman.monitor_dxact_interval` seconds and attempts to check and resolve any prepared transactions that did not complete and became outdated for some reason. To resolve these transactions, the Shardman monitor process determines the coordinator of the transaction and requests the transaction status from the coordinator. Based on the status of the transaction, Shardman monitor will either roll back or commit the transaction.

To disable the prepared transaction resolution logic, set `shardman.monitor_dxact_interval` to `0`.

Default: `5` (seconds).

`shardman.monitor_trim_csnxid_map_interval` (`integer`)

Each cluster node freezes its own `xmin` value for `csn_snapshot_defer_time` seconds to support global transactions. Large `csn_snapshot_defer_time` values can negatively impact the performance. Shardman monitor has a routine that every `shardman.monitor_trim_csnxid_map_interval` seconds updates `xmin` on all nodes to the minimum possible value (taking into account active transactions).

The background routine will run on only one node in the Shardman cluster. Note that this will give an additional load on this node.

To disable such updates, set `shardman.monitor_trim_csnxid_map_interval` to `0`.

Default: `5` (seconds).

`shardman.monitor_dxact_timeout` (`integer`)

Maximum allowed age of prepared transactions before a resolution attempt.

During the resolution of a prepared transaction, Shardman monitor determines whether the transaction is outdated or not. A transaction becomes outdated if it was prepared more than `shardman.monitor_dxact_timeout` seconds ago.

Default: `5` (seconds).

`shardman.trim_csnxid_map_naptime` (`integer`)

Specifies the minimum delay between `xmin` updates on all nodes. See shardman.monitor_trim_csnxid_map_interval for more information.

Possible values are from `1` to `600`.

Default: `5`.

`shardman.monitor_deadlock_interval` (`integer`)

Interval between checks for distributed deadlock conditions.

The Shardman monitor background process wakes up every `shardman.monitor_deadlock_interval` seconds and searches for distributed deadlocks in the cluster. It gathers information about mutual locks from all nodes and looks for circular dependencies between transactions. If it detects a deadlock, it resolves it by canceling one of the backend processes involved in the lock.

To disable the distributed deadlock resolution logic, set `shardman.monitor_deadlock_interval` to `0`.

Default: `2` (seconds).

`postgres_fdw.remote_plan_cache` (`boolean`) — EXPERIMENTAL

Enables remote plan caching for FDW queries produced by locally cached plans.

Default: `off`.

`shardman.plan_cache_mem` (`integer`) — EXPERIMENTAL

Specifies how much memory per worker can be used for remote plan caches.

Default: `0` (caches are disabled).

`shardman.gt_batch_size` (`integer`) —

Specifies the buffer size for `INSERT` and `DELETE` commands executed on a global table.

Default: `64K`.

`postgres_fdw.enable_always_shippable` (`boolean`) — EXPERIMENTAL

Always allow some expressions to be evaluated on a remote. Right now this is limited to just a few functions. All nodes should have identical `timezone` settings for this feature to work correctly.

> **Warning**
>
> Do not turn this on unless all `postgres_fdw` remotes are Shardman-managed.

Default: `false`.

`track_fdw_wait_timing` (`boolean`)

The statistics for the network latency (wait time) for inter-cluster operations, in milliseconds. It can be accessed by running the `EXPLAIN` command with the `network` parameter enabled, and via the pgpro_stats view pgpro_stats_sdm_statements.

Default: `on`.

`track_xact_time` (`boolean`)

Enables or disables statistics collection for time spent on a transaction.

Default: `off`.

`enable_non_equivalence_filters` (`boolean`)

Enables the optimizer to generate additional non-equivalence conditions using equivalence classes.

Default: `off`.

`optimize_row_in_expr` (`boolean`)

Enables the optimizer to generate additional conditions from the `IN ()` expression.

Default: `off`.

Examples

Spec File for a Cluster with Enabled scram-sha-256 Authentication

> **Note**
>
> The initial configuration file should be generated with the following command:
>
> ```
> shardmanctl config generate > sdmspec.json
> ```
>
> The example below is for educational purposes only and may lack the latest updates.

This is the contents of an example `sdmspec.json` configuration file:

```
{
   "ConfigVersion": "1",
   "Repfactor": 1,
   "PlacementPolicy": "manual",
   "PGsInitialPort": 5432,
   "SilkInitialPort": 8000,
   "HTTP": {
     "Port": 15432,
     "PortMetrics": 15432
   },
   "Users": [
  {
   "Name": "postgres",
   "Groups": [ "su"],
   "AuthMethod": "scram-sha-256",
   "Password": "changeMe"
  },
  {
   "Name": "repluser",
   "Groups": ["repl"],
   "AuthMethod": "scram-sha-256",
   "Password": "changeMe"
  }
 ],
   "ShardSpec": {
    "synchronousReplication": true,
    "usePgrewind": true,
    "pgParameters": {
     "csn_snapshot_defer_time": "300",
     "enable_csn_snapshot": "on",
     "enable_csn_wal": "true",
     "shardman.query_engine_mode": "text",
     "shardman.silk_num_workers": "8",
     "max_connections": "600",
     "max_files_per_process": "65535",
     "max_logical_replication_workers": "14",
```

```
      "max_prepared_transactions": "200",
      "max_worker_processes": "24",
      "shared_preload_libraries": "postgres_fdw, shardman"
     },
     "pgHBA": [
      "host replication postgres 0.0.0.0/0 scram-sha-256",
      "host replication postgres ::0/0 scram-sha-256"
     ],
     "automaticPgRestart": true,
     "masterDemotionEnabled": false
    },
    "FDWOptions": {
     "async_capable": "on",
     "batch_size": "100",
     "connect_timeout": "5",
     "fdw_tuple_cost": "0.2",
     "fetch_size": "50000",
     "tcp_user_timeout": "10000"
    }
}
```

From that configuration file, you can see that a Shardman cluster initialized with this spec file has `Repfactor` equal to 1 (one replica for each master). The configuration file also shows that two special users are created in this cluster — superuser `postgres` and replication user `repluser` with `ChangeMe` passwords. They can be authenticated using the `md5` or `scram-sha-256` authorization method. One postgres_fdw fetch operation will get up to 50000 rows from the remote server. The cost of fetching one row is set to a reasonably high value to make PostgreSQL planner consider conditions pushdown-attractive. `pg_hba.conf` settings allow `postgres` user access from anywhere using a replication protocol; all other users can access any database from anywhere. Since `defaultSUReplAccessMode` is not set to `strict`, utilities will automatically add entries that allow `PgSuUsername` user's (`postgres`) access to any database from anywhere and `PgReplUsername` user's (`repluser`) replication access from anywhere.

Several important Shardman-specific parameters are set in the `pgParameters` hash table. These are:

`wal_level`

Should be set to `logical` for Shardman to work correctly.

`shared_preload_libraries`

Should include postgres_fdw and shardman extensions in the specified order.

`max_logical_replication_workers`

Should be rather high since the rebalance process uses up to `max(max_replication_slots, max_logical_replication_workers, max_worker_processes, max_wal_senders)/3` concurrent threads.

`max_prepared_transactions`

Should be rather high since Shardman utilities use the 2PC protocol. If `postgres_fdw.use_twophase` is true, postgres_fdw also uses 2PC.

`enable_csn_snapshot`

Should be enabled to achieve a true `REPEATABLE READ` isolation level in a distributed system.

`csn_snapshot_defer_time`

All global transactions must start on all participant nodes within `csn_snapshot_defer_time` seconds after start, otherwise they will be aborted.

`enable_partitionwise_aggregate`
`enable_partitionwise_join`

Set to `on` to enable optimizations for partitioned tables.

Spec File for a Cluster with Enabled Certificate Authentication

This is the contents of an example `sdmspec.json` configuration file:

```
 {
   "ConfigVersion": "1",
   "HTTP": {
     "Port": 15432,
     "PortMetrics": 15432
     "SSLKey": "/pgpro/ssl/server.key",
     "SSLCert": "/pgpro/ssl/server.crt"
   },
   "Users": [
     {
       "Name": "postgres",
       "SSLKey": "/var/lib/postgresql/.ssh/client.key",
       "SSLCert": "/var/lib/postgresql/.ssh/client.crt",
       "Groups": ["su"],
       "AuthMethod":"scram-sha-256"
     },
     {
      "Name": "repluser",
       "SSLKey": "/var/lib/postgresql/.ssh/repluser.key",
       "SSLCert": "/var/lib/postgresql/.ssh/repluser.crt",
       "Groups": ["repl"],
       "AuthMethod":"scram-sha-256"
     }
   ],
    "ShardSpec": {
     "synchronousReplication": true,
     "usePgrewind": true,
     "pgParameters": {
      "ssl": "on",
      "ssl_cert_file": "/var/lib/postgresql/.ssh/server.crt",
      "ssl_key_file": "/var/lib/postgresql/.ssh/server.key",
      "ssl_ca_file": "/var/lib/postgresql/.ssh/ca.crt",
      "csn_snapshot_defer_time": "300",
      "enable_csn_snapshot": "on",
      "enable_csn_wal": "true",
      "log_line_prefix": "%m [%r][%p]",
      "log_min_messages": "INFO",
      "log_statement": "none",
      "maintenance_work_mem": "1GB",
      "max_connections": "600",
      "max_files_per_process": "65535",
      "max_logical_replication_workers": "9",
      "max_prepared_transactions": "200",
      "max_wal_size": "4GB",
      "max_worker_processes": "16",
      "min_wal_size": "512MB",
      "postgres_fdw.subplan_pushdown": "off",
      "shardman.query_engine_mode": "text",
      "shardman.silk_num_workers": "8",
      "shared_buffers": "4GB",
      "shared_preload_libraries": "postgres_fdw, shardman"
     },
     "strictUserHBA": true,
     "pgHBA": [
      "hostssl all postgres 0.0.0.0/0 cert clientcert=verify-full",
```

```
      "hostssl all repluser 0.0.0.0/0 cert clientcert=verify-full",
      "hostssl replication postgres 0.0.0.0/0 cert clientcert=verify-full",
      "hostssl replication postgres ::0/0 cert clientcert=verify-full",
      "hostssl replication repluser 0.0.0.0/0 cert clientcert=verify-full",
      "hostssl replication repluser ::0/0 cert clientcert=verify-full",
      "hostnossl all all 0.0.0.0/0 reject",
      "local postgres postgres scram-sha-256",
      "local replication repluser scram-sha-256"
     ],
     "automaticPgRestart": true,
     "masterDemotionEnabled": false
    },
    "FDWOptions": {
     "async_capable": "on",
     "batch_size": "100",
     "connect_timeout": "5",
     "fdw_tuple_cost": "0.2",
     "fetch_size": "50000",
     "tcp_user_timeout": "10000"
    }
 }
```

pgpro_stats parameters

`pgpro_stats.track_sharded` (`boolean`)

Specifies whether the sharded statements are tracked and aggregated by pgpro_stats.

Default: `on`.

`pgpro_stats.pgss_max_nodes_tracked` (`integer`)

Sets the maximum number of nodes that are tracked by pgpro_stats for query fragments.

It actually sets the maximum amount of the status entries that pgpro_stats can store for the `pgpro_stats_sdm_stats_updated` function. It does not affect the statistics tracking itself.

Default: `2048`.

`pgpro_stats.transport_compression` (`string`)

Sets algorithm for transport compression during statistics transferring between nodes.

Transport compression is used to compress statistical entries passed from the shard nodes to the coordinator. The possible values are `pglz`, `zlib`, `lz4`, `zstd` or `off`.

Default: `pglz`.

`pgpro_stats.enable_wait_counters` (`boolean`)

Enables or disables statistics collection for wait counters by enabling or disabling functions that calculate metrics of wait events.

Default: `off`.

`pgpro_stats.enable_inval_msgs_counters` (`boolean`)

Enables or disables statistics collection the invalidation messages by enabling or disabling functions that calculate metrics of invalidation messages.

If disabled, the `pgpro_stats_inval_status` view is empty.

Default: `off`.

`pgpro_stats.enable_rusage_counters` (`boolean`)

Enables or disables statistics collection for resource usage counters by enabling or disabling functions that calculate metrics of OS resource usage.

Default: `off`.

`pgpro_stats.track_shardman_connections` (`enum`)

Enables or disables Shardman-specific statements processing. This parameter has three possible values. `none` with no processing, `normalized` (default) with generalized statements being processed, and `all` with all statements being processed.

See Also

shardmanctl

shardmand

shardmand — Shardman configuration daemon

Synopsis

`shardmand` [*common_options*] [`--system-bus`] [`--user` *user_name*]

Here *common_options* are:

[`--cluster-name` *cluster_name*] [`--log-level` `error`|`warn`|`info`|`debug`] [`--retries` *retries_number*] [`--session-timeout` *seconds*] [`--store-endpoints` *store_endpoints*] [`--store-ca-file` *store_ca_file*] [`--store-cert-file` *store_cert_file*] [`--store-key` *client_private_key*] [`--store-timeout` *duration*] [`--version`] [`-h`|`--help`] [`--log-format`]

Description

shardmand is a Shardman configuration daemon. It runs on each node in a Shardman cluster, subscribes for changes of `shardman/cluster0/data/ladle` and `shardman/cluster0/data/cluster` keys in the etcd store (`cluster0` is the default cluster name used by Shardman utils) and manages Shardman processes on the node where it is running according to the configuration described in these JSON documents.

shardmand manages integrated `keepers` and `sentinels`. On startup and when one of the monitored etcd keys changes, shardmand reconfigures them as follows:

- It calculates the expected node configuration, i. e., the list of `keepers` and `sentinels` expected to run and their configurations, from the `shardman/cluster0/data/ladle` and `shardman/cluster0/data/cluster` values.
- It receives the list of running `keepers` and `sentinels` with their configurations from the internal process manager.
- It stops processes that are not expected to run. This can be a process that belongs to a cluster with the same name, but a different UUID, or a process whose description is no longer present in the expected node configuration. For `keeper` processes, shardmand purges their data directory.
- If a process should be running, but its settings are different from the expected ones, shardmand updates the configuration and restarts the process. If a process should be running, but it is not running, shardmand starts it.

Also, a separate thread of shardmand periodically updates the `shardman/cluster0/data/shardmand/NODENAME` etcd key with the `ClusterUUID` of the last cluster to which the configuration was applied. So, before the shardmanctl `nodes add` command tries to initialize new stolon clusters for a clover, the command can ensure that no alive stolon threads from a previous cluster configuration are left on all nodes in the clover.

Additionaly, shardmand starts two http servers in separate threads. If servers ports match, a single server running both roles is started. The first server provides following metrics: `shardmand_etcd_unavailable_time_seconds`, `shardmand_healthy_keepers`, `shardmand_sentinels`, `shardmand_uptime`, `shardmand_etcd_errors_total`, `shardmand_reconfigurations_number_total`, `shardmand_demotions_number_total`. Also server provides a `/healthz` endpoint for shardmand health-check. The second server provides the following endponts:

- `/shardmand/v1/replica` — returns 200 status code if a secondary instance is running on node , 500 status code if a master instance is running on node, `/shardmand/v1/master` — returns 200 status code if a master instance is running on node , 500 status code if a secondary instance is running on node. If node both master and secondary instances are running on node `/shardmand/v1/replica` and `shardmand/v1/master` endpoints return 404 status code.
- `/shardmand/v1/status` — getting information about shardmand status.

All Shardman services are managed by `shardmand@cluster0.service`, so when it is started, stopped or restarted, it also starts, stops or restarts all other Shardman processes (including DBMS instances).

Command-line Reference

This section describes `shardmand`-specific command-line options. For Shardman common options used by the commands, see the section called “Common Options”.

`--log-format`

Specifies the log output format, `json` or `text`. The default is `text`.

`--system-bus`

Not used. Left for compatibility. Ignored.

`--user` *`user_name`*

Not used. Left for compatibility. Ignored.

Common Options

shardmand common options are optional parameters that are not specific to the utility. They specify etcd connection settings, cluster name and a few more settings. By default shardmand tries to connect to the etcd store `127.0.0.1:2379` and use the `cluster0` cluster name. The default log level is `info` .

`-h, --help`

Show brief usage information.

`--cluster-name` *`cluster_name`*

Specifies the name for a cluster to operate on. The default is `cluster0`.

`--log-level` *`level`*

Specifies the log verbosity. Possible values of *`level`* are (from minimum to maximum): `error`, `warn`, `info` and `debug`. The default is `info`.

`--retries` *`number`*

Specifies how many times shardmanctl retries a failing etcd request. If an etcd request fails, most likely, due to a connectivity issue, shardmanctl retries it the specified number of times before reporting an error. The default is 5.

`--session-timeout` *`seconds`*

Specifies the session timeout for shardmanctl locks. If there is no connectivity between shardmanctl and the etcd store for the specified number of seconds, the lock is released. The default is 30.

`--store-endpoints` *`string`*

Specifies the etcd address in the format: `http[s]://`*`address`*`[:`*`port`*`](,http[s]://`*`address`*`[:`*`port`*`])*`. The default is `http://127.0.0.1:2379`.

`--store-ca-file` *`string`*

Verify the certificate of the HTTPS-enabled etcd store server using this CA bundle.

`--store-cert-file` *`string`*

Specifies the certificate file for client identification by the etcd store.

`--store-key` *`string`*

Specifies the private key file for client identification by the etcd store.

`--store-timeout` *`duration`*

Specifies the timeout for a etcd request. The default is 5 seconds.

`--monitor-port` *`number`*

Specifies the port for the shardmand http server for metrics and probes. The default is 15432.

`--api-port` *`number`*

Specifies the port for the shardmand http api server. The default is 15432.

`--version`

Show shardman-utils version information.

Environment

A shardmand service reads the environment from `/etc/shardman/shardmand-cluster0.env`. The following environment variables affect the behavior of shardmand.

`SDM_CLUSTER_NAME`

An alternative to setting the `--cluster-name` option

`SDM_LOG_LEVEL`

An alternative to setting the `--log-level` option

`SDM_RETRIES`

An alternative to setting the `--retries` option

`SDM_SYSTEM_BUS`

An alternative to setting the `--system-bus` option

`SDM_STORE_ENDPOINTS`

An alternative to setting the `--store-endpoints` option

`SDM_STORE_CA_FILE`

An alternative to setting the `--store-ca-file` option

`SDM_STORE_CERT_FILE`

An alternative to setting the `--store-cert-file` option

`SDM_STORE_KEY`

An alternative to setting the `--store-key` option

`SDM_STORE_TIMEOUT`

An alternative to setting the `--store-timeout` option

`SDM_SESSION_TIMEOUT`

An alternative to setting the `--session-timeout` option

`SDM_USER`

An alternative to setting the `--user` option

Examples

Configuring a shardmand Service

shardmand settings are usually specified in the `/etc/shardman/shardmand-cluster0.env` file. If you want shardmand to connect to an etcd cluster at hosts `n1`-`n3` using port 2379 and all Shardman services to use the `debug` log level, you can use the following `env` file:

```
SDM_STORE_ENDPOINTS=http://n1:2379,http://n2:2379,http://n3:2379
SDM_LOG_LEVEL=debug
```

Note that you need to restart `shardmand@cluster0` service to apply new settings from the `env` file.

Showing shardmand Logs

To look at shardmand logs, you can use a `journalctl` command:

```
$ journalctl -u shardmand@cluster0.service
```

Restarting Shardman Services

You can restart all Shardman services on a node using a `systemctl` command:

```
$ systemctl restart shardmand@cluster0.service
```

See Also

shardmanctl , sdmspec.json

Chapter 7. Shardman Internals

The Shardman software comprises these main components: PostgreSQL core with additional features, shardman extension, management services and utilities. This section considers Shardman cluster as a group of PostgreSQL instances or shards. Each shard may also have one or more replicas and to emphasize this the term replication group is used. The support for highly available configurations is currently done on the level of tools and services and will be covered in the Management section.

7.1. Table Types

In a distributed database managed by Shardman the following special table types are used: sharded tables and global tables.

7.1.1. Sharded Tables

Sharded tables are just usual PostgreSQL partitioned tables where a few partitions, making up a *shard*, are regular local tables and the other partitions are foreign tables available from remote servers via *postgres_fdw*. Sharded tables are registered in the `shardman.sharded_tables` dictionary. Use the `CREATE TABLE` statement with the `distributed_by` parameter to create a sharded table. Several sharded tables can be created as colocated. This means that they have the same number of partitions and that their partitions corresponding to the same sharding key should reside together. During a rebalance, Shardman management utilities ensure that corresponding partitions of colocated tables are moved to the same node. (Such a rebalance happens, for example, when a new node is added to the cluster). Colocation is necessary to ensure that joins of several tables are propagated to the node where the actual data resides. To define one sharded table colocated with another one, first, create one table and then use the `colocate_with` parameter of the `CREATE TABLE` statement while creating the second table. Chains of colocated tables are not supported, all related tables should be marked as colocated to one of the tables instead. Note that `colocate_with` property is symmetric and transitive.

7.1.1.1. Partitions

A sharded table consists of several partitions. Some of them are regular tables, and others are foreign tables. By default, the number of partitions is determined by the `shardman.num_parts` parameter, but it can be overwritten by the `num_parts` `CREATE TABLE` parameter. Most of DDL operations are restricted on partitions of a sharded table. You should modify the parent table instead.

The number of partitions in a sharded table is defined when it is created and cannot be changed afterwards. When new nodes are added to the cluster, some partitions are moved from existing nodes to the new ones to balance the load. So, to allow scaling of clusters, the initial number of partitions should be high enough, but not too high since an extremely large number of partitions significantly slows down query planning. For example, if you expect the number of nodes in your cluster to grow by 4 times at a maximum, create sharded tables with the number of partitions equal to `4 * N`, where `N` is the number of nodes. A cluster becomes unable to scale when the number of cluster nodes reaches the number of partitions in the sharded table with the minimal number of them.

7.1.1.2. Subpartitions

Partitions of a sharded table can be partitioned by range. In this case, each partition of a sharded table is a partitioned table consisting only of regular or only of foreign subpartitions. All subpartitions of a partition are located on the same node. Use the `partition_by` `CREATE TABLE` parameter to specify a column that should be used as a subpartition key column and the `partition_bounds` parameter to set bounds of the second-level table partitions. New subpartitions can be added or removed from a table as necessary. So you can omit the `partition_bounds` parameter during table creation and create partitions later using the `shardman.create_subpart()` function. Other subpartition management functions allow you to drop, detach or attach subpartitions of a sharded table. Subpartition management is cluster-wide.

7.1.2. Global Tables

Global tables are available to all nodes of a cluster. Now a global table is a set of regular tables synchronized by triggers. The main use case for a global table is to store a relatively rarely updated set of data that is used by all cluster nodes. When a sharded table is joined to a global table, joins between sharded table partitions and the global table can be performed on nodes where individual partitions reside. The implementation of trigger-based replication requires a non-deferrable primary key on a global table to be defined. Currently when a global table is modified, an after-statement trigger fires and propagates changes to other nodes of the cluster via foreign tables. When new nodes are added to a cluster, global table data is transferred to the new nodes via logical replication. When some nodes are removed from a cluster, global tables get locked for writes for a brief time. Use the `global` `CREATE TABLE` parameter to create a global table. Global tables are registered in the `shardman.global_tables` dictionary. Partitioned global tables are not supported.

7.1.3. Distributed DDL

Shardman extension allows creating several kinds of global objects. These are sharded and global tables, roles and tablespaces. The list of operations allowed on global objects is limited particularly to protect consistency of a global schema. For the same reason, most operations on global objects are cluster-wide. The list of cluster-wide operations includes:

- `CREATE` for sharded and global tables, global roles and tablespaces or indexes on sharded or global tables.
- `DROP` for sharded and global tables, global roles and tablespaces or indexes on sharded or global tables.
- `ALTER TABLE` for sharded and global tables.
- `ALTER TABLESPACE` for global tablespaces.
- `ALTER ROLE` for global roles.
- `RENAME` for sharded and global tables or indexes on them.
- `SET CONSTRAINTS ALL` inside a transaction block.

These configuration settings control execution of the distributed DDL: shardman.broadcast_ddl and shardman.sync_schema. The first one can be used for a cluster-wide broadcast of all regular DDL operations (for example, creating schemas or functions). The second one controls broadcasting of statements related to global objects and should never be turned off without consulting the Postgres Pro Shardman support team.

7.2. Query Processing

Shardman uses the standard PostgreSQL *query execution pipeline*. Other nodes in the cluster are accessed via the modified postgres_fdw extension.

Shardman query planner takes the query abstract syntax tree (AST) and creates a query plan, which is used by the executor. While evaluating query execution methods, the planner operates with so-called paths, which specify how relations should be accessed. While processing a query join tree, the planner looks at different combinations of how relations can be joined. Each time it examines a join of two relations, one of which can be a join relation itself. After choosing the order and strategies for joining relations the planner considers the group by, order by and limit operations. When the cheapest path is selected, it is transformed to a query plan. A plan consists of a tree of nodes, each of which has methods to get one next result row (or NULL if there are no more results).

7.2.1. Push-down Technique

7.2.1.1. Joins

The efficiency of query execution in a distributed DBMS is determined by how many operations can be executed on nodes that hold the actual data. For Shardman, a lot of effort is devoted to pushing down join operations. When the planner finds a relation that is accessible via a foreign data wrapper (FDW), it creates `ForeignPath` to access it. Later, when it examines a join of two relations and both of them are available via `ForeignPath` from the same foreign server, it can consider pushing down this join to the server and generating a so-called `ForeignJoinPath`. The planner can fail to do it if the join type is not supported, if filters attached to the relation should be applied locally, or if the relation scan result contains fields that cannot be evaluated on the remote server. An example of a currently unsupported join type is anti-join. Local filters attached to the relation should be applied locally when remote execution can lead to a different result or if the postgres_fdw module cannot create SQL expressions to apply some of the filters. An example of fields that cannot be evaluated on a remote server are attributes of semi-join inner relation that are not accessible via an outer relation. If the foreign_join_fast_path configuration parameter is set to on (which is the default value), the Shardman planner stops searching for other join strategies of two relations once it finds a foreign join possible for them. When the postgres_fdw.enforce_foreign_join configuration parameter is set to on (which is also the default), the cost of a foreign join is estimated so as to be always less than the cost of a local join.

When several sharded tables are joined on a sharding key, a partitionwise join can be possible. This means that instead of joining original tables, we can join their matching partitions. Partitionwise join currently applies only when the join conditions include all the partition keys, which must be of the same data type and have exactly matching sets of child partitions. Partitionwise join is crucial to the efficient query execution as it allows pushing down joins of table partitions. Evidently, to push down a join of several partitions, these partitions should reside on the same node. This is usually the case when sharded tables are created with the same `num_parts` parameter. However, for a rebalance process to move the corresponding partitions to the same nodes, sharded tables should be

marked as colocated when created (see Section 7.1.1). Partitionwise join is enabled with the *enable_partitionwise_join* configuration parameter, which is turned on by default in Shardman.

When a sharded table is joined to a plain global table, asymmetric partitionwise join is possible. This means that instead of joining original tables, we can join each partition of the sharded table with the global table. This makes it possible to push down a join of sharded table partitions - with a global table to the foreign server.

7.2.1.2. Aggregations

After planning joins, the planner considers paths for post-join operations, such as aggregations, limiting, sorting and grouping. Not all such operations reach FDW pushdown logic. For example, currently partitioning efficiently prevents the `LIMIT` clause from being pushed down. There are two efficient strategies for executing aggregates on remote nodes. The first one is a partitionwise aggregation — when a `GROUP BY` clause includes a partitioning key, the aggregate can be pushed down together with the `GROUP BY` clause (this behavior is controlled by the *enable_partitionwise_aggregate* configuration parameter, which is turned on by default in Shardman). Alternatively, the planner can decide to execute partial aggregation on each partition of a sharded table and then combine the results. In Shardman, such a partial aggregate can be pushed down if the partial aggregate efficiently matches the main aggregate. For example, partial `sum()` aggregate can always be pushed down, but `avg()` cannot. Also the planner refuses pushing down partial aggregates if they contain additional clauses, such as `ORDER BY` or `DISTINCT`, or if the statement has the `HAVING` clause.

7.2.1.3. Subqueries

Generally, subqueries cannot be pushed down to other cluster nodes. However, Shardman uses two approaches to alleviate this limitation.

The first is subquery unnesting. In PostgreSQL, non-correlated subqueries can be transformed into semi-joins. In the following example, `ANY` subquery on non-partitioned tables is transformed to `Hash Semi Join`:

```
EXPLAIN (COSTS OFF) SELECT * FROM pgbench_branches WHERE bid = ANY (SELECT bid FROM
 pgbench_tellers);
                         QUERY PLAN
------------------------------------------------------------
 Hash Semi Join
   Hash Cond: (pgbench_branches.bid = pgbench_tellers.bid)
   ->  Seq Scan on pgbench_branches
   ->  Hash
         ->  Seq Scan on pgbench_tellers
```

When optimize_correlated_subqueries is on (which is the default), Shardman planner also tries to convert correlated subqueries (i.e., subqueries that reference upper-level relations) into semi-joins. This optimization works for `IN` and `=` operators. The transformation has some restrictions. For example, it is not considered if a subquery contains aggregates or references upper-level relations from outside of a `WHERE` clause. This optimization allows transforming more complex subqueries into semi-joins, like in the following example:

```
EXPLAIN (COSTS OFF) SELECT * FROM pgbench_branches WHERE bid = ANY (SELECT bid FROM
 pgbench_tellers WHERE tbalance = bbalance);
                                                        QUERY PLAN

-----------------------------------------------------------------------------------------------
 Hash Semi Join
   Hash Cond: ((pgbench_branches.bid = pgbench_tellers.bid) AND
 (pgbench_branches.bbalance = pgbench_tellers.tbalance))
   ->  Seq Scan on pgbench_branches
   ->  Hash
         ->  Seq Scan on pgbench_tellers
(5 rows)
```

After applying subquery unnesting, semi-join can be pushed down for execution to a remote node.

The second approach is to push down the entire subquery. This is possible when the optimizer has already figured out that the subquery references only partitions from the same foreign server as the upper-level query and corresponding foreign scans do not

have local conditions. The optimization is controlled by postgres_fdw.subplan_pushdown (which is off by default). When a decision to push down a subquery is made by postgres_fdw, it has to deparse this subquery. A subquery that contains plan nodes for which deparsing is not implemented will not be pushed down. An example of a subquery pushdown looks as follows:

```
EXPLAIN (VERBOSE ON, COSTS OFF)
SELECT * FROM pgbench_accounts a WHERE a.bid=90 AND abalance =
    (SELECT min(tbalance) FROM pgbench_tellers t WHERE t.bid=90 and a.bid=t.bid);
                                    QUERY PLAN

-------------------------------------------------------------------------------------
 Foreign Scan on public.pgbench_accounts_5_fdw a
   Output: a.aid, a.bid, a.abalance, a.filler
   Remote SQL: SELECT aid, bid, abalance, filler FROM public.pgbench_accounts_5
 r2 WHERE ((r2.bid = 90)) AND ((r2.abalance = ((SELECT min(sp0_2.tbalance) FROM
 public.pgbench_tellers_5 sp0_2 WHERE ((sp0_2.bid = 90)) AND ((r2.bid = 90))))))
   Transport: Silk
   SubPlan 1
     ->  Finalize Aggregate
           Output: min(t.tbalance)
           ->  Foreign Scan
                 Output: (PARTIAL min(t.tbalance))
                 Relations: Aggregate on (public.pgbench_tellers_5_fdw t)
                 Remote SQL: SELECT min(tbalance) FROM public.pgbench_tellers_5 WHERE
 ((bid = 90)) AND (($1::integer = 90))
                 Transport: Silk
```

Note that in the plan above there are no references to `SubPlan 1`.

7.2.2. Asynchronous Execution

When a sharded table is queried, the Shardman planner creates `Append` plans to scan all partitions of the table and combine the result. When some of partitions are foreign tables, the planner can decide to use an asynchronous execution. This means that when an `Append` node for the first time after initialization is asked for the tuples, it asks asynchronous child nodes to start fetching the result. For postgres_fdw async `ForeignScan` nodes, it means that a remote cursor is declared and a fetch request is sent to the remote server. If Silk transport is used, this means that the query is sent for execution to the remote server as an `MT_SPI` message.

After sending a request to the remote servers, `Append` returns to fetching data from synchronous child nodes — local scan nodes or synchronous `ForeignScan` nodes. Data from such nodes is fetched in a blocking manner. When `Append` ends getting data from synchronous nodes, it looks if async nodes have some data. If they do not, it waits for async nodes to produce results.

Shardman can execute several types of plans asynchronously. These are asynchronous `ForeignScans`, projections and trivial subquery scans (`select * from subquery`) over asynchronous plans.

The asynchronous execution is turned on by default on the level of a foreign server. This is controlled by `async_capable` postgres_fdw option. For now, only `Append` plans support asynchronous execution. `MergeAppend` does not support asynchronous execution.

While examining query plans, pay attention to the presence of non-asynchronous `ForeignScan` nodes in the plan. Asynchronous execution can significantly increase query execution time.

Examples:

```
EXPLAIN (COSTS OFF) SELECT * FROM pgbench_accounts;
                              QUERY PLAN
----------------------------------------------------------------------
 Append
   ->  Seq Scan on pgbench_accounts_0 pgbench_accounts_1
   ->  Async Foreign Scan on pgbench_accounts_1_fdw pgbench_accounts_2
   ->  Async Foreign Scan on pgbench_accounts_2_fdw pgbench_accounts_3
   ->  Seq Scan on pgbench_accounts_3 pgbench_accounts_4
```

```
   ->  Async Foreign Scan on pgbench_accounts_4_fdw pgbench_accounts_5
   ->  Async Foreign Scan on pgbench_accounts_5_fdw pgbench_accounts_6
   ->  Seq Scan on pgbench_accounts_6 pgbench_accounts_7
   ->  Async Foreign Scan on pgbench_accounts_7_fdw pgbench_accounts_8
   ->  Async Foreign Scan on pgbench_accounts_8_fdw pgbench_accounts_9
   ->  Seq Scan on pgbench_accounts_9 pgbench_accounts_10
   ->  Async Foreign Scan on pgbench_accounts_10_fdw pgbench_accounts_11
   ->  Async Foreign Scan on pgbench_accounts_11_fdw pgbench_accounts_12
   ->  Seq Scan on pgbench_accounts_12 pgbench_accounts_13
   ->  Async Foreign Scan on pgbench_accounts_13_fdw pgbench_accounts_14
   ->  Async Foreign Scan on pgbench_accounts_14_fdw pgbench_accounts_15
   ->  Seq Scan on pgbench_accounts_15 pgbench_accounts_16
   ->  Async Foreign Scan on pgbench_accounts_16_fdw pgbench_accounts_17
   ->  Async Foreign Scan on pgbench_accounts_17_fdw pgbench_accounts_18
   ->  Seq Scan on pgbench_accounts_18 pgbench_accounts_19
   ->  Async Foreign Scan on pgbench_accounts_19_fdw pgbench_accounts_20
```

Here we see a typical asynchronous plan. There are asynchronous foreign scans and local sequential scans, which are executed synchronously.

```
EXPLAIN (COSTS OFF) SELECT * FROM pgbench_accounts ORDER BY aid;
                            QUERY PLAN
------------------------------------------------------------------
 Merge Append
   Sort Key: pgbench_accounts.aid
   ->  Sort
         Sort Key: pgbench_accounts_1.aid
         ->  Seq Scan on pgbench_accounts_0 pgbench_accounts_1
   ->  Foreign Scan on pgbench_accounts_1_fdw pgbench_accounts_2
   ->  Foreign Scan on pgbench_accounts_2_fdw pgbench_accounts_3
   ->  Sort
         Sort Key: pgbench_accounts_4.aid
         ->  Seq Scan on pgbench_accounts_3 pgbench_accounts_4
   ->  Foreign Scan on pgbench_accounts_4_fdw pgbench_accounts_5
   ->  Foreign Scan on pgbench_accounts_5_fdw pgbench_accounts_6
   ->  Sort
         Sort Key: pgbench_accounts_7.aid
         ->  Seq Scan on pgbench_accounts_6 pgbench_accounts_7
   ->  Foreign Scan on pgbench_accounts_7_fdw pgbench_accounts_8
   ->  Foreign Scan on pgbench_accounts_8_fdw pgbench_accounts_9
   ->  Sort
         Sort Key: pgbench_accounts_10.aid
         ->  Seq Scan on pgbench_accounts_9 pgbench_accounts_10
   ->  Foreign Scan on pgbench_accounts_10_fdw pgbench_accounts_11
   ->  Foreign Scan on pgbench_accounts_11_fdw pgbench_accounts_12
   ->  Sort
         Sort Key: pgbench_accounts_13.aid
         ->  Seq Scan on pgbench_accounts_12 pgbench_accounts_13
   ->  Foreign Scan on pgbench_accounts_13_fdw pgbench_accounts_14
   ->  Foreign Scan on pgbench_accounts_14_fdw pgbench_accounts_15
   ->  Sort
         Sort Key: pgbench_accounts_16.aid
         ->  Seq Scan on pgbench_accounts_15 pgbench_accounts_16
   ->  Foreign Scan on pgbench_accounts_16_fdw pgbench_accounts_17
   ->  Foreign Scan on pgbench_accounts_17_fdw pgbench_accounts_18
   ->  Sort
         Sort Key: pgbench_accounts_19.aid
         ->  Seq Scan on pgbench_accounts_18 pgbench_accounts_19
   ->  Foreign Scan on pgbench_accounts_19_fdw pgbench_accounts_20
```

Here `merge append` is used, and so the execution cannot be asynchronous.

7.2.3. Fetch-all Fallback

There are a lot of cases when operations on data cannot be executed remotely (for example, when some non-immutable function is used in filters, when several sharded tables are joined by an attribute that is not a sharding key, when pushdown of a particular join type is not supported) or when the planner considers local execution to be cheaper. In such cases different operations (selection, joins or aggregations) are not pushed down, but executed locally. This can lead to inefficient query execution due to large inter-cluster traffic and high processing cost on a coordinator. When this happens, you should check if an optimizer has fresh statistics, consider rewriting the query to benefit from different forms of pushdown or at least check that the suggested query plan is reasonable enough. To make DBMS analyze data for the whole cluster, you can use shardman.global_analyze function.

7.3. Distributed Transactions

7.3.1. Visibility and CSN

7.3.1.1. *CSN* — Commit Sequence Number

A Shardman cluster uses a snapshot isolation mechanism for distributed transactions. The mechanism provides a way to synchronize snapshots between different nodes of a cluster and a way to atomically commit such a transaction with respect to other concurrent global and local transactions. These global transactions can be coordinated by using provided SQL functions or through postgres_fdw, which uses these functions on remote nodes transparently.

Assume that each node uses the CSN-based visibility: the database tracks the counter for each transaction commit (`CSN`). With such a setting, a snapshot is just a single number — a copy of the current `CSN` at the moment when the snapshot was taken. Visibility rules are boiled down to checking whether the current tuple's `CSN` is less than our snapshot's `CSN`.

Let's assume that `CSN` is the current physical time on the node and call it `GlobalCSN`. If the physical time on different nodes is perfectly synchronized, then such a snapshot obtained on one node can be used on other nodes to provide the necessary level of transaction isolation. But unfortunately physical time is never perfectly sync and can drift, and this should be taken into account. Also, there is no easy notion of lock or atomic operation in the distributed environment, so commit atomicity on different nodes with respect to concurrent snapshot acquisition should be handled somehow. This is addressed in the following way:

1. To achieve commit atomicity of different nodes, intermediate step is introduced: at the first run, a transaction is marked as `InDoubt` on all nodes, and only after that each node commits it and stamps with a given `GlobalCSN`. All readers that ran into tuples of an `InDoubt` transaction should wait until it ends and recheck the visibility.
2. When the coordinator is marking transactions as `InDoubt` on other nodes, it collects `ProposedGlobalCSN` from each participant, which is the local time on those nodes. Next, it selects the maximal value of all `ProposedGlobalCSN`s and commits the transaction on all nodes with that maximal `GlobalCSN` even if that value is greater than the current time on this node due to clock drift. So the `GlobalCSN` for the given transaction will be the same on all nodes. Each node records its last generated `CSN` (`last_csn`) and cannot generate `CSN` $\leq$ `last_csn`. When a node commits a transaction with `CSN` > `last_csn`, `last_csn` is adjusted to record this `CSN`. Due to this mechanism, a node cannot generate a `CSN`, that is less than `CSN`s of already committed transactions.
3. When a local transaction imports a foreign global snapshot with some `GlobalCSN` and the current time on this node is smaller than the incoming `GlobalCSN`, then the transaction should wait until this `GlobalCSN` time comes to the local clock.

The two last rules provide protection against time drift.

7.3.1.2. Commit Delay and External Consistency

The rules above still do not guarantee recency for snapshots genereted on nodes that do not participate in a transaction. A read operation that originates from such a node can see stale data. The probability of the anomaly directly depends on the system clock skew in the Shardman cluster.

Particular attention should be paid to the synchronization of system clocks on all cluster nodes. The size of the clock skew must be measured. If an external consistency is required, then the clock skew can be compensated with a commit delay. This delay is added

before every commit in the system, so it has a negative impact on the latency of transactions. Read-only transactions are not affected by this delay. The delay can be set using the configuration parameter csn_commit_delay.

7.3.1.3. CSN Map

The CSN visibility mechanism described above is not a general way to check the visibility of all transactions. It is used to provide isolation only for distributed transactions. As a result, each cluster node uses a visibility checking mechanism based on xid and xmin. To be able to use the CSN snapshot that points to the past, we need to keep old versions of tuples on all nodes and therefore defer vacuuming them. To do this, each node in a Shardman cluster maintains a CSN to xid mapping. The map is called CSNSnapshotXidMap. This map is a ring buffer, and it stores the correspondence between the current snapshot_csn and xmin in a sparse way: snapshot_csn is rounded to seconds (and here we use the fact that snapshot_csn is just a timestamp), and xmin is stored in the circular buffer where rounded snapshot_csn acts as an offset from the current circular buffer head. The size of the circular buffer is controlled by the csn_snapshot_defer_time configuration setting. VACUUM is not allowed to clean up tuples whose xmax is newer than the oldest xmin in CSNSnapshotXidMap.

When a CSN snapshot arrives, we check that its snapshot_csn is still in our map, otherwise, we will error out with “snapshot too old” message. If the snapshot_csn is successfully mapped, we fill backend's xmin with the value from the map. That way we can take into account backends with an imported CSN snapshot, and old tuple versions will be preserved.

7.3.1.4. CSN Map Trimming

To support global transactions, each node keeps old versions of tuples for at least csn_snapshot_defer_time seconds. With large values of csn_snapshot_defer_time, this negatively affects performance. This is because nodes save all row versions during the last csn_snapshot_defer_time seconds, but there may not be more transactions in the cluster that can read them. A special task of the monitor periodically recalculates xmin in the cluster and sets it on all nodes to the minimum possible value. This allows the vacuuming routine to remove a row version that is no longer of interest to any transaction. The shardman.monitor_trim_csnxid_map_interval configuration setting controls the worker. The worker wakes up every monitor_interval seconds and performs the following operations:

1. Checks if the current node's repgroup ID is the smallest among all IDs in the cluster. If this condition is not met, then the work on the current node is terminated. So only one node in the cluster can perform a horizon negotiation.
2. From each node of the Shardman cluster, the coordinator collects the oldest snapshot CSN among all active transactions on the node.
3. The coordinator chooses the smallest CSN and sends it to each node. Each node discards its csnXidMap values that are less than this value.

7.3.2. 2PC and Prepared Transaction Resolution

Shardman implements a two-phase commit protocol to ensure the atomicity of distributed transactions. During the execution of a distributed transaction, the coordinator node sends the command BEGIN to participant nodes to initiate their local transactions.

> The term "participant nodes" herein and subsequently refers to a subset of cluster nodes that participate in the execution of a transaction's command while the node is engaged in writing activity.

Additionally, a local transaction is created on the coordinator node. This ensures that there are corresponding local transactions on all nodes participating in the distributed transaction.

During the two-phase transaction commit, the coordinator node sends the command PREPARE TRANSACTION to the participant nodes to initiate the preparation of their local transactions for commit. If the preparation is successful, the local transaction data is stored in a disk storage, making it persistent. If all participant nodes report successful preparation to the coordinator node, the coordinator node will commit its local transaction. Subsequently, the coordinator node will also commit the previously prepared transactions on the participant nodes using the command COMMIT PREPARED.

If a failure occurs during the PREPARE TRANSACTION command on any of the participant nodes, the distributed transaction is considered aborted. The coordinator node then broadcasts the command to abort the previously prepared transactions using the ROLLBACK PREPARED command. If the local transaction was already prepared, it is aborted. However, if there was no prepared transaction with the specified name, the command to rollback is simply ignored. Subsequently, the coordinator node rolls back its local transaction.

After a successful preparation phase, there will be an object `prepared transaction` on the each of participant nodes. These objects are actually disk files and records in the server memory.

It is possible to have a prepared transaction that was created earlier through a two-phase operation and will never be completed. This can occur, for example, if the coordinator node fails exactly after the preparation step but before the commit step. It can also occur as a result of network connectivity issues. For instance, if the command `COMMIT PREPARED` from the coordinator node to a participant node ends with an error, local transactions will be committed on all participant nodes except for the one with the error. The local transaction will also be committed on the coordinator node. All participants, except for the one with the error, believe that the distributed transaction was completed. However, the one participant still waiting for `COMMIT PREPARED` will never receive it, resulting in a prepared transaction that will never be completed.

A prepared transaction consumes system resources, such as memory and disk space. An incomplete prepared transaction causes other transactions that access rows modified by that transaction to wait until the distributed operation completes. Therefore, it is necessary to complete prepared transactions, even in cases where there were failures during commit, to free up resources and ensure that other transactions can proceed.

To resolve such situations, there is a mechanism for resolving prepared transactions that is implemented as part of the Shardman monitor. It is implemented as a background worker that wakes up periodically, acting as an internal “crontab” job. By default, the period is set to 5 seconds, but it can be configured using the `shardman.monitor_dxact_interval` configuration parameter. The worker checks the presence of prepared transactions that were created earlier by a certain amount of time, specified by the `shardman.monitor_dxact_timeout` configuration parameter (which is also set to 5 seconds by default), on the same node where the Shardman monitor is running.

When the `PREPARE TRANSACTION` command is sent to a participant node, a special name is assigned to the prepared transaction. This name encodes useful information, which allows identifying the coordinator node and its local transaction.

If the Shardman monitor finds outdated prepared transactions, it extracts the coordinator's replication group ID and transaction ID of the coordinator's local transaction. The monitor then sends a query to the coordinator

```
SELECT shardman.xact_status(TransactionId)
```

which requests the current status of the coordinator's local transaction. If the query fails, for example, due to network connectivity issues, then the prepared transaction will remain untouched until the next time when the monitor wakes up.

In the case of a successful query, the coordinator node can reply with one of the following statuses:

`committed`

The local transaction on the coordinator node was completed successfully. Therefore, the Shardman monitor also commits this prepared transaction using the `COMMIT PREPARED` command.

`aborted`

The local transaction on the coordinator node was aborted. Therefore, the monitor also aborts this transaction using the `ROLLBACK PREPARED` command.

`unknown`

The transaction with such an identifier never existed on the coordinator node. Therefore, the monitor aborts this transaction using the `ROLLBACK PREPARED` command.

`active`

The local transaction on the coordinator node is still somewhere inside the `CommitTransaction()` flow. Therefore, the monitor does nothing with this transaction. The monitor will try again with this transaction at the next wake-up.

`ambiguous`

This status can be returned when `CLOG`'s truncating is enabled on the coordinator node. The `CLOG` is a bitmap that stores the status of completed local transactions. When a transaction is committed or aborted, its status is marked in the `CLOG`. However, the `CLOG` can be truncated (garbage collected) by the `VACUUM` process to discard statuses of old transactions that do not affect the visibility of data for any existing transaction.

When the `CLOG` is truncated, there is a possibility that the `shardman.xact_status()` function may not be able to unambiguously decide if a transaction exists in the past (with some status) or if it never existed. In such cases, the function returns

an ambiguous status. This can lead to uncertainty about the actual status of the transaction and can make it difficult to resolve the prepared transaction.

When the `shardman.xact_status()` function returns the `ambiguous` status for a prepared transaction, the monitor node logs a warning message indicating that the status could not be determined unambiguously. The prepared transaction is left untouched, and the monitor will try again with this transaction at the next wake-up. It is important to properly configure the `min_clog_size` parameter with the value of `1024000` (which means "never truncate CLOG") to avoid ambiguity in the status of prepared transactions.

In situations where the prepared transaction resolution mechanism is unable to resolve prepared transactions due to constant errors or ambiguous status, the administrator will need to manually intervene to resolve these transactions. This may involve examining the server logs and performing a manual rollback or commit operation on the prepared transaction. Note that leaving prepared transactions unresolved can lead to resource-consumption and performance issues, so it is important to address these situations as soon as possible.

7.4. Silk

7.4.1. Concept

Silk (Shardman InterLinK) is an experimental transport feature. It is injected at the point where `postgres_fdw` decides to transmit deparsed piece of query through `libpq` connection to the remote node, replacing `libpq` connection with itself. It is designed to decrease the count of idle `postgres_fdw` connections during transaction execution, minimize latency and boost overall throughput.

Silk implementation uses several background processes. The main routing/multiplexing process (one per PostgreSQL instance), called `silkroad`, and a bunch of background workers, called `silkworms`. While `postgres_fdw` uses `libpq`, it spawns multiple `libpq` connections from each backend to the remote node (where multiple backend processes are spawned accordingly). But if `silk` replaces `libpq` - every `silkroad` process is connected to only one remote `silkroad`. In this scheme, remote `silkworms` play the role of remote backends otherwise spawned by `postgres_fdw`.

`Silkroad` wires local backend with remote node's workers this way:

1. Backend process uses regular `postgres_fdw` API to access remote data as usual. But `postgres_fdw`, when silk is enabled, writes the query into shared memory queue instead of `libpq` connection;
2. `Silkroad` process parses incoming shared memory queue from that backend and routes the message to appropriate network connection with remote `silkroad` process.
3. Remote `silkroad` process grabs incoming message from network and (if it is a new one) redirects it to available worker's shared memory queue (or in a special "unassigned jobs" queue if all of the workers are busy).
4. At last, remote worker gets the message through its shared memory queue, executes it and sends back the result tuples (or an error) the same way.

`Silkroad` acts here like a common network switch, tossing packets between backend's shared memory and appropriate network socket. It knows nothing about content of a message relying only on the message header.

7.4.2. Event Loop

`Silkroad` process runs an event loop powered by the `libev` library. Each backend's shared memory queue is exposed at the event loop with the `eventfd` descriptor, and each network connection - with a socket descriptor.

During startup, the backend registers itself (its `eventfd` descriptors) at a local `silkroad` process. `Silkroad` responds by specifying which memory segments to use for the backend's message queue. From this moment `silkroad` will respond to events from the queue associated with this backend. Network connections between local and remote `silkroads` will be established at once on the first request from the backend to the remote node and stay alive until both of participants (`silkroad` processes) exist.

7.4.3. Routing and Multiplexing

For each subquery, we expect a subset of tuples, and therefore represent the interaction within the subquery as a bidirectional data stream. `Silkroad` uses an internal routing table to register these streams. A unique stream ID (within the Shardman cluster) is formed as a pair of "origin node address, target node address" and a locally (within the node) unique number. Each particular subquery

from a backend to remote nodes will be registered by `silkroad` as such a stream. So, any backend can be associated with many streams at the time.

When a local `silkroad` process got a message with a new stream ID from backend, it registers it in a local routing table and then redirects this message to an appropriate socket. If the connection with the remote `silkroad` does not exist, it is established using a handshake procedure. The original message that initiated a handshake is placed into a special internal buffer until the handshake succeeds. The remote `silkroad` process receiving a packet with the new ID registers it in its own table, then assigns a `silkworm` worker from a pool of available workers and places the message into the worker's shared memory queue. If all of the `silkworm` workers are busy at the moment, the message will be postponed, i.e., placed into a special "unassigned jobs queue" (note that the configuration parameter `shardman.silk_unassigned_job_queue_size` is 1024). If there is no free space in the queue, an error message will be generated and sent back to the source backend. A job from this queue will be assigned later to the first available worker when it gets rid of the previous job.

When the worker got a new “job”, it executes it through `SPI` subsystem, organizing result tuples into batches and sends them back through shared memory to the local `silkroad` process. The rest is trivial due to the whole route is known. The last resulting packet with tuples in a stream is marked as “closing”. It is an order to `silkroads` to wipe out this route from their tables.

Note that backend and remote workers stay “subscribed” to their streams until they are explicitly closed. So the backend has the opportunity to send an abort message or notify the remote worker to prematurely close the transaction. And it makes it possible to discard obsolete data packets, possibly from previous aborted transactions.

To observe the current state of the `silkroad` multiplexer process, Silk diagnostics views are available, as explained in Section 6.4.2.

7.4.4. Error Handling and Route Integrity

Besides the routing table `silkroad` tracks endpoints (backends and network connections) that were involved in some particular stream. So when some connection is closed, all the involved backends (and/or workers) will be notified of that event with a special error message, and all routes/streams related to this connection will be dismissed. The same way, if the backend crashes, its shared memory queue become detached and `silkroad` reacts by sending error messages to remote participants of every stream related to the crashed backend. So remote workers are not left doing useless work when the requester has already died.

7.4.5. Data Transmitting/batching/splitting Oversized Tuples

The resulting tuples are transmitted by `silkworm` in a native binary mode. Tuples with `external` storage attribute will be deTOASTed, but those that were compressed stay compressed.

Small tuples will be organized in batches (about 256k). Big tuples will be cut into pieces by the sender and assembled into a whole by the receiving backend.

7.4.6. Streams Flow Control

It may happen that when the next message is received from a backend, it will not fit the target network buffer. Or the message received from the network does not fit into the target shared memory queue. In such a case, the stream that caused this situation will be “suspended”. This means that the `silkroad` pauses the reaction to events from the source endpoint (connection or backend) until the target endpoint drains their messages. The rest backends and connections not affected by this route are kept working. Receiving modules of backends are designed to minimize these situations. The backend periodically checks and drains the incoming queue even when the plan executor is busy processing other plan nodes. Received tuples are stored in backend's tuplestores according the plan nodes until the executor requests the next tuple for a particular plan node execution.

When enough space is freed on the target queue, the suspended stream gets resumed, endpoint's events get unblocked and the process of receiving and sorting packets continues.

7.4.7. Implementation details

7.4.7.1. State Transferring and CSNs

When postgres_fdw works over Silk transport, only one connection between `silkroad` routing daemons is used to transfer user requests to `silkworm` workers and get their responses. Each request contains a transaction state, a replication group ID of the node where the request is formed (coordinator), a query itself and query parameters (if present). A response is either an error response message with a specific error message and error code or a bunch of tuples followed by “end of tuples” message. This means that `silkworm` has to switch to the transaction state coming with the request prior to executing the request.

For now, Silk transport is used only for read-only `SELECT` queries. All modifying requests are processed via a usual libpq connection and handled mostly as all other DML requests in PostgreSQL postgres_fdw. The only distinction is that when a DML request is processed by postgres_fdw, it resets the saved transaction state for the connection cache entry corresponding to the connection where this request is sent. Also a read-only flag is set to false for such a connection cache entry. When a request is sent over Silk transport, Shardman extension asks for the transaction state for a pair of serverid and userid from postgres_fdw. If such a connection cache entry is found in the postgres_fdw connection cache, it is not a read-only cache entry and transaction state is present in this entry, the state is returned. If it is not present, postgres_fdw retreives a full transaction state from the remote server, saves it in the connection cache entry and returns to the Shardman extension.

The full transaction state is similar to the parallel worker transaction state and contains:

- information related to the current user (uid, username)
- pid of the current backend
- transaction start timestamp
- current snapshot CSN
- flags indicating that invalidation messages are present
- backend private state:
  - array of ComboCIDs
  - internal transaction state (full transaction ID, isolation level, current command ID, etc.)
  - information about reindexed indexes

If the connection is not found in the postgres_fdw connection cache (i.e., it is a new connection) or the entry in the connection cache is marked as read-only, only these characteristics form the transaction state:

- information related to the current user (username)
- transaction start timestamp
- current snapshot CSN
- flags indicating that invalidation messages are present

Using such transaction states, `silkworm` can attach to a running transaction or start a new read-only transaction with the provided snapshot `CSN` and retreive the result.

Note that the full transaction state can be imported only on the server that exported it. Also note that due to this transaction state transferring method, you cannot use Silk transport without enabling `CSN` snapshots.

7.4.7.2. Integration with Asynchronous FDW Engine

In the Section 7.2.2, asynchronous `ForeignScan` plan nodes were presented as a way to optimize data retrieval from multiple hosts while these plan nodes were located under a single `Append` node. In the standard PostgreSQL architecture, the execution of `ForeignScan` plan nodes is implemented using the network protocol based on libpq. To improve the system performance during data transfer and reduce resource consumption, Shardman employs a different method for exchanging data with remote hosts. The mechanism for executing `ForeignScan` nodes is implemented using the Silk protocol.

To incorporate Silk transport into the asynchronous executor, modifications were made to the postgres_fdw extension. A pluggable transport was implemented as a set of interface functions included as part of the Shardman extension. During execution of callbacks that interact with remote hosts, these functions are called by the postgres_fdw extension. The pluggable Silk transport is activated if the Shardman extension is preloaded and if the foreign server has the attribute `extended_features` (applicable for any FDW server in the Shardman cluster). For all other cases, the postgres_fdw extension uses the standard exchange protocol based on libpq.

To disable the pluggable Silk transport in the Shardman cluster, it is necessary to set the `query_engine_mode` configuration parameter to the value of `ENGINE_NONE`.

In the current implementation, the pluggable Silk transport is only used for read-only queries, specifically during the execution of the `ForeignScan` node. The standard exchange protocol based on libpq is used for modifying queries.

When receiving query execution result rows using the Silk transport, the data is stored in a `TupleStoreState` storage as a complete result set, which is the same size as that returned by the remote host. The `TupleStoreState` is implemented as a

data structure that can spill data to the disk in case of memory shortage. If the remote host returns a large result set, it does not lead to an out-of-memory (OOM) condition. Once the result set is received in the `TupleStoreState`, the data is copied into the `ForeignScan` executor's in-memory buffer. The size of this buffer is defined by the `fetch_size` attribute of the foreign server. The default value of `50000` rows can be adjusted to find a balance between the performance (number of `ForeignScan` node calls) and memory consumption.

Utilizing the pluggable Silk transport for the asynchronous FDW engine results in an increase of the network exchange performance and a reduction of the system resource consumption due to better utilization of system resources, including the number of network connections.

7.5. Distributed Deadlock Detection

Distributed deadlocks may occur during the processing of distributed transactions. Let us consider the following example:

```
    create table players(id int, username text, pass text) with (distributed_by='id');
    insert into players select id, 'user_' || id, 'pass_' || id from
 generate_series(1,1000) id;
```

Assume that the record with `id=2` belongs to `node1` and the record with `id=3` belongs to `node2`.

Let us execute the following commands on different nodes:

```
    node1=# begin;
    node1=# update players set pass='someval' where id=3;

    node2=# begin;
    node2=# update players set pass='someval' where id=2;

    -- it should stuck because transaction on node1 locked record with id=3
    node2=# update players set pass='someval2' where id=3;

    -- it should stuck because transaction on node2 locked record with id=2
    node1=# update players set pass='someval2' where id=2;
```

A distributed deadlock situation arises when transactions are mutually locked by each other. PostgreSQL has an internal mechanism for deadlock detection, which detects mutual locking between child processes of a single PostgreSQL instance (backend) and resolves it. However, this mechanism is not applicable to the discovered situation because mutual locking is distributed, i.e., backends from different nodes are involved. From the point of view of the PostgreSQL lock manager, there is no deadlock condition because processes of the single instance are not locking each other. Therefore, Shardman has its own mechanism for distributed deadlock resolution.

We can represent the interaction between processes in the entire cluster as a graph. A graph vertex represents a process (backend), which we can identify with a couple of attributes `{rgid; vxid}`, where `rgid` is the replication group ID, and `vxid` is the virtual transaction ID of the currently executed transaction. Graph edges represent directional connections between vertices. Each connection is directed from the locked process to the locking process.

It is obvious that any process can be locked by only one process. In other words, if the backend is waiting for a lock, it can only wait for a specific lock. On the other hand, a locking process can acquire multiple locks, meaning that it can lock multiple backends simultaneously.

With that said, the lock graph acts as a singly linked list. If this list contains a closed loop, then here is a deadlock condition. To detect a deadlock, it is necessary to build such a list and detect closed loops in it.

The distributed deadlock detector in Shardman is implemented as a separate task inside the Shardman monitor. If a process is unable to acquire a lock within a specified amount of time (which is one second by default, but can be adjusted using the `deadlock_timeout` configuration parameter), the internal PostgreSQL deadlock detector attempts to detect a local deadlock. If no local deadlock is found, the distributed deadlock detector is activated.

The distributed deadlock detector builds a graph (list) of locks in the cluster. It queries views `pg_locks` and `pg_stat_activity` on the local node and on each of the remote cluster nodes.

The process of building the list of locks involves sequentially querying nodes in the cluster, and it is not atomic, so the list is not consistent. This means that the distributed deadlock detector may produce false positives. During the building of the list, we can store a lock that can disappear before the end of the list building process. To guarantee the reliability of deadlock detection, after the detection of a closed loop, it is necessary to re-query the nodes involved in the closed loop.

After finding the closed loop, the distributed deadlock detector chooses the process belonging to the local node and cancels it. The user process served by the cancelled backend will receive a message:

```
canceling statement due distributed deadlock was found
```

A verbose message about the detected deadlock will be recorded in the server logs:

```
LOG:  distributed deadlock detected
DETAIL:  repgroup 1, PID 95264 (application 'psql'), executed query 'update
players set pass='qqq' where id=2;' is blocked by repgroup 1, PID 95283 (application
'pgfdw:2:95278:9/2'), executed query 'UPDATE public.players_0 SET pass = 'qqq'::text
WHERE ((id = 2))'
   repgroup 1, PID 95283 (application 'pgfdw:2:95278:9/2'), executed query 'UPDATE
public.players_0 SET pass = 'qqq'::text WHERE ((id = 2))' is blocked by repgroup 2,
PID 95278 (application 'psql'), executed query 'update players set pass='qqq' where
id=3;'
   repgroup 2, PID 95278 (application 'psql'), executed query 'update players
set pass='qqq' where id=3;' is blocked by repgroup 2, PID 95267 (application
'pgfdw:1:95264:8/4'), executed query 'UPDATE public.players_1 SET pass = 'qqq'::text
WHERE ((id = 3))'
   repgroup 2, PID 95267 (application 'pgfdw:1:95264:8/4'), executed query 'UPDATE
public.players_1 SET pass = 'qqq'::text WHERE ((id = 3))' is blocked by repgroup 1,
PID 95264 (application 'psql'), executed query 'update players set pass='qqq' where
id=2;'
```

7.6. Global Sequences

Global sequences in Shardman are implemented on top of regular PostgreSQL sequences with some additional cluster-wide metadata, which among other things holds the interval of globally unused sequence elements.

When `CREATE SEQUENCE` is issued, an ordinary PostgreSQL sequence with the same name is created on every cluster node. The range of this local sequence is a bounded sub-interval of the global sequence (as defined by `MINVALUE` and `MAXVALUE` parameters), and it contains at most `block_size` elements. The `nextval` function returns values from the local sequence until it runs out, then a new sub-interval with `block_size` elements is allocated from the global sequence using a broadcast query involving all cluster nodes. So, smaller block size values make the generated numbers more monotonic across the cluster, but incur a performance penalty since the broadcast query may be rather expensive. Another way to describe the block size parameter is to say that it controls the size of the second cache level, similarly to how the `CACHE` parameter works, except at the level of an entire Shardman cluster.

Also note, that every time a new sub-interval is allocated the underlying local sequence is modified (as in `ALTER SEQUENCE`), which will lock it for the transaction duration, preventing any other local concurrent transactions from obtaining next sequence values.

7.7. Syncpoints and Consistent Backup

To ensure that cluster binary backup is consistent, Shardman implements the *syncpoint*s mechanism.

To achieve consistent visibility of distributed transactions, the technique of global snapshots based on physical clocks is used. Similarly, it is possible to get a consistent snapshot for backups, only the time corresponding to the global snapshot must be mapped to a set of `LSN` for each node. Such a set of consistent LSN in a cluster is called a *syncpoint*.

In a Shardman cluster, each node can generate its own independent local CSN, which does not guarantee the global ordering of values in time. Therefore, we cannot take this arbitrary local CSN as the basis for a *syncpoint*. Instead, Shardman chooses only those CSNs that match distributed transaction commit records as the basis of the syncpoint. These CSNs have the property of global ordering and can be used to obtain a syncpoint. The main points of this mechanism are described below.

The commit record of each completed transaction in Shardman contains the assigned `CSN` for this transaction. This value, together with the `LSN` of this record, forms a pair of values `(CSN, LSN)`. Each of the cluster nodes stores a certain number of such pairs in

RAM in a special structure - the `CSNLSNMap`. `CSNLSNMap` is a circular buffer. Each element of the map is a `(CSN, LSN)` pair. The map size is set by the configuration settings csn_lsn_map_size. A `(CSN, LSN)` pair can be added to the map only if there are no transactions on the node that can receive a CSN less than the one added. This important condition guarantees monotonous growth of CSN and LSN in CSNLSNmap, but does not guarantee that every commit record will get into the map.

When a user submits a request to create a *syncpoint*, a search by every `CSNLSNMap` is made for a largest possible CSN_g for which there is an entry (CSN_n, LSN) in each node and the condition $CSN_n <= CSN_g$ is true. The monotonic growth property of every `CSNLSNMap` ensures that each found pair (CSN_n, LSN) corresponds to the state of the global data at the time corresponding to CSN_g. If no such value of CSN_g is found, the get *syncpoint* operation fails and can be retried later. If such a value CSN_g is found, then a syncpoint is generated as a special type of `WAL` record, which is duplicated on all nodes of the cluster.

By getting a *syncpoint* and taking the LSN for each node in the cluster from it, we can make a backup of each node, which must necessarily contain that `LSN`. We can also recover to this LSN using the point in time recovery (PITR) mechanism.

7.8. Collecting Distributed Statement Statistics Using the pgpro_stats Extension

During execution of distributed queries, Shardman sends derived SQL queries to remote nodes that hold data partitions involved in the query execution. Let's call these SQL queries query fragments. Shardman sends such queries using the postgres_fdw extension. The node that queries the sharded table is called the *`coordinator`*, while the nodes that accept query fragments are called *`shards`*.

When the pgpro_stats extension is enabled on a Shardman cluster node, it collects statistics about local and distributed queries. The information about distributed queries initiated by this node is incomplete because it misses data about remote query fragments. The statistics concerning queries initiated by other nodes is also ambiguous because there is no simple way for a user to determine the distributed query to which the fragment corresponds.

To address these issues, pgpro_stats for Shardman introduces an aggregation of statistics for the distributed queries. These aggregated statistics can be accesses with the `pgpro_stats_sdm_statements` view. However, each Shardman node collects statistics for all the statements, so that the `pgpro_stats_statements` view can work the way it did before.

When a node receives a query fragment, it saves its statistics to a separate shared hash table. Periodically and asynchronously, each node sends this information from a separate table to the coordinator corresponding to the query. The coordinator aggregates the statistical data obtained from the query fragments with the statistics of its parent query, which is the query initiated by the client.

The pgpro_stats extension starts a separate background worker. This worker is responsible for sending the accumulated statistics to the coordinator nodes either every 5 seconds or when triggered by the guard latch. The collecting function sets this latch when the hash table is almost full.

To reduce the network traffic initiated by a statistics sender, compression is applied to the statistics data sent. The compression method can be selected by the `pgpro_stats.transport_compression` configuration parameter.

Each node stores the total number of statistics entries received from the shard node and the timestamp of when they were last received. When a coordinator node receives a statistics message, it updates the appropriate values, which are accessible using the SQL interface.

There are additional pgpro_stats SQL functions introduced by Shardman additions described in Section 6.2 and configuration parameters described in the section called “pgpro_stats parameters”.

7.9. Advisory Locks

PostgreSQL provides ways of creating locks that have application-defined meanings. These are cluster-wide advisory locks because the system does not enforce their use. Advisory locks and global locks work simultaneously and do not conflict with each other. Both these locks can be viewed with the *pg_locks* view and have the `shradman` value in `locktype`.

To see the advisory lock functions, refer to Advisory Lock Functions.

Appendix A. Release Notes

A.1. Postgres Pro Shardman 14.17.2

Release date: 2025-04-14

This release is based on PostgreSQL 14.17 and Shardman 14.17.1 and provides new features, optimizations and bug fixes. Major changes are as follows:

A.1.1. Core and Extensions

- Added the csn_max_shift and csn_max_shift_error configuration parameters to work with CSN snapshots for the distributed queries and imported snapshots.
- Added the shardman.context_log configuration parameter that allows the coordinator to see the error context on a worker.
- Deleted the `csn_max_commit_shift` and `csn_max_snapshot_shift` configuration parameters.
- Forbade access to global views from standby servers.
- Updated the `ABORT` command output for workers that now shows a detailed information about the abort reasons on a coordinator.
- Optimized the `MergeAppend` behavior to consider the cheapest sorted total path. Previously the most efficient path could not be chosen by the planner.

A.1.2. Management Utilities

- Fixed the GO-2025-3553 vulnerability.
- Fixed the invalid shardmanctl nodes command behavior. Now the same node cannot be specified more than once in the shradmanctl nodes start, shradmanctl nodes stop, and shradmanctl nodes restart commands.
- Added cluster configuration parameters related to replication slots: `additionalReplicationSlots` to specify an array of names for replication slots to be created on the master, `createSlotsOnFollowers` to also create replication slots on standby nodes, and `additionalSlotsLagLimit` to limit lagging behind for additional replication slots.
- Added placeholder support for pgParameters.
- Optimized the `pgwaldump` adapter to avoid sending the entire `pg_waldump` output to the buffer.

A.2. Postgres Pro Shardman 14.17.1

Release date: 2025-03-17

This release is based on PostgreSQL 14.17 and Shardman 14.15.4 and provides new features, optimizations and bug fixes. Major changes are as follows:

A.2.1. Core and Extensions

- The `shardman.silk_shmem_size`, `shardman.silk_netbuf_size` `shardman.silk_suspend_shmqe_limit`, `shardman.silk_resume_shmqe_limit`, `shardman.silk_suspend_netqe_limit`, `shardman.silk_resume_netqe_limit` parameters now cannot change their values and are only kept for compatibility purposes.
- Added a detailed description for the following configuration parameters: shardman.silk_unassigned_job_queue_size, shardman.silk_max_message, shardman.silkworm_fetch_size, and shardman.silk_hello_timeout.
- Optimized mechanisms to result in receiving of a consistent syncpoint.
- Improved error messages for temporary sharded or global tables creation failures.
- Updated the `CREATE USER MAPPING`, `ALTER USER MAPPING`, and `DROP USER MAPPING` commands that are now prohibited when applied to mappings for foreign servers from the Shardman cluster.

- Updated the `shardman.users` and `pg_user_mapping` catalogs that are now not stored in plain-text.
- Fixed an issue with the processing of `ALTER INDEX` commands for the sharded tables.
- Fixed a bug related to the Silk transport that previously resulted in a recursive error and the postmaster crash.
- Fixed the BDU:2025-01601 vulnerabilitiy.

A.2.2. Management Utilities

- Added a new parameter `--lock-lifetime` to the probackup backup command to allow setting the maximum time that pg_probackup can hold the lock, in seconds.
- Updated the shardmanctl forall, shardmanctl load, and shardmanctl history commands so they can run concurrently and do not block other processes.
- Added a new option `-n|--node` to the commands shardmanctl shard stop and shardmanctl shard start to specify the node to start or stop.
- Improved error messaging for the pg_probackup-related tools.
- Implemented safe restoration of the `etcd` cluster from the dump for the cold backup by adding a shardmanctl store restore command.
- Optimized the backup validation process by adding new options `--data-validate`, `-remote-port`, `--remote-user`, and `--ssh-key` to the shardmanctl probackup restore command.
- Added a new filter `restart_required_params` to the shardmanctl status command that checks that all the `postgres` parameters requiring a `postgres` instance restart are applied. The successful output shows no pending restart parameters.
- Implemented the automatic confirmation of the restart for the shardmanctl probackup archive-command add and the probackup backup commands with the `-y|--yes` option.
- Updated the shardmanctl history output to show whether the listed commands succeeded or failed.
- Implemented the automatic confirmation of the restart necessary for the parameters to take effect for the shardmanctl config update and shardmanctl config set command with the `-y|--yes` option. If this option is not specified, and the parameters update requires a restart, the manual confirmation will be requested. If not confirmed, the cluster will continue to work, yet the new parameter values will only take effect after the restart.
- Fixed an issue that previously resulted in the `pg_hba.conf` row duplicates.
- Fixed a bug that previously resulted in the shardmanctl status command failure.
- Updated the supported version of pg_probackup to 2.8.8.

A.3. Postgres Pro Shardman 14.15.4

Release date: 2025-02-19

This release is based on PostgreSQL 14.15 and Shardman 14.15.3 and provides optimizations and bug fixes. Major changes are as follows:

A.3.1. Core and Extensions

- Fixed a bug that previously resulted in the incorrect reusing of the tracepoint memory while executing a prepared statement with shardman.silk_tracepoints enabled.

A.3.2. Management Utilities

- Fixed an issue that previously resulted in the backup failure after the primary nodes were switched.
- Updated the supported version of pg_probackup to 2.8.7.

A.4. Postgres Pro Shardman 14.15.3

Release date: 2025-02-10

This release is based on PostgreSQL 14.15 and Shardman 14.15.2 and provides new features, optimizations and bug fixes. Major changes are as follows:

A.4.1. Core and Extensions

- Added the shardman.pg_indoubt_xacts view that displays information about transactions that are currently in the `InDoubt` state.
- Added the global views for the system catalog and statistics-related views.
- Added new fields to the `shardman.silk_connects`, `shardman.silk_backends`, and `shardman.silk_routing` views that show time from the last reading or writing event of a connect or a backend.
- Added a new error message for the coordinator if the `MT_SPI` message size exceeds the `silk_max_message` value, if a query is executed via Silk.
- Added new diagnostic messages for the scenarios where the exported transaction state size is more than half of `shardman.silk_max_message`.
- Updated the maximum values of the `shardman.silk_num_workers`, `shardman.silk_unassigned_job_queue_size`, `shardman.silk_max_message`, `shardman.silk_shmem_size`, `shardman.silk_netbuf_size`, `shardman.silk_suspend_shmqe_limit`, `shardman.silk_resume_shmqe_limit`, `shardman.silk_suspend_netqe_limit`, and `shardman.silk_resume_netqe_limit` parameters.
- Added a feature to pushdown the type conversion operations to a remote server.
- Added a new limitation for the self-referencing sharded tables that are allowed only if a foreign key is referencing the same partition of the sharded table.
- Upgraded etcd to version 3.5.18.

A.4.2. Management Utilities

- Added a new subcommand show-config to the shardmanctl probackup command. It displays all the current pg_probackup configuration settings, including those that are specified in the `pg_probackup.conf`, and those that were provided on a command line.
- Updated the backup retention policies with the new parameters of the sharmanctl probackup delete and shardmanctl probackup backup subcommands: --retention-redundancy, --retention-window, --wal-depth, --delete-expired, and --merge-expired.
- Fixed the CVE-2024-24790 and CVE-2024-45337 vulnerabilities.

A.5. Postgres Pro Shardman 14.15.2

Release date: 2024-12-16

This release is based on PostgreSQL 14.15 and provides new features, optimizations and bug fixes. Major changes are as follows:

A.5.1. Core and Extensions

- Added the `in_queue_used` and `out_queue_used` fields to the shardman.silk_backends view that show the number of queued data bytes in the incoming or outgoing queue in the shared memory between the backend and multiplexer.
- Added a new shardman.silk_routing function along with the corresponding views shardman.silk_routing and `gv_silk_routing`. They show information about the current active routes.
- Added a new shardman.silk_rbc_snap function that retrieves a consistent snapshot of all the connects, backends and routes that can be used by `silk_connects`, `silk_backends`, and `silk_routes` functions.
- Added `shardman.silk_state` and `shardman.silk_statinfo` views, the `shardman.silk_statinfo_reset()` function and the shardman.silk_track_time configuration parameter that cover the multiplexer process state.
- Added two new configuration parameters, shardman.silk_tracelog configuration parameter that enables or disables Silk tracing and debug logging, and shardman.silk_tracelog_category that defines the Silk message categories to be traced.

- Added two new configuration parameters, enable_non_equivalence_filters that enables the optimizer to generate additional non-equivalence conditions using equivalence classes, and optimize_row_in_expr that enables the optimizer to generate additional conditions from the `IN ()` expression.
- Added a new configuration parameter track_xact_time, the shardman.pg_stat_xact_time view, and the `shardman.gv_stat_xact_time` global view for showing statistics for the time spent on transactions.
- Added the `attached` field to the shardman.silk_backends view and the `silk_backends` function that shows the actual attaching of a backend to the multiplexer.
- Added a new shardman.silk_stream_work_mem configuration parameter that sets the base maximum amount of memory to be used by a Silk stream before writing to the temporary disk files.
- Updated the `EXPLAIN` command output to show `server` and `transport` blocks in one row, if set to `verbose`.
- Updated the supported version of pgpro_pwr to 4.8.
- Updated the supported version of pg_query_state to 1.1.
- Updated the supported version of pgpro_stats to 1.8-sdm4.
- Updated the supported version of pg_probackup to 2.8.5.
- Sped up planning for the queries *`field`* `= ANY (ARRAY[`*`values`*`])` for the arrays with a big number of records.
- Updated the postgres_fdw.foreign_explain configuration parameter type from `boolean` to `enum`, the default value being `collapsed`. Also updated the `EXPLAIN` command output to comply with the new values.
- Fixed a bug that previously resulted in the multiplexer hanging.
- Updated the `nextval` function that can be used to generate next sequence values that are unique across the entire cluster.

A.5.2. Management Utilities

- Fixed a bug that previously resulted in the command line key being ignored if a corresponding environment variable was set.
- Fixed a bug that previously resulted in shardmanctl bench run failure due to its memory buffers overflow.
- Fixed a bug that previously resulted in shardmanctl bench failure if the command wasn't executed under the `postgres` user.
- Fixed a bug that previously resulted in the full resync of a replica and was caused by saving invalid data to the `postgresql.auto.conf` file.
- Fixed a bug that previously resulted in the PANIC-level error when calling any commands that modify configurations of a cluster that was not yet initialized.
- Updated the shardmanctl bench run command flag `-f|--file` *`file_name`* to add a transaction script read from *`file_name`* to the list of scripts to be executed and to write an integer weight for each file.
- Updated the shardmanctl bench run command with `-P|--progress`, `-R|--rate`, and `-M|--protocol` flags.

A.6. Postgres Pro Shardman 14.15.1

Release date: 2024-11-25

This release provides new features, optimizations and bug fixes. Major changes are as follows:

A.6.1. Core and Extensions

- Added a new metric to the `shardman.pg_stat_csn` view that counts transactions with an exceeded time in the `inDoubt` state.
- Added new fields to the shardman.silk_pending_jobs view: `query`, `pending_queue_bytes`, and `pending_queue_messages` for the first queued message, the pending queue size, in bytes, and the number of pending queue messages.
- Added tracing for the queries processed via the Silk transport and added a new configuration parameter shardman.silk_tracepoints that enables it.

- Updated the function `current_date` that now can be pre-evaluated locally on coordinator. `timestamp` and `timestamptz` comparisons are now considered safe for the remote execution.
- Added pg_query_state support.
- Introduced cluster-wide advisory locks which are recommended locks that have application-defined meanings. Also added advisory lock functions.
- Fixed a bug that previously resulted in uncontrolled memory usage and allocation by `silkworm` while processing messages.
- Fixed a bug that previously resulted in unstable Silk connectivity and potential queries hanging in case `shardman.silk_flow_control` was enabled.

A.6.2. Management Utilities

- Added a new shardmanctl history command that shows history of the commands that updated the cluster. By default, they are sorted from the most recent to the oldest ones.
- Updated etcd version to 3.5.13.
- Added the normalization for the rebalance process. It allows properly resuming it if it was interrupted.
- Added a new feature for the shardmand application that allows configuring a port in `sdmspec.json` with encryption option.
- Updated the PostgreSQL parameter validation mechanism that now uses data returned by PostgreSQL instance.

A.7. Postgres Pro Shardman 14.13.4

Release date: 2024-11-13

This release provides new features, optimizations and bug fixes. Major changes are as follows:

A.7.1. Core and Extensions

- #dded support for asynchronous execution of `ForeignScan` operations under `MergeAppend`, controlled by the enable_async_merge_append parameter, which is enabled by default. If the operations under `MergeAppend` support asynchronous execution, requests are sent asynchronously at the start of the `MergeAppend` operation, and the results are cached as they are received. These cached results are then used, just as they would be in synchronous `MergeAppend`, for merge sorting.
- Implemented the ability to use sorting on the remote server if it allows performing `MergeJoin` or `MergeAppend` operations. This is controlled by the postgres_fdw.additional_ordered_paths parameter, which is enabled by default in new installations but must be explicitly enabled in upgraded clusters.
- Added support for the limit clause pushdown under `Append`and `MergeAppend` when there is a `Sort` plan node between `LIMIT` and `Append`. It is possible when rows in subplans of `Append`/`MergeAppend` are already sorted in the necessary order.
- Sped up `INSERT`, `UPDATE`, and `DELETE` operations with global tables. Added the shardman.gt_batch_size configuration parameter that specifies the buffer size for `INSERT` and `DELETE` commands executed on global tables.
- Added a limitation on creating sharded and local partitioned tables based on the same attribute.
- Added a new shardman.broadcast_query function that returns an executed SQL *`statement`* results.
- Added a new field `CSNXidMap_last_trim` to the `shardman.pg_stat_csn` view that shows the last time when the `shardman.trim_csnxid_map()` function was called.
- Improved the state consistency checks for the shardman application.
- Fixed an issue with inappropriate resource allocation, which could cause errors in some corner cases when tuples were spilled to disk.
- Fixed a bug in pg_rewind that previously resulted in the former primary server full resync on replica promotion.
- Upgraded supported version of pgpro_pwr to 4.7.

A.7.2. Management Utilities

- Added logging of the updated parameters in case it results in `postgresql` restart.

- Improved the logic for obtaining the state of the PostgreSQL instance.
- Improved shardmand log messaging.
- Fixed a bug that previously resulted in the shardmanctl psql command failure.
- Added support for compression level values depending on the compression algorithm when creating a backup with shardmanctl probackup backup.
- Updated the shardmanctl benchmark with a new dependency between the `pgbench_branches` number of records and the number of nodes. This allows a better distribution of data between nodes.
- Added the shardmanctl shard reset command that resets nodes of a replication group if they are in a state of hanging.
- Added the shardmanctl daemon set command that allows updating the log parameters without restart.

A.8. Postgres Pro Shardman 14.13.3

Release date: 2024-10-28

This release provides new features, optimizations and bug fixes. Major changes are as follows:

A.8.1. Core and Extensions

- Added configuration parameters to enable getting information on crashes of a backend. The crash_info parameter turns on this functionality, while crash_info_dump and crash_info_location specify the contents and location of crash information files, respectively.

A.8.2. Management Utilities

- Fixed a bug that affected switching from primary to replica server in cases when attempts to receive server configuration parameters failed.

A.9. Postgres Pro Shardman 14.13.2

Release date: 2024-10-22

This release provides new features, optimizations and bug fixes. Major changes are as follows:

A.9.1. Core and Extensions

- Added a new configuration parameter `shardman.silk-flow-control` that controls the mode of handling read events. It has three possible values: `none`, `round_robin`, and `shortest_job_first`.
- Added the shardman.pg_stat_foreign_stat_bytes view that shows the amount of statistics for foreign relations transferred over the network between Shardman cluster nodes. Also added the corresponding global view `shardman.gv_stat_foreign_bytes`.
- Added a new configuration parameter `shardman.sync_cluster_settings` that enables cluster-wide synchronization of configuration parameters set by user.
- Added a new configuration parameter `shardman.sync_cluster_settings_blacklist` that excludes the options not to be propagated to a remote cluster.
- Added a new configuration parameter `enable_sql_func_custom_plans`. If enabled, custom plans can be created to run SQL functions. Enabled by default for the new clusters and disabled for the old ones.
- Fixed a bug that previously resulted in shardmand hanging in case an etcd cluster looses quorum.
- Allowed `ALTER COLUMN SET STATISTICS` for global and sharded tables.
- Introduces the limitation for the privilege management per columns that is not supported for global tables.
- Introduced a limitation that global tables cannot inherit other tables.
- Removed the limitation for using of `DEFERRABLE` constraints for global tables that is now allowed.

- Added a new field `CSNXidMap_last_trim` to the `shardman.pg_stat_csn` view that shows when the most recent `shardman.trim_csnxid_map()` function was called.

A.9.2. Management Utilities

- Added the shardmanctl psql command that creates a connection to the first available master node if no options are specified. If `--shard` is specified, the connection is installed with the shard current master.
- Enabled the `lz4` compression method for the `default_toast_compression`.
- Fixed a bug that previously resulted in a failure of the shardmanctl probackup checkdb command when a custom port was specified in Shardman configuration.
- Fixed a shardmand bug that previously resulted in the application failing with PANIC-level error in case of insufficient access rights to the `DataDir` directory.
- Fixed a bug that previously resulted in primary server switching to a replica after restart. Also, added a new option `--no-wait` to the `shardmanctl restart` command (disabled by default).

A.10. Postgres Pro Shardman 14.13.1

Release date: 2024-09-12

This release provides new features, optimizations and bug fixes. Major changes are as follows:

A.10.1. Core and Extensions

- Added a possibility to push down the joins like `JOIN UNIQUE INNER` to a remote server.
- Added the shardman.pg_stat_monitor view showing metrics of the Shardman monitor; shardman.pg_stat_net_usage view showing the cumulative network traffic between Shardman cluster nodes; and `shardman.gv_lock_graph` view that displays a graph of locks between processes on Shardman cluster nodes including external locks.
- Added the shardman.oldest_csn view that shows tuple `csn`, `xid`, and `rgid` containing CSN and XID of the oldest transaction in the cluster, along with transaction's replication group number.
- Added the `csn_max_snapshot_shift` configuration parameter that enables checking the imported snapshots in `pg_csn_snapshot_import()`.
- Introduced new limitations on the types of tables that can be included in logical replication.
- Upgraded supported version of pg_probackup to 2.8.3.

A.10.2. Management Utilities

- Updated the text of the messages sent when trying to get the topology configuration sent by the `shardmanctl cluster topology` command on an uninitialized cluster, as well as lowered the logging level for this case.
- Fixed the `shardmanctl bench run` command to avoid long delays before its execution.
- Added the `shardmanctl config update credentials` command that updates password or certificate/key of a user to connect to a Shardman cluster.
- Added the `shardmanctl config revisions`, `shardmanctl config rollback`, `shardmanctl config revisions rm`, and `shardmanctl config get` commands, and added to the console output the information about the host from which the appropriate command was executed and the user who executed it.
  - The `shardmanctl config rollback` command makes a rollback of Shardman to one of the previous states of Shardman cluster configuration. This command has the replicas reinitialized when rolling back to the config revision that has `max_connections`, `max_prepared_transactions`, or `max_worker_processes` parameters.
  - The `shardmanctl config revisions` command outputs `revision_id` that is the timestamp of the command that resulted in the Shardman configuration change, `host` that is the host from which the appropriate command was executed, `user` that is the user who executed the command, and `command` that is the command itself.
  - The `shardmanctl config revisions set` command allows setting the length of the configuration revision history. Added a hard lower limit on the revision history length of a Shardman cluster configuration. This value cannot be

lower than 5. For clusters where the configuration revision history was not tracked, the length is automatically set to the default value of 20.

- The `shardmanctl config get` command outputs the current full cluster specification or the configuration of the specified revision. The `--choose-revision` option enables an interactive mode of choosing the configuration of the specified revision.
- The `shardmanctl config revisions rm` command deletes a specified configuration revision from the history.

- Modified the role description in `sdmspec.json`.

A.11. Postgres Pro Shardman 14.12.2

Release date: 2024-08-01

This release provides new features, optimizations and bug fixes. Major changes are as follows:

A.11.1. Core and Extensions

- Added a possibility to create a global or sharded table like another global, sharded or local table. Creation of a table like a local table currently has certain limitations.
- Fixed processing of the `IF NOT EXISTS` parameter of the CREATE TABLE command for sharded and global tables. Earlier a table with an incorrect structure could be created if a partitioned table with the same name existed on one of the cluster nodes.
- Changed the default value of the num_parts storage parameter to 24 to achieve a more even data distribution for 2, 3, 4, 6, and 8-node clusters.
- Added enable_merge_append configuration parameter that enables or disables the use of `MergeAppend` plans by the query planner. Specifically, this allows disabling the use of these plans when they are too expensive.
- Added the pgpro_stats.track_shardman_connections configuration parameter that enables or disables Shardman-specific statement processing.
- Enabled pushing down join queries with `VALUES` to a remote server.
- Removed a limit of about 64K on the number of tables in a query.
- Added the `shardman.pg_stat_monitor` view that provides statistics on the work of the distributed deadlock detector and of the prepared transaction resolution services.
- Added the `shardman.gv_stats_sdm_statements` global view that allows accessing the aggregated statistics for the distributed queries.
- Updated the pgpro_stats `pgpro_stats_sdm_statements` view to only contain statistics on queries involving sharded tables.
- Upgraded supported version of pg_probackup to 2.8.2.

A.11.2. Management Utilities

- Implemented the ability to backup clusters with tablespaces. Now the tablespaces are located under the backup directory.
- Enabled `shardmanctl probackup` restore a fully/partially working cluster from a backup made on a partially working cluster.
- Added the `--no-wait` option for the `shardmanctl shard add` command that sets shardmanctl not to wait for the shard to start and lifts the lock on other commands.
- Added the `s|--scale` option for the `shardmanctl bench run` command. It multiplies the number of generated rows by the scale factor.
- Added the `shardmanctl script` command that executes non-transactional commands from a file or from the command-line on the specified shards.
- Updated the `sdmspec.json` configuration file generated by the `shardmanctl config generate` command to exclude the parameters that depend on the hardware resources and the workload on the cluster node. These parameters are now

set to their default values. Previously, cluster initialization could fail on nodes with lower capacity due to setting these values too high.

- Enabled restoring other clusters from a cluster backup if they have the same topology. Added the `shardmanctl config update ip` command that updates the specified node IPs in the cluster.
- Added the `--log-format` option to shardmand that specifies the log output format, `json` or `text`.

A.12. Postgres Pro Shardman 14.12.1

Release date: 2024-06-06

This release provides new features, optimizations and bug fixes. Major changes are as follows:

A.12.1. Core and Extensions

- Added the new `REMOTE` parameter of `EXPLAIN`, enabled by default, which allows the `EXPLAIN` output for queries executed on foreign servers.
- Implemented a Shardman-specific estimation logic for plan costs. It may help the planner choose generic plans more often when the overall shape of a generic plan is similar to that of a custom plan.
- Added support for initial pruning of foreign aggregate plan nodes.
- Added cumulative metrics for the network traffic between Shardman cluster nodes in the `shardman.pg_stat_netusage` view.
- Updated the `pg_stat_activity` view to show the status of the monitor's worker processes.

A.12.2. Management Utilities

- Prevented the CVE-2023-45288 and CVE-2023-44487 vulnerabilities.
- Fixed a bug in the `shardmanctl cleanup` command that could make it impossible to delete replication groups.
- Improved the output of the `shardmanctl forall` command in the cases where the result is empty.
- Fixed shardmand failures that could occur when the Shardman cluster was underconfigured.

A.13. Postgres Pro Shardman 14.11.2

Release date: 2024-04-18

This release provides new features, optimizations and bug fixes. Major changes are as follows:

A.13.1. Core and Extensions

- Added the foreign_analyze_interval setting, in seconds, indicating how often to gather foreign statistics during autovacuum.
- Added a possibility to create a foreign key between a sharded and a global table or between two global tables with the `ON DELETE CASCADE` action.
- Added support for `MergeAppend` node pruning in generic plans.
- Added support for a pushdown (remote execution) of `to_timestamp()` functions.
- Implemented global views. Fetching from a global view returns a union of rows from the corresponding local views with the rows fetched from each of the local view cluster nodes.
- Added a description of Silk multiplexer diagnostics views.
- Improved error messages related to updating cluster parameters.

A.13.2. Management Utilities

- Added the `--no-validate` and `--skip-block-validation` flags to the `shardmanctl probackup restore` command.

- Improved the process of `restore` to a cluster compatible with the source one.
- Added the `shardmanctl probackup checkdb` command to verify the Shardman cluster correctness by detecting physical and logical corruption.
- Enabled shardmanctl `set` and `config update` commands to work on a cluster that was stopped using `shardmanctl stop`.
- Added the `--all` flag to the `shardmanctl getconnstr` command to add information on replicas to the command output.
- Added new commands `nodes start`, `nodes restart` and `nodes stop` for nodes, as well as `start` and `stop` for shards to shardmanctl.
- Extended permissions for the shardmand data directory.

A.14. Postgres Pro Shardman 14.11.1

Release date: 2024-03-14

This release provides new features, optimizations and bug fixes. Major changes are as follows:

A.14.1. Core and Extensions

- Enhanced the pgpro_stats extension to give a better understanding of what system resources are used for distributed queries. Now the regular `pgpro_stats_statements` view shows the gathered statistics for individual statements on the current Shardman node (they can be part of some distributed query), while the `pgpro_stats_sdm_statements` view shows the gathered statistics for the distributed queries originating from the current node, that is, aggregated from all the participating nodes.
- Added the *pgpro_pwr* package compatible with Shardman. This allows Shardman users to build workload reports.
- Improved the `EXPLAIN` output. If a query plan contains `ForeignScan` nodes, the `EXPLAIN` output for queries executed on the remote server can now be included.
- Added a new configuration parameter `enable_partition_pruning_extra` that enables extended subplan pruning logic when building and executing generic plans where the set of useful partitions depends on the prepared query parameters. This allows Shardman to do initial pruning of complex subplans, joins and partial aggregates, in particular.
- Added metrics to the `shardman.pg_stat_csn` view that show delays of the global horizon and the transaction that may cause that delay. They may be useful to research autovacuum issues.

A.14.2. Management Utilities

- Considerably improved backup and restore with the `shardmanctl probackup` command. Notable changes are as follows:
  - Added support of backups to an S3-compatible object storage.
  - Implemented selective WAL archiving on the specified shards by the `probackup` subcommand.
  - Added two new commands `shardmanctl probackup delete` and `shardmanctl probackup merge`. The `delete` command deletes a backup with a specified ID and the archived WAL files that are no longer in use. The `merge` command merges the backups that belong to a common incremental backup chain.
  - Added new `shardmanctl probackup set-config` command that adds the specified settings to the `pg_probackup.conf` or modifies the existing ones.
  - Added a new option `log-to-console` for the `validate` subcommand. Set the log rotation file size to 20 MB. If this value is reached, the log file is rotated once a `validate` or `backup` subcommand is launched.
  - Increased the number of retries for some subcommands to avoid backup failures caused by large database sizes.
  - Added topology compatibility checks between the current Shardman cluster and the one in the backup directory to `backup` and `restore` subcommands.
  - Set the default value for the number of concurrent pg_probackup processes to the number of logical CPUs of the system.

  - Fixed data cleanup after a failure of a `backup` subcommand. Previously, some data of a failed backup could still remain in the repository.
  - Fixed hanging that could occur during metadata-only restore of a Shardman cluster.
  - Fixed the pg_probackup issue that could occur during the schema recovery process.
  - Changed the behavior of metadata-only restore to avoid losing a cluster. Now the cluster is stopped before such a restore and restarted after it, a cluster that has no nodes cannot be restored from the etcd dump, and if cluster IDs of the dump and the current cluster are different, the user is asked whether restoring the cluster with the changed ID is OK.
  - Added new options for `archive-command`: `--compress`, `--compress-algorithm`, `--compress-level`, `--batch-size`, and `-j|--jobs`. This helps to reduce the WAL size.
  - Improved the `show` subcommand output. Added new flags `-archive` to output the log information, `-instance` and `-i|--backup-id` to output information for the specified backups and instances.
- Updated the `getconnstr` and `cluster topology` commands so that they do not issue a lock on other processes. Previously, some commands failed to receive a connection string because of the locks.
- Fixed a panic that could occur on a Shardman cluster configured with `PlacementPolicy` = `manual` when a user executed the command `shardmanctl cluster repfactor set`.
- Hid uninformative warnings that pg_dump displayed during execution of `shardmanctl nodes add` and `shardmanctl probackup backup` commands.
- Removed a lock that was required by the `shardmanctl status` command. Previously `shardmanctl status` did not provide any useful information in case a process hung as it was waiting for the lock from that process.
- Added the `forceSuUserLocalPeerAuth` configuration parameter. When enabled, it sets a peer authentication via unix socket for the `postgres` user unless `strictUserHBA` is set to `true`. See sdmspec.json for details.
- Added a URL for Prometheus automatic service discovery metrics to shardmand.

A.15. Postgres Pro Shardman 14.10.3

Release date: 2024-02-02

This release is based on Shardman 14.10.2 and provides optimizations and bug fixes. Major changes are as follows.

- Fixed an issue that prevented Shardman from working with *pg_probackup* when PostgresSQL ran on port different from `5432`.
- Fixed Shardman to enable pg_probackup run on a node that is not in the Shardman cluster.
- Fixed hanging of `shardmanctl probackup restore` that took place in some cases.
- Added cleanup of the backup directory in case of a `shardmanctl probackup init` failure.
- Improved error handling of `probackup` backups. Now if a backup fails on one shard, it gets terminated on the others.
- Improved the behavior of `shardmanctl probackup show` to display a message informing of no backups when the `backup_info` file is missing.

A.16. Postgres Pro Shardman 14.10.2

Release date: 2024-01-25

This is the first public release of Shardman software. It is shipped as packages with Shardman DBMS and management utilities.

Shardman DBMS is based on PostgreSQL with additional patches where most of the functionality is implemented in shardman and *postgres_fdw* extensions.

Major features are as follows:

- Distributed ACID transactions.
- Distributed DDL to manage cluster-wide objects, including sharded and global tables, sequences and users.

- Efficient multiplexing transport for intercluster communication.
- Efficient distributed query planning and execution.
- Automatic resolution of prepared transactions and distributed deadlock detection.
- Aggregation of distributed statement statistics and internal network metrics in *pgpro_stats* extension.
- Support for global tablespaces and Compressed File System (CFS).

Management utilities are implemented as shardmand service and shardmanctl tool. They use third-party etcd service to store global cluster configuration and exchange information.

Major features are as follows:

- Initial cluster configuration and setup.
- Managing and displaying the current configuration of shards and replicas.
- Updating and setting parameters in the cluster.
- Ensuring fault tolerance and high availability of shards.
- Consistent data backup and restore (*pg_basebackup* and *pg_probackup* support).
- Fast data load and automatic schema migration.
- Benchmarking tools.
- Updating database metadata on DBMS updates.

Appendix B. Glossary

This is a list of terms and their meaning in the context of Shardman. For terms that are used in this document in the general context of PostgreSQL and relational databases, see *PostgreSQL Glossary*.

ACID	Atomicity, Consistency, Isolation and Durability. This set of properties of database transactions is intended to guarantee validity in concurrent operation and even in event of errors, power failures, etc. For more information, see *PostgreSQL Glossary*.
Clover	A set of nodes where each node holds a PostgreSQL instance that is the master for one of the replication groups and PostgreSQL instances that are replicas for all the other replication groups. The total number of nodes in a clover is equal to the replication factor.
etcd	A distributed reliable key-value store for the most critical data of a distributed system. For more information, see *etcd home page*.
Global role	A role such that operations on it are always performed on all replication groups simultaneously.
Global user	A user such that operations on it are always performed on all replication groups simultaneously.
Replication group	A stolon cluster with one master and one or more replicas. Replication groups are organized in Clovers. Shardman utilities often refer to replication groups as "repgroups".
Shard	In Sharding, some table partitions located on one node being the master for them.
Sharded table	A partitioned table where some partitions are regular local tables that make up a Shard and the other partitions are foreign tables available from remote servers via `postgres_fdw`.
Sharding	A database design principle where rows of a table are held separately in different databases that are potentially managed by different DBMS instances.
Silk (Shardman InterLinK)	Experimental transport that can be used in a Shardman cluster for communication between nodes.
stolon	A cloud native PostgreSQL manager for PostgreSQL high availability. For more information, see *stolon on github*.
syncpoint	A set of consistent LSNs in a cluster corresponding to a global snapshot.

Appendix C. FAQ

C.1. General Questions

C.1.1. What is Shardman?

Shardman is a PostgreSQL-based distributed database management system (DBMS) that implements sharding. *Sharding* is a database design principle where rows of a table are held separately in different databases that are potentially managed by different DBMS instances. The main purpose of Shardman is to make querying sharded distributed databases efficient and ease the complexity of managing them.

C.1.2. What does Shardman consist of?

Shardman is composed of several software components:

- PostgreSQL 14 DBMS with a set of patches.
- Shardman extension.
- Management tools and services, including built-in stolon manager to provide high availability.

C.1.3. When to use Shardman?

- The working volume of data does not fit in the RAM of one server, but several shards can fit (or at least reading is parallelized).
- Number of sessions is too large for one instance of PostgreSQL.
- Intensive writing to WAL takes place.
- Complex logic consuming too much CPU, and one server is not enough.

C.1.4. When is Shardman not appropriate?

- If the memory, session, CPU load can be pulled by a single PostgreSQL server, this will be both faster and simpler. (This applies to testing too!)

C.1.5. How many nodes does it take to deploy Shardman?

A minimum of three nodes are required to deploy Shardman. One node is required for an etcd cluster (single-node etcd cluster), and a minimum of two nodes is required for the RDBMS cluster. It is possible to reduce the minimum deployment to two nodes by placing etcd on one of the RDBMS cluster nodes. The minimal deployment is described in section Get Started with Shardman.

C.1.6. Does Shardman support fault tolerance?

Yes, Shardman is fault-tolerant at the level of each shard. Each shard is a fault-tolerant cluster.

C.1.7. How is sharding structured?

In Shardman, tables are divided into partitions, and the partitions are distributed between shards.

C.1.8. Is it possible to change the number of partitions?

No, the number of partitions of sharded tables is set when creating them and remains unchanged. If you expect that the amount of data you have will grow significantly, you should create the necessary number of partitions (by default - 20) in advance.

C.1.9. Does Shardman support resharding?

No, Shardman currently does not support automatic change of a sharding key. In order to change the sharding key, you need to create new tables with a new sharding key and migrate data from old tables to new ones.

C.1.10. Is it possible to convert an unsharded (local) table to a sharded one?

No, Shardman currently does not support this feature.

C.1.11. Does Shardman support adding and removing shards?

Minimally a Shardman cluster can consist of a single node without fault tolerance, but such a configuration makes little sense. You can add or remove shards, Shardman will automatically (by default, this is adjustable) redistribute data between nodes. Replicas can be added to Shardman, then shards will be fault-tolerant.

C.1.12. What is the status of data balancing?

When adding new shards, data will be redistributed between all shards, including new ones.

C.1.13. How is a Shardman cluster accessed?

Shardman can be accessed through any node in the cluster, all nodes in the cluster are equal. Use the shardmanctl getconnstr command to get the cluster connection string.

C.1.14. How is balancing between cluster nodes implemented?

There is no built-in balancing solution at the moment. But you can organize balancing at the application level, for example, see *JDBC driver options* (`loadBalanceHosts`). For libpq, this functionality will be implemented in PostgreSQL 16 release.

C.1.15. Is mass data loading supported in Shardman?

Yes, this functionality is built in the management utility, see shardmanctl load.

C.2. Databases

C.2.1. Is it possible to create multiple databases in a Shardman cluster?

For now, sharding works only for a database named `postgres` (default), creating other databases is in development.

C.3. Tables

C.3.1. What kind of tables are there in Shardman?

In addition to local table types Shardman supports distributed tables: global and sharded.

C.3.2. What are global tables?

A global table in Shardman is a table that has the same schema and contents on all shards in the cluster. Global tables are created as follows:

```
CREATE TABLE g(id bigint PRIMARY KEY, t text) WITH(global);
```

A copy of such a table is created on each shard. Data replication of global tables is based on triggers. When data is inserted into such a table on any node of the cluster, data replication to other nodes occurs. When creating a global table, it is necessary to specify non deferrable primary key.

C.3.3. What are global tables suitable for?

Global tables are suitable for directories and other relatively small and infrequently modified tables. Global tables are NOT suitable for storing large amounts of data and for intensive `INSERT`/`UPDATE`/`DELETE` workload, especially with highly competitive access (storefronts, queues, etc.)

C.3.4. What are sharded tables?

Sharded tables are tables whose parts are hosted on different shards. Each shard stores its own piece of data from such a table. A sharded table can be created as follows:

```
CREATE TABLE ... WITH(distributed_by = 'column_name', num_parts =
 number_of_partitions);
```

Where:

`distributed_by` — table field being the sharding key,

`num_parts` — (`default = 24`) number of partitions into which the table is initially divided.

These parts are then distributed to shards.

C.3.5. Which partitioning parameters are optimal when creating a sharded table?

The number of partitions should be not less than the number of shards including the shards that can be added later. In general it may be a number with quite a few divisors like 12 or 24, so you can evenly divide the table into 2, 3, 4 or 6 shards. Large amount of partitions adds overhead on planning and execution, so it is preferable to keep it reasonable.

C.3.6. What are colocated tables?

Colocated tables are used when a table is often joined with another sharded table (usually by foreign keys) and therefore it is better to physically place their parts on the same shards.

C.3.7. How to create a colocated table?

```
CREATE TABLE ... WITH(distributed_by = 'column_name', num_parts = number_of_partitions,
 colocate_with = 'distributed_table');
```

Here:

`distributed_by = 'column_name'` — the name of the sharding key as it is called in the colocated table (not the colocating table) being created,

`colocate_with = 'distributed_table'` — the name of the table with which you want to colocate parts of the colocated table.

C.3.8. What are local tables?

A local table is a table only hosted on the shard where it was created.

C.3.9. Are foreign keys supported in Shardman?

Foreign keys are allowed in Shardman but with some limitations:

- On global tables, both from sharded tables and from other global tables
- Between sharded colocated tables.

Foreign keys are NOT allowed:

- From global to sharded tables
- Between sharded tables if they are not colocated.

C.4. Sequences

C.4.1. Are global sequences supported in Shardman?

Yes, they are supported. However, there are specifics of their work that should be taken into account. Under the hood of global sequences, there are regular sequences on each shard, and they are allocated by sequential blocks (of 65536 numbers by default). When numbers are passed to the sequence, the local sequential block is given to the local sequential block on the shard. I.e., numbers

from the global sequences are unique, but there is no strict monotony (unlike in PostgreSQL). Well, there may be "holes" in the values given by the sequencer.

C.4.2. How to create a global sequence?

```
CREATE SEQUENCE ... WITH (GLOBAL);
```

The `nextval` function can be used to fetch the next value of a sequence:

```
SELECT nextval('acl_id_seq'::regclass);
```

Data types `bigserial`, `smallserial`, and `serial` (for automatic creation of sequences and output of default values from it) are implemented and work for both sharded and global tables. It is recommended to use `bigserial` unless there are special requirements.

C.5. User Management

C.5.1. Does Shardman support global user roles?

Yes, global user roles are supported.

C.5.2. How do I create a global user in Shardman?

```
postgres=# create role my_user with login password 'my_user123' in role global;
CREATE ROLE
```

C.5.3. How do I grant permissions to a global user?

The following commands can be run on one shard and will be cascaded to the other shards automatically:

```
GRANT\REVOKE
CREATE ROLE ... IN ROLE GLOBAL / ALTER ROLE (for global role)
```

```
postgres=# grant CONNECT on DATABASE postgres TO my_user;
GRANT ROLE
postgres=# grant pg_monitor TO my_user;
GRANT ROLE
postgres=# \du
                                 List of roles
 Role name |                         Attributes                         |  Member of
-----------+------------------------------------------------------------+--------------
 my_user   |                                                            | {pg_monitor}
 postgres  | Superuser, Create role, Create DB, Replication, Bypass RLS | {}
 repluser  | Replication                                                | {}
```

The list of cascadable commands is being finalized and will be changed in future versions of Shardman.

C.6. Useful Functions and Tables

C.6.1. How do I see which tables and sequences are distributed?

Here are lists of some useful internal Shardman tables.

- `shardman.sequence` — the list of global sequences
- `shardman.sharded_tables` — the list of sharded tables

- `shardman.global_tables` — the list of global tables

For example:

```
postgres=# select * from shardman.sequence;
 seqns  | seqname | seqmin |       seqmax        | seqblk
--------+---------+--------+---------------------+--------
 public | s       |      1 | 9223372036854775807 |  65536
(1 row)

postgres=# select * from shardman.sharded_tables;
  rel  | nparts | colocated_with | relname | nspname
-------+--------+----------------+---------+---------
 16648 |     20 |                | d       | public
 16741 |     20 |          16648 | c       | public
(2 rows)

postgres=# select * from shardman.global_tables;
 relid | main_rgid | relname | nspname
-------+-----------+---------+---------
 16636 |           | g       | public
(1 row)
```

C.6.2. How do I execute some SQL command on all nodes in the cluster?

To do this, use the `shardman.broadcast_all_sql` function and shardmanctl `forall` option. For example:

```
postgres$ shardmanctl forall --sql "select name,setting from pg_settings where name =
 'max_connections'"
SQL: select name,setting from pg_settings where name = 'max_connections'
Node 1 says:
[max_connections 90]
Node 2 says:
[max_connections 90]
```

C.6.3. How do I get Shardman configuration parameters on a selected node?

The standard `SHOW` command can be used to obtain Shardman-specific parameters that are listed in the `section`.

Besides, you can use shardmanctl `config get` command to obtain cluster configuration from etcd. You can view the parameters, but it is better to customize them after consulting with Posgres Pro engineers.

C.6.4. How do I update Shardman configuration parameters?

You can use shardmanctl `config update` functionality, see an example.

C.7. Disaster Recovery Cluster Requirements

The underlying functionality is under development. For the production usage contact Support.

C.7.1. Terms and Abbreviations

DB — Database.

DBMS — Database management system.

DC — Data center.

MDC — Main data center.

BDC — Backup data center.

HaC — Hight availability cluster.

DRC Disaster — recovery cluster.

C.7.2. High-level Description of the DRC

MDC hosts the main cluster shards and the etcd cluster. Shards are high-availability clusters that consist of two nodes with Postgres Pro DBMS instances, one as a primary node, one as a synchronous standby. Every shard has the shardmand service running that checks the Postgres Pro DBMS instances and exchanges the information with the etcd cluster, thus providing Shardman clustering. The etcd cluster consists of three nodes that ensures a quorum.

To ensure disaster recovery, the customer's BDC must host an identical cluster with the identical configuration and set of components. By default, the standby Shardman cluster nodes are disabled. A continuous logs delivery from MDC to BDC is asynchronous and uses the physical replication mechanisms. It is based on the standart Shardman utility pg_receivewal. It writes WALs to the default instance directory `$PGDATA/pg_wal`. This utility is managed by the cluster software. When a syncpoint is detected under the standby etcd cluster, a standby Shardman cluster nodes are started by shardmand. It results in WAL update till LSN received from the syncpoin. In different DCs the etcd clusters are isolated, therefore, to distribute the syncpoint updated information, a script is periodically run from the MDC to BDC etcd.

C.7.3. Replication Topology

Streaming physical replication is provided:

- From the Postgres Pro DBMS shard nodes to MDC (synchronous)
- From the Postgres Pro DBMS shard nodes to BDC (synchronous)
- From the Postgres Pro DBMS shard nodes to DC (asynchronous)

C.7.4. Hardware and Network Requirements

MDC and BDC hardware must have identical system resources and configuration for all the DRC components.

DCs must be connected with fiber optic network with the capacity not less than 20 Gbit per second. A backup channel is also required.

C.7.5. Replication Mechanisms

To provide high-availability and disaster recovery clusters Shardman uses the Postgres Pro built-in streaming physical replication mechanism, for BDC it is also asynchronous.

Automatic recovery of a high-availability Shardman cluster is ensured by the cluster software.

DRC cluster recovery is only provided in manual semi-automatic mode.

C.7.6. Monitoring and Management

Shardman cluster monitoring and management is provided within one DC with the shardmanctl utility.

C.7.7. Security

C.7.7.1. Encrypting Data Across A Network (TLS/SSL)

A secure channel between DCs is required.

C.7.7.2. Inter-nodes Authentication and Authorization

Inter-nodes authentication and authorization is ensured by the built-in Postgres Pro DMBS tools.

C.7.7.3. Protection from Unauthorized Access to Standby Servers

Protection from unauthorized access to standby servers is provided by the operation system and network tools.

C.7.8. QA and Rollback

It is recommended to do periodical switchovers.

C.7.8.1. Data Integrity Check After Failover

Data integrity check after a failover is provided by the backup utility `shardmanctl probackup`.

C.7.8.2. Switchover to BDC

Should the MDC fail, the administrator must make sure it is, indeed, unavailable and initiate the promote of the standby nodes. The standby cluster upgrades its state from `standby` to `master`. This process is only initiated and managed by the shardmanctl utility, no other procedures required.

C.7.8.3. MDC Recovery

To recover remote nodes to the MDC, create a backup of the main cluster and restore it on these nodes. The backup can be either created as a cold backup or with the pg_probackup repository. Both options require a backup recovery to the MDC. Once the DB is restored from the backup, run pg_receivewal that connects to a special primary or standby shard replication slot in the BDC, then it receives WAL segments asynchronously and writes to the `$PGDATA/pg_wal` directory of the main node.

In the BDC cluster, a script creates a syncpoint each specified period of time. It is written to the BDC etcd and sent to the MDC etcd. Once a syncpoint is in etcd, the MDC stanby cluster nodes check if a WAL with this record is received. If it is received by all the MDC standby cluster nodes, the cluster software initiates the DBMS server startup in the recovery with WAL mode until the syncpoint. Once the syncpoint is reached, no more WALs are applied. If all nodes successfully applied the WAL records, the DBMS server is stopped, followed by another cycle of receiving WAL, syncpoint check and recovery mode.

C.7.8.4. Switching Back to MDC

To switch back to the MDC, create and transfer a cluster backup from BDS to MDC, run the nodes in the standby node mode. Once the lacking WALs are received, the BDC cluster nodes are stopped, and the MDC cluster nodes are promoted.

C.7.9. Backup in Geografically Distributed System

Within the GDS (Geografically distributed systemt), BDC cluster must have the storage for the backups identical to one of the MDC. Regular syncing between the main and backup storage is also required.

C.7.9.1. Storing Backups in Geographically Distributed Storages

The period of time the backups are stored is defined by the backup policy.

C.7.10. Documentation and Regulations

For more information on disaster failover and normal switchover to MDC instructions, contact Prostres Pro Support.

Index

T