Citus Documentation

Release 5.0.0

Citus Data

Dec 20, 2018

About Citus

Welcome to the documentation for Citus 5.0! Citus horizontally scales PostgreSQL across commodity servers using sharding and replication. Its query engine parallelizes incoming SQL queries across these servers to enable real-time responses on large datasets.

The documentation explains how you can install Citus and then provides instructions to design, build, query, and maintain your Citus cluster. It also includes a Reference section which provides quick information on several topics.

CHAPTER 1

What is Citus?

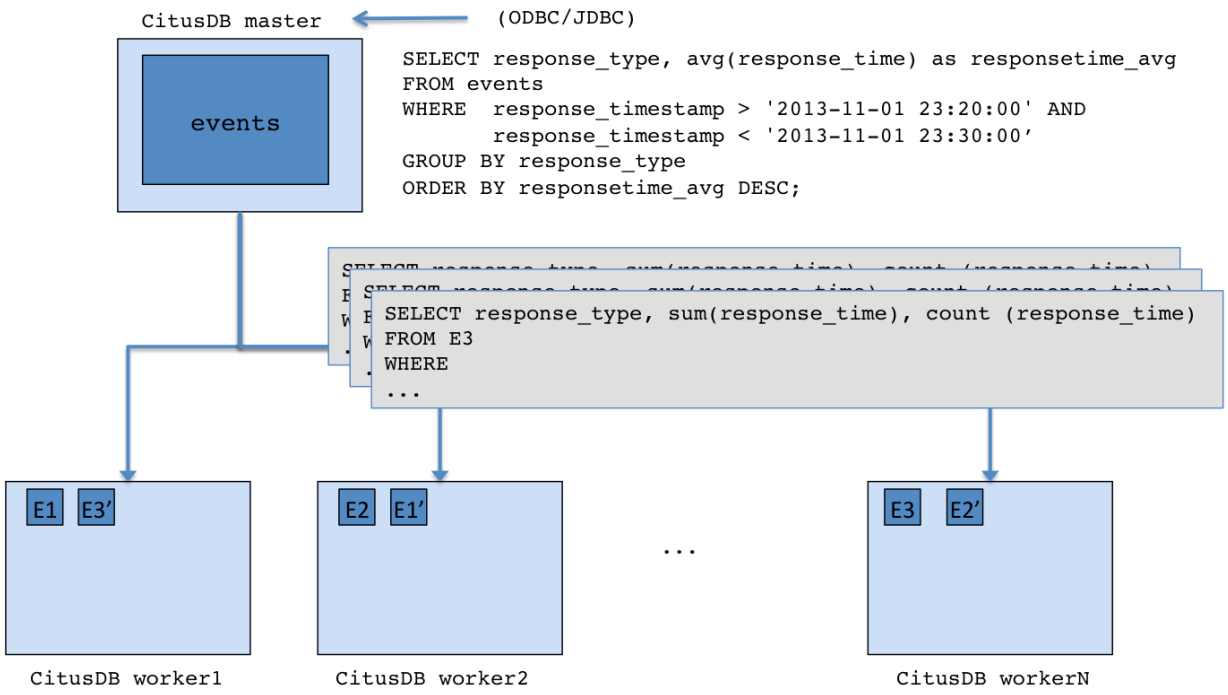
Citus horizontally scales PostgreSQL across commodity servers using sharding and replication. Its query engine parallelizes incoming SQL queries across these servers to enable real-time responses on large datasets.

Citus extends the underlying database rather than forking it, which gives developers and enterprises the power and familiarity of a traditional relational database. As an extension, Citus supports new PostgreSQL releases, allowing users to benefit from new features while maintaining compatibility with existing PostgreSQL tools.

CHAPTER 2

Architecture

At a high level, Citus distributes the data across a cluster of commodity servers. Incoming SQL queries are then parallel processed across these servers.



In the sections below, we briefly explain the concepts relating to Citus's architecture.

2.1 Master / Worker Nodes

You first choose one of the PostgreSQL instances in the cluster as the Citus master. You then add the DNS names of worker PostgreSQL instances (Citus workers) to a membership file on the master. From that point on, you interact with the master through standard PostgreSQL interfaces for data loading and querying. All the data is distributed across the workers. The master only stores metadata about the shards.

2.2 Logical Sharding

Citus utilizes a modular block architecture which is similar to Hadoop Distributed File System blocks but uses PostgreSQL tables on the workers instead of files. Each of these tables is a horizontal partition or a logical "shard". The Citus master then maintains metadata tables which track all the workers and the locations of the shards on the workers.

Each shard is replicated on at least two of the workers (Users can configure this to a higher value). As a result, the loss of a single machine does not impact data availability. The Citus logical sharding architecture also allows new workers to be added at any time to increase the capacity and processing power of the cluster.

2.3 Metadata Tables

The Citus master maintains metadata tables to track all the workers and the locations of the database shards on them. These tables also maintain statistics like size and min/max values about the shards which help Citus's distributed query planner to optimize the incoming queries. The metadata tables are small (typically a few MBs in size) and can be replicated and quickly restored if the master ever experiences a failure.

You can view the metadata by running the following queries on the Citus master.

```
SELECT * from pg_dist_partition;
 logicalrelid | partmethod |                                                        ␣
↪partkey
--------------+------------+------------------------------------------------------------------
↪--------------------------------------------------------------
       488843 | r          | {VAR :varno 1 :varattno 4 :vartype 20 :vartypmod -1␣
↪:varcollid 0 :varlevelsup 0 :varnoold 1 :varoattno 4 :location 232}
(1 row)

SELECT * from pg_dist_shard;
 logicalrelid | shardid | shardstorage | shardalias | shardminvalue | shardmaxvalue
--------------+---------+--------------+------------+---------------+---------------
       488843 |  102065 | t            |            | 27            | 14995004
       488843 |  102066 | t            |            | 15001035      | 25269705
       488843 |  102067 | t            |            | 25273785      | 28570113
       488843 |  102068 | t            |            | 28570150      | 28678869
(4 rows)

SELECT * from pg_dist_shard_placement;
 shardid | shardstate | shardlength | nodename  | nodeport
---------+------------+-------------+-----------+----------
  102065 |          1 |     7307264 | localhost |   9701
  102065 |          1 |     7307264 | localhost |   9700
  102066 |          1 |     5890048 | localhost |   9700
  102066 |          1 |     5890048 | localhost |   9701
  102067 |          1 |     5242880 | localhost |   9701
  102067 |          1 |     5242880 | localhost |   9700
```

(continues on next page)

(continued from previous page)

```
  102068 |          1 |     3923968 | localhost |   9700
  102068 |          1 |     3923968 | localhost |   9701
(8 rows)
```

To learn more about the metadata tables and their schema, please visit the *Metadata Tables Reference* section of our documentation.

2.4 Query Processing

When the user issues a query, the Citus master partitions it into smaller query fragments where each query fragment can be run independently on a worker shard. This allows Citus to distribute each query across the cluster, utilizing the processing power of all of the involved nodes and also of individual cores on each node. The master then assigns the query fragments to workers, oversees their execution, merges their results, and returns the final result to the user. To ensure that all queries are executed in a scalable manner, the master also applies optimizations that minimize the amount of data transferred across the network.

2.5 Failure Handling

Citus can easily tolerate worker failures because of its logical sharding-based architecture. If a worker fails mid-query, Citus completes the query by re-routing the failed portions of the query to other workers which have a copy of the shard. If a worker is permanently down, users can easily rebalance the shards onto other workers to maintain the same level of availability.

CHAPTER 3

Start Demo Cluster

To do the tutorials you'll need a single-machine Citus cluster with a master and two worker PostgreSQL instances. Follow these instructions to create a temporary installation which is quick to try and easy to remove.

1. Download the package

Download and unzip it into a directory of your choosing. Then, enter that directory:

```
cd citus-tutorial
```

2. Initialize the cluster

Citus has two kinds of components, the master and the workers. The master coordinates queries and maintains metadata on where in the cluster each row of data is. The workers hold your data and respond to queries.

Let's create directories for those nodes to store their data in:

```
bin/initdb -D data/master
bin/initdb -D data/worker
```

The above commands will give you warnings about trust authentication. Those will become important when you're setting up a production instance of Citus but for now you can safely ignore them.

The master needs to know where it can find the worker. To tell it you can run:

```
echo "localhost 9701" >> data/master/pg_worker_list.conf
```

We assume that ports 9700 (for the master) and 9701 (for the worker) are available on your machine. Feel free to use different ports if they are in use.

Citus is a Postgres extension. To tell Postgres to use this extension, you'll need to add it to a configuration variable called `shared_preload_libraries`:

```
echo "shared_preload_libraries = 'citus'" >> data/master/postgresql.conf
echo "shared_preload_libraries = 'citus'" >> data/worker/postgresql.conf
```

3. Start the master and worker

Let's start the databases:

```
bin/pg_ctl -D data/master -o "-p 9700" -l master_logfile start
bin/pg_ctl -D data/worker -o "-p 9701" -l worker_logfile start
```

And initialize them:

```
bin/createdb -p 9700 $(whoami)
bin/createdb -p 9701 $(whoami)
```

Above you added Citus to `shared_preload_libraries`. That lets it hook into some deep parts of Postgres, swapping out the query planner and executor. Here, we load the user-facing side of Citus (such as the functions you'll soon call):

```
bin/psql -p 9700 -c "CREATE EXTENSION citus;"
bin/psql -p 9701 -c "CREATE EXTENSION citus;"
```

4. Ready for the tutorials!

Your cluster is running and eagerly awaiting data. Proceed to the *Real Time Aggregation* tutorial to begin learning how to use Citus.

CHAPTER 4

Real Time Aggregation

In this tutorial we'll look at a stream of live wikipedia edits. Wikimedia is kind enough to publish all changes happening across all their sites in real time; this can be a lot of events!

This tutorial assumes you've set up a *single-machine demo cluster*. Our first task is to get the cluster ready to accept a stream of wikipedia edits. First, open a psql shell to the master:

```
cd citus-tutorial
bin/psql postgresql://:9700
```

This will open a new prompt. You can leave psql at any time by hitting `Ctrl` + `D`.

Create a table for the wikipedia data to be entered into:

```
CREATE TABLE wikipedia_edits (
  time TIMESTAMP WITH TIME ZONE, -- When the edit was made

  editor TEXT, -- The editor who made the change
  bot BOOLEAN, -- Whether the editor is a bot

  wiki TEXT, --  Which wiki was edited
  namespace TEXT, -- Which namespace the page is a part of
  title TEXT, -- The name of the page

  comment TEXT, -- The message they described the change with
  minor BOOLEAN, -- Whether this was a minor edit (self-reported)
  type TEXT, -- "new" if this created the page, "edit" otherwise

  old_length INT, -- How long the page used to be
  new_length INT -- How long the page is as of this edit
);
```

The `wikipedia_edits` table is currently a regular Postgres table. Its growth is limited by how much data the master can hold and queries against it don't benefit from any parallelism.

Tell Citus that it should be a distributed table:

```
SELECT master_create_distributed_table(
  'wikipedia_edits', 'time', 'append'
);
```

This says to append distribute the `wikipedia_edits` table using the `time` column. The table will be stored as a collection of shards, each responsible for a range of `time` values. The page on *Append Distribution* goes into more detail.

Each shard can be on a different worker, letting the table grow to sizes too big for any one node to handle. Queries against this table run across all shards in parallel. Even on a single machine, this can have some significant performance benefits!

By default, Citus will replicate each shard across multiple workers. Since we only have one worker, we have to tell Citus that not replicating is okay:

```
SET citus.shard_replication_factor = 1;
```

If we didn't do the above, when we went to create a shard Citus would give us an error rather than accepting data it can't backup.

Now we create a shard for the data to be inserted into:

```
SELECT master_create_empty_shard('wikipedia_edits');
```

Citus is eagerly awaiting data, let's give it some! **Open a separate terminal** and run the data ingest script we've made for you.

```
# - in a new terminal -

cd citus-tutorial
scripts/insert-live-wikipedia-edits postgresql://:9700
```

This should continue running and adding edits, let's run some queries on them! If you run any of these queries multiple times you'll see the results update. Data is available to be queried in Citus as soon as it is ingested. Returning to our psql session on the master node we can ask who the most prolific editors are:

```
-- back in the original (psql) terminal

SELECT count(1) AS edits, editor
FROM wikipedia_edits
GROUP BY 2 ORDER BY 1 DESC LIMIT 20;
```

This is likely to be dominated by bots, so we can look at just the sources which represent actual users:

```
SELECT count(1) AS edits, editor
FROM wikipedia_edits WHERE bot IS false
GROUP BY 2 ORDER BY 1 DESC LIMIT 20;
```

Unfortunately, 'bot' is a user-settable flag which many bots forget to send, so this list is usually also dominated by bots.

Another user-settable flag is "minor", which users can hit to indicate they've made a small change which doesn't need to be reviewed as carefully. Let's see if they're actually following instructions:

```
SELECT
  avg(
    CASE WHEN minor THEN abs(new_length - old_length) END
```

(continues on next page)

(continued from previous page)

```
  ) AS average_minor_edit_size,
  avg(
    CASE WHEN NOT minor THEN abs(new_length - old_length) END
  ) AS average_normal_edit_size
FROM wikipedia_edits
WHERE old_length IS NOT NULL AND new_length IS NOT NULL;
```

Or how about combining the two? What are the top contributors, and how big are their edits?

```
SELECT
  COUNT(1) AS total_edits,
  editor,
  avg(abs(new_length - old_length)) AS average_edit_size
FROM wikipedia_edits
WHERE new_length IS NOT NULL AND old_length IS NOT NULL
GROUP BY 2 ORDER BY 1 DESC LIMIT 20;
```

That's all for now. To learn more about Citus continue to the *next tutorial*, or, if you're done with the cluster, run this to stop the worker and master:

```
bin/pg_ctl -D data/master stop
bin/pg_ctl -D data/worker stop
```

CHAPTER 5

Updatable User Data

In this tutorial we'll continue looking at wikipedia edits. The previous tutorial ingested a stream of all live edits happening across wikimedia. We'll continue looking at that stream but store it in a different way.

This tutorial assumes you've set up a *single-machine demo cluster*. Once your cluster is running, open a prompt to the master instance:

```
cd citus-tutorial
bin/psql postgresql://:9700
```

This will open a new prompt. You can leave it at any time by hitting `Ctrl + D`.

This time we're going to make two tables.

```
CREATE TABLE wikipedia_editors (
  editor TEXT UNIQUE, -- The name of the editor
  bot BOOLEAN, -- Whether they are a bot (self-reported)

  edit_count INT, -- How many edits they've made
  added_chars INT, -- How many characters they've added
  removed_chars INT, -- How many characters they've removed

  first_seen TIMESTAMPTZ, -- The time we first saw them edit
  last_seen TIMESTAMPTZ -- The time we last saw them edit
);

CREATE TABLE wikipedia_changes (
  editor TEXT, -- The editor who made the change
  time TIMESTAMP WITH TIME ZONE, -- When they made it

  wiki TEXT, -- Which wiki they edited
  title TEXT, -- The name of the page they edited

  comment TEXT, -- The message they described the change with
  minor BOOLEAN, -- Whether this was a minor edit (self-reported)
  type TEXT, -- "new" if this created the page, "edit" otherwise
```

(continues on next page)

(continued from previous page)

```
  old_length INT, -- how long the page used to be
  new_length INT -- how long the page is as of this edit
);
```

These tables are regular Postgres tables. We need to tell Citus that they should be distributed tables, stored across the cluster.

```
SELECT master_create_distributed_table(
  'wikipedia_changes', 'editor', 'hash'
);
SELECT master_create_distributed_table(
  'wikipedia_editors', 'editor', 'hash'
);
```

These say to store each table as a collection of shards, each responsible for holding a different subset of the data. The shard a particular row belongs in will be computed by hashing the `editor` column. The page on *Hash Distribution* goes into more detail.

Finally, create the shards:

```
SELECT master_create_worker_shards('wikipedia_editors', 16, 1);
SELECT master_create_worker_shards('wikipedia_changes', 16, 1);
```

This tells Citus to create 16 shards for each table, and save 1 replica of each. You can ask Citus to store multiple copies of each shard, which allows it to recover from worker failures without losing data or dropping queries. However, in this example cluster we only have 1 worker, so Citus would error out if we asked it to store any more than 1 replica.

Now we're ready to accept some data! **Open a separate terminal** and run the data ingest script we've made for you in this new terminal:

```
# - in a new terminal -

cd citus-tutorial
scripts/collect-wikipedia-user-data postgresql://:9700
```

This should keep running and aggregating data on the users who are editting right now. Let's run some queries! If you run any of these queries multiple times you'll see the results update as more data is ingested. Returning to our existing psql terminal we can ask some simple questions, such as finding edits which were made by bots:

```
-- back in the original (psql) terminal

SELECT comment FROM wikipedia_changes c, wikipedia_editors e
WHERE c.editor = e.editor AND e.bot IS true LIMIT 10;
```

Above, when we created our two tables, we partitioned them along the same column and created an equal number of shards for each. Doing this means that all data for each editor is kept on the same machine, or, colocated.

How many pages have been created by bots? By users?

```
SELECT bot, count(*) as pages_created
FROM wikipedia_changes c,
     wikipedia_editors e
WHERE c.editor = e.editor
      AND type = 'new'
GROUP BY bot;
```

Citus can also perform joins where the rows to be joined are not stored on the same machine. But, joins involving colocated rows usually run *faster* than their non-distributed versions, because they can run across all machines and shards in parallel.

A surprising amount of the content in wikipedia is written by users who stop by to make just one or two edits and don't even bother to create an account. Their username is just their ip address, which will look something like '95.180.5.193' or '2607:FB90:25C8:8785:0:42:36E9:7E01'.

We can (using a very rough regex), find their edits:

```
SELECT editor SIMILAR TO '[0-9.A-F:]+' AS ip_editor,
       COUNT(1) AS edit_count,
       SUM(added_chars) AS added_chars
FROM wikipedia_editors WHERE bot is false
GROUP BY ip_editor;
```

Usually, around a fifth of all non-bot edits are made from unregistered editors. The real percentage is a lot higher, since "bot" is a user-settable flag which many bots neglect to set.

That's all for now. This script showed a data layout which many Citus users choose. One table stored a stream of events while another table stored some aggregations of those events and made queries over them quick.

We hope you enjoyed working through our tutorials. Once you're ready to stop the cluster run these commands:

```
bin/pg_ctl -D data/master stop
bin/pg_ctl -D data/worker stop
```

CHAPTER 6

Requirements

Citus works with modern 64-bit Linux and most Unix based operating systems. Citus 5.0 offically supports PostgreSQL 9.5 and later versions. The extension will also work against PostgreSQL 9.4 but versions older than 9.4 are not supported.

Before setting up a Citus cluster, you should ensure that the network and firewall settings are configured to allow:

- The database clients (eg. psql, JDBC / OBDC drivers) to connect to the master.
- All the nodes in the cluster to connect to each other over TCP without any interactive authentication.

CHAPTER 7

Single-Machine Cluster

If you are a developer looking to try Citus out on your machine, the guides below will help you get started quickly.

7.1 OS X

This section will show you how to create a Citus cluster on a single OS X machine.

1. Install PostgreSQL 9.5 and the Citus extension

Use our Homebrew package to extend PostgreSQL with Citus.

```
brew install citus
```

2. Initialize the Cluster

Citus has two kinds of components, the master and the workers. The master coordinates queries and maintains metadata on where in the cluster each row of data is. The workers hold your data and respond to queries.

Let's create directories for those nodes to store their data. For convenience we suggest making subdirectories in your home folder, but feel free to choose another path.

```
cd ~
mkdir -p citus/master citus/worker1 citus/worker2

# create three normal postgres instances
initdb -D citus/master
initdb -D citus/worker1
initdb -D citus/worker2
```

The master needs to know where it can find the workers. To tell it you can run:

```
echo "localhost 9701" >> citus/master/pg_worker_list.conf
echo "localhost 9702" >> citus/master/pg_worker_list.conf
```

We will configure the PostgreSQL instances to use ports 9700 (for the master) and 9701, 9702 (for the workers). We assume those ports are available on your machine. Feel free to use different ports if they are in use.

Citus is a Postgres extension, to tell Postgres to use this extension you'll need to add it to a configuration variable called `shared_preload_libraries`:

```
echo "shared_preload_libraries = 'citus'" >> citus/master/postgresql.conf
echo "shared_preload_libraries = 'citus'" >> citus/worker1/postgresql.conf
echo "shared_preload_libraries = 'citus'" >> citus/worker2/postgresql.conf
```

3. Start the master and workers

Let's start the databases:

```
pg_ctl -D citus/master -o "-p 9700" -l master_logfile start
pg_ctl -D citus/worker1 -o "-p 9701" -l worker1_logfile start
pg_ctl -D citus/worker2 -o "-p 9702" -l worker2_logfile start
```

And initialize them:

```
createdb -p 9700 $(whoami)
createdb -p 9701 $(whoami)
createdb -p 9702 $(whoami)
```

Above you added Citus to `shared_preload_libraries`. That lets it hook into some deep parts of Postgres, swapping out the query planner and executor. Here, we load the user-facing side of Citus (such as the functions you'll soon call):

```
psql -p 9700 -c "CREATE EXTENSION citus;"
psql -p 9701 -c "CREATE EXTENSION citus;"
psql -p 9702 -c "CREATE EXTENSION citus;"
```

4. Verify that installation has succeeded

To verify that the installation has succeeded we check that the master node has picked up the desired worker configuration. First start the psql shell on the master node:

```
psql -p 9700 -c "select * from master_get_active_worker_nodes();"
```

You should see a row for each worker node including the node name and port.

7.2 Fedora, CentOS, or Red Hat

This section describes the steps needed to set up a single-node Citus cluster on your own Linux machine from RPM packages.

1. Install PostgreSQL 9.5 and the Citus extension

Locate the PostgreSQL 9.5 YUM repository for your Linux distribution in this list. Copy its URL and add the repository:

```
# add repository from URL you previously copied
sudo yum install -y <repository-url>

# install Citus extension
sudo yum install -y citus_95
```

2. Initialize the Cluster

Citus has two kinds of components, the master and the workers. The master coordinates queries and maintains metadata on where in the cluster each row of data is. The workers hold your data and respond to queries.

Let's create directories for those nodes to store their data. For convenience in using PostgreSQL Unix domain socket connections we'll use the postgres user.

```
# this user has access to sockets in /var/run/postgresql
sudo su - postgres

# include path to postgres binaries
export PATH=$PATH:/usr/pgsql-9.5/bin

cd ~
mkdir -p citus/master citus/worker1 citus/worker2

# create three normal postgres instances
initdb -D citus/master
initdb -D citus/worker1
initdb -D citus/worker2
```

The master needs to know where it can find the workers. To tell it you can run:

```
echo "localhost 9701" >> citus/master/pg_worker_list.conf
echo "localhost 9702" >> citus/master/pg_worker_list.conf
```

We will configure the PostgreSQL instances to use ports 9700 (for the master) and 9701, 9702 (for the workers). We assume those ports are available on your machine. Feel free to use different ports if they are in use.

Citus is a Postgres extension, to tell Postgres to use this extension you'll need to add it to a configuration variable called `shared_preload_libraries`:

```
echo "shared_preload_libraries = 'citus'" >> citus/master/postgresql.conf
echo "shared_preload_libraries = 'citus'" >> citus/worker1/postgresql.conf
echo "shared_preload_libraries = 'citus'" >> citus/worker2/postgresql.conf
```

3. Start the master and workers

Let's start the databases:

```
pg_ctl -D citus/master -o "-p 9700" -l master_logfile start
pg_ctl -D citus/worker1 -o "-p 9701" -l worker1_logfile start
pg_ctl -D citus/worker2 -o "-p 9702" -l worker2_logfile start
```

Above you added Citus to `shared_preload_libraries`. That lets it hook into some deep parts of Postgres, swapping out the query planner and executor. Here, we load the user-facing side of Citus (such as the functions you'll soon call):

```
psql -p 9700 -c "CREATE EXTENSION citus;"
psql -p 9701 -c "CREATE EXTENSION citus;"
psql -p 9702 -c "CREATE EXTENSION citus;"
```

4. Verify that installation has succeeded

To verify that the installation has succeeded we check that the master node has picked up the desired worker configuration. First start the psql shell on the master node:

```
psql -p 9700 -c "select * from master_get_active_worker_nodes();"
```

You should see a row for each worker node including the node name and port.

7.3 Ubuntu or Debian

This section describes the steps needed to set up a single-node Citus cluster on your own Linux machine from deb packages.

1. Install PostgreSQL 9.5 and the Citus extension

```
# add postgresql-9.5-citus pgdg repository
echo "deb http://apt.postgresql.org/pub/repos/apt/ $(lsb_release -cs)-pgdg main" |␣
↪sudo tee -a /etc/apt/sources.list.d/pgdg.list
sudo apt-get install wget ca-certificates
wget --quiet --no-check-certificate -O - https://www.postgresql.org/media/keys/
↪ACCC4CF8.asc | sudo apt-key add -
sudo apt-get update

# install the server and initialize db
sudo apt-get -y install postgresql-9.5-citus
```

2. Initialize the Cluster

Citus has two kinds of components, the master and the workers. The master coordinates queries and maintains metadata on where in the cluster each row of data is. The workers hold your data and respond to queries.

Let's create directories for those nodes to store their data. For convenience in using PostgreSQL Unix domain socket connections we'll use the postgres user.

```
# this user has access to sockets in /var/run/postgresql
sudo su - postgres

# include path to postgres binaries
export PATH=$PATH:/usr/lib/postgresql/9.5/bin

cd ~
mkdir -p citus/master citus/worker1 citus/worker2

# create three normal postgres instances
initdb -D citus/master
initdb -D citus/worker1
initdb -D citus/worker2
```

The master needs to know where it can find the workers. To tell it you can run:

```
echo "localhost 9701" >> citus/master/pg_worker_list.conf
echo "localhost 9702" >> citus/master/pg_worker_list.conf
```

We will configure the PostgreSQL instances to use ports 9700 (for the master) and 9701, 9702 (for the workers). We assume those ports are available on your machine. Feel free to use different ports if they are in use.

Citus is a Postgres extension, to tell Postgres to use this extension you'll need to add it to a configuration variable called `shared_preload_libraries`:

```
echo "shared_preload_libraries = 'citus'" >> citus/master/postgresql.conf
echo "shared_preload_libraries = 'citus'" >> citus/worker1/postgresql.conf
echo "shared_preload_libraries = 'citus'" >> citus/worker2/postgresql.conf
```

3. Start the master and workers

Let's start the databases:

```
pg_ctl -D citus/master -o "-p 9700" -l master_logfile start
pg_ctl -D citus/worker1 -o "-p 9701" -l worker1_logfile start
pg_ctl -D citus/worker2 -o "-p 9702" -l worker2_logfile start
```

Above you added Citus to `shared_preload_libraries`. That lets it hook into some deep parts of Postgres, swapping out the query planner and executor. Here, we load the user-facing side of Citus (such as the functions you'll soon call):

```
psql -p 9700 -c "CREATE EXTENSION citus;"
psql -p 9701 -c "CREATE EXTENSION citus;"
psql -p 9702 -c "CREATE EXTENSION citus;"
```

4. Verify that installation has succeeded

To verify that the installation has succeeded we check that the master node has picked up the desired worker configuration. First start the psql shell on the master node:

```
psql -p 9700 -c "select * from master_get_active_worker_nodes();"
```

You should see a row for each worker node including the node name and port.

7.4 Docker

This section describes setting up a Citus cluster on a single machine using docker-compose.

1. Install docker and docker-compose

The easiest way to install docker-compose on Mac or Windows is to use the docker toolbox installer. Ubuntu users can follow this guide; other Linux distros have a similar procedure.

Note that Docker runs in a virtual machine on Mac, and to use it in your terminal you first must run

```
# for mac only
eval $(docker-machine env default)
```

This exports environment variables which tell the docker command-line tools how to connect to the virtual machine.

2. Start the Citus Cluster

Citus uses docker-compose to run and connect containers holding the database master node, workers, and a persistent data volume. To create a local cluster download our docker-compose configuration file and run it

```
wget https://raw.githubusercontent.com/citusdata/docker/master/docker-compose.yml
docker-compose -p citus up -d
```

The first time you start the cluster it builds its containers. Subsequent startups take a matter of seconds.

3. Verify that installation has succeeded

To verify that the installation has succeeded we check that the master node has picked up the desired worker configuration. First start the psql shell on the master node:

```
docker exec -it citus_master psql -U postgres
```

Then run this query:

```
select * from master_get_active_worker_nodes();
```

You should see a row for each worker node including the node name and port.

Alternatively try running a *demo cluster* which is temporary and easier to set up. It comes with tutorials to load a stream of live wikipedia edits into Citus and run interesting queries on them.

CHAPTER 8

Multi-Machine Cluster

The *Single-Machine Cluster* section has instructions on installing a Citus cluster on one machine. If you are looking to deploy Citus across multiple nodes, you can use the guide below.

8.1 Amazon Web Services

There are two approaches for running Citus on AWS. You can provision it manually using our CloudFormation template, or use Citus Cloud for automated provisioning, backup, and monitoring.

8.1.1 Managed Citus Cloud Deployment

Citus Cloud is a fully managed "Citus-as-a-Service" built on top of Amazon Web Services. It's an easy way to provision and monitor a high-availability cluster.

8.1.2 Manual CloudFormation Deployment

Alternately you can manage a Citus cluster manually on EC2 instances using CloudFormation. The CloudFormation template for Citus enables users to start a Citus cluster on AWS in just a few clicks, with also cstore_fdw extension for columnar storage is pre-installed. The template automates the installation and configuration process so that the users don't need to do any extra configuration steps while installing Citus.

Please ensure that you have an active AWS account and an Amazon EC2 key pair before proceeding with the next steps.

Introduction

CloudFormation lets you create a "stack" of AWS resources, such as EC2 instances and security groups, from a template defined in a JSON file. You can create multiple stacks from the same template without conflict, as long as they have unique stack names.

Below, we explain in detail the steps required to setup a multi-node Citus cluster on AWS.

1. Start a Citus cluster

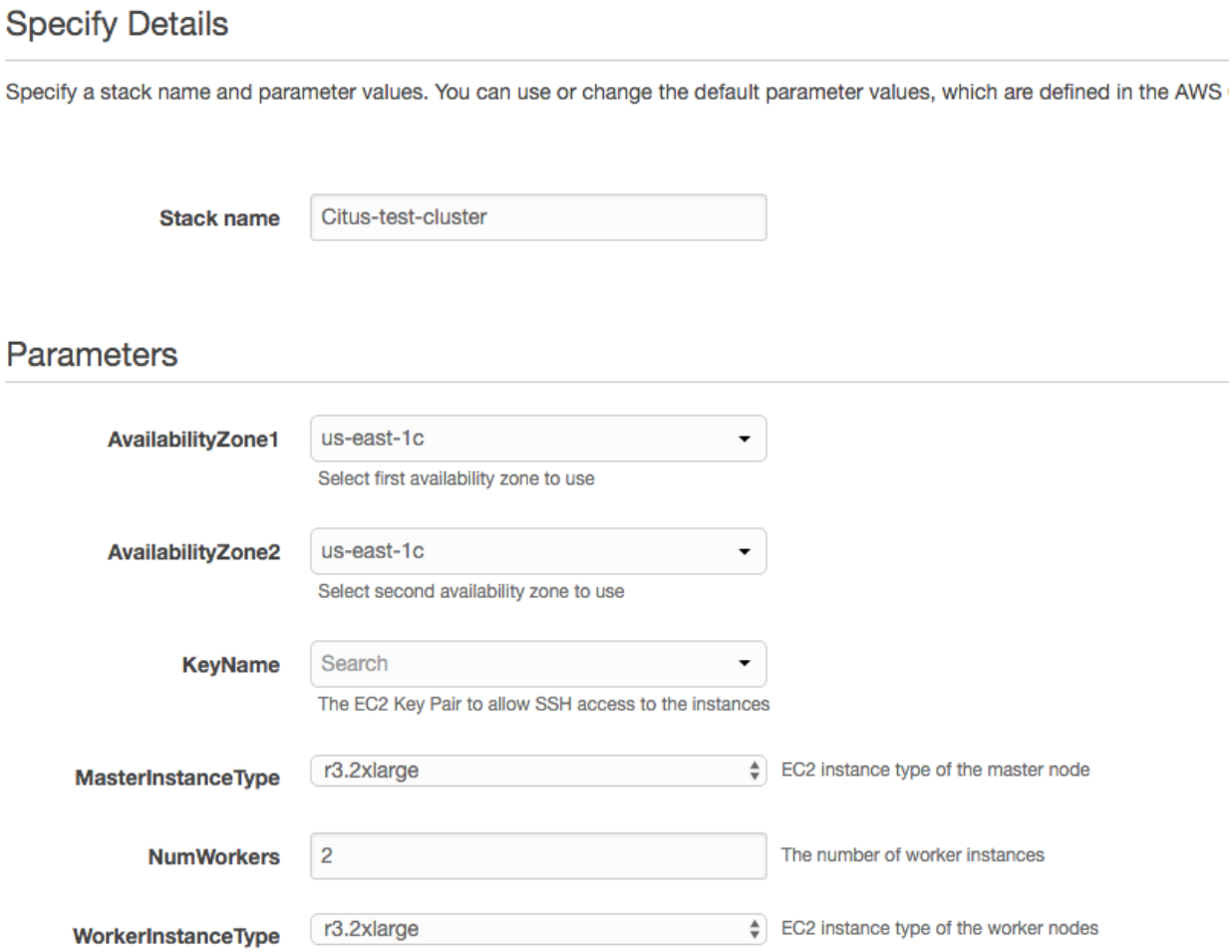
Note: You might need to login to AWS at this step if you aren't already logged in.

2. Select Citus template

You will see select template screen. Citus template is already selected, just click Next.

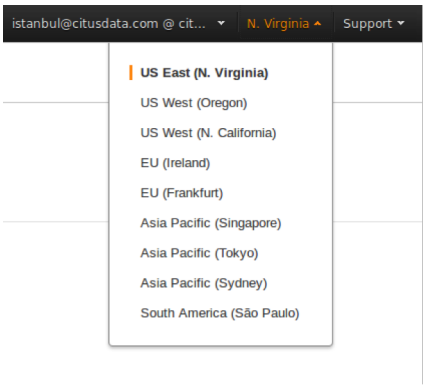
3. Fill the form with details about your cluster

In the form, pick a unique name for your stack. You can customize your cluster setup by setting availability zones, number of workers and the instance types. You also need to fill in the AWS keypair which you will use to access the cluster.



Note: Please ensure that you choose unique names for all your clusters. Otherwise, the cluster creation may fail with the error "Template_name template already created".

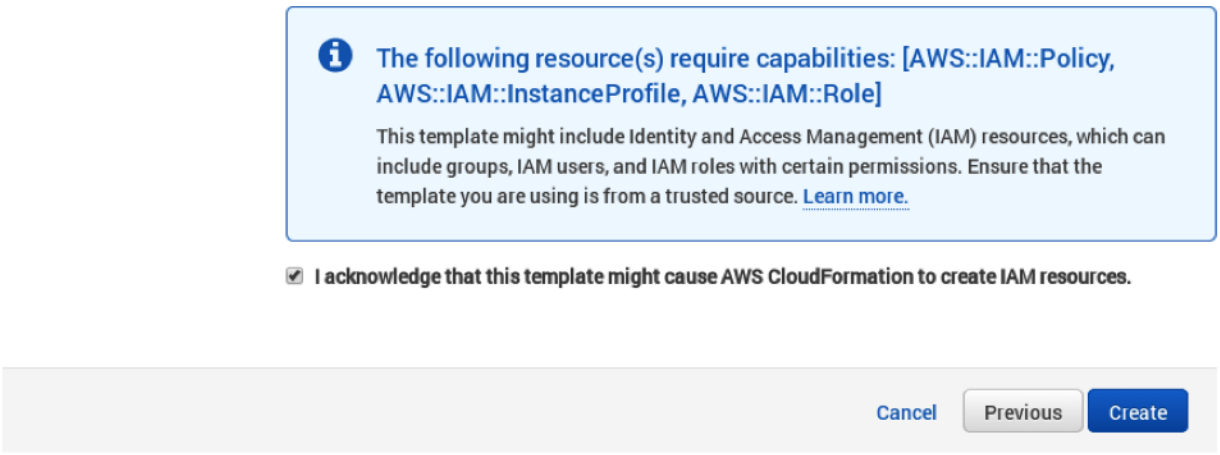
Note: If you want to launch a cluster in a region other than US East, you can update the region in the top-right corner of the AWS console as shown below.



4. Acknowledge IAM capabilities

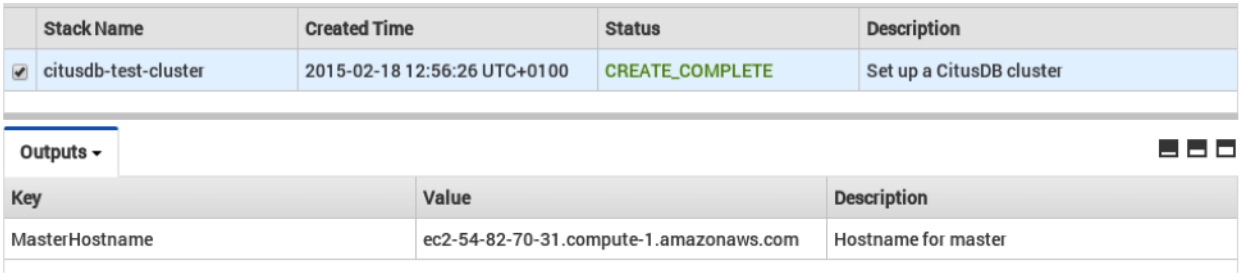
The next screen displays Tags and a few advanced options. For simplicity, we use the default options and click Next.

Finally, you need to acknowledge IAM capabilities, which give the master node limited access to the EC2 APIs to obtain the list of worker IPs. Your AWS account needs to have IAM access to perform this step. After ticking the checkbox, you can click on Create.



5. Cluster launching

After the above steps, you will be redirected to the CloudFormation console. You can click the refresh button on the top-right to view your stack. In about 10 minutes, stack creation will complete and the hostname of the master node will appear in the Outputs tab.



Note: Sometimes, you might not see the outputs tab on your screen by default. In that case, you should click on "restore" from the menu on the bottom right of your screen.

Troubleshooting:

You can use the cloudformation console shown above to monitor your cluster.

If something goes wrong during set-up, the stack will be rolled back but not deleted. In that case, you can either use a different stack name or delete the old stack before creating a new one with the same name.

6. Login to the cluster

Once the cluster creation completes, you can immediately connect to the master node using SSH with username ec2-user and the keypair you filled in. For example:

```
ssh -i your-keypair.pem ec2-user@ec2-54-82-70-31.compute-1.amazonaws.com
```

7. Ready to use the cluster

At this step, you have completed the installation process and are ready to use the Citus cluster. You can now login to the master node and start executing commands. The command below, when run in the psql shell, should output the worker nodes mentioned in the pg_worker_list.conf.

```
/usr/pgsql-9.5/bin/psql -h localhost -d postgres
select * from master_get_active_worker_nodes();
```

8. Cluster notes

The template automatically tunes the system configuration for Citus and sets up RAID on the SSD drives where appropriate, making it a great starting point even for production systems.

The database and its configuration files are stored in /data/base. So, to change any configuration parameters, you need to update the postgresql.conf file at /data/base/postgresql.conf.

Similarly to restart the database, you can use the command:

```
/usr/pgsql-9.5/bin/pg_ctl -D /data/base -l logfile restart
```

Note: You typically want to avoid making changes to resources created by CloudFormation, such as terminating EC2 instances. To shut the cluster down, you can simply delete the stack in the CloudFormation console.

8.2 Multi-node setup on Fedora, CentOS, or Red Hat

This section describes the steps needed to set up a multi-node Citus cluster on your own Linux machines from RPM packages.

8.2.1 Steps to be executed on all nodes

1. Add PGDG repository

Locate the PostgreSQL 9.5 YUM repository for your Linux distribution in this list. Copy its URL and add the repository:

```
# add repository from URL you previously copied
sudo yum install -y <repository-url>

# e.g on RHEL 7:
# sudo yum install -y https://download.postgresql.org/pub/repos/yum/9.5/redhat/rhel-7-
↪x86_64/pgdg-redhat95-9.5-2.noarch.rpm
```

2. Install PostgreSQL + Citus and initialize a database

```
# install PostgreSQL with Citus extension
sudo yum install -y citus_95
# initialize system database (using RHEL 6 vs 7 method as necessary)
sudo service postgresql-9.5 initdb || sudo /usr/pgsql-9.5/bin/postgresql95-setup␣
↪initdb
# preload citus extension
echo "shared_preload_libraries = 'citus'" | sudo tee -a /var/lib/pgsql/9.5/data/
↪postgresql.conf
```

PostgreSQL adds version-specific binaries in */usr/pgsql-9.5/bin*, but you'll usually just need psql, whose latest version is added to your path, and managing the server itself can be done with the *service* command.

3. Configure connection and authentication

Before starting the database let's change its access permissions. By default the database server listens only to clients on localhost. As a part of this step, we instruct it to listen on all IP interfaces, and then configure the client authentication file to allow all incoming connections from the local network.

```
sudo vi /var/lib/pgsql/9.5/data/postgresql.conf
```

```
# Uncomment listen_addresses for the changes to take effect
listen_addresses = '*'
```

```
sudo vi /var/lib/pgsql/9.5/data/pg_hba.conf
```

```
# Allow unrestricted access to nodes in the local network. The following ranges
# correspond to 24, 20, and 16-bit blocks in Private IPv4 address spaces.
host    all             all             10.0.0.0/8              trust
```

Note: Your DNS settings may differ. Also these settings are too permissive for some environments. The PostgreSQL manual explains how to make them more restrictive.

4. Start database servers, create Citus extension

```
# start the db server
sudo service postgresql-9.5 restart
# and make it start automatically when computer does
sudo chkconfig postgresql-9.5 on
```

You must add the Citus extension to **every database** you would like to use in a cluster. The following example adds the extension to the default database which is named *postgres*.

```
sudo -i -u postgres psql -c "CREATE EXTENSION citus;"
```

8.2.2 Steps to be executed on the master node

The steps listed below must be executed **only** on the master node after the previously mentioned steps have been executed.

1. Add worker node information

We need to inform the master about its workers. To add this information, we append the worker database names and server ports to the *pg_worker_list.conf* file in the data directory. For our example, we assume that there are two workers (named worker-101, worker-102). Add the workers' DNS names and server ports to the list.

```
echo "worker-101 5432" | sudo -u postgres tee -a /var/lib/pgsql/9.5/data/pg_worker_
↪list.conf
echo "worker-102 5432" | sudo -u postgres tee -a /var/lib/pgsql/9.5/data/pg_worker_
↪list.conf
```

Note that you can also add this information by editing the file using your favorite editor.

2. Reload master database settings

```
sudo service postgresql-9.5 reload
```

3. Verify that installation has succeeded

To verify that the installation has succeeded, we check that the master node has picked up the desired worker configuration. This command when run in the psql shell should output the worker nodes mentioned in the *pg_worker_list.conf* file.

```
sudo -i -u postgres psql -c "SELECT * FROM master_get_active_worker_nodes();"
```

Ready to use Citus

At this step, you have completed the installation process and are ready to use your Citus cluster. To help you get started, we have a *real-time aggregation tutorial* which has instructions on setting up a Citus cluster with sample data in minutes.

Your new Citus database is accessible in psql through the postgres user:

```
sudo -i -u postgres psql
```

8.3 Multi-node setup on Ubuntu or Debian

This section describes the steps needed to set up a multi-node Citus cluster on your own Linux machines using deb packages.

8.3.1 Steps to be executed on all nodes

1. Add PGDG repository

```
# add postgresql-9.5-citus pgdg repository
echo "deb http://apt.postgresql.org/pub/repos/apt/ $(lsb_release -cs)-pgdg main" |␣
↪sudo tee -a /etc/apt/sources.list.d/pgdg.list
sudo apt-get install wget ca-certificates
wget --quiet --no-check-certificate -O - https://www.postgresql.org/media/keys/
↪ACCC4CF8.asc | sudo apt-key add -
sudo apt-get update
```

2. Install PostgreSQL + Citus and initialize a database

```
# install the server and initialize db
sudo apt-get -y install postgresql-9.5-citus

# preload citus extension
sudo pg_conftool 9.5 main set shared_preload_libraries citus
```

This installs centralized configuration in */etc/postgresql/9.5/main*, and creates a database in */var/lib/postgresql/9.5/main*.

3. Configure connection and authentication

Before starting the database let's change its access permissions. By default the database server listens only to clients on localhost. As a part of this step, we instruct it to listen on all IP interfaces, and then configure the client authentication file to allow all incoming connections from the local network.

```
sudo pg_conftool 9.5 main set listen_addresses '*'
```

```
sudo vi /etc/postgresql/9.5/main/pg_hba.conf
```

```
# Allow unrestricted access to nodes in the local network. The following ranges
# correspond to 24, 20, and 16-bit blocks in Private IPv4 address spaces.
host    all             all             10.0.0.0/8              trust
```

Note: Your DNS settings may differ. Also these settings are too permissive for some environments. The PostgreSQL manual explains how to make them more restrictive.

4. Start database servers, create Citus extension

```
# start the db server
sudo service postgresql restart
# and make it start automatically when computer does
sudo update-rc.d postgresql enable
```

You must add the Citus extension to **every database** you would like to use in a cluster. The following example adds the extension to the default database which is named *postgres*.

```
# add the citus extension
sudo -i -u postgres psql -c "CREATE EXTENSION citus;"
```

8.3.2 Steps to be executed on the master node

The steps listed below must be executed **only** on the master node after the previously mentioned steps have been executed.

1. Add worker node information

We need to inform the master about its workers. To add this information, we append the worker database names and server ports to the *pg_worker_list.conf* file in the data directory. For our example, we assume that there are two workers (named worker-101, worker-102). Add the workers' DNS names and server ports to the list.

```
echo "worker-101 5432" | sudo -u postgres tee -a /var/lib/postgresql/9.5/main/pg_
↪worker_list.conf
echo "worker-102 5432" | sudo -u postgres tee -a /var/lib/postgresql/9.5/main/pg_
↪worker_list.conf
```

Note that you can also add this information by editing the file using your favorite editor.

2. Reload master database settings

```
sudo service postgresql reload
```

3. Verify that installation has succeeded

To verify that the installation has succeeded, we check that the master node has picked up the desired worker configuration. This command when run in the psql shell should output the worker nodes mentioned in the *pg_worker_list.conf* file.

```
sudo -i -u postgres psql -c "SELECT * FROM master_get_active_worker_nodes();"
```

Ready to use Citus

At this step, you have completed the installation process and are ready to use your Citus cluster. To help you get started, we have a *real-time aggregation tutorial* which has instructions on setting up a Citus cluster with sample data in minutes.

Your new Citus database is accessible in psql through the postgres user:

```
sudo -i -u postgres psql
```

CHAPTER 9

Working with distributed tables

Citus provides distributed functionality by extending PostgreSQL using the hook and extension APIs. This allows users to benefit from the features that come with the rich PostgreSQL ecosystem. These features include, but aren't limited to, support for a wide range of data types (including semi-structured data types like jsonb and hstore), operators and functions, full text search, and other extensions such as PostGIS and HyperLogLog. Further, proper use of the extension APIs enable compatibility with standard PostgreSQL tools such as pgAdmin, pg_backup, and pg_upgrade.

Citus users can leverage standard PostgreSQL interfaces with minimal modifications to enable distributed behavior. This includes commands for creating tables, loading data, updating rows, and also for querying. You can find a full reference of the PostgreSQL constructs here. We also discuss the relevant commands in our documentation as needed. Before we dive into the syntax for these commands, we briefly discuss two important concepts which must be decided during distributed table creation: the distribution column and distribution method.

9.1 Distribution Column

Every distributed table in Citus has exactly one column which is chosen as the distribution column. This informs the database to maintain statistics about the distribution column in each shard. Citus's distributed query optimizer then leverages these statistics to determine how best a query should be executed.

Typically, you should choose that column as the distribution column which is the most commonly used join key or on which most queries have filters. For filters, Citus uses the distribution column ranges to prune away unrelated shards, ensuring that the query hits only those shards which overlap with the WHERE clause ranges. For joins, if the join key is the same as the distribution column, then Citus executes the join only between those shards which have matching / overlapping distribution column ranges. This helps in greatly reducing both the amount of computation on each node and the network bandwidth involved in transferring shards across nodes. In addition to joins, choosing the right column as the distribution column also helps Citus push down several operations directly to the worker shards, hence reducing network I/O.

Note: Citus also allows joining on non-distribution key columns by dynamically repartitioning the tables for the query. Still, joins on non-distribution keys require shuffling data across the cluster and therefore aren't as efficient as joins on distribution keys.

The best option for the distribution column varies depending on the use case and the queries. In general, we find two common patterns: (1) **distributing by time** (timestamp, review date, order date), and (2) **distributing by identifier**

(user id, order id, application id). Typically, data arrives in a time-ordered series. So, if your use case works well with batch loading, it is easiest to distribute your largest tables by time, and load it into Citus in intervals of N minutes. In some cases, it might be worth distributing your incoming data by another key (e.g. user id, application id) and Citus will route your rows to the correct shards when they arrive. This can be beneficial when most of your queries involve a filter or joins on user id or order id.

9.2 Distribution Method

The next step after choosing the right distribution column is deciding the right distribution method. Citus supports two distribution methods: append and hash. Citus also provides the option for range distribution but that currently requires manual effort to set up.

As the name suggests, append based distribution is more suited to append-only use cases. This typically includes event based data which arrives in a time-ordered series. You can then distribute your largest tables by time, and batch load your events into Citus in intervals of N minutes. This data model can be generalized to a number of time series use cases; for example, each line in a website's log file, machine activity logs or aggregated website events. Append based distribution supports more efficient range queries. This is because given a range query on the distribution key, the Citus query planner can easily determine which shards overlap that range and send the query to only to relevant shards.

Hash based distribution is more suited to cases where you want to do real-time inserts along with analytics on your data or want to distribute by a non-ordered column (eg. user id). This data model is relevant for real-time analytics use cases; for example, actions in a mobile application, user website events, or social media analytics. In this case, Citus will maintain minimum and maximum hash ranges for all the created shards. Whenever a row is inserted, updated or deleted, Citus will redirect the query to the correct shard and issue it locally. This data model is more suited for doing co-located joins and for queries involving equality based filters on the distribution column.

Citus uses slightly different syntaxes for creation and manipulation of append and hash distributed tables. Also, the operations supported on the tables differ based on the distribution method chosen. In the sections that follow, we describe the syntax for creating append and hash distributed tables, and also describe the operations which can be done on them. We also briefly discuss how you can setup range distribution manually.

CHAPTER 10

Append Distribution

Append distributed tables are best suited to append-only event data which arrives in a time-ordered series. In the next few sections, we describe how users can create append distributed tables, load data into them and also expire old data from them.

Note: The instructions below assume that the PostgreSQL installation is in your path. If not, you will need to add it to your PATH environment variable. For example:

```
export PATH=/usr/lib/postgresql/9.5/:$PATH
```

We use the github events dataset to illustrate the commands below. You can download that dataset by running:

```
wget http://examples.citusdata.com/github_archive/github_events-2015-01-01-{0..5}.csv.
↪gz
gzip -d github_events-2015-01-01-*.gz
```

10.1 Creating and Distributing Tables

To create an append distributed table, you need to first define the table schema. To do so, you can define a table using the CREATE TABLE statement in the same way as you would do with a regular PostgreSQL table.

```
csql -h localhost -d postgres
CREATE TABLE github_events
(
    event_id bigint,
    event_type text,
    event_public boolean,
    repo_id bigint,
    payload jsonb,
    repo jsonb,
```

(continues on next page)

(continued from previous page)

```
    actor jsonb,
    org jsonb,
    created_at timestamp
);
```

Next, you can use the master_create_distributed_table() function to mark the table as an append distributed table and specify its distribution column.

```
SELECT master_create_distributed_table('github_events', 'created_at', 'append');
```

This function informs Citus that the github_events table should be distributed by append on the created_at column. Note that this method doesn't enforce a particular distribution; it merely tells the database to keep minimum and maximum values for the created_at column in each shard which are later used by the database for optimizing queries.

10.2 Data Loading

Citus supports two methods to load data into your append distributed tables. The first one is suitable for bulk loads from CSV/TSV files and involves using the stage command. For use cases requiring smaller, incremental data loads, Citus provides two user defined functions. We describe each of the methods and their usage below.

10.2.1 Bulk load using \stage

The stage command is used to copy data from a file to a distributed table while handling replication and failures automatically. It is not available in the standard psql client. Instead, Citus provides a database client called csql which supports all the commands of psql plus the stage command.

Note: You can use the psql client if you do not wish to use stage to ingest data. In upcoming releases, stage will be replaced by more native ingest methods which will remove the need for csql.

The stage command borrows its syntax from the client-side copy command in PostgreSQL. Behind the scenes, stage first opens a connection to the master and fetches candidate workers on which to create new shards. Then, the command connects to these workers, creates at least one shard there, and uploads the data to the shards. The command then replicates these shards on other workers until the replication factor is satisfied and fetches statistics for these shards. Finally, the command stores the shard metadata with the master.

```
SET citus.shard_max_size TO '64MB';
SET citus.shard_replication_factor TO 1;
\stage github_events from 'github_events-2015-01-01-0.csv' WITH CSV;
```

Citus assigns a unique shard id to each new shard and all its replicas have the same shard id. Each shard is represented on the worker node as a regular PostgreSQL table with name 'tablename_shardid' where tablename is the name of the distributed table and shardid is the unique id assigned to that shard. One can connect to the worker postgres instances to view or run commands on individual shards.

By default, the stage command depends on two configuration parameters for its behavior. These are called citus.shard_max_size and citus.shard_replication_factor.

1. **citus.shard_max_size :-** This parameter determines the maximum size of a shard created using stage, and defaults to 1 GB. If the file is larger than this parameter, stage will break it up into multiple shards.
2. **citus.shard_replication_factor :-** This parameter determines the number of nodes each shard gets replicated to, and defaults to two. The ideal value for this parameter depends on the size of the cluster and rate of node failure. For example, you may want to increase the replication factor if you run large clusters and observe node failures on a more frequent basis.

Please note that you can load several files in parallel through separate database connections or from different nodes. It is also worth noting that stage always creates at least one shard and does not append to existing shards. You can use the method described below to append to previously created shards.

10.2.2 Incremental loads by appending to existing shards

The stage command always creates a new shard when it is used and is best suited for bulk loading of data. Using stage to load smaller data increments will result in many small shards which might not be ideal. In order to allow smaller, incremental loads into append distributed tables, Citus provides 2 user defined functions. They are master_create_empty_shard() and master_append_table_to_shard().

master_create_empty_shard() can be used to create new empty shards for a table. This function also replicates the empty shard to citus.shard_replication_factor number of nodes like the stage command.

master_append_table_to_shard() can be used to append the contents of a PostgreSQL table to an existing shard. This allows the user to control the shard to which the rows will be appended. It also returns the shard fill ratio which helps to make a decision on whether more data should be appended to this shard or if a new shard should be created.

To use the above functionality, you can first insert incoming data into a regular PostgreSQL table. You can then create an empty shard using master_create_empty_shard(). Then, using master_append_table_to_shard(), you can append the contents of the staging table to the specified shard, and then subsequently delete the data from the staging table. Once the shard fill ratio returned by the append function becomes close to 1, you can create a new shard and start appending to the new one.

```
SELECT * from master_create_empty_shard('github_events');
master_create_empty_shard
---------------------------
                102089
(1 row)

SELECT * from master_append_table_to_shard(102089, 'github_events_temp', 'master-101',
↪ 5432);
master_append_table_to_shard
------------------------------
        0.100548
(1 row)
```

To learn more about the two UDFs, their arguments and usage, please visit the *User Defined Functions Reference* section of the documentation.

10.2.3 Increasing data loading performance

The methods described above enable you to achieve high bulk load rates which are sufficient for most use cases. If you require even higher data load rates, you can use the functions described above in several ways and write scripts to better control sharding and data loading. For more information, you can consult the *Scaling Out Data Ingestion* section of our documentation.

10.3 Dropping Shards

In append distribution, users typically want to track data only for the last few months / years. In such cases, the shards that are no longer needed still occupy disk space. To address this, Citus provides a user defined function master_apply_delete_command() to delete old shards. The function takes a DELETE command as input and deletes all the shards that match the delete criteria with their metadata.

The function uses shard metadata to decide whether or not a shard needs to be deleted, so it requires the WHERE clause in the DELETE statement to be on the distribution column. If no condition is specified, then all shards are selected for deletion. The UDF then connects to the worker nodes and issues DROP commands for all the shards which need to be deleted. If a drop query for a particular shard replica fails, then that replica is marked as TO DELETE. The shard replicas which are marked as TO DELETE are not considered for future queries and can be cleaned up later.

Please note that this function only deletes complete shards and not individual rows from shards. If your use case requires deletion of individual rows in real-time, please consider using the hash distribution method.

The example below deletes those shards from the github_events table which have all rows with created_at <= '2014-01-01 00:00:00'. Note that the table is distributed on the created_at column.

```
SELECT * from master_apply_delete_command('DELETE FROM github_events WHERE created_at
↪<= ''2014-01-01 00:00:00''');
 master_apply_delete_command
-----------------------------
                           3
(1 row)
```

To learn more about the function, its arguments and its usage, please visit the *User Defined Functions Reference* section of our documentation.

10.4 Dropping Tables

You can use the standard PostgreSQL DROP TABLE command to remove your append distributed tables. As with regular tables, DROP TABLE removes any indexes, rules, triggers, and constraints that exist for the target table. In addition, it also drops the shards on the worker nodes and cleans up their metadata.

```
DROP TABLE github_events;
```

CHAPTER 11

Hash Distribution

Hash distributed tables are best suited for use cases which require real-time inserts and updates. They also allow for faster key-value lookups and efficient joins on the distribution column. In the next few sections, we describe how you can create and distribute tables using the hash distribution method, and do real time inserts and updates to your data in addition to analytics.

Note: The instructions below assume that the PostgreSQL installation is in your path. If not, you will need to add it to your PATH environment variable. For example:

```
export PATH=/usr/lib/postgresql/9.5/:$PATH
```

We use the github events dataset to illustrate the commands below. You can download that dataset by running:

```
wget http://examples.citusdata.com/github_archive/github_events-2015-01-01-{0..5}.csv.
↪gz
gzip -d github_events-2015-01-01-*.gz
```

11.1 Creating And Distributing Tables

To create a hash distributed table, you need to first define the table schema. To do so, you can define a table using the CREATE TABLE statement in the same way as you would do with a regular PostgreSQL table.

```
psql -h localhost -d postgres
CREATE TABLE github_events
(
    event_id bigint,
    event_type text,
    event_public boolean,
    repo_id bigint,
    payload jsonb,
```

(continues on next page)

(continued from previous page)

```
    repo jsonb,
    actor jsonb,
    org jsonb,
    created_at timestamp
);
```

Next, you can use the master_create_distributed_table() function to mark the table as a hash distributed table and specify its distribution column.

```
SELECT master_create_distributed_table('github_events', 'repo_id', 'hash');
```

This function informs Citus that the github_events table should be distributed by hash on the repo_id column.

Then, you can create shards for the distributed table on the worker nodes using the master_create_worker_shards() UDF.

```
SELECT master_create_worker_shards('github_events', 16, 1);
```

This UDF takes two arguments in addition to the table name; shard count and the replication factor. This example would create a total of sixteen shards where each shard owns a portion of a hash token space and gets replicated on one worker. The shard replica created on the worker has the same table schema, index, and constraint definitions as the table on the master. Once the replica is created, this function saves all distributed metadata on the master.

Each created shard is assigned a unique shard id and all its replicas have the same shard id. Each shard is represented on the worker node as a regular PostgreSQL table with name 'tablename_shardid' where tablename is the name of the distributed table and shardid is the unique id assigned to that shard. You can connect to the worker postgres instances to view or run commands on individual shards.

After creating the worker shard, you are ready to insert data into the hash distributed table and run queries on it. You can also learn more about the UDFs used in this section in the *User Defined Functions Reference* of our documentation.

11.2 Inserting Data

11.2.1 Single row inserts

To insert data into hash distributed tables, you can use the standard PostgreSQL INSERT commands. As an example, we pick two rows randomly from the Github Archive dataset.

```
INSERT INTO github_events VALUES (2489373118,'PublicEvent','t',24509048,'{}','{"id":␣
↪24509048, "url": "https://api.github.com/repos/SabinaS/csee6868", "name": "SabinaS/
↪csee6868"}','{"id": 2955009, "url": "https://api.github.com/users/SabinaS", "login
↪": "SabinaS", "avatar_url": "https://avatars.githubusercontent.com/u/2955009?",
↪"gravatar_id": ""}',NULL,'2015-01-01 00:09:13');

INSERT INTO github_events VALUES (2489368389,'WatchEvent','t',28229924,'{"action":
↪"started"}','{"id": 28229924, "url": "https://api.github.com/repos/inf0rmer/blanket
↪", "name": "inf0rmer/blanket"}','{"id": 1405427, "url": "https://api.github.com/
↪users/tategakibunko", "login": "tategakibunko", "avatar_url": "https://avatars.
↪githubusercontent.com/u/1405427?", "gravatar_id": ""}',NULL,'2015-01-01 00:00:24');
```

When inserting rows into hash distributed tables, the distribution column of the row being inserted must be specified. Based on the distribution column, Citus determines the right shard to which the insert should be routed to. Then, the query is forwarded to the right shard, and the remote insert command is executed on all the replicas of that shard.

11.2.2 Bulk inserts

Sometimes, you may want to bulk load several rows together into your hash distributed tables. To facilitate this, a script named copy_to_distributed_table is provided for loading many rows of data from a file, similar to the functionality provided by PostgreSQL's COPY command. It is automatically installed into the bin directory for your PostgreSQL installation.

Before invoking the script, you should set the environment variables which will be used as connection parameters while connecting to your Postgres server. For example, to set the default database to postgres, you can run the command shown below.

```
export PGDATABASE=postgres
```

As an example usage for the script, the invocation below would copy rows into the github_events table from a CSV file.

```
copy_to_distributed_table -C github_events-2015-01-01-0.csv github_events
```

To learn more about the different options supported by the script, you can call the script with -h for usage information.

```
copy_to_distributed_table -h
```

Note that hash distributed tables are optimised for real-time ingestion, where users typically have to do single row inserts into distributed tables. Bulk loading, though supported, is generally slower than tables using the append distribution method. For use cases involving bulk loading of data, please consider using *Append Distribution*.

11.3 Updating and Deleting Data

You can also update or delete rows from your tables, using the standard PostgreSQL UPDATE and DELETE commands.

```
UPDATE github_events SET org = NULL WHERE repo_id = 24509048;
DELETE FROM github_events WHERE repo_id = 24509048;
```

Currently, Citus requires that an UPDATE or DELETE involves exactly one shard. This means commands must include a WHERE qualification on the distribution column that restricts the query to a single shard. Such qualifications usually take the form of an equality clause on the table's distribution column.

11.4 Maximizing Write Performance

Both INSERT and UPDATE/DELETE statements can be scaled up to around 50,000 queries per second on large machines. However, to achieve this rate, you will need to use many parallel, long-lived connections and consider how to deal with locking. For more information, you can consult the *Scaling Out Data Ingestion* section of our documentation.

11.5 Dropping Tables

You can use the standard PostgreSQL DROP TABLE command to remove your hash distributed tables. As with regular tables, DROP TABLE removes any indexes, rules, triggers, and constraints that exist for the target table. In addition, it also drops the shards on the worker nodes and cleans up their metadata.

```
DROP TABLE github_events;
```

CHAPTER 12

Range Distribution (Manual)

Citus also supports range based distribution, but this currently requires manual effort to set up. In this section, we briefly describe how you can set up range distribution and where it can be useful.

Note: The instructions below assume that the PostgreSQL installation is in your path. If not, you will need to add it to your PATH environment variable. For example:

```
export PATH=/usr/lib/postgresql/9.5/:$PATH
```

To create a range distributed table, you need to first define the table schema. To do so, you can define a table using the CREATE TABLE statement in the same way as you would do with a regular postgresql table.

```
psql -h localhost -d postgres
CREATE TABLE github_events
(
    event_id bigint,
    event_type text,
    event_public boolean,
    repo_id bigint,
    payload jsonb,
    repo jsonb,
    actor jsonb,
    org jsonb,
    created_at timestamp
);
```

Next, you can use the master_create_distributed_table() function to mark the table as a range distributed table and specify its distribution column.

```
SELECT master_create_distributed_table('github_events', 'repo_id', 'range');
```

This function informs Citus that the github_events table should be distributed by range on the repo_id column.

Range distribution signifies to the database that all the shards have non-overlapping ranges of the distribution key. Currently, the stage command for data loading does not impose that the shards have non-overlapping distribution key ranges. Hence, the user needs to make sure that the shards don't overlap.

To set up range distributed shards, you first have to sort the data on the distribution column. This may not be required if the data already comes sorted on the distribution column (eg. facts, events, or time-series tables distributed on time). The next step is to split the input file into different files having non-overlapping ranges of the distribution key and run the stage command for each file separately. As stage always creates a new shard, the shards are guaranteed to have non-overlapping ranges.

As an example, we'll describe how to stage data into the github_events table shown above. First, you download and extract two hours of github data.

```
wget http://examples.citusdata.com/github_archive/github_events-2015-01-01-0.csv.gz
wget http://examples.citusdata.com/github_archive/github_events-2015-01-01-1.csv.gz
gzip -d github_events-2015-01-01-0.csv.gz
gzip -d github_events-2015-01-01-1.csv.gz
```

Then, you merge both files into a single file by using the cat command.

```
cat github_events-2015-01-01-0.csv github_events-2015-01-01-1.csv > github_events-
↪2015-01-01-0-1.csv
```

Next, you should sort the data on the repo_id column using the linux sort command.

```
sort -n --field-separator=',' --key=4 github_events-2015-01-01-0-1.csv > github_
↪events_sorted.csv
```

Finally, you can split the input data such that no two files have any overlapping partition column ranges. This can be done by writing a custom script or manually ensuring that files don't have overlapping ranges of the repo_id column.

For this example, you can download and extract data that has been previously split on the basis of repo_id.

```
wget http://examples.citusdata.com/github_archive/github_events_2015-01-01-0-1_range1.
↪csv.gz
wget http://examples.citusdata.com/github_archive/github_events_2015-01-01-0-1_range2.
↪csv.gz
wget http://examples.citusdata.com/github_archive/github_events_2015-01-01-0-1_range3.
↪csv.gz
wget http://examples.citusdata.com/github_archive/github_events_2015-01-01-0-1_range4.
↪csv.gz
gzip -d github_events_2015-01-01-0-1_range1.csv.gz
gzip -d github_events_2015-01-01-0-1_range2.csv.gz
gzip -d github_events_2015-01-01-0-1_range3.csv.gz
gzip -d github_events_2015-01-01-0-1_range4.csv.gz
```

Then, you can connect to the master using csql and load the files using the stage command.

The client application csql is provided with Citus. It is similar to and supports all the functionality provided by psql. The only difference is that it adds a client side stage command which is useful for batch loading data into range distributed tables.

Note: You can use the psql client if you do not wish to use stage to ingest data. In upcoming releases, stage will be replaced by more native ingest methods which will remove the need for csql.

```
csql -h localhost -d postgres
SET citus.shard_replication_factor TO 1;
\stage github_events from 'github_events_2015-01-01-0-1_range1.csv' with csv;
\stage github_events from 'github_events_2015-01-01-0-1_range2.csv' with csv;
```

(continues on next page)

(continued from previous page)

```
\stage github_events from 'github_events_2015-01-01-0-1_range3.csv' with csv;
\stage github_events from 'github_events_2015-01-01-0-1_range4.csv' with csv;
```

After this point, you can run queries on the range distributed table. To generate per-repository metrics, your queries would generally have filters on the repo_id column. Then, Citus can easily prune away unrelated shards and ensure that the query hits only one shard. Also, groupings and orderings on repo_id can be easily pushed down the workers leading to more efficient queries. We also note here that all the commands which can be run on tables using the append distribution method can be run on tables using range distribution. This includes stage, the append and shard creation UDFs and the delete UDF.

The difference between range and append methods is that Citus's distributed query planner has extra knowledge that the shards have distinct non-overlapping distribution key ranges. This allows the planner to push down more operations to the workers so that they can be executed in parallel. This reduces both the amount of data transferred across network and the amount of computation to be done for aggregation on the master.

CHAPTER 13

Querying Distributed Tables

As discussed in the previous sections, Citus is an extension which extends the latest PostgreSQL for distributed execution. This means that you can use standard PostgreSQL SELECT queries on the Citus master for querying. Citus will then parallelize the SELECT queries involving complex selections, groupings and orderings, and JOINs to speed up the query performance. At a high level, Citus partitions the SELECT query into smaller query fragments, assigns these query fragments to workers, oversees their execution, merges their results (and orders them if needed), and returns the final result to the user.

In the following sections, we discuss the different types of queries you can run using Citus.

13.1 Aggregate Functions

Citus supports and parallelizes most aggregate functions supported by PostgreSQL. Citus's query planner transforms the aggregate into its commutative and associative form so it can be parallelized. In this process, the workers run an aggregation query on the shards and the master then combines the results from the workers to produce the final output.

13.1.1 Count (Distinct) Aggregates

Citus supports count(distinct) aggregates in several ways. If the count(distinct) aggregate is on the distribution column, Citus can directly push down the query to the workers. If not, Citus needs to repartition the underlying data in the cluster to parallelize count(distinct) aggregates and avoid pulling all rows to the master.

To address the common use case of count(distinct) approximations, Citus provides an option of using the HyperLogLog algorithm to efficiently calculate approximate values for the count distincts on non-distribution key columns.

To enable count distinct approximations, you can follow the steps below:

1. Download and install the hll extension on all PostgreSQL instances (the master and all the workers).

Please visit the PostgreSQL hll github repository for specifics on obtaining the extension.

2. Create the hll extension on all the PostgreSQL instances

```
CREATE EXTENSION hll;
```

3. Enable count distinct approximations by setting the citus.count_distinct_error_rate configuration value. Lower values for this configuration setting are expected to give more accurate results but take more time for computation. We recommend setting this to 0.005.

```
SET citus.count_distinct_error_rate to 0.005;
```

After this step, you should be able to run approximate count distinct queries on any column of the table.

13.1.2 HyperLogLog Column

Certain users already store their data as HLL columns. In such cases, they can dynamically roll up those data by creating custom aggregates within Citus.

As an example, if you want to run the hll_union aggregate function on your data stored as hll, you can define an aggregate function like below :

```
CREATE AGGREGATE sum (hll)
(
sfunc = hll_union_trans,
stype = internal,
finalfunc = hll_pack
);
```

You can then call sum(hll_column) to roll up those columns within the database. Please note that these custom aggregates need to be created both on the master and the workers.

13.2 Limit Pushdown

Citus also pushes down the limit clauses to the shards on the workers wherever possible to minimize the amount of data transferred across network.

However, in some cases, SELECT queries with LIMIT clauses may need to fetch all rows from each shard to generate exact results. For example, if the query requires ordering by the aggregate column, it would need results of that column from all shards to determine the final aggregate value. This reduces performance of the LIMIT clause due to high volume of network data transfer. In such cases, and where an approximation would produce meaningful results, Citus provides an option for network efficient approximate LIMIT clauses.

LIMIT approximations are disabled by default and can be enabled by setting the configuration parameter citus.limit_clause_row_fetch_count. On the basis of this configuration value, Citus will limit the number of rows returned by each task for aggregation on the master. Due to this limit, the final results may be approximate. Increasing this limit will increase the accuracy of the final results, while still providing an upper bound on the number of rows pulled from the workers.

```
SET citus.limit_clause_row_fetch_count to 10000;
```

13.3 Joins

Citus supports equi-JOINs between any number of tables irrespective of their size and distribution method. The query planner chooses the optimal join method and join order based on the statistics gathered from the distributed tables. It

evaluates several possible join orders and creates a join plan which requires minimum data to be transferred across network.

To determine the best join strategy, Citus treats large and small tables differently while executing JOINs. The distributed tables are classified as large and small on the basis of the configuration entry citus.large_table_shard_count (default value: 4). The tables whose shard count exceeds this value are considered as large while the others small. In practice, the fact tables are generally the large tables while the dimension tables are the small tables.

13.3.1 Broadcast joins

This join type is used while joining small tables with each other or with a large table. This is a very common use case where you want to join the keys in the fact tables (large table) with their corresponding dimension tables (small tables). Citus replicates the small table to all workers where the large table's shards are present. Then, all the joins are performed locally on the workers in parallel. Subsequent join queries that involve the small table then use these cached shards.

13.3.2 Colocated joins

To join two large tables efficiently, it is advised that you distribute them on the same columns you used to join the tables. In this case, the Citus master knows which shards of the tables might match with shards of the other table by looking at the distribution column metadata. This allows Citus to prune away shard pairs which cannot produce matching join keys. The joins between remaining shard pairs are executed in parallel on the workers and then the results are returned to the master.

Note: In order to benefit most from colocated joins, you should hash distribute your tables on the join key and use the same number of shards for both tables. If you do this, each shard will join with exactly one shard of the other table. Also, the shard creation logic will ensure that shards with the same distribution key ranges are on the same workers. This means no data needs to be transferred between the workers, leading to faster joins.

13.3.3 Repartition joins

In some cases, you may need to join two tables on columns other than the distribution column. For such cases, Citus also allows joining on non-distribution key columns by dynamically repartitioning the tables for the query.

In such cases, the best partition method (hash or range) and the table(s) to be partitioned is determined by the query optimizer on the basis of the distribution columns, join keys and sizes of the tables. With repartitioned tables, it can be ensured that only relevant shard pairs are joined with each other reducing the amount of data transferred across network drastically.

In general, colocated joins are more efficient than repartition joins as repartition joins require shuffling of data. So, you should try to distribute your tables by the common join keys whenever possible.

13.4 Data Warehousing Queries

Citus's current version works well for real-time analytics use cases. We are continuously working to increase SQL coverage to better support data warehousing use-cases. In the mean-time, since Citus is an extension on top of PostgreSQL, we can usually offer workarounds that work well for a number of use cases. So, if you can't find documentation for a SQL construct or run into an unsupported feature, please send us an email at engage@citusdata.com.

Here, we would like to illustrate one such example which works well when queries have restrictive filters i.e. when very few results need to be transferred to the master. In such cases, it is possible to run unsupported queries in two

steps by storing the results of the inner queries in regular PostgreSQL tables on the master. Then, the next step can be executed on the master like a regular PostgreSQL query.

For example, currently Citus does not have out of the box support for window functions on queries involving distributed tables. Suppose you have a query on the github_events table that has a window function like the following:

```
SELECT
    repo_id, actor->'id', count(*)
OVER
    (PARTITION BY repo_id)
FROM
    github_events
WHERE
    repo_id = 1 OR repo_id = 2;
```

You can re-write the query like below:

Statement 1:

```
CREATE TEMP TABLE results AS
(SELECT
    repo_id, actor->'id' as actor_id
FROM
    github_events
WHERE
    repo_id = 1 OR repo_id = 2
);
```

Statement 2:

```
SELECT
    repo_id, actor_id, count(*)
OVER
    (PARTITION BY repo_id)
FROM
    results;
```

Similar workarounds can be found for other data warehousing queries involving unsupported constructs.

Note: The above query is a simple example intended at showing how meaningful workarounds exist around the lack of support for a few query types. Over time, we intend to support these commands out of the box within Citus.

13.5 Query Performance

Citus parallelizes incoming queries by breaking it into multiple fragment queries ("tasks") which run on the worker shards in parallel. This allows Citus to utilize the processing power of all the nodes in the cluster and also of individual cores on each node for each query. Due to this parallelization, you can get performance which is cumulative of the computing power of all of the cores in the cluster leading to a dramatic decrease in query times versus PostgreSQL on a single server.

Citus employs a two stage optimizer when planning SQL queries. The first phase involves converting the SQL queries into their commutative and associative form so that they can be pushed down and run on the workers in parallel. As discussed in previous sections, choosing the right distribution column and distribution method allows the distributed query planner to apply several optimizations to the queries. This can have a significant impact on query performance due to reduced network I/O.

Citus's distributed executor then takes these individual query fragments and sends them to worker PostgreSQL instances. There are several aspects of both the distributed planner and the executor which can be tuned in order to improve performance. When these individual query fragments are sent to the workers, the second phase of query optimization kicks in. The workers are simply running extended PostgreSQL servers and they apply PostgreSQL's standard planning and execution logic to run these fragment SQL queries. Therefore, any optimization that helps PostgreSQL also helps Citus. PostgreSQL by default comes with conservative resource settings; and therefore optimizing these configuration settings can improve query times significantly.

We discuss the relevant performance tuning steps in the *Query Performance Tuning* section of the documentation.

CHAPTER 14

PostgreSQL extensions

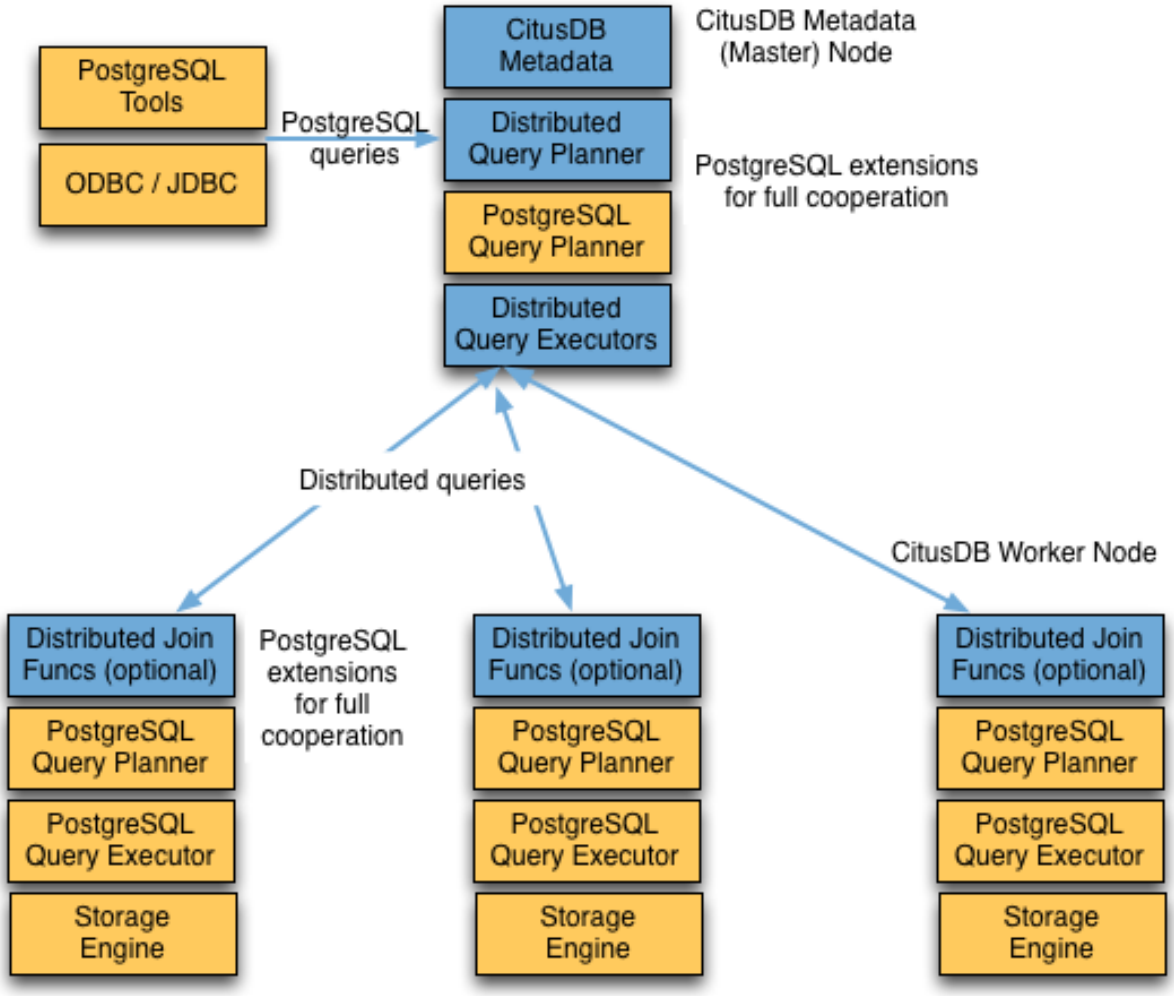
As Citus is an extension which can be installed on any PostgreSQL instance, you can directly use other extensions such as hstore, hll, or PostGIS with Citus. However, there are two things to keep in mind. First, while including other extensions in shared_preload_libraries, you should make sure that Citus is the first extension. Secondly, you should create the extension on both the master and the workers before starting to use it.

Note: Sometimes, there might be a few features of the extension that may not be supported out of the box. For example, a few aggregates in an extension may need to be modified a bit to be parallelized across multiple nodes. Please contact us at engage@citusdata.com if some feature from your favourite extension does not work as expected with Citus.

CHAPTER 15

Citus Query Processing

A Citus cluster consists of a master instance and multiple worker instances. The data is sharded and replicated on the workers while the master stores metadata about these shards. All queries issued to the cluster are executed via the master. The master partitions the query into smaller query fragments where each query fragment can be run independently on a shard. The master then assigns the query fragments to workers, oversees their execution, merges their results, and returns the final result to the user. The query processing architecture can be described in brief by the diagram below.



Citus's query processing pipeline involves the two components:

- **Distributed Query Planner and Executor**
- **PostgreSQL Planner and Executor**

We discuss them in greater detail in the subsequent sections.

15.1 Distributed Query Planner

Citus's distributed query planner takes in a SQL query and plans it for distributed execution.

For SELECT queries, the planner first creates a plan tree of the input query and transforms it into its commutative and associative form so it can be parallelized. It also applies several optimizations to ensure that the queries are executed in a scalable manner, and that network I/O is minimized.

Next, the planner breaks the query into two parts - the master query which runs on the master and the worker query fragments which run on individual shards on the workers. The planner then assigns these query fragments to the workers such that all their resources are used efficiently. After this step, the distributed query plan is passed on to the distributed executor for execution.

The planning process for key-value lookups on the distribution column or modification queries is slightly different as they hit exactly one shard. Once the planner receives an incoming query, it needs to decide the correct shard to which the query should be routed. To do this, it extracts the distribution column in the incoming row and looks up the metadata to determine the right shard for the query. Then, the planner rewrites the SQL of that command to reference the shard table instead of the original table. This re-written plan is then passed to the distributed executor.

15.2 Distributed Query Executor

Citus's distributed executors run distributed query plans and handle failures that occur during query execution. The executors connect to the workers, send the assigned tasks to them and oversee their execution. If the executor cannot assign a task to the designated worker or if a task execution fails, then the executor dynamically re-assigns the task to replicas on other workers. The executor processes only the failed query sub-tree, and not the entire query while handling failures.

Citus has 3 different executor types - real time, task tracker and router. The first two are better suited for larger SELECT queries while the router executor is useful for handling simple key-value lookups and INSERT, UPDATE, and DELETE queries. We briefly discuss the executors below.

15.2.1 Real-time Executor

The real-time executor is the default executor used by Citus. It is well suited for getting fast responses to queries involving filters, aggregations and colocated joins. The real time executor opens one connection per shard to the workers and sends all fragment queries to them. It then fetches the results from each fragment query, merges them, and gives the final results back to the user.

Since the real time executor maintains an open connection for each shard to which it sends queries, it may reach file descriptor / connection limits while dealing with high shard counts. In such cases, the real-time executor throttles on assigning more tasks to workers to avoid overwhelming them with too many tasks. One can typically increase the file descriptor limit on modern operating systems to avoid throttling, and change Citus configuration to use the real-time executor. But, that may not be ideal for efficient resource management while running complex queries. For queries that touch thousands of shards or require large table joins, you can use the task tracker executor.

15.2.2 Task Tracker Executor

The task tracker executor is well suited for long running, complex data warehousing queries. This executor opens only one connection per worker, and assigns all fragment queries to a task tracker daemon on the worker. The task tracker daemon then regularly schedules new tasks and sees through their completion. The executor on the master regularly checks with these task trackers to see if their tasks completed.

Each task tracker daemon on the workers also makes sure to execute at most citus.max_running_tasks_per_node concurrently. This concurrency limit helps in avoiding disk I/O contention when queries are not served from memory. The task tracker executor is designed to efficiently handle complex queries which require repartitioning and shuffling intermediate data among workers.

15.2.3 Router Executor

The router executor is used by Citus for handling simple key-value lookups and INSERT, UPDATE and DELETE queries. This executor assigns the incoming query to the worker which has the target shard. The query is then handled by the worker PostgreSQL server and the results are returned back to the user. In case a modification fails on a shard replica, the executor marks the corresponding shard replica as invalid in order to maintain data consistency.

15.3 PostgreSQL planner and executor

Once the distributed executor sends the query fragments to the workers, they are processed like regular PostgreSQL queries. The PostgreSQL planner on that worker chooses the most optimal plan for executing that query locally on the corresponding shard table. The PostgreSQL executor then runs that query and returns the query results back to the distributed executor. You can learn more about the PostgreSQL planner and executor from the PostgreSQL manual. Finally, the distributed executor passes the results to the master for final aggregation.

CHAPTER 16

Scaling Out Data Ingestion

Citus lets you scale out data ingestion to very high rates, but there are several trade-offs to consider in terms of the throughput, durability, consistency and latency. In this section, we discuss several approaches to data ingestion and give examples of how to use them.

The best method to distribute tables and ingest your data depends on your use case requirements. Citus supports two distribution methods: append and hash; and the data ingestion methods differ between them. You can visit the *Working with distributed tables* section to learn about the tradeoffs associated with each distribution method.

16.1 Hash Distributed Tables

Hash distributed tables support ingestion using standard single row INSERT and UPDATE commands. This subsection describes how you maximize insert throughput for hash distributed tables.

16.1.1 Real-time Inserts (0-50k/s)

On the Citus master, you can perform INSERT commands directly on hash distributed tables. The advantage of using INSERT is that the new data is immediately visible to SELECT queries, and durably stored on multiple replicas.

When processing an INSERT, Citus first finds the right shard placements based on the value in the distribution column, then it connects to the workers storing the shard placements, and finally performs an INSERT on each of them. From the perspective of the user, the INSERT takes several milliseconds to process because of the round-trips to the workers, but the master can process other INSERTs in other sessions while waiting for a response. The master also keeps connections to the workers open within the same session, which means subsequent queries will see lower response times.

```
-- Set up a distributed table containing counters
CREATE TABLE counters (c_key text, c_date date, c_value int, primary key (c_key, c_
↪date));
SELECT master_create_distributed_table('counters', 'c_key', 'hash');
SELECT master_create_worker_shards('counters', 128, 2);
```

(continues on next page)

(continued from previous page)

```
-- Enable timing to see reponse times
\timing

-- First INSERT requires connection set-up, second will be faster
INSERT INTO counters VALUES ('num_purchases', '2016-03-04', 12); -- Time: 10.314 ms
INSERT INTO counters VALUES ('num_purchases', '2016-03-05', 5); -- Time: 3.132 ms
```

INSERT is currently the only way of adding data to hash-distributed tables. To load data from a file into a hash-distributed table, Citus comes with a command-line tool called copy_to_distributed_table, which mimicks the behaviour of COPY by performing an INSERT for each row in an input (CSV) file. However, the script only uses a single connection, meaning every INSERT waits for several round-trips and throughput is very low by default. However, you can parallelize the script by splitting the input and doing the INSERTs in parallel using xargs, which gives vastly better throughput.

For example, for Linux systems you can use the split command to split the input file into 64 pieces.

```
mkdir chunks
split -n l/64 github_events-2015-01-01-0.csv chunks/
```

Then, you can load each of the chunks in parallel using xargs.

```
export PGDATABASE=postgres
find chunks/ -type f | xargs -n 1 -P 64 sh -c 'echo $0 `copy_to_distributed_table -C
↪$0 github_events`'
```

To learn more about the copy_to_distributed_table script, you can visit the *Hash Distribution* section of our documentation.

To reach high throughput rates, applications should send INSERTs over a many separate connections and keep connections open to avoid the initial overhead of connection set-up.

16.1.2 Real-time Updates (0-50k/s)

On the Citus master, you can also perform UPDATE, DELETE, and INSERT ... ON CONFLICT (UPSERT) commands on distributed tables. By default, these queries take an exclusive lock on the shard, which prevents concurrent modifications to guarantee that the commands are applied in the same order on all shard placements.

Given that every command requires several round-trips to the workers, and no two commands can run on the same shard at the same time, update throughput is very low by default. However, if you know that the order of the queries doesn't matter (they are commutative), then you can turn on citus.all_modifications_commutative, in which case multiple commands can update the same shard concurrently.

For example, if your distributed table contains counters and all your DML queries are UPSERTs that add to the counters, then you can safely turn on citus.all_modifications_commutative since addition is commutative:

```
SET citus.all_modifications_commutative TO on;
INSERT INTO counters VALUES ('num_purchases', '2016-03-04', 1)
ON CONFLICT (c_key, c_date) DO UPDATE SET c_value = counters.c_value + 1;
```

Note that this query also takes an exclusive lock on the row in PostgreSQL, which may also limit the throughput. When storing counters, consider that using INSERT and summing values in a SELECT does not require exclusive locks.

When the replication factor is 1, it is always safe to enable citus.all_modifications_commutative. Citus does not do this automatically yet.

16.1.3 Masterless Citus (50k/s-500k/s)

Note: This section is currently experimental and not a guide to setup masterless clusters in production. We are working on providing official support for masterless clusters including replication and automated fail-over solutions. Please contact us at engage@citusdata.com if your use case requires multiple masters.

It is technically possible to create the distributed table on every node in the cluster. The big advantage is that all queries on distributed tables can be performed at a very high rate by spreading the queries across the workers. In this case, the replication factor should always be 1 to ensure consistency, which causes data to become unavailable when a node goes down. All nodes should have a hot standby and automated fail-over to ensure high availability.

To allow DML commands on the distribute table from any node, first create a distributed table on both the master and the workers:

```
CREATE TABLE data (key text, value text);
SELECT master_create_distributed_table('data','key','hash');
```

Then on the master, create shards for the distributed table with a replication factor of 1.

```
/* Create 128 shards with a single replica on the workers */
SELECT master_create_worker_shards('data', 128, 1);
```

Finally, you need to copy and convert the shard metadata from the master to the workers. The logicalrelid column in pg_dist_shard may differ per node. If you have the dblink extension installed, then you can run the following commands on the workers to get the metadata from master-node.

```
INSERT INTO pg_dist_shard SELECT * FROM
dblink('host=master-node port=5432',
       'SELECT logicalrelid::regclass,shardid,shardstorage,shardalias,shardminvalue,
↪shardmaxvalue FROM pg_dist_shard')
AS (logicalrelid regclass, shardid bigint, shardstorage char, shardalias text,␣
↪shardminvalue text, shardmaxvalue text);

INSERT INTO pg_dist_shard_placement SELECT * FROM
dblink('host=master-node port=5432',
       'SELECT * FROM pg_dist_shard_placement')
AS (shardid bigint, shardstate int, shardlength bigint, nodename text, nodeport int);
```

After these commands, you can connect to any node and perform both SELECT and DML commands on the distributed table. However, DDL commands won't be supported.

16.2 Append Distributed Tables

If your use-case does not require real-time ingests, then using append distributed tables will give you the highest ingest rates. This approach is more suitable for use-cases which use time-series data and where the database can be a few minutes or more behind.

16.2.1 Master Node Bulk Ingestion (50k/s-100k/s)

To ingest data into an append-distributed table, your application can first create a staging table, copy or insert data into the staging table, and finally append the staging table to the distributed table. The simplest approach is to create the staging table on the master and append the table to a new shard using master_append_table_to_shard:

```
-- Set up the events table
CREATE TABLE events (time timestamp, data jsonb);
SELECT master_create_distributed_table('events', 'time', 'append');

-- Add data into a new staging table
CREATE UNLOGGED TABLE stage_1 (LIKE events);
COPY stage_1 FROM 'path-to-csv-file' WITH CSV; -- followed by CSV data

-- Add the data in the staging table to a new shard in the events table and drop the␣
↪staging table
SELECT master_append_table_to_shard(master_create_empty_shard('events'), 'stage_1',
↪'master-node', 5432);
DROP TABLE stage_1;
```

Note that copying to the staging table and appending to a shard need to be in separate transactions. Otherwise, the data would not yet be visible to the workers when trying to append. You can choose to make the staging table unlogged for better performance if you can easily reload the data. To learn more about the master_append_table_to_shard and master_create_empty_shard UDFs, please visit the *User Defined Functions Reference* section of the documentation.

The example above uses master_create_empty_shard to create a new shard every time new data is ingested, which allows many files to be ingested simultaneously, but may cause issues if queries end up involving thousands of shards.

One way to solve this problem is to define a function that selects either a new or existing shard:

```
SELECT master_append_table_to_shard(choose_shard('events'), 'stage_1', 'master-node',␣
↪5432);
```

An example of a shard selection function is given below. It appends to a shard until its size is greater than 1GB and then creates a new one, which has the drawback of only allowing one append at a time, but the advantage of bounding shard sizes.

```
CREATE OR REPLACE FUNCTION choose_shard(table_id regclass) RETURNS bigint AS $$
DECLARE
  shard_id bigint;
BEGIN
  SELECT shardid INTO shard_id
  FROM pg_dist_shard JOIN pg_dist_shard_placement USING (shardid)
  WHERE logicalrelid = table_id AND shardlength < 1024*1024*1024;

  IF shard_id IS NULL THEN
    /* no shard smaller than 1GB, create a new one */
    SELECT master_create_empty_shard(table_id::text) INTO shard_id;
  END IF;

  RETURN shard_id;
END;
$$ LANGUAGE plpgsql;
```

It may also be useful to create a sequence to generate a unique name for the staging table. This way each ingestion can be handled independently.

```
-- Create stage table name sequence
CREATE SEQUENCE stage_id_sequence;

-- Generate a stage table name
SELECT 'stage_'||nextval('stage_id_sequence');
```

16.2.2 Worker Node Bulk Ingestion (100k/s-1M/s)

For very high data ingestion rates, data can be staged via the workers. This method scales out horizontally and provides the highest ingestion rates, but is more complex to setup. Hence, we recommend trying this method only if your data ingestion rates cannot be addressed by the previously described methods.

A relatively simple way to ingest data files directly into new shards in the distributed table is to use csql, a database client that comes with Citus. csql is the same as psql, but with an additional STAGE command that copies data directly from the client to the workers into a new shard. STAGE can break up files larger than the configured citus.shard_max_size into multiple shards. The main drawbacks of STAGE are that it requires going through the command-line, only ingests files, and always creates one or more new shards.

```
csql -h master-node -c "\\STAGE events FROM 'data.csv' WITH CSV"
```

An alternative to using STAGE is to create a staging table and use standard SQL clients to append it to the distributed table, which is similar to staging data via the master. An example of staging a file via a worker using psql is as follows:

```
stage_table=$(psql -tA -h worker-node-1 -c "SELECT 'stage_'||nextval('stage_id_
↪sequence')")
psql -h worker-node-1 -c "CREATE TABLE $stage_table (time timestamp, data jsonb)"
psql -h worker-node-1 -c "\\COPY $stage_table FROM 'data.csv' WITH CSV"
psql -h master-node -c "SELECT master_append_table_to_shard(choose_shard('events'), '
↪$stage_table', 'worker-node-1', 5432)"
psql -h worker-node-1 -c "DROP TABLE $stage_table"
```

The example above again uses a choose_shard function to select the shard to which to append. To ensure parallel data ingestion, this function should balance across many different shards.

An example choose_shard function belows randomly picks one of the 20 smallest shards or creates a new one if there are less than 20 under 1GB. This allows 20 concurrent appends, which allows data ingestion of up to 1 million rows/s (depending on indexes, size, capacity).

```
/* Choose a shard to which to append */
CREATE OR REPLACE FUNCTION choose_shard(table_id regclass)
RETURNS bigint LANGUAGE plpgsql
AS $function$
DECLARE
  shard_id bigint;
  num_small_shards int;
BEGIN
  SELECT shardid, count(*) OVER () INTO shard_id, num_small_shards
  FROM pg_dist_shard JOIN pg_dist_shard_placement USING (shardid)
  WHERE logicalrelid = table_id AND shardlength < 1024*1024*1024
  GROUP BY shardid ORDER BY RANDOM() ASC;

  IF num_small_shards IS NULL OR num_small_shards < 20 THEN
    SELECT master_create_empty_shard(table_id::text) INTO shard_id;
  END IF;

  RETURN shard_id;
END;
$function$;
```

A drawback of this approach is that shards may span longer time periods, which means that queries for a specific time period may involve shards that contain a lot of data outside of that period.

In addition to copying into temporary staging tables, it is also possible to set up tables on the workers which can continuously take INSERTs. In that case, the data has to be periodically moved into a staging table and then appended,

but this requires more advanced scripting.

16.3 Pre-processing Data in Citus

The format in which raw data is delivered often differs from the schema used in the database. For example, the raw data may be in the form of log files in which every line is a JSON object, while in the database table it is more efficient to store common values in separate columns. Moreover, a distributed table should always have a distribution column. Fortunately, PostgreSQL is a very powerful data processing tool. You can apply arbitrary pre-processing using SQL before putting the results into a staging table.

For example, assume we have the following table schema and want to load the compressed JSON logs from githubarchive.org:

```
CREATE TABLE github_events
(
    event_id bigint,
    event_type text,
    event_public boolean,
    repo_id bigint,
    payload jsonb,
    repo jsonb,
    actor jsonb,
    org jsonb,
    created_at timestamp
);
SELECT master_create_distributed_table('github_events', 'created_at', 'append');
```

To load the data, we can download the data, decompress it, filter out unsupported rows, and extract the fields in which we are interested into a staging table using 3 commands:

```
CREATE TEMPORARY TABLE prepare_1 (data jsonb);

/* Load a file directly from Github archive and filter out rows with unescaped 0-
↪bytes */
COPY prepare_1 FROM PROGRAM
'curl -s http://data.githubarchive.org/2016-01-01-15.json.gz | zcat | grep -v "\\u0000
↪"'
CSV QUOTE e'\x01' DELIMITER e'\x02';

/* Prepare a staging table */
CREATE UNLOGGED TABLE stage_1 AS
SELECT (data->>'id')::bigint event_id,
       (data->>'type') event_type,
       (data->>'public')::boolean event_public,
       (data->'repo'->>'id')::bigint repo_id,
       (data->'payload') payload,
       (data->'actor') actor,
       (data->'org') org,
       (data->>'created_at')::timestamp created_at FROM prepare_1;
```

You can then use the master_append_table_to_shard function to append this staging table to the distributed table.

This approach works especially well when staging data via the workers, since the pre-processing itself can be scaled out by running it on many workers in parallel for different chunks of input data.

CHAPTER 17

Query Performance Tuning

In this section, we describe how you can tune your Citus cluster to get maximum performance. We begin by explaining how choosing the right distribution column and method affect performance. We then describe how you can first tune your database for high performance on one PostgreSQL server and then scale it out across all the CPUs in the cluster. In this section, we also discuss several performance related configuration parameters wherever relevant.

17.1 Table Distribution and Shards

The first step while creating a distributed table is choosing the right distribution column and distribution method. Citus supports both append and hash based distribution; and both are better suited to certain use cases. Also, choosing the right distribution column helps Citus push down several operations directly to the worker shards and prune away unrelated shards which lead to significant query speedups. We discuss briefly about choosing the right distribution column and method below.

Typically, you should pick that column as the distribution column which is the most commonly used join key or on which most queries have filters. For filters, Citus uses the distribution column ranges to prune away unrelated shards, ensuring that the query hits only those shards which overlap with the WHERE clause ranges. For joins, if the join key is the same as the distribution column, then Citus executes the join only between those shards which have matching / overlapping distribution column ranges. All these shard joins can be executed in parallel on the workers and hence are more efficient.

In addition, Citus can push down several operations directly to the worker shards if they are based on the distribution column. This greatly reduces both the amount of computation on each node and the network bandwidth involved in transferring data across nodes.

For distribution methods, Citus supports both append and hash distribution. Append based distribution is more suited to append-only use cases. This typically includes event based data which arrives in a time-ordered series. Users then distribute their largest tables by time, and batch load their events into distributed tables in intervals of N minutes. This data model can be applied to a number of time series use cases; for example, each line in a website's log file, machine activity logs or aggregated website events. In this distribution method, Citus stores min / max ranges of the distribution column in each shard, which allows for more efficient range queries on the distribution column.

Hash based distribution is more suited to cases where users want to do real-time inserts along with analytics on their data or want to distribute by a non-ordered column (eg. user id). This data model is relevant for real-time analytics use

cases; for example, actions in a mobile application, user website events, or social media analytics. This distribution method allows users to perform co-located joins and efficiently run queries involving equality based filters on the distribution column.

Once you choose the right distribution method and column, you can then proceed to the next step, which is tuning single node performance.

17.2 PostgreSQL tuning

The Citus master partitions an incoming query into fragment queries, and sends them to the workers for parallel processing. The workers are just extended PostgreSQL servers and they apply PostgreSQL's standard planning and execution logic for these queries. So, the first step in tuning Citus is tuning the PostgreSQL configuration parameters on the workers for high performance.

This step involves loading a small portion of the data into regular PostgreSQL tables on the workers. These tables would be representative of shards on which your queries would run once the full data has been distributed. Then, you can run the fragment queries on the individual shards and use standard PostgreSQL tuning to optimize query performance.

The first set of such optimizations relates to configuration settings. PostgreSQL by default comes with conservative resource settings; and among these settings, shared_buffers and work_mem are probably the most important ones in optimizing read performance. We discuss these parameters in brief below. Apart from them, several other configuration settings impact query performance. These settings are covered in more detail in the PostgreSQL manual and are also discussed in the PostgreSQL 9.0 High Performance book.

shared_buffers defines the amount of memory allocated to the database for caching data, and defaults to 128MB. If you have a worker node with 1GB or more RAM, a reasonable starting value for shared_buffers is 1/4 of the memory in your system. There are some workloads where even larger settings for shared_buffers are effective, but given the way PostgreSQL also relies on the operating system cache, it's unlikely you'll find using more than 25% of RAM to work better than a smaller amount.

If you do a lot of complex sorts, then increasing work_mem allows PostgreSQL to do larger in-memory sorts which will be faster than disk-based equivalents. If you see lot of disk activity on your worker node inspite of having a decent amount of memory, then increasing work_mem to a higher value can be useful. This will help PostgreSQL in choosing more efficient query plans and allow for greater amount of operations to occur in memory.

Other than the above configuration settings, the PostgreSQL query planner relies on statistical information about the contents of tables to generate good plans. These statistics are gathered when ANALYZE is run, which is enabled by default. You can learn more about the PostgreSQL planner and the ANALYZE command in greater detail in the PostgreSQL documentation.

Lastly, you can create indexes on your tables to enhance database performance. Indexes allow the database to find and retrieve specific rows much faster than it could do without an index. To choose which indexes give the best performance, you can run the query with EXPLAIN to view query plans and optimize the slower parts of the query. After an index is created, the system has to keep it synchronized with the table which adds overhead to data manipulation operations. Therefore, indexes that are seldom or never used in queries should be removed.

For write performance, you can use general PostgreSQL configuration tuning to increase INSERT rates. We commonly recommend increasing checkpoint_timeout and max_wal_size settings. Also, depending on the reliability requirements of your application, you can choose to change fsync or synchronous_commit values.

17.3 Scaling Out Performance

Once you have achieved the desired performance for a single shard, you can set similar configuration parameters on all your workers. As Citus runs all the fragment queries in parallel across the worker nodes, users can scale out

the performance of their queries to be the cumulative of the computing power of all of the CPU cores in the cluster assuming that the data fits in memory.

Users should try to fit as much of their working set in memory as possible to get best performance with Citus. If fitting the entire working set in memory is not feasible, we recommend using SSDs over HDDs as a best practice. This is because HDDs are able to show decent performance when you have sequential reads over contiguous blocks of data, but have significantly lower random read / write performance. In cases where you have a high number of concurrent queries doing random reads and writes, using SSDs can improve query performance by several times as compared to HDDs. Also, if your queries are highly compute intensive, it might be beneficial to choose machines with more powerful CPUs.

To measure the disk space usage of your database objects, you can log into the worker nodes and use PostgreSQL administration functions for individual shards. The pg_total_relation_size() function can be used to get the total disk space used by a table. You can also use other functions mentioned in the PostgreSQL docs to get more specific size information. On the basis of these statistics for a shard and the shard count, users can compute the hardware requirements for their cluster.

Another factor which affects performance is the number of shards per worker node. Citus partitions an incoming query into its fragment queries which run on individual worker shards. Hence, the degree of parallelism for each query is governed by the number of shards the query hits. To ensure maximum parallelism, you should create enough shards on each node such that there is at least one shard per CPU core. Another consideration to keep in mind is that Citus will prune away unrelated shards if the query has filters on the distribution column. So, creating more shards than the number of cores might also be beneficial so that you can achieve greater parallelism even after shard pruning.

17.4 Distributed Query Performance Tuning

Once you have distributed your data across the cluster, with each worker optimized for best performance, you should be able to see high performance gains on your queries. After this, the final step is to tune a few distributed performance tuning parameters.

Before we discuss the specific configuration parameters, we recommend that you measure query times on your distributed cluster and compare them with the single shard performance. This can be done by enabling timing and running the query on the master node and running one of the fragment queries on the worker nodes. This helps in determining the amount of time spent on the worker nodes and the amount of time spent in fetching the data to the master node. Then, you can figure out what the bottleneck is and optimize the database accordingly.

In this section, we discuss the parameters which help optimize the distributed query planner and executors. There are several relevant parameters and we discuss them in two sections:- general and advanced. The general performance tuning section is sufficient for most use-cases and covers all the common configs. The advanced performance tuning section covers parameters which may provide performance gains in specific use cases.

17.4.1 General

For higher INSERT performance, the factor which impacts insert rates the most is the level of concurrency. You should try to run several concurrent INSERT statements in parallel. This way you can achieve very high insert rates if you have a powerful master node and are able to use all the CPU cores on that node together.

Citus has 3 different executor types for running SELECT queries. The desired executor can be selected by setting the citus.task_executor_type configuration parameter. If your use case mainly requires simple key-value lookups, you can choose the router executor. The router executor has a very simple architecture and hence can provide minimum latencies for simple queries. For sub-second responses to aggregations and joins, you should use the real-time executor as much as possible. If there are long running queries which require repartitioning and shuffling of data across the workers, then you can switch to the the task tracker executor.

An important performance tuning parameter in context of SELECT query performance is citus.remote_task_check_interval. The Citus master assigns tasks to workers, and then regularly checks with them about each task's progress. This configuration value sets the time interval between two consequent checks. Setting this parameter to a lower value reduces query times significantly for sub-second queries. For relatively long running queries (which take minutes as opposed to seconds), reducing this parameter might not be ideal as this would make the master contact the workers more often, incurring a higher overhead.

Other than the above, there are two configuration parameters which can be useful in cases where approximations produce meaningful results. These two parameters are citus.limit_clause_row_fetch_count and citus.count_distinct_error_rate. The former sets the number of rows to fetch from each task while calculating limits while the latter sets the desired error rate when calculating approximate distinct counts. You can learn more about the applicability and usage of these parameters in the user guide sections: *Count (Distinct) Aggregates* and *Limit Pushdown*.

17.4.2 Advanced

In this section, we discuss advanced performance tuning parameters. These parameters are applicable to specific use cases and may not be required for all deployments.

Task Assignment Policy

The Citus query planner assigns tasks to the worker nodes based on shard locations. The algorithm used while making these assignments can be chosen by setting the citus.task_assignment_policy configuration parameter. Users can alter this configuration parameter to choose the policy which works best for their use case.

The **greedy** policy aims to distribute tasks evenly across the workers. This policy is the default and works well in most of the cases. The **round-robin** policy assigns tasks to workers in a round-robin fashion alternating between different replicas. This enables much better cluster utilization when the shard count for a table is low compared to the number of workers. The third policy is the **first-replica** policy which assigns tasks on the basis of the insertion order of placements (replicas) for the shards. With this policy, users can be sure of which shards will be accessed on each machine. This helps in providing stronger memory residency guarantees by allowing you to keep your working set in memory and use it for querying.

Intermediate Data Transfer Format

There are two configuration parameters which relate to the format in which intermediate data will be transferred across workers or between workers and the master. Citus by default transfers intermediate query data in the text format. This is generally better as text files typically have smaller sizes than the binary representation. Hence, this leads to lower network and disk I/O while writing and transferring intermediate data.

However, for certain data types like hll or hstore arrays, the cost of serializing and deserializing data is pretty high. In such cases, using binary format for transferring intermediate data can improve query performance due to reduced CPU usage. There are two configuration parameters which can be used to tune this behaviour, citus.binary_master_copy_format and citus.binary_worker_copy_format. Enabling the former uses binary format to transfer intermediate query results from the workers to the master while the latter is useful in queries which require dynamic shuffling of intermediate data between workers.

Real Time Executor

If you have SELECT queries which require sub-second response times, you should try to use the real-time executor.

The real-time executor opens one connection and uses two file descriptors per unpruned shard (Unrelated shards are pruned away during planning). Due to this, the executor may need to open more connections than max_connections or use more file descriptors than max_files_per_process if the query hits a high number of shards.

In such cases, the real-time executor will begin throttling tasks to prevent overwhelming resources on the workers. Since this throttling can reduce query performance, the real-time executor will issue a warning suggesting that max_connections or max_files_per_process should be increased. On seeing these warnings, you should increase the suggested parameters to maintain the desired query performance.

Task Tracker Executor

If your queries require repartitioning of data or more efficient resource management, you should use the task tracker executor. There are two configuration parameters which can be used to tune the task tracker executor's performance.

The first one is the citus.task_tracker_delay. The task tracker process wakes up regularly, walks over all tasks assigned to it, and schedules and executes these tasks. This parameter sets the task tracker sleep time between these task management rounds. Reducing this parameter can be useful in cases when the shard queries are short and hence update their status very regularly.

The second parameter is citus.max_running_tasks_per_node. This configuration value sets the maximum number of tasks to execute concurrently on one worker node node at any given time. This configuration entry ensures that you don't have many tasks hitting disk at the same time and helps in avoiding disk I/O contention. If your queries are served from memory or SSDs, you can increase citus.max_running_tasks_per_node without much concern.

With this, we conclude our discussion about performance tuning in Citus. To learn more about the specific configuration parameters discussed in this section, please visit the *Configuration Reference* section of our documentation.

CHAPTER 18

Cluster Management

In this section, we discuss how you can add or remove nodes from your Citus cluster and how you can deal with node failures.

Note: To make moving shards across nodes or re-replicating shards on failed nodes easier, Citus Enterprise comes with a shard rebalancer extension. We discuss briefly about the functions provided by the shard rebalancer as and when relevant in the sections below. You can learn more about these functions, their arguments and usage, in the *Cluster Management And Repair Functions* reference section.

18.1 Scaling out your cluster

Citus's logical sharding based architecture allows you to scale out your cluster without any down time. This section describes how you can add more nodes to your Citus cluster in order to improve query performance / scalability.

18.1.1 Adding a worker

Citus stores all the data for distributed tables on the worker nodes. Hence, if you want to scale out your cluster by adding more computing power, you can do so by adding a worker.

To add a new node to the cluster, you first need to add the DNS name of that node to the pg_worker_list.conf file in your data directory on the master node.

Next, you can call the pg_reload_conf UDF to cause the master to reload its configuration.

```
select pg_reload_conf();
```

After this point, Citus will automatically start assigning new shards to that worker.

In addition to the above, if you want to move existing shards to the newly added worker, Citus Enterprise provides an additional rebalance_table_shards function to make this easier. This function will move the shards of the given table to make them evenly distributed among the workers.

```
select rebalance_table_shards('github_events');
```

18.1.2 Adding a master

The Citus master only stores metadata about the table shards and does not store any data. This means that all the computation is pushed down to the workers and the master does only final aggregations on the result of the workers. Therefore, it is not very likely that the master becomes a bottleneck for read performance. Also, it is easy to boost up the master by shifting to a more powerful machine.

However, in some write heavy use cases where the master becomes a performance bottleneck, users can add another master. As the metadata tables are small (typically a few MBs in size), it is possible to copy over the metadata onto another node and sync it regularly. Once this is done, users can send their queries to any master and scale out performance. If your setup requires you to use multiple masters, please contact us at engage@citusdata.com.

18.2 Dealing With Node Failures

In this sub-section, we discuss how you can deal with node failures without incurring any downtime on your Citus cluster. We first discuss how Citus handles worker failures automatically by maintaining multiple replicas of the data. We also briefly describe how users can replicate their shards to bring them to the desired replication factor in case a node is down for a long time. Lastly, we discuss how you can setup redundancy and failure handling mechanisms for the master.

18.2.1 Worker Node Failures

Citus can easily tolerate worker node failures because of its logical sharding-based architecture. While loading data, Citus allows you to specify the replication factor to provide desired availability for your data. In face of worker node failures, Citus automatically switches to these replicas to serve your queries. It also issues warnings like below on the master so that users can take note of node failures and take actions accordingly.

```
WARNING:  could not connect to node localhost:9700
```

On seeing such warnings, the first step would be to remove the failed worker from the pg_worker_list.conf file in the data directory.

```
vi $PGDATA/pg_worker_list.conf
```

Note: The instruction above assumes that the data directory is in the PGDATA environment variable. If not, you will need to set it. For example:

```
export PGDATA=/usr/lib/postgresql/9.5/data
```

Then, you can reload the configuration so that the master picks up the desired configuration changes.

```
SELECT pg_reload_conf();
```

After this step, Citus will stop assigning tasks or storing data on the failed node. Then, you can log into the failed node and inspect the cause of the failure.

Once you remove the failed worker from pg_worker_list.conf, Citus will then automatically re-route the work to the healthy workers. Also, if Citus is not able to connect to a worker, it will assign that task to another node having a copy

of that shard. If the failure occurs mid-query, Citus does not re-run the whole query but assigns only the failed query fragments leading to faster responses in face of failures.

Once the node is back up, you can add it to the pg_worker_list.conf and reload the configuration. If you want to add a new node to the cluster to replace the failed node, you can follow the instructions described in the *Adding a worker* section.

While the node is down, you may wish to retain the same level of replication so that your application can tolerate more failures. To make this simpler, Citus enterprise provides a replicate_table_shards UDF which can be called after removing the failed worker from pg_worker_list.conf. This function copies the shards of a table across the healthy nodes so they all reach the configured replication factor.

```
select replicate_table_shards('github_events');
```

18.2.2 Master Node Failures

The Citus master maintains metadata tables to track all of the cluster nodes and the locations of the database shards on those nodes. The metadata tables are small (typically a few MBs in size) and do not change very often. This means that they can be replicated and quickly restored if the node ever experiences a failure. There are several options on how users can deal with master failures.

1. **Use PostgreSQL streaming replication:** You can use PostgreSQL's streaming replication feature to create a hot standby of the master. Then, if the primary master node fails, the standby can be promoted to the primary automatically to serve queries to your cluster. For details on setting this up, please refer to the PostgreSQL wiki.

2. Since the metadata tables are small, users can use EBS volumes, or PostgreSQL backup tools to backup the metadata. Then, they can easily copy over that metadata to new nodes to resume operation.

3. Citus's metadata tables are simple and mostly contain text columns which are easy to understand. So, in case there is no failure handling mechanism in place for the master node, users can dynamically reconstruct this metadata from shard information available on the worker nodes. To learn more about the metadata tables and their schema, you can visit the *Metadata Tables Reference* section of our documentation.

CHAPTER 19

Upgrading to Citus 5

This section describes how you can upgrade your existing Citus installation to Citus 5.0.

If you are upgrading from CitusDB 4.0 to Citus 5.0, you can use the standard PostgreSQL pg_upgrade utility. pg_upgrade uses the fact that the on-disk representation of the data has probably not changed, and copies over the disk files as is, thus making the upgrade process faster.

Citus 5.0 is not a standalone application but a PostgreSQL extension. Therefore, you should first install PostgreSQL 9.5 before installing Citus 5.0.

Apart from running pg_upgrade, there are 3 manual steps to be accounted for in order to update Citus.

1. Create and configure Citus extension.
2. Copy over Citus medadata tables.
3. Set pg_dist_shardid_seq current sequence to max shard value.

The others are known pg_upgrade manual steps, i.e. manually updating configuration files, pg_hba etc.

We discuss the step by step process of upgrading the cluster below. Please note that you need to run the steps on all the nodes in the cluster. Some steps need to be run only on the master and they are explicitly marked as such.

1. Download and install Citus 5.0 on the server having the to-be-upgraded 4.0 data directory

Download and install PostgreSQL 9.5.x from http://www.postgresql.org/download/.

Note: The instructions below assume that the PostgreSQL installation is in your path. If not, you will need to add it to your PATH environment variable. For example:

```
export PATH=/usr/lib/postgresql/9.5/:$PATH
```

You can then download and install the new Citus packages from our Downloads page. Please visit the *Multi-Machine Cluster* section for specific instructions.

Note that the Citus 5.0 extension will be installed at the PostgreSQL install location.

2. Setup environment variables for the data directories

Please set appropriate values for data location like below. This makes accessing data directories in subsequent commands easier.

```
export PGDATA5_0=/usr/lib/postgresql/9.5/data
export PGDATA4_0=/opt/citusdb/4.0/data
```

3. Stop loading data on to that instance

If you are upgrading the master, then you should stop all data-loading/appending and staging before copying out the metadata. If data-loading continues after step 4 below, then the metadata will be out of date.

4. Copy out pg_dist catalog metadata from the 4.0 server (Only needed for master)

```
COPY pg_dist_partition TO '/var/tmp/pg_dist_partition.data';
COPY pg_dist_shard TO '/var/tmp/pg_dist_shard.data';
COPY pg_dist_shard_placement TO '/var/tmp/pg_dist_shard_placement.data';
```

5. Initialize a new data directory for 5.0

```
initdb -D $PGDATA5_0
```

Note: On some platforms, PostgreSQL creates a data directory by default during installation. You can ignore this step if you want to use that data directory.

6. Check upgrade compatibility

```
pg_upgrade -b /opt/citusdb/4.0/bin/ -B /usr/lib/postgresql/9.5/bin/ -d $PGDATA4_0 -D
↪$PGDATA5_0 --check
```

This should return **Clusters are compatible**. If this doesn't return that message, you need to stop and check what the error is.

Note: This may return the following warning if the 4.0 server has not been stopped. This warning is OK:

```
*failure*
Consult the last few lines of "pg_upgrade_server.log" for the probable cause of the␣
↪failure.
```

7. Shutdown the running 4.0 server

```
/opt/citusdb/4.0/bin/pg_ctl stop -D $PGDATA4_0
```

8. Run pg_upgrade, remove the –check flag

```
pg_upgrade -b /opt/citusdb/4.0/bin/ -B /usr/lib/postgresql/9.5/bin/ -d $PGDATA4_0 -D
↪$PGDATA5_0
```

9. Copy over pg_worker_list.conf (Only needed for master)

```
cp $PGDATA4_0/pg_worker_list.conf $PGDATA5_0/pg_worker_list.conf
```

10. Re-do changes to config settings in postgresql.conf and pg_hba.conf in the 5.0 data directory

- listen_addresses
- performance tuning parameters
- enabling connections

- Copy all non-default settings below **DISTRIBUTED DATABASE** section (eg. shard_max_size, shard_replication_factor etc) to the end of the new postgresql.conf. Also note that for usage with Citus 5.0, each setting name must be prefixed with "citus". i.e. **shard_max_size** becomes **citus.shard_max_size**.

11. Add citus to shared_preload_libraries in postgresql.conf

```
vi $PGDATA5_0/postgresql.conf
```

```
shared_preload_libraries = 'citus'
```

Make sure **citus** is the first extension if there are more extensions you want to add there.

12. Start the new 5.0 server

```
pg_ctl -D $PGDATA5_0 start
```

13. Connect to postgresql instance and create citus extension

```
psql -d postgres -h localhost
create extension citus;
```

14. Copy over pg_dist catalog tables to the new server using the PostgreSQL 9.5.x psql client (Only needed for master)

```
psql -d postgres -h localhost
COPY pg_dist_partition FROM '/var/tmp/pg_dist_partition.data';
COPY pg_dist_shard FROM '/var/tmp/pg_dist_shard.data';
COPY pg_dist_shard_placement FROM '/var/tmp/pg_dist_shard_placement.data';
```

15. Restart the sequence pg_dist_shardid_seq (Only needed for master)

```
SELECT setval('pg_catalog.pg_dist_shardid_seq', (SELECT MAX(shardid) AS max_shard_id␣
↪FROM pg_dist_shard)+1, false);
```

This is needed since the sequence value doesn't get copied over. So we restart the sequence from the largest shardid (+1 to avoid collision). This will come into play when staging data, not when querying data.

If you are using hash distributed tables, then this step may return an error :

```
ERROR:  setval: value 100** is out of bounds for sequence "pg_dist_shardid_seq"␣
↪(102008..9223372036854775807)
```

You can ignore this error and continue with the process below.

16. Ready to run queries/create tables/load data

At this step, you have successfully completed the upgrade process. You can run queries, create new tables or add data to existing tables. Once everything looks good, the old 4.0 data directory can be deleted.

Running in a mixed mode

For users who don't want to take a cluster down and upgrade all nodes at the same time, there is the possibility of running in a mixed 4.0 / 5.0 mode. To do so, you can first upgrade the master. Then, you can upgrade the workers one at a time. This way you can upgrade the cluster with no downtime. However, we recommend using 5.0 version in whole cluster.

CHAPTER 20

Transitioning From PostgreSQL to Citus

As Citus is a PostgreSQL extension, PostgreSQL users can start using Citus by simply installing the extension on their existing PostgreSQL database. Once you create the extension, you can create and use distributed tables through standard PostgreSQL interfaces while maintaining compatibility with existing PostgreSQL tools. Please look at *Working with distributed tables* for specific instructions.

To move your data from a PostgreSQL table to a distributed table, you can copy out the data into a csv file and then use the stage command or the copy script to load it into a distributed table. Citus also provides the *master_append_table_to_shard* function for users who want to append their PostgreSQL tables to distributed tables directly. One thing to note as you transition from a single node to multiple nodes is that you should create your extensions, operators, user defined functions, and custom data types on all nodes.

If you have any questions or require assistance in scaling out your existing PostgreSQL installation, please get in touch with us at engage@citusdata.com.

CHAPTER 21

Citus SQL Language Reference

As Citus provides distributed functionality by extending PostgreSQL, it is compatible with PostgreSQL constructs. This means that users can use the tools and features that come with the rich and extensible PostgreSQL ecosystem for distributed tables created with Citus. These features include but are not limited to :-

- support for wide range of data types (including support for semi-structured data types like jsonb, hstore)
- full text search
- operators and functions
- foreign data wrappers
- extensions

To learn more about PostgreSQL and its features, you can visit the PostgreSQL 9.5 documentation.

For a detailed reference of the PostgreSQL SQL command dialect (which can be used as is by Citus users), you can see the SQL Command Reference.

Note: PostgreSQL has a wide SQL coverage and Citus may not support the entire SQL spectrum out of the box for distributed tables. We aim to continuously improve Citus's SQL coverage in the upcoming releases. In the mean time, if you have a use case which requires support for these constructs, please get in touch with us by dropping a note to engage@citusdata.com.

CHAPTER 22

User Defined Functions Reference

This section contains reference information for the User Defined Functions provided by Citus. These functions help in providing additional distributed functionality to Citus other than the standard SQL commands.

22.1 Table and Shard DDL

22.1.1 master_create_distributed_table

The master_create_distributed_table() function is used to define a distributed table. This function takes in a table name, the distribution column and distribution method and inserts appropriate metadata to mark the table as distributed.

Arguments

table_name: Name of the table which needs to be distributed.

distribution_column: The column on which the table is to be distributed.

distribution_method: The method according to which the table is to be distributed. Permissible values are append, hash or range.

Return Value

N/A

Example

This example informs the database that the github_events table should be distributed by hash on the repo_id column.

```
SELECT master_create_distributed_table('github_events', 'repo_id', 'hash');
```

22.1.2 master_create_worker_shards

The master_create_worker_shards() function creates a specified number of worker shards with the desired replication factor for a *hash* distributed table. While doing so, the function also assigns a portion of the hash token space (which spans between -2 Billion and 2 Billion) to each shard. Once all shards are created, this function saves all distributed metadata on the master.

Arguments

table_name: Name of hash distributed table for which shards are to be created.

shard_count: Number of shards to create.

replication_factor: Desired replication factor for each shard.

Return Value

N/A

Example

This example usage would create a total of 16 shards for the github_events table where each shard owns a portion of a hash token space and gets replicated on 2 workers.

```
SELECT master_create_worker_shards('github_events', 16, 2);
```

22.1.3 master_create_empty_shard

The master_create_empty_shard() function can be used to create an empty shard for an *append* distributed table. Behind the covers, the function first selects shard_replication_factor workers to create the shard on. Then, it connects to the workers and creates empty placements for the shard on the selected workers. Finally, the metadata is updated for these placements on the master to make these shards visible to future queries. The function errors out if it is unable to create the desired number of shard placements.

Arguments

table_name: Name of the append distributed table for which the new shard is to be created.

Return Value

shard_id: The function returns the unique id assigned to the newly created shard.

Example

This example creates an empty shard for the github_events table. The shard id of the created shard is 102089.

```
SELECT * from master_create_empty_shard('github_events');
 master_create_empty_shard
---------------------------
                102089
(1 row)
```

22.1.4 master_append_table_to_shard

The master_append_table_to_shard() function can be used to append a PostgreSQL table's contents to a shard of an *append* distributed table. Behind the covers, the function connects to each of the workers which have a placement of that shard and appends the contents of the table to each of them. Then, the function updates metadata for the shard placements on the basis of whether the append succeeded or failed on each of them.

If the function is able to successfully append to at least one shard placement, the function will return successfully. It will also mark any placement to which the append failed as INACTIVE so that any future queries do not consider that placement. If the append fails for all placements, the function quits with an error (as no data was appended). In this case, the metadata is left unchanged.

Arguments

shard_id: Id of the shard to which the contents of the table have to be appended.

source_table_name: Name of the PostgreSQL table whose contents have to be appended.

source_node_name: DNS name of the node on which the source table is present ("source" node).

source_node_port: The port on the source worker node on which the database server is listening.

Return Value

shard_fill_ratio: The function returns the fill ratio of the shard which is defined as the ratio of the current shard size to the configuration parameter shard_max_size.

Example

This example appends the contents of the github_events_local table to the shard having shard id 102089. The table github_events_local is present on the database running on the node master-101 on port number 5432. The function returns the ratio of the the current shard size to the maximum shard size, which is 0.1 indicating that 10% of the shard has been filled.

```
SELECT * from master_append_table_to_shard(102089,'github_events_local','master-101',␣
↪5432);
 master_append_table_to_shard
------------------------------
                  0.100548
(1 row)
```

22.1.5 master_apply_delete_command

The master_apply_delete_command() function is used to delete shards which match the criteria specified by the delete command. This function deletes a shard only if all rows in the shard match the delete criteria. As the function uses

shard metadata to decide whether or not a shard needs to be deleted, it requires the WHERE clause in the DELETE statement to be on the distribution column. If no condition is specified, then all shards of that table are deleted.

Behind the covers, this function connects to all the worker nodes which have shards matching the delete criteria and sends them a command to drop the selected shards. Then, the function updates the corresponding metadata on the master. If the function is able to successfully delete a shard placement, then the metadata for it is deleted. If a particular placement could not be deleted, then it is marked as TO DELETE. The placements which are marked as TO DELETE are not considered for future queries and can be cleaned up later.

Arguments

delete_command: valid SQL DELETE command

Return Value

deleted_shard_count: The function returns the number of shards which matched the criteria and were deleted (or marked for deletion). Note that this is the number of shards and not the number of shard placements.

Example

The first example deletes all the shards for the github_events table since no delete criteria is specified. In the second example, only the shards matching the criteria (3 in this case) are deleted.

```
SELECT * from master_apply_delete_command('DELETE FROM github_events');
 master_apply_delete_command
-----------------------------
                           5
(1 row)

SELECT * from master_apply_delete_command('DELETE FROM github_events WHERE review_
↪date < ''2009-03-01''');
 master_apply_delete_command
-----------------------------
                           3
(1 row)
```

22.2 Metadata / Configuration Information

22.2.1 master_get_active_worker_nodes

The master_get_active_worker_nodes() function returns a list of active worker host names and port numbers. Currently, the function assumes that all the worker nodes in pg_worker_list.conf are active.

Arguments

N/A

Return Value

List of tuples where each tuple contains the following information:

node_name: DNS name of the worker node

node_port: Port on the worker node on which the database server is listening

Example

```
SELECT * from master_get_active_worker_nodes();
 node_name | node_port
-----------+-----------
 localhost |      9700
 localhost |      9702
 localhost |      9701

(3 rows)
```

22.2.2 master_get_table_metadata

The master_get_table_metadata() function can be used to return distribution related metadata for a distributed table. This metadata includes the relation id, storage type, distribution method, distribution column, replication count, maximum shard size and the shard placement policy for that table. Behind the covers, this function queries Citus metadata tables to get the required information and concatenates it into a tuple before returning it to the user.

Arguments

table_name: Name of the distributed table for which you want to fetch metadata.

Return Value

A tuple containing the following information:

logical_relid: Oid of the distributed table. This values references the relfilenode column in the pg_class system catalog table.

part_storage_type: Type of storage used for the table. May be 't' (standard table), 'f' (foreign table) or 'c' (columnar table).

part_method: Distribution method used for the table. May be 'a' (append), 'h' (hash) or 'r' (range).

part_key: Distribution column for the table.

part_replica_count: Current shard replication count.

part_max_size: Current maximum shard size in bytes.

part_placement_policy: Shard placement policy used for placing the table's shards. May be 1 (local-node-first) or 2 (round-robin).

Example

The example below fetches and displays the table metadata for the github_events table.

```
SELECT * from master_get_table_metadata('github_events');
 logical_relid | part_storage_type | part_method | part_key | part_replica_count |␣
↪part_max_size | part_placement_policy
---------------+-------------------+-------------+----------+--------------------+----
↪-----------+------------------------
         24180 | t                 | h           | repo_id  |                  2 |  ␣
↪1073741824 |                      2
(1 row)
```

22.3 Cluster Management And Repair Functions

22.3.1 master_copy_shard_placement

If a shard placement fails to be updated during a modification command or a DDL operation, then it gets marked as inactive. The master_copy_shard_placement function can then be called to repair an inactive shard placement using data from a healthy placement.

To repair a shard, the function first drops the unhealthy shard placement and recreates it using the schema on the master. Once the shard placement is created, the function copies data from the healthy placement and updates the metadata to mark the new shard placement as healthy. This function ensures that the shard will be protected from any concurrent modifications during the repair.

Arguments

shard_id: Id of the shard to be repaired.

source_node_name: DNS name of the node on which the healthy shard placement is present ("source" node).

source_node_port: The port on the source worker node on which the database server is listening.

target_node_name: DNS name of the node on which the invalid shard placement is present ("target" node).

target_node_port: The port on the target worker node on which the database server is listening.

Return Value

N/A

Example

The example below will repair an inactive shard placement of shard 12345 which is present on the database server running on 'bad_host' on port 5432. To repair it, it will use data from a healthy shard placement present on the server running on 'good_host' on port 5432.

```
SELECT master_copy_shard_placement(12345, 'good_host', 5432, 'bad_host', 5432);
```

22.3.2 rebalance_table_shards

Note: The rebalance_table_shards function is a part of Citus Enterprise. Please contact engage@citusdata.com to obtain this functionality.

The rebalance_table_shards() function moves shards of the given table to make them evenly distributed among the workers. The function first calculates the list of moves it needs to make in order to ensure that the cluster is balanced within the given threshold. Then, it moves shard placements one by one from the source node to the destination node and updates the corresponding shard metadata to reflect the move.

Arguments

table_name: The name of the table whose shards need to be rebalanced.

threshold: (Optional) A float number between 0.0 and 1.0 which indicates the maximum difference ratio of node utilization from average utilization. For example, specifying 0.1 will cause the shard rebalancer to attempt to balance all nodes to hold the same number of shards $\pm 10\%$. Specifically, the shard rebalancer will try to converge utilization of all worker nodes to the (1 - threshold) * average_utilization ... (1 + threshold) * average_utilization range.

max_shard_moves: (Optional) The maximum number of shards to move.

excluded_shard_list: (Optional) Identifiers of shards which shouldn't be moved during the rebalance operation.

Return Value

N/A

Example

The example below will attempt to rebalance the shards of the github_events table within the default threshold.

```
SELECT rebalance_table_shards('github_events');
```

This example usage will attempt to rebalance the github_events table without moving shards with id 1 and 2.

```
SELECT rebalance_table_shards('github_events', excluded_shard_list:='{1,2}');
```

22.3.3 replicate_table_shards

Note: The replicate_table_shards function is a part of Citus Enterprise. Please contact engage@citusdata.com to obtain this functionality.

The replicate_table_shards() function replicates the under-replicated shards of the given table. The function first calculates the list of under-replicated shards and locations from which they can be fetched for replication. The function then copies over those shards and updates the corresponding shard metadata to reflect the copy.

Arguments

table_name: The name of the table whose shards need to be replicated.

shard_replication_factor: (Optional) The desired replication factor to achieve for each shard.

max_shard_copies: (Optional) Maximum number of shards to copy to reach the desired replication factor.

excluded_shard_list: (Optional) Identifiers of shards which shouldn't be copied during the replication operation.

Return Value

N/A

Examples

The example below will attempt to replicate the shards of the github_events table to shard_replication_factor.

```
SELECT replicate_table_shards('github_events');
```

This example will attempt to bring the shards of the github_events table to the desired replication factor with a maximum of 10 shard copies. This means that the rebalancer will copy only a maximum of 10 shards in its attempt to reach the desired replication factor.

```
SELECT replicate_table_shards('github_events', max_shard_copies:=10);
```

CHAPTER 23

Metadata Tables Reference

Citus divides each distributed table into multiple logical shards based on the distribution column. The master then maintains metadata tables to track statistics and information about the health and location of these shards. In this section, we describe each of these metadata tables and their schema. You can view and query these tables using SQL after logging into the master node.

23.1 Partition table

The pg_dist_partition table stores metadata about which tables in the database are distributed. For each distributed table, it also stores information about the distribution method and detailed information about the distribution column.

Name	Type	Description
logicalrelid	oid	Distributed table to which this row corresponds. This value references the relfilenode column in the pg_class system catalog table.
partmethod	char	The method used for partitioning / distribution. The values of this column corresponding to different distribution methods are :- append: 'a' hash: 'h' range: 'r'
partkey	text	Detailed information about the distribution column including column number, type and other relevant information.

```
SELECT * from pg_dist_partition;
 logicalrelid | partmethod |                                                        ␣
↪partkey
--------------+------------+------------------------------------------------------------
↪-----------------------------------------------------------
       488843 |      r     | {VAR :varno 1 :varattno 4 :vartype 20 :vartypmod -1␣
↪:varcollid 0 :varlevelsup 0 :varnoold 1 :varoattno 4 :location 232}

(1 row)
```

23.2 Shard table

The pg_dist_shard table stores metadata about individual shards of a table. This includes information about which distributed table the shard belongs to and statistics about the distribution column for that shard. For append distributed tables, these statistics correspond to min / max values of the distribution column. In case of hash distributed tables, they are hash token ranges assigned to that shard. These statistics are used for pruning away unrelated shards during SELECT queries.

Name	Type	Description
logicalrelid	oid	Distributed table to which this shard belongs. This value references the relfilenode column in the pg_class system catalog table.
shardid	bigint	Globally unique identifier assigned to this shard.
shardstorage	char	Type of storage used for this shard. Different storage types are discussed in the table below.
shardalias	text	Determines the name used on the worker PostgreSQL database to refer to this shard. If NULL, the default name is "tablename_shardid".
shardminvalue	text	For append distributed tables, minimum value of the distribution column in this shard (inclusive). For hash distributed tables, minimum hash token value assigned to that shard (inclusive).
shardmaxvalue	text	For append distributed tables, maximum value of the distribution column in this shard (inclusive). For hash distributed tables, maximum hash token value assigned to that shard (inclusive).

```
SELECT * from pg_dist_shard;
 logicalrelid | shardid | shardstorage | shardalias | shardminvalue | shardmaxvalue
--------------+---------+--------------+------------+---------------+---------------
```

(continues on next page)

(continued from previous page)

```
       488843 |  102065 | t            |            | 27           | 14995004
       488843 |  102066 | t            |            | 15001035     | 25269705
       488843 |  102067 | t            |            | 25273785     | 28570113
       488843 |  102068 | t            |            | 28570150     | 28678869

(4 rows)
```

23.2.1 Shard Storage Types

The shardstorage column in pg_dist_shard indicates the type of storage used for the shard. A brief overview of different shard storage types and their representation is below.

Storage Type	Shardstorage value	Description
TABLE	't'	Indicates that shard stores data belonging to a regular distributed table.
COLUMNAR	'c'	Indicates that shard stores columnar data. (Used by distributed cstore_fdw tables)
FOREIGN	'f'	Indicates that shard stores foreign data. (Used by distributed file_fdw tables)

23.3 Shard placement table

The pg_dist_shard_placement table tracks the location of shard replicas on worker nodes. Each replica of a shard assigned to a specific node is called a shard placement. This table stores information about the health and location of each shard placement.

Name	Type	Description
shardid	bigint	Shard identifier associated with this placement. This values references the shardid column in the pg_dist_shard catalog table.
shardstate	int	Describes the state of this placement. Different shard states are discussed in the section below.
shardlength	bigint	For append distributed tables, the size of the shard placement on the worker node in bytes. For hash distributed tables, zero.
nodename	text	DNS host name of the worker node PostgreSQL server hosting this shard placement.
nodeport	int	Port number on which the worker node PostgreSQL server hosting this shard placement is listening.

```
SELECT * from pg_dist_shard_placement;
 shardid | shardstate | shardlength | nodename  | nodeport
---------+------------+-------------+-----------+----------
  102065 |          1 |     7307264 | localhost |   9701
  102065 |          1 |     7307264 | localhost |   9700
  102066 |          1 |     5890048 | localhost |   9700
  102066 |          1 |     5890048 | localhost |   9701
  102067 |          1 |     5242880 | localhost |   9701
  102067 |          1 |     5242880 | localhost |   9700
  102068 |          1 |     3923968 | localhost |   9700
  102068 |          1 |     3923968 | localhost |   9701

(8 rows)
```

23.3.1 Shard Placement States

Citus manages shard health on a per-placement basis and automatically marks a placement as unavailable if leaving the placement in service would put the cluster in an inconsistent state. The shardstate column in the pg_dist_shard_placement table is used to store the state of shard placements. A brief overview of different shard placement states and their representation is below.

State name	Shardstate value	Description
FINALIZED	1	This is the state new shards are created in. Shard placements in this state are considered up-to-date and are used in query planning and execution.
INACTIVE	3	Shard placements in this state are considered inactive due to being out-of-sync with other replicas of the same shard. This can occur when an append, modification (INSERT, UPDATE or DELETE) or a DDL operation fails for this placement. The query planner will ignore placements in this state during planning and execution. Users can synchronize the data in these shards with a finalized replica as a background activity.
TO_DELETE	4	If Citus attempts to drop a shard placement in response to a master_apply_delete_command call and fails, the placement is moved to this state. Users can then delete these shards as a subsequent background activity.

CHAPTER 24

Configuration Reference

There are various configuration parameters that affect the behaviour of Citus. These include both standard PostgreSQL parameters and Citus specific parameters. To learn more about PostgreSQL configuration parameters, you can visit the run time configuration section of PostgreSQL documentation.

The rest of this reference aims at discussing Citus specific configuration parameters. These parameters can be set similar to PostgreSQL parameters by modifying postgresql.conf or by using the SET command.

24.1 Node configuration

24.1.1 pg_worker_list.conf

The Citus master needs to have information about the worker nodes in the cluster so that it can communicate with them. This information is stored in the pg_worker_list.conf file in the data directory on the master. To add this information, you need to append the DNS names and port numbers of the workers to this file. You can then call pg_reload_conf() or restart the master to allow it to refresh its worker membership list.

The example below adds worker-101 and worker-102 as worker nodes in the pg_worker_list.conf file on the master.

```
vi $PGDATA/pg_worker_list.conf
# HOSTNAME      [PORT]  [RACK]
worker-101
worker-102
```

24.1.2 citus.max_worker_nodes_tracked (integer)

Citus tracks worker nodes' locations and their membership in a shared hash table on the master node. This configuration value limits the size of the hash table, and consequently the number of worker nodes that can be tracked. The default for this setting is 2048. This parameter can only be set at server start and is effective on the master node.

24.2 Data Loading

24.2.1 citus.shard_replication_factor (integer)

Sets the replication factor for shards i.e. the number of nodes on which shards will be placed and defaults to 2. This parameter can be set at run-time and is effective on the master. The ideal value for this parameter depends on the size of the cluster and rate of node failure. For example, you may want to increase this replication factor if you run large clusters and observe node failures on a more frequent basis.

24.2.2 citus.shard_max_size (integer)

Sets the maximum size to which a shard will grow before it gets split and defaults to 1GB. When the source file's size (which is used for staging) for one shard exceeds this configuration value, the database ensures that a new shard gets created. This parameter can be set at run-time and is effective on the master.

24.2.3 citus.shard_placement_policy (enum)

Sets the policy to use when choosing nodes for placing newly created shards. When using the stage command, the master needs to choose the worker nodes on which it will place the new shards. This configuration value is applicable on the master and specifies the policy to use for selecting these nodes. The supported values for this parameter are :-

- **round-robin:** The round robin policy is the default and aims to distribute shards evenly across the cluster by selecting nodes in a round-robin fashion. This allows you to stage from any node including the master node.
- **local-node-first:** The local node first policy places the first replica of the shard on the client node from which the stage command is being run. As the master node does not store any data, the policy requires that the command be run from a worker node. As the first replica is always placed locally, it provides better shard placement guarantees.

24.3 Planner Configuration

24.3.1 citus.large_table_shard_count (integer)

Sets the shard count threshold over which a table is considered large and defaults to 4. This criteria is then used in picking a table join order during distributed query planning. This value can be set at run-time and is effective on the master.

24.3.2 citus.limit_clause_row_fetch_count (integer)

Sets the number of rows to fetch per task for limit clause optimization. In some cases, select queries with limit clauses may need to fetch all rows from each task to generate results. In those cases, and where an approximation would produce meaningful results, this configuration value sets the number of rows to fetch from each shard. Limit approximations are disabled by default and this parameter is set to -1. This value can be set at run-time and is effective on the master.

24.3.3 citus.count_distinct_error_rate (floating point)

Citus can calculate count(distinct) approximates using the postgresql-hll extension. This configuration entry sets the desired error rate when calculating count(distinct). 0.0, which is the default, disables approximations for count(distinct); and 1.0 provides no guarantees about the accuracy of results. We recommend setting this parameter to 0.005 for best results. This value can be set at run-time and is effective on the master.

24.3.4 citus.task_assignment_policy (enum)

Sets the policy to use when assigning tasks to workers. The master assigns tasks to workers based on shard locations. This configuration value specifies the policy to use when making these assignments. Currently, there are three possible task assignment policies which can be used.

- **greedy:** The greedy policy is the default and aims to evenly distribute tasks across workers.
- **round-robin:** The round-robin policy assigns tasks to workers in a round-robin fashion alternating between different replicas. This enables much better cluster utilization when the shard count for a table is low compared to the number of workers.
- **first-replica:** The first-replica policy assigns tasks on the basis of the insertion order of placements (replicas) for the shards. In other words, the fragment query for a shard is simply assigned to the worker which has the first replica of that shard. This method allows you to have strong guarantees about which shards will be used on which nodes (i.e. stronger memory residency guarantees).

This parameter can be set at run-time and is effective on the master.

24.4 Intermediate Data Transfer Format

24.4.1 citus.binary_worker_copy_format (boolean)

Use the binary copy format to transfer intermediate data between workers. During large table joins, Citus may have to dynamically repartition and shuffle data between different workers. By default, this data is transferred in text format. Enabling this parameter instructs the database to use PostgreSQL's binary serialization format to transfer this data. This parameter is effective on the workers and needs to be changed in the postgresql.conf file. After editing the config file, users can send a SIGHUP signal or restart the server for this change to take effect.

24.4.2 citus.binary_master_copy_format (boolean)

Use the binary copy format to transfer data between master and the workers. When running distributed queries, the workers transfer their intermediate results to the master for final aggregation. By default, this data is transferred in text format. Enabling this parameter instructs the database to use PostgreSQL's binary serialization format to transfer this data. This parameter can be set at runtime and is effective on the master.

24.5 Executor Configuration

24.5.1 citus.all_modifications_commutative

Citus enforces commutativity rules and acquires appropriate locks for modify operations in order to guarantee correctness of behavior. For example, it assumes that an INSERT statement commutes with another INSERT statement, but not with an UPDATE or DELETE statement. Similarly, it assumes that an UPDATE or DELETE statement does not

commute with another UPDATE or DELETE statement. This means that UPDATEs and DELETEs require Citus to acquire stronger locks.

If you have UPDATE statements that are commutative with your INSERTs or other UPDATEs, then you can relax these commutativity assumptions by setting this parameter to true. When this parameter is set to true, all commands are considered commutative and claim a shared lock, which can improve overall throughput. This parameter can be set at runtime and is effective on the master.

24.5.2 citus.remote_task_check_interval (integer)

Sets the frequency at which Citus checks for job statuses and defaults to 10ms. The master assigns tasks to workers, and then regularly checks with them about each task's progress. This configuration value sets the time interval between two consequent checks. This parameter is effective on the master and can be set at runtime.

The ideal value of remote_task_check_interval depends on the workload. If your queries take a few seconds on average, then reducing this value makes sense. On the other hand, if an average query over a shard takes minutes as opposed to seconds then reducing this value may not be ideal. This would make the master contact each worker more frequently, which is an overhead you may not want to pay in this case.

24.5.3 citus.task_executor_type (enum)

Citus has 3 different executor types for running distributed SELECT queries. The desired executor can be selected by setting this configuration parameter. The accepted values for this parameter are:

- **router:** The router executor simply routes the incoming select query to a single target shard. It is optimal for key-value lookups requiring swift responses.
- **real-time:** The real-time executor is the default executor and is optimal when you require fast responses to queries that involve aggregations and colocated joins spanning across multiple shards.
- **task-tracker:** The task-tracker executor is well suited for long running, complex queries which require shuffling of data across worker nodes and efficient resource management.

This parameter can be set at run-time and is effective on the master. For more details about the executors, you can visit the *Distributed Query Executor* section of our documentation.

24.5.4 Real-time executor configuration

The Citus query planner first prunes away the shards unrelated to a query and then hands the plan over to the real-time executor. For executing the plan, the real-time executor opens one connection and uses two file descriptors per unpruned shard. If the query hits a high number of shards, then the executor may need to open more connections than max_connections or use more file descriptors than max_files_per_process.

In such cases, the real-time executor will begin throttling tasks to prevent overwhelming the worker resources. Since this throttling can reduce query performance, the real-time executor will issue an appropriate warning suggesting that increasing these parameters might be required to maintain the desired performance. These parameters are discussed in brief below.

max_connections (integer)

Sets the maximum number of concurrent connections to the database server. The default is typically 100 connections, but might be less if your kernel settings will not support it (as determined during initdb). The real time executor maintains an open connection for each shard to which it sends queries. Increasing this configuration parameter will

allow the executor to have more concurrent connections and hence handle more shards in parallel. This parameter has to be changed on the workers as well as the master, and can be done only during server start.

max_files_per_process (integer)

Sets the maximum number of simultaneously open files for each server process and defaults to 1000. The real-time executor requires two file descriptors for each shard it sends queries to. Increasing this configuration parameter will allow the executor to have more open file descriptors, and hence handle more shards in parallel. This change has to be made on the workers as well as the master, and can be done only during server start.

Note: Along with max_files_per_process, one may also have to increase the kernel limit for open file descriptors per process using the ulimit command.

24.5.5 Task tracker executor configuration

citus.task_tracker_delay (integer)

This sets the task tracker sleep time between task management rounds and defaults to 200ms. The task tracker process wakes up regularly, walks over all tasks assigned to it, and schedules and executes these tasks. Then, the task tracker sleeps for a time period before walking over these tasks again. This configuration value determines the length of that sleeping period. This parameter is effective on the workers and needs to be changed in the postgresql.conf file. After editing the config file, users can send a SIGHUP signal or restart the server for the change to take effect.

This parameter can be decreased to trim the delay caused due to the task tracker executor by reducing the time gap between the management rounds. This is useful in cases when the shard queries are very short and hence update their status very regularly.

citus.max_tracked_tasks_per_node (integer)

Sets the maximum number of tracked tasks per node and defaults to 1024. This configuration value limits the size of the hash table which is used for tracking assigned tasks, and therefore the maximum number of tasks that can be tracked at any given time. This value can be set only at server start time and is effective on the workers.

This parameter would need to be increased if you want each worker node to be able to track more tasks. If this value is lower than what is required, Citus errors out on the worker node saying it is out of shared memory and also gives a hint indicating that increasing this parameter may help.

citus.max_assign_task_batch_size (integer)

The task tracker executor on the master synchronously assigns tasks in batches to the deamon on the workers. This parameter sets the maximum number of tasks to assign in a single batch. Choosing a larger batch size allows for faster task assignment. However, if the number of workers is large, then it may take longer for all workers to get tasks. This parameter can be set at runtime and is effective on the master.

citus.max_running_tasks_per_node (integer)

The task tracker process schedules and executes the tasks assigned to it as appropriate. This configuration value sets the maximum number of tasks to execute concurrently on one node at any given time and defaults to 8. This parameter is effective on the worker nodes and needs to be changed in the postgresql.conf file. After editing the config file, users can send a SIGHUP signal or restart the server for the change to take effect.

This configuration entry ensures that you don't have many tasks hitting disk at the same time and helps in avoiding disk I/O contention. If your queries are served from memory or SSDs, you can increase max_running_tasks_per_node without much concern.

citus.partition_buffer_size (integer)

Sets the buffer size to use for partition operations and defaults to 8MB. Citus allows for table data to be re-partitioned into multiple files when two large tables are being joined. After this partition buffer fills up, the repartitioned data is flushed into files on disk. This configuration entry can be set at run-time and is effective on the workers.