CitusDB Documentation

Release 4.0.1

August 13, 2015

CONTENTS

Welcome to the documentation for CitusDB 4.0.1! CitusDB is hybrid operational and analytics database for Big Data built on PostgreSQL. CitusDB scales out PostgreSQL across a cluster of physical or virtual servers through sharding and replication. Its advanced query engine then parallelizes incoming SQL queries across these servers to enable real-time responses.

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

CHAPTER

ONE

INSTALLATION GUIDE

This section provides instructions on installing and configuring CitusDB. There are two possible setups for CitusDB; namely single-node and multi-node cluster. Note that both the clusters execute the exact same logic i.e. they both run independent master and worker databases that communicate over the PostgreSQL communication libraries (libpq). They primarily differ in that the single node cluster is bound by the node's hardware resources while the multi-node cluster requires setting up authentication between the different nodes.

The single-node installation is a quick way to setup a test CitusDB cluster. However, a single node cluster can quickly get bottlenecked on hardware resources. We therefore recommend setting up a multi-node cluster for performance testing and production use cases. For setting up multi-node clusters on AWS easily, we recommend using the *CloudFormation template*.

Before diving into the exact setup instructions, we first describe the OS requirements for running CitusDB.

1.1 Supported Operating Systems

In general, any modern 64-bit Linux based operating system should be able to install and run CitusDB. Below is the list of platforms that have received specific testing at the time of release:

- Amazon Linux
- Redhat / Centos 6+
- Fedora 19+
- Ubuntu 10.04+
- Debian 6+

Note: The above is not an exhaustive list and only includes platforms where CitusDB packages are extensively tested. If your operating system is supported by PostgreSQL and you'd like to use CitusDB, please get in touch with us at engage@citusdata.com.

1.2 Single Node Cluster

The tutorials below provide instructions for installing and setting up a single node CitusDB cluster on various Linux systems. These instructions differ slightly depending on whether your operating system supports .rpm packages or .deb packages.

Note: Single node installations are recommended only for test clusters. Please use multi-node clusters for production deployments.

Single-node setup from .rpm packages (Fedora, CentOS, or Redhat)

Single-node setup from .deb packages (Ubuntu / Debian)

1.2.1 Single-node setup from .rpm packages (Fedora, CentOS, or Redhat)

This section describes the procedure for setting up a CitusDB cluster on a single node from .rpm packages.

We recommend using Fedora 11+, Redhat 6+ or Centos 6+ for running CitusDB.

1. Download CitusDB packages

Please note that by downloading the packages below, you agree that you have read, understand and accept the CitusDB License Agreement.

```
wget https://packages.citusdata.com/readline-6.0/citusdb-4.0.1-1.x86_64.rpm
wget https://packages.citusdata.com/contrib/citusdb-contrib-4.0.1-1.x86_64.rpm
```

Note: We also download the contrib package in addition to the core CitusDB package. The contrib package contains extensions to core CitusDB, such as the pg_shard extension for real-time inserts, the hstore type, HyperLogLog module for distinct count approximations, and administration tools such as shard rebalancer and pg_upgrade.

2. Install CitusDB packages using the rpm package manager

```
sudo rpm --install citusdb-4.0.1-1.x86_64.rpm
sudo rpm --install citusdb-contrib-4.0.1-1.x86_64.rpm
```

The CitusDB package puts all binaries under /opt/citusdb/4.0, and also creates a data subdirectory to store the newly initialized database's contents. The data directory created is owned by the current user. If CitusDB cannot find a non-root user, then the directory is owned by the 'postgres' user.

3. Setup worker database instances

In the single node setup, we use multiple database instances on the same node to demonstrate CitusDB's distributed logic. We use the already installed database as the master, and then initialize two more worker nodes. Note that we use the standard postgres initdb utility to initialize the databases.

```
/opt/citusdb/4.0/bin/initdb -D /opt/citusdb/4.0/data.9700
/opt/citusdb/4.0/bin/initdb -D /opt/citusdb/4.0/data.9701
```

4. Add worker node information

CitusDB uses a configuration file to inform the master node about the worker databases. To add this information, we append the worker database names and server ports to the pg_worker_list.conf file in the data directory.

```
echo 'localhost 9700' >> /opt/citusdb/4.0/data/pg_worker_list.conf
echo 'localhost 9701' >> /opt/citusdb/4.0/data/pg_worker_list.conf
```

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

5. Add pg_shard related configuration to enable real time inserts

```
vi /opt/citusdb/4.0/data/postgresql.conf
```

```
# Add the two below lines to the config file
shared_preload_libraries = 'pg_shard'
pg_shard.use_citusdb_select_logic = true
```

6. Start the master and worker databases

```
/opt/citusdb/4.0/bin/pg_ctl -D /opt/citusdb/4.0/data -l logfile start
/opt/citusdb/4.0/bin/pg_ctl -D /opt/citusdb/4.0/data.9700 -o "-p 9700" -l logfile.9700 start
/opt/citusdb/4.0/bin/pg_ctl -D /opt/citusdb/4.0/data.9701 -o "-p 9701" -l logfile.9701 start
```

Note that we use the standard postgresql pg_ctl utility to start the database. You can use the same utility to stop, restart or reload the cluster as specified in the PostgreSQL documentation.

7. Create the pg_shard extension on the master node

```
/opt/citusdb/4.0/bin/psql -h localhost -d postgres
```

```
CREATE EXTENSION pg_shard;
```

8. 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.

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

Ready to use CitusDB

At this step, you have completed the installation process and are ready to use CitusDB. To help you get started, we have an *Examples* section which has instructions on setting up a CitusDB cluster with sample data in minutes. You can also visit the *User Guide* section of our documentation to learn about CitusDB commands in detail.

1.2.2 Single-node setup from .deb packages (Ubuntu / Debian)

This section describes the procedure for setting up a CitusDB cluster on a single node from .deb packages.

We recommend using Ubuntu 12.04+ or Debian 7+ for running CitusDB. However, older versions of Ubuntu 10.04+ and Debian 6+ are also supported.

1. Download CitusDB packages

Please note that by downloading the packages below, you agree that you have read, understand and accept the CitusDB License Agreement.

```
wget https://packages.citusdata.com/readline-6.0/citusdb-4.0.1-1.amd64.deb
wget https://packages.citusdata.com/contrib/citusdb-contrib-4.0.1-1.amd64.deb
```

Note: We also download the contrib package in addition to the core CitusDB package. The contrib package contains extensions to core CitusDB, such as the pg_shard extension for real-time inserts, the hstore type, HyperLogLog module for distinct count approximations, and administration tools such as shard rebalancer and pg_upgrade.

2. Install CitusDB packages using the debian package manager

```
sudo dpkg --install citusdb-4.0.1-1.amd64.deb
sudo dpkg --install citusdb-contrib-4.0.1-1.amd64.deb
```

The CitusDB package puts all binaries under /opt/citusdb/4.0, and also creates a data subdirectory to store the newly initialized database's contents. The data directory created is owned by the current user. If CitusDB cannot find a non-root user, then the directory is owned by the 'postgres' user.

3. Setup worker database instances

In the single node setup, we use multiple database instances on the same node to demonstrate CitusDB's distributed logic. We use the already installed database as the master, and then initialize two more worker nodes. Note that we use the standard postgres initdb utility to initialize the databases.

```
/opt/citusdb/4.0/bin/initdb -D /opt/citusdb/4.0/data.9700
/opt/citusdb/4.0/bin/initdb -D /opt/citusdb/4.0/data.9701
```

4. Add worker node information

CitusDB uses a configuration file to inform the master node about the worker databases. To add this information, we append the worker database names and server ports to the pg_worker_list.conf file in the data directory.

```
echo 'localhost 9700' >> /opt/citusdb/4.0/data/pg_worker_list.conf
echo 'localhost 9701' >> /opt/citusdb/4.0/data/pg_worker_list.conf
```

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

5. Add pg_shard related configuration to enable real time inserts

```
vi /opt/citusdb/4.0/data/postgresql.conf
```

```
# Add the two below lines to the config file
shared_preload_libraries = 'pg_shard'
pg_shard.use_citusdb_select_logic = true
```

6. Start the master and worker databases

```
/opt/citusdb/4.0/bin/pg_ctl -D /opt/citusdb/4.0/data -l logfile start
/opt/citusdb/4.0/bin/pg_ctl -D /opt/citusdb/4.0/data.9700 -o "-p 9700" -l logfile.9700 start
/opt/citusdb/4.0/bin/pg_ctl -D /opt/citusdb/4.0/data.9701 -o "-p 9701" -l logfile.9701 start
```

Note that we use the standard postgresql pg_ctl utility to start the database. You can use the same utility to stop, restart or reload the cluster as specified in the PostgreSQL documentation.

7. Create the pg_shard extension on the master node

```
/opt/citusdb/4.0/bin/psql -h localhost -d postgres
```

```
CREATE EXTENSION pg_shard;
```

8. 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.

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

Ready to use CitusDB

At this step, you have completed the installation process and are ready to use CitusDB. To help you get started, we have an *Examples* guide which has instructions on setting up a CitusDB cluster with sample data in minutes. You can also visit the *User Guide* section of our documentation to learn about CitusDB commands in detail.

1.3 Multi Node Cluster

The tutorials below provide instructions to install and setup a multi-node CitusDB cluster. You can either setup the cluster on AWS using our CloudFormation templates or install CitusDB on premise on your machines. We recommend using the CloudFormation templates for cloud deployments as it quickly sets up a fully operational CitusDB cluster.

Multi-node setup on AWS (Recommended)

Multi-node setup on-premise

1.3.1 Multi-node setup on AWS (Recommended)

This section describes the steps needed to set up a CitusDB cluster on AWS using the CloudFormation console.

To simplify the process of setting up a CitusDB cluster on EC2, AWS users can use AWS CloudFormation. The CloudFormation template for CitusDB enables users to start a CitusDB cluster on AWS in just a few clicks, with pg_shard, for real-time transactional workloads, cstore_fdw, for columnar storage, and contrib extensions 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 CitusDB.

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 CitusDB cluster on AWS.

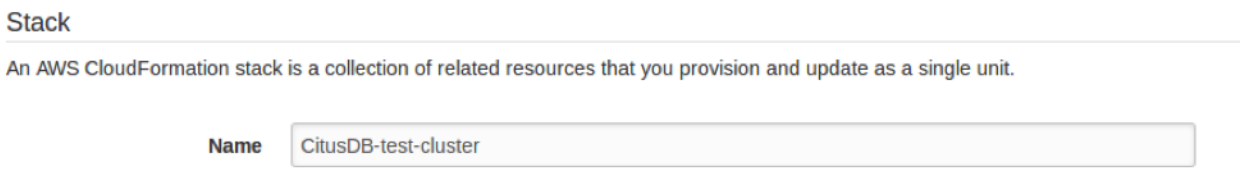
1. Start a CitusDB cluster

To begin, you can start a CitusDB cluster using CloudFormation by clicking the "Launch CitusDB on AWS" button on our downloads page. This will take you directly to the AWS CloudFormation console.

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

2. Pick unique stack name

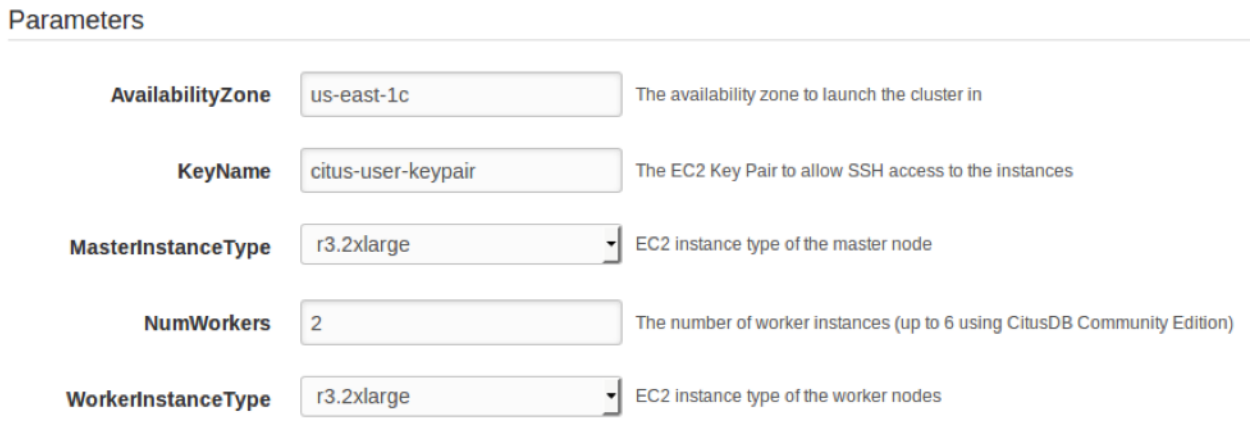
In the console, pick a unique name for your stack and click Next.



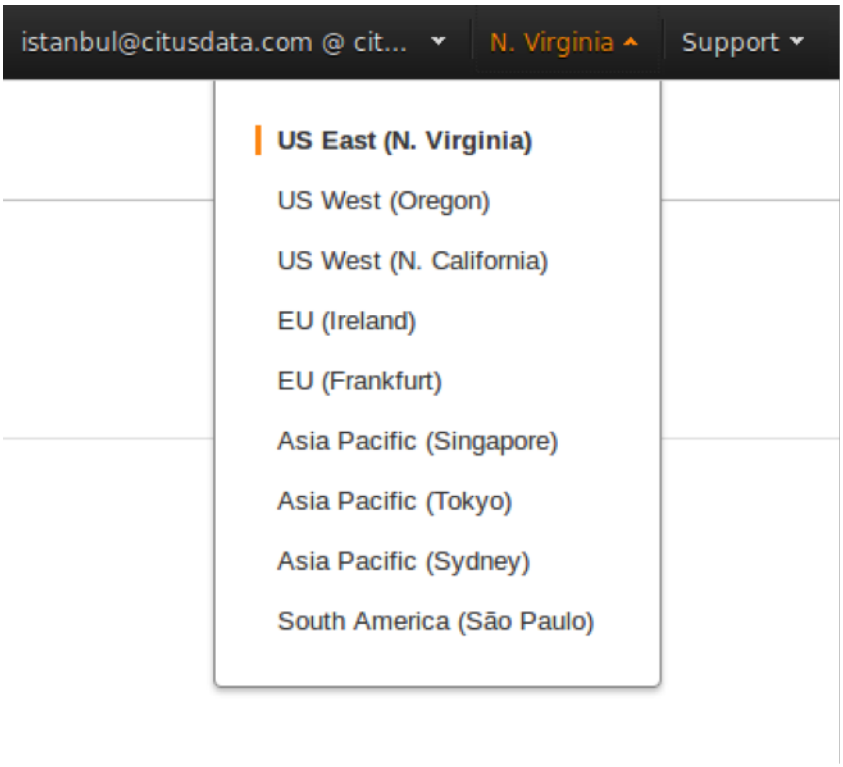
Note: Please ensure that you choose unique names for all your clusters. Otherwise, the cluster creation may fail with the error "Template_name template already created".

3. Fill the form with details about your cluster

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



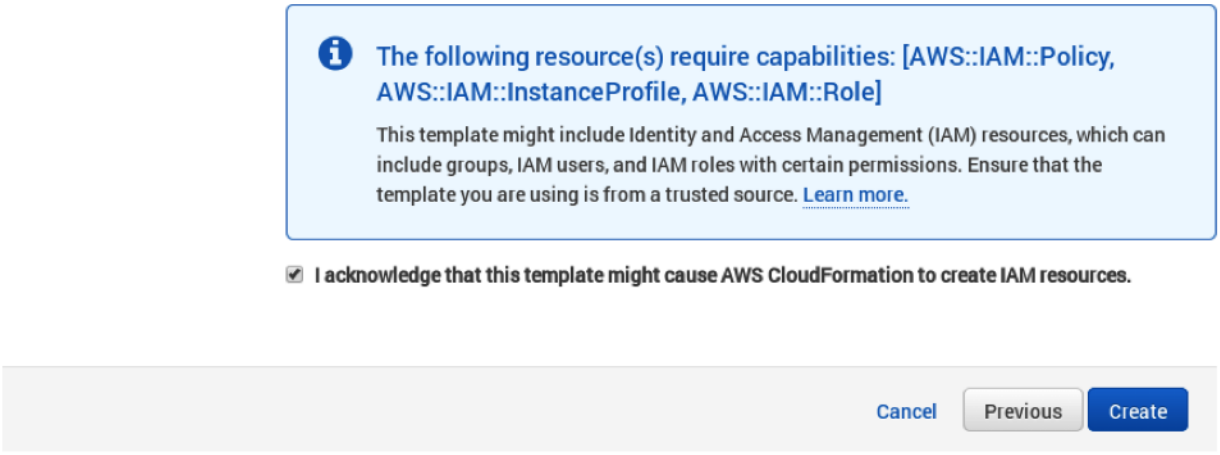
Note: If you want to launch a cluster in a region other than US East, you can update the region in the top-right corner of the AWS console as shown below.



4. Acknowledge IAM capabilities

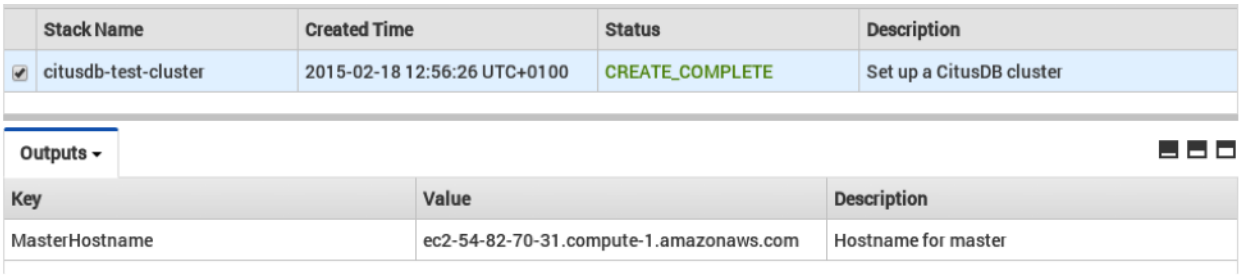
The next screen displays Tags and a few advanced options. For simplicity, we use the default options and click Next.

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 citus-user-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 CitusDB 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.

```
/opt/citusdb/4.0/bin/psql -h localhost -d postgres
select * from master_get_active_worker_nodes();
```

To help you get started, we have an *Examples* guide which has instructions on setting up a cluster with sample data in minutes. You can also visit the *User Guide* section of our documentation to learn about CitusDB commands in detail.

8. Cluster notes

The template automatically tunes the system configuration for CitusDB 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:

```
/opt/citusdb/4.0/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.

1.3.2 Multi-node setup on-premise

At a high level, the multi-node setup involves two steps that differ from those of a single node cluster. First, you need to either manually log in to all the nodes and issue the installation and configuration commands, or rely on tools such as pssh to run the remote commands in parallel for you. Second, you need to configure authentication settings on the nodes to allow for them to talk to each other.

In the following tutorials, we assume that you will manually log in to all the nodes and issue commands. We also assume that the master node has a DNS name of master-100 and the worker nodes have DNS names worker-101, worker-102 and so on. Lastly, we note that you can edit the hosts file in /etc/hosts if your nodes don't already have their DNS names assigned.

You can choose the appropirate tutorial below depending on whether your operating system supports .rpm packages or .deb packages.

Setup multiple node cluster from .rpm packages (Fedora / Centos / Redhat)

Setup multiple node cluster from .deb packages (Ubuntu / Debian)

Setup multiple node cluster from .rpm packages (Fedora / Centos / Redhat)

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

We recommend using Fedora 11+, Redhat 6+ or Centos 6+ for running CitusDB.

Steps to be executed on all nodes

1. Download CitusDB packages

Please note that by downloading the packages below, you agree that you have read, understand and accept the CitusDB License Agreement.

```
wget https://packages.citusdata.com/readline-6.0/citusdb-4.0.1-1.x86_64.rpm
wget https://packages.citusdata.com/contrib/citusdb-contrib-4.0.1-1.x86_64.rpm
```

Note: We also download the contrib package in addition to the core CitusDB package. The contrib package contains extensions to core CitusDB, such as the pg_shard extension for real-time inserts, the hstore type, HyperLogLog module for distinct count approximations, and administration tools such as shard rebalancer and pg_upgrade.

2. Install CitusDB packages using the rpm package manager

```
sudo rpm --install citusdb-4.0.1-1.x86_64.rpm
sudo rpm --install citusdb-contrib-4.0.1-1.x86_64.rpm
```

The CitusDB package puts all binaries under /opt/citusdb/4.0, and also creates a data subdirectory to store the newly initialized database's contents. The data directory created is owned by the current user. If CitusDB cannot find a non-root user, then the directory is owned by the 'postgres' user.

3. Configure connection and authentication

By default, the CitusDB server only listens to clients on localhost. As a part of this step, we instruct CitusDB to listen on all IP interfaces, and then configure the client authentication file to allow all incoming connections from the local network.

```
vi /opt/citusdb/4.0/data/postgresql.conf
```

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

```
vi /opt/citusdb/4.0/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: Admittedly, these settings are too permissive for certain types of environments, and the PostgreSQL manual explains in more detail on how to restrict them further.

4. Start database servers

Now, we start up all the databases in the cluster.

```
/opt/citusdb/4.0/bin/pg_ctl -D /opt/citusdb/4.0/data -l logfile start
```

Note that we use the standard postgresql pg_ctl utility to start the database. You can use the same utility to stop, restart or reload the cluster as specified in the PostgreSQL documentation.

Steps to be executed on the master node

The steps listed below must be executed **only** on the master node (i.e. master-100 in our example) after the above mentioned steps have been executed.

5. Add worker node information

CitusDB uses a configuration file to inform the master node about the worker databases. 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 (worker-101 and worker-102) and add their DNS names and server ports to the list.

```
echo 'worker-101 5432' >> /opt/citusdb/4.0/data/pg_worker_list.conf
echo 'worker-102 5432' >> /opt/citusdb/4.0/data/pg_worker_list.conf
```

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

6. Add pg_shard related configuration to enable real time inserts

```
vi /opt/citusdb/4.0/data/postgresql.conf
```

```
# Add the two below lines to the config file
shared_preload_libraries = 'pg_shard'
pg_shard.use_citusdb_select_logic = true
```

7. Restart the master database

```
/opt/citusdb/4.0/bin/pg_ctl -D /opt/citusdb/4.0/data -l logfile restart
```

8. Create the pg_shard extension

```
/opt/citusdb/4.0/bin/psql -h localhost -d postgres
```

```
CREATE EXTENSION pg_shard;
```

9. 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.

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

Ready to use CitusDB

At this step, you have completed the installation process and are ready to use your CitusDB cluster. To help you get started, we have an *Examples* guide which has instructions on setting up a CitusDB cluster with sample data in minutes. You can also visit the *User Guide* section of our documentation to learn about CitusDB commands in detail.

Setup multiple node cluster from .deb packages (Ubuntu / Debian)

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

We recommend using Ubuntu 12.04+ or Debian 7+ for running CitusDB. However, older versions of Ubuntu 10.04+ and Debian 6+ are also supported.

Steps to be executed on all nodes

1. Download CitusDB packages

Please note that by downloading the packages below, you agree that you have read, understand and accept the CitusDB License Agreement.

```
wget https://packages.citusdata.com/readline-6.0/citusdb-4.0.1-1.amd64.deb
wget https://packages.citusdata.com/contrib/citusdb-contrib-4.0.1-1.amd64.deb
```

Note: We also download the contrib package in addition to the core CitusDB package. The contrib package contains extensions to core CitusDB, such as the pg_shard extension for real-time inserts, the hstore type, HyperLogLog module for distinct count approximations, and administration tools such as shard rebalancer and pg_upgrade.

2. Install CitusDB packages using the debian package manager

```
sudo dpkg --install citusdb-4.0.1-1.amd64.deb
sudo dpkg --install citusdb-contrib-4.0.1-1.amd64.deb
```

The CitusDB package puts all binaries under /opt/citusdb/4.0, and also creates a data subdirectory to store the newly initialized database's contents. The data directory created is owned by the current user. If CitusDB cannot find a non-root user, then the directory is owned by the 'postgres' user.

3. Configure connection and authentication

By default, the CitusDB server only listens to clients on localhost. As a part of this step, we instruct CitusDB to listen on all IP interfaces, and then configure the client authentication file to allow all incoming connections from the local network.

```
vi /opt/citusdb/4.0/data/postgresql.conf
```

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

```
vi /opt/citusdb/4.0/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: Admittedly, these settings are too permissive for certain types of environments, and the PostgreSQL manual explains in more detail on how to restrict them further.

4. Start database servers

Now, we start up all the databases in the cluster.

```
/opt/citusdb/4.0/bin/pg_ctl -D /opt/citusdb/4.0/data -l logfile start
```

Note that we use the standard postgresql pg_ctl utility to start the database. You can use the same utility to stop, restart or reload the cluster as specified in the PostgreSQL documentation.

Steps to be executed on the master node

The steps listed below must be executed **only** on the master node (i.e. master-100 in our example) after the above mentioned steps have been executed.

5. Add worker node information

CitusDB uses a configuration file to inform the master node about the worker databases. To add this information, we append the worker database names server ports to the pg_worker_list.conf file in the data directory. For our example, we assume that there are two workers (worker-101 and worker-102) and add their DNS names and server ports to the list.

```
echo 'worker-101 5432' >> /opt/citusdb/4.0/data/pg_worker_list.conf
echo 'worker-102 5432' >> /opt/citusdb/4.0/data/pg_worker_list.conf
```

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

6. Add pg_shard related configuration to enable real time inserts

```
vi /opt/citusdb/4.0/data/postgresql.conf
```

```
# Add the two below lines to the config file
shared_preload_libraries = 'pg_shard'
pg_shard.use_citusdb_select_logic = true
```

7. Restart the master databases

```
/opt/citusdb/4.0/bin/pg_ctl -D /opt/citusdb/4.0/data -l logfile restart
```

8. Create the pg_shard extension

```
/opt/citusdb/4.0/bin/psql -h localhost -d postgres
```

```
CREATE EXTENSION pg_shard;
```

9. 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.

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

Ready to use CitusDB

At this step, you have completed the installation process and are ready to use your CitusDB cluster. To help you get started, we have an *Examples* guide which has instructions on setting up a CitusDB cluster with sample data in minutes. You can also visit the *User Guide* section of our documentation to learn about CitusDB commands in detail.

CHAPTER

TWO

EXAMPLES

In this section, we discuss how users can use CitusDB to address their real-time use cases. These examples include powering real-time analytic dashboards on events or metrics data; tracking unique counts; performing session analytics; rendering real-time maps using geospatial data types and indexes; and querying distributed and local PostgreSQL tables in the same database.

In our examples, we use public events data from GitHub. This data has information for about 20+ event types that range from new commits and fork events to opening new tickets, commenting, and adding members to a project. These events are aggregated into hourly archives, with each archive containing JSON encoded events as reported by the GitHub API. For our examples, we slightly modified these data to include both structured tabular and semi-structured jsonb fields.

We begin by describing how users can use CitusDB to power their real-time customer-facing analytics dashboards.

2.1 Powering real-time analytics dashboards

A common use case where customers use CitusDB is to power their customer facing real-time analytics dashboards. This involves providing human real-time (few seconds or sub-second) responses to queries over billions of "events". This event-based data model is applicable to a variety of use cases, like user actions in mobile application, user clicks on websites / ads, events in an event stream, or each line in a log file.

We generally see two approaches to capturing and querying high-volume events data and presenting insights through dashboards or graphs. The first approach involves distributing the data by the time dimension and works well with batch loading of data. The second approach distributes the data by identifier and is more suited to use cases which require users to insert / update their data in real-time.

We describe these approaches below and provide instructions in context of the github events example for users who want to try them out. Note that the sections below assume that you have already downloaded and installed CitusDB. If you haven't, please visit the *Installation Guide* before proceeding.

2.1.1 Distributing by Time Dimension (Incremental Data Loading)

Events data generally arrives in a time-ordered series. So, if your use case works well with batch loading, it is easiest to append distribute your largest tables by time, and load data into them in intervals of N minutes / hours. Since the data already comes sorted on the time dimension, you don't have to do any extra preprocessing before loading the data.

In the next few sections, we demonstrate how you can setup a CitusDB cluster which uses the above approach with the Github events data.

Querying Raw Data

We first insert raw events data into the database distributed on the time dimension. This assumes that we have enough hardware to generate query results in real-time. This approach is the simplest to setup and provides the most flexibility in terms of dimensions on which the queries can be run.

1. Download sample data

We begin by downloading and extracting the sample data for 6 hours of github data. We have also transformed more sample data for users who want to try the example with larger volumes of data. Please contact us at engage@citusdata.com if you want to run this example with a larger dataset.

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

github_events-2015-01-01-*.gz
```

2. Connect to the master node and create a distributed table

```
/opt/citusdb/4.0/bin/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
) DISTRIBUTE BY APPEND (created_at);
```

This command creates a new distributed table, and has almost the same syntax as PostgreSQL's CREATE TABLE command. The only difference is that the command has a distribute by append clause at the end, which tells the database the column to use for distribution. 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. This helps CitusDB to optimize queries with time qualifiers by pruning away unrelated shards.

To learn more about different distribution methods in CitusDB, you can visit the *related section* in our user guide.

3. Create desired indexes

Most of your queries may have an inherent time dimension, but you may still want to query data on secondary dimensions at times, and that's when creating an index comes in handy. In this example, we create an index on repo_id as we have queries where we want to generate statistics / graphs only for a particular repository.

```
CREATE INDEX repo_id_index ON github_events (repo_id);
```

Note that indexes visibly increase data loading times in PostgreSQL, so you may want to skip this step if you don't need them.

4. Load Data

Now, we're ready to load these events into the database by specifying the correct file path. Please note that you can load these files in parallel through separate database connections or from different nodes. We also enable timing so that we can view the loading and query times.

```
\timing
\stage github_events from 'github_events-2015-01-01-0.csv' with CSV

SELECT count(*) from github_events;
```

```
\stage github_events from 'github_events-2015-01-01-1.csv' with CSV

SELECT count(*) from github_events;

\stage github_events from 'github_events-2015-01-01-2.csv' with CSV
\stage github_events from 'github_events-2015-01-01-3.csv' with CSV
\stage github_events from 'github_events-2015-01-01-4.csv' with CSV
\stage github_events from 'github_events-2015-01-01-5.csv' with CSV
```

The stage command borrows its syntax from the client-side copy command in PostgreSQL. Behind the covers, the command opens a connection to the master node and fetches candidate worker nodes to create new shards on. The command then connects to these worker nodes, creates at least one shard there, and uploads the github events to the shards. The command then replicates these shards on other worker nodes till the replication factor is satisfied, and then finalizes the shard's metadata with the master node.

Also, you can notice that the times for the count query remain similar even after adding more data into the cluster. This is because CitusDB now parallelizes the same query across more cores on your machine.

5. Run queries

After creating the table and staging data into it, you are now ready to run analytics queries on it. We use the psql command prompt to issue queries, but you could also use a graphical user interface.

```
-- Find the number of each event type on a per day and per hour graph for a particular repository.
SELECT
    event_type, date_part('hour', created_at) as hour, to_char(created_at, 'day'), count(*)
FROM
    github_events
WHERE
    repo_id = '724712'
GROUP BY
    event_type, date_part('hour', created_at), to_char(created_at, 'day')
ORDER BY

    event_type, date_part('hour', created_at), to_char(created_at, 'day');

-- Find total count of issues opened, closed, and reopened in the first 3 hours of Jan 1, 2015.
SELECT
    payload->'action', count(*)
FROM
    github_events
WHERE

    event_type = 'IssuesEvent' AND
    created_at >= '2015-01-01 00:00:00' AND
    created_at <= '2015-01-01 02:00:00'
GROUP BY
    payload->'action';
```

Other than the sample queries mentioned above, you can also run several other interesting queries over the github dataset.

With huge volumes of raw data, the hardware requirements of a cluster to get real-time query responses might not be cost effective. Also, you might not use all the fields being captured about your events. Hence, you can aggregate their data on the time dimension and query the aggregated data to get fast query responses. In the next section, we discuss several approaches to aggregate the data and also provide instructions for trying them out.

Querying Aggregated Data

This type of data model is more applicable to users who want to load data into their database at minutely, hourly or daily intervals. Users can aggregate their data on the time interval using one of the approaches below, and store the aggregated data in CitusDB. They can then serve real-time graphs grouped at the desired time intervals by running queries on the aggregated data.

To aggregate the raw data into hourly / daily digests, users can choose one of the following approaches:

- **Use an external system:** One option is to use an external system to capture the raw events data and aggregate it. Then, the aggregated data can be loaded into CitusDB. There are several tools which can be used for this step. You can see an example setup for this approach where Cloudflare uses Kafka queues and Go aggregators to consume their log data and insert aggregated data with 1-minute granularity into their CitusDB cluster.
- **Use PostgreSQL to aggregate the data:** Another common approach is to append your raw events data into a PostgreSQL table and then run an aggregation query on top of the raw data. Then, you can load the aggregated data into a distributed table.

Below, we give an example of how you can use PostgreSQL to aggregate the Github data and then run queries on it.

1. Create local aggregation tables

We first create a regular PostgreSQL table to store the incoming raw data. This has the same schema as the table discussed in the previous section.

```
/opt/citusdb/4.0/bin/psql -h localhost -d postgres
CREATE TABLE github_events_local
 (
     event_id bigint,
     event_type text,
     event_public boolean,
     repo_id bigint,
     payload jsonb,
     repo jsonb,
     actor jsonb,
     org jsonb,
     created_at timestamp
 );
```

Next, we create the aggregation table. The schema for the aggregate table depends on the type of queries and intervals of aggregations. We will generate reports / graphs similar to github i.e. statistics (per-repo and global) for different types of events over different time filters. Hence, we aggregate the data on the repo_id and event_type columns on an hourly basis.

```
CREATE TABLE github_events_hourly_local
(
    event_type text,
    repo_id bigint,
    created_hour timestamp,
    event_count bigint,
    statistics jsonb
);
```

Please note that this table is also created as a regular PostgreSQL table (i.e. without the distribute by append clause). We will append the aggregated data into a distributed table in subsequent steps.

One interesting field in this table schema is the statistics column. In the raw data schema, there is a payload column which stores event specific data. The statistics column can be used to store statistics about keys in that payload column for the set of events which have been aggregated into that row.

For example, the github events data has a event_type named IssuesEvent which is triggered when an issue's status changes eg. opened, closed, or reopened. The statistics field in the rows corresponding to IssuesEvent will have a jsonb object having information about how many of those events were opened, how many were closed and how many were reopened ({"closed": "1", "opened": "3"}).

Such a schema allows you to store any kinds of statistics by just changing the aggregation query. You can choose to store total number of lines changed, name of the branch on which the event occurred, or any other information you would like.

2. Create distributed aggregate table

Next, we create the distributed aggregate table with the same schema as the PostgreSQL aggregate table but with the DISTRIBUTE BY APPEND clause.

```
CREATE TABLE github_events_hourly
(
    event_type text,
    repo_id bigint,
    created_hour timestamp,
    event_count bigint,
    statistics jsonb
) DISTRIBUTE BY APPEND (created_hour);
```

3. Load raw data into the PostgreSQL table

The users can then insert their incoming raw events data into the github_events table using the regular PostgreSQL INSERT commands. For the example however, we use the copy command to copy one hour of data into the table.

```
\copy github_events_local from 'github_events-2015-01-01-0.csv' WITH CSV
```

4. Aggregation on a hourly basis

Once the github_events table has an hour of data, users can run an aggregation query on that table and store the aggregated results into the github_events_hourly table.

To do this aggregation step, we first create a user defined function which takes in a array of strings and returns a json object with each object and its count in that array. This function will be used later to generate the statistics column in the aggregated table.

```
CREATE FUNCTION count_elements(text[]) RETURNS json AS
$BODY$
select ('{' || a || '}')::json
    from
    (
    select string_agg('"' || i || '":"' || c || '"', ',') a
        FROM
        (
            SELECT i, count(*) c
            FROM
            (SELECT unnest($1::text[]) i) i GROUP BY i ORDER BY c DESC
        ) foo
    ) bar;
$BODY$
LANGUAGE SQL;
```

In the next step, we run a SQL query which aggregates the raw events data on the basis of event_type, repo_id and the hour, and then stores the relevant statistics about those events in the statistics column of the github_events_hourly_local table. Please note that you can store more information from the events' payload column by modifying the aggregation query.

```
INSERT INTO github_events_hourly_local (
select
    event_type, repo_id, date_trunc('hour', created_at) as created_hour, count(*),
    (CASE
        when
            event_type = 'IssuesEvent'
        then count_elements(array_agg(payload->>'action'))::jsonb
        when event_type = 'GollumEvent'
        then count_elements(array_agg(payload->'pages'->0->>'action'))::jsonb
        when event_type = 'PushEvent'
        then count_elements(array_agg(payload->>'ref'))::jsonb
        when event_type = 'ReleaseEvent'
        then count_elements(array_agg(payload->'release'->>'tag_name'))::jsonb
        when event_type = 'CreateEvent'
        then count_elements(array_agg(payload->>'ref_type'))::jsonb
        when event_type = 'DeleteEvent'
        then count_elements(array_agg(payload->>'ref_type'))::jsonb
        when event_type = 'PullRequestEvent'
        then count_elements(array_agg(payload->>'action'))::jsonb
        else null
    end)
from
    github_events_local
where
    date_trunc('hour',created_at) = '2015-01-01 00:00:00'
group by
    event_type, repo_id, date_trunc('hour',created_at));
```

This way users can convert the raw data into aggregated tables on a hourly basis simply by changing the time filter in the above query.

Once you have the aggregate table, you can stage or append the raw data into another distributed table. This way, if you have queries on a dimension other than the aggregated dimension, you can always lookup those results in the raw data. Once that is done, you can truncate the local table or delete those rows from it which have already been aggregated.

```
TRUNCATE github_events_local ;
```

5. Create an empty shard for the distributed table

We then create a new shard for the distributed table by using the master_create_empty_shard UDF.

```
select * from master_create_empty_shard('github_events_hourly');
 master_create_empty_shard
---------------------------
                  102014
(1 row)
```

This function creates a new shard on the worker nodes which has the same schema as the master node table. Then, it replicates the empty shard to shard_replication_factor nodes. Finally, the function returns the shard id of the newly created shard.

6. Append the aggregated table into a distributed table

Then, we append the local aggregate table to the newly created shard using the master_append_table_to_shard UDF.

```
select * from master_append_table_to_shard(102014, 'github_events_hourly_local', 'source-node-name',
   master_append_table_to_shard
-----------------------------
```

```
                    0.000473022
(1 row)
```

master_append_table_to_shard() appends the contents of a PostgreSQL table to a shard of a distributed table. In this example, the function fetches the github_events_hourly_local table from the database server running on node 'source-node-name' with port number 5432 and appends it to the shard 102014. This source node name can be set to the hostname of the node which has the github_events_hourly_local table.

The function then returns the shard fill ratio which which helps to make a decision on whether more data should be appended to this shard or if a new shard should be created. The maximum desired shard size can be configured using the shard_max_size parameter.

Once this data is appended to the distributed table, we can truncate the local table.

```
TRUNCATE github_events_hourly_local;
```

7. Download pre-aggregated data

In the above steps, we have demonstrated how users can aggregate data within PostgreSQL for one hour of data. You can aggregate the data for all the six hours using the process described in step 4.

For ease of use, we download the already aggregated data and use it to describe the next steps.

```
wget http://examples.citusdata.com/github_archive/github_events_aggregated-2015-01-01-{1..5}.csv.gz gzip -d
github_events_aggregated-2015-01-01-*.gz
```

8. Load pre-aggregated data into distributed table

If you are doing aggregation similar to the process described above, then your aggregated data will be in the github_events_hourly_local table. If you are using some other external tool for aggregation or downloading the pre-aggregated files, then you can load the data into that table before appending it to a distributed table.

```
\copy github_events_hourly_local from 'github_events_aggregated-2015-01-01-1.csv' WITH CSV;
```

Please note that we store this data into a local table using copy instead of using the stage command. This is because the stage command always creates a new shard when invoked. In this case, creating new shards for every hour of data will lead to a very high number of small shards in the system which might not be desirable. So, we load the data into PostgreSQL tables and then append them to a shard until we reach the desired size.

We then append this data into the previously created shard of the distributed table and then empty the local table.

```
SELECT * from master_append_table_to_shard(102014, 'github_events_hourly_local', 'source-node-name', TRUNCATE

github_events_hourly_local;
```

Similarly, you can repeat the above steps for the next four hours of data. Please note that you can run the master_create_empty_shard() to create new shards when the shard fill ratio returned by the UDF is close to one and the shard size is close to the shard_max_size. Then, you can begin appending to that shard by using the new shard id in the master_append_table_to_shard() UDF.

9. Run queries on the distributed aggregate table

After the above steps, you can run queries on your aggregated data. To generate the same reports as you did on the raw data, you can use the following queries on the aggregated table.

```
SELECT
    event_type, date_part('hour', created_hour), to_char(created_hour,'day'), sum(event_count) FROM
    github_events_hourly
WHERE


    repo_id = '724712'
```

```
GROUP BY
    event_type, date_part('hour', created_hour), to_char(created_hour, 'day')
ORDER BY
    event_type, date_part('hour', created_hour), to_char(created_hour, 'day');

SELECT
    sum((statistics->>'opened')::int) as opened_issue_count,
    sum((statistics->>'closed')::int) as closed_issue_count,
    sum((statistics->>'reopened')::int) as reopened_issue_count
FROM
    github_events_hourly
WHERE
    event_type = 'IssuesEvent' AND
    created_hour >= '2015-01-01 00:00:00' AND
    created_hour <= '2015-01-01 02:00:00';
```

With this, we conclude our discussion about distributing data on the time dimension and running analytics queries for generating hourly / daily reports. In the next example, we discuss how we can partition the data on an identifier to do real time inserts into distributed tables.

2.1.2 Distributing by Identifier (Real-time Data Loading)

This data model 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. In this case, CitusDB will maintain hash token ranges for all the created shards. Whenever a row is inserted, updated or deleted, CitusDB (with the help of pg_shard) will redirect the query to the correct shard and issue it locally.

Since, users generally want to generate per-repo statistics, we distribute their data by hash on the repo_id column. Then, CitusDB can easily prune away the unrelated shards and speed up the queries with a filter on the repo id.

As in the previous case, users can choose to store their raw events data in CitusDB or aggregate their data on the desired time interval.

Querying Raw Data

In this section, we discuss how users can setup hash partitioned tables on the repo_id column with the same github events data.

1. Create hash partitioned raw data tables

We define a hash partitioned table using the CREATE TABLE statement in the same way as you would do with a regular postgresql table. Please note that we use the same table schema as we used in the previous example.

```
/opt/citusdb/4.0/bin/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, we need to specify the distribution column for the table. This can be done by using the master_create_distributed_table() UDF.

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

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

2. Create worker shards for the distributed table

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

This UDF takes two arguments in addition to the table name; shard count and the replication factor. This example would create a total of 16 shards where each shard owns a portion of a hash token space and gets replicated on 2 worker nodes.

3. Download sample data

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

4. Insert raw events data into github_events table

Users can then use the standard PostgreSQL insert command to insert data into the github_events table. For now, we will use the copy_to_distributed_table script to populate the github_events table.

Before invoking the script, we set the PATH environment variable to include the CitusDB bin directory.

```
export PATH=/opt/citusdb/4.0/bin/:$PATH
```

Next, 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
```

After setting these environment variables, we load the 6 hours of data into the github_events table by invoking the copy_to_distributed_table script.

```
/opt/citusdb/4.0/bin/copy_to_distributed_table -C github_events-2015-01-01-0.csv github_events &
/opt/citusdb/4.0/bin/copy_to_distributed_table -C github_events-2015-01-01-1.csv github_events &
/opt/citusdb/4.0/bin/copy_to_distributed_table -C github_events-2015-01-01-2.csv github_events &
/opt/citusdb/4.0/bin/copy_to_distributed_table -C github_events-2015-01-01-3.csv github_events &
/opt/citusdb/4.0/bin/copy_to_distributed_table -C github_events-2015-01-01-4.csv github_events &
/opt/citusdb/4.0/bin/copy_to_distributed_table -C github_events-2015-01-01-5.csv github_events &
```

Please note that we run all the commands in background for them to execute in parallel. Also, it is worth noting here that hash partitioned tables are more suited for single row inserts as compared to bulk loads. However, we use the copy script here for simplicity reasons.

5. Run queries on raw data

We are now ready to run queries on the events data.

```
-- Find the number of each event type on a per day and per hour graph for a particular repository.
SELECT
    event_type, date_part('hour', created_at) as hour, to_char(created_at, 'day'), count(*)
FROM
    github_events
WHERE
    repo_id = '724712'
GROUP BY

    event_type, date_part('hour', created_at), to_char(created_at, 'day')
```

```
ORDER BY
    event_type, date_part('hour', created_at), to_char(created_at, 'day');
```

```
-- Find total count of issues opened, closed, and reopened in the first 3 hours.
SELECT
    payload->'action', count(*)
FROM
    github_events
WHERE
    event_type = 'IssuesEvent' AND
    created_at >= '2015-01-01 00:00:00' AND
    created_at <= '2015-01-01 02:00:00'
GROUP BY
    payload->'action';
```

As discussed in the previous sections, the hardware requirements of a cluster to get real-time query responses might not be cost effective. Therefore, users can create tables to aggregate their data on the time dimension and query the aggregated data to get fast query responses. In the next section, we discuss an approach to do this for hash partitioned tables partitioned on the repo_id column.

Querying Aggregated Data

As discussed in the pervious section, users may want to aggregate their distributed tables in order to receive fast query responses. To do that with hash distributed tables, you can follow the steps below.

1. Create Hash Partitioned Aggregate table and worker shards

```
/opt/citusdb/4.0/bin/psql -h localhost -d postgres
CREATE TABLE github_events_hourly
(
    event_type text,
    repo_id bigint,
    created_hour timestamp,
    event_count bigint,
    statistics jsonb
);

SELECT master_create_distributed_table('github_events_hourly', 'repo_id');
SELECT master_create_worker_shards('github_events_hourly', 16, 2);
```

There are few points to note here. Firstly, we use the same aggregate table schema as we used in the previous example.

Secondly, we distribute the aggregate table by the same column as the raw data tables and create the same number of worker shards i.e. 16 for both the tables. This ensures that shards for both the tables are colocated. Hence, all the shards having the same hash ranges will reside on the same node. This will allow us to run the aggregation queries in a distributed way on the worker nodes without moving any data across the nodes.

2. Use a custom UDF to aggregate data from the github_events table into the github_events_hourly table

To make the process of aggregating distributed tables in parallel easier, CitusDB's next release will come with a user defined function master_aggregate_table_shards(). The signature of the function is as defined below.

```
master_aggregate_table_shards(source table, destination table, aggregation_query);
```

Since, we created two tables with the same number of shards, there is a 1:1 mapping between shards of the raw data table and the aggregate table. This function will take the aggregation query passed as an argument and run it on each shard of the source table. The results of this query will then be stored into the corresponding destination table shard.

This UDF is under development and will be a part of CitusDB's next release. If you want to use this functionality or learn more about the UDF and its implementation, please get in touch with us at engage@citusdata.com.

3. Query the aggregated table

Once the data is stored into the aggregate table, users can run their queries on them.

```
SELECT
    event_type, date_part('hour', created_hour), to_char(created_hour,'day'), sum(event_count)
FROM
    github_events_hourly
WHERE
    repo_id = '724712'
GROUP BY
    event_type, date_part('hour', created_hour), to_char(created_hour,'day')
ORDER BY

    event_type, date_part('hour', created_hour), to_char(created_hour, 'day');

SELECT
    sum((statistics->>'opened')::int) as opened_issue_count,
    sum((statistics->>'closed')::int) as closed_issue_count,
    sum((statistics->>'reopened')::int) as reopened_issue_count
FROM
    github_events_hourly
WHERE
    event_type = 'IssuesEvent' AND
    created_hour >= '2015-01-01 00:00:00' AND
    created_hour <= '2015-01-01 02:00:00';
```

With this we end our discussion about how users can use CitusDB to address different type of real-time dashboard use cases. We will soon be adding another example, which explains how users can run session analytics on their data using CitusDB. You can also visit the *User Guide* section of our documentation to learn about CitusDB commands in detail.

CHAPTER

THREE

USER GUIDE

This section describes how CitusDB users can create, manipulate and query their distributed tables. We first begin by describing basics of the CitusDB architecture, how it processes queries and handles failures. We then discuss how users can create append and hash partitioned tables and load data into them. We finally describe the different types of queries CitusDB supports and how users can run them.

3.1 Introduction To CitusDB

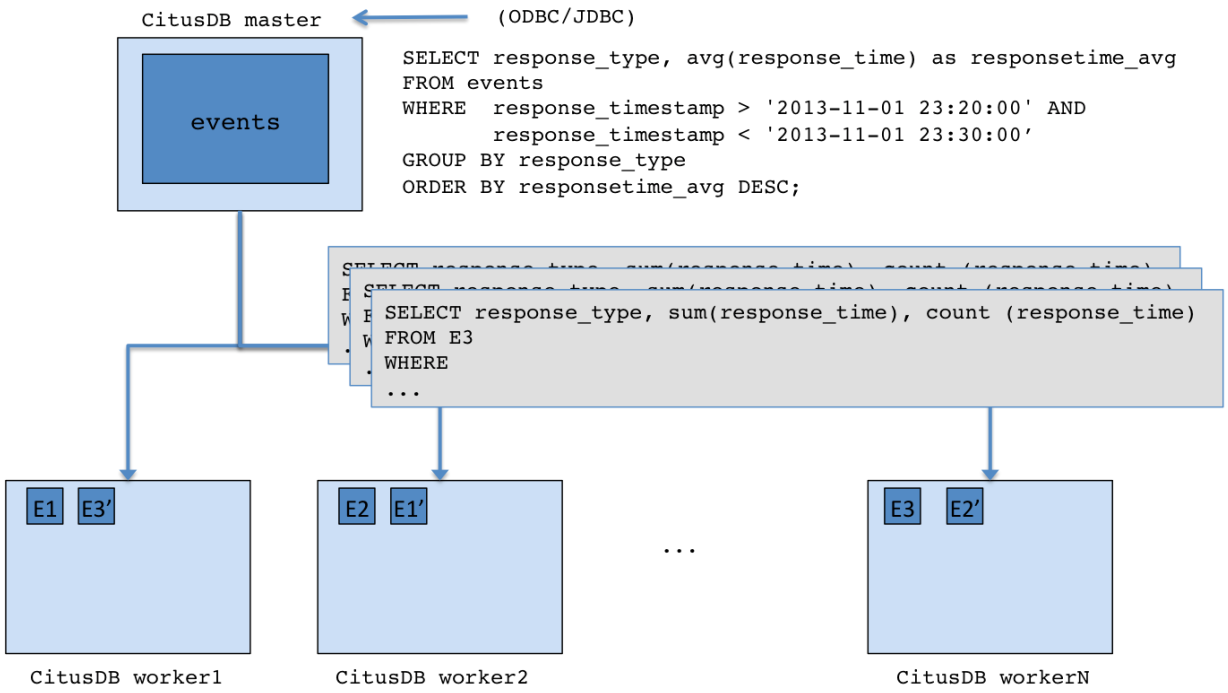
3.1.1 What is CitusDB?

CitusDB is a distributed database that is optimized for real-time big data workloads. CitusDB scales out PostgreSQL across a cluster of physical or virtual servers through sharding and replication. The advanced CitusDB query engine then parallelizes incoming SQL queries across these servers to enable real-time responses. CitusDB also intelligently recovers from mid-query failures by automatically failing over to other replicas, allowing users to maintain high availability.

CitusDB is not a fork of PostgreSQL but rather extends it by using the hook and extension APIs. As a result, CitusDB can be used as a drop in replacement for PostgreSQL without making any changes at the application layer. CitusDB is rebased on all major PostgreSQL releases, allowing users to benefit from new PostgreSQL features and maintain compatibility with existing PostgreSQL tools.

3.1.2 Architecture

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



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

Master / Worker Nodes

The user chooses one of the nodes in the cluster as the master node. He then adds the names of worker nodes to a membership file on the master node. From that point on, the user interacts with the master node through standard PostgreSQL interfaces for data loading and querying. All the data is distributed across the worker nodes. The master node only stores metadata about the shards.

Logical Sharding

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

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

Metadata Tables

The CitusDB master node maintains metadata tables to track all of the cluster nodes and the locations of the database shards on those nodes. These tables also maintain statistics like size and min/max values about the shards which help CitusDB'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 node ever experiences a failure.

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

```
SELECT * from pg_dist_partition;
 logicalrelid | partmethod |                                                                                partkey
 -------------+------------+--------------------------------------------------------------------------------
       488843 | r          | {VAR :varno 1 :varattno 4 :vartype 20 :vartypmod -1 :varcollid 0 :varlevelsup
(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
  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 *Reference* section of our documentation.

Query processing

When the user issues a query, the master node partitions the query into smaller query fragments where each query fragment can be run independently on a shard. This allows CitusDB to distribute each query across the cluster nodes, utilizing the processing power of all of the involved nodes and also of individual cores on each node. The master node then assigns the query fragments to worker nodes, 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 node also applies optimizations that minimize the amount of data transferred across the network.

Failure Handling

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

3.2 Distributed DDL and DML

CitusDB 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 full compatibility with standard PostgreSQL tools such as pgAdmin, pg_backup, and pg_upgrade.

CitusDB users can leverage standard PostgreSQL interfaces with no or minimal modifications. 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 schema creation: the distribution column and distribution method.

3.2.1 Distribution Column

Every distributed table in CitusDB 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. CitusDB's distributed query optimizer then leverages these distribution column ranges to determine how best a query should be executed.

Typically, users 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, CitusDB 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 CitusDB 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 CitusDB to push down several operations directly to the worker shards, hence reducing network I/O.

Note: CitusDB 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 CitusDB 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 CitusDB 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.

3.2.2 Distribution Method

The next step after choosing the right distribution column is deciding the right distribution method. CitusDB supports two distribution methods: append and hash. CitusDB 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. Users then distribute their largest tables by time, and batch load their events into CitusDB 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 CitusDB 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 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. In this case, CitusDB will maintain minimum and maximum hash ranges for all the created shards. Whenever a row is inserted,

updated or deleted, CitusDB 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.

CitusDB uses different syntaxes for creation and manipulation of append and hash distributed tables. Also, the operations / commands supported on the tables differ based on the distribution method chosen. In the sections below, 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 users can setup range partitioning manually.

3.2.3 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.

Creating Tables

To create an append distributed table, users can connect to the master node and create a table by issuing a CREATE TABLE command like below.

```
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
) DISTRIBUTE BY APPEND (created_at);
```

This command creates a new distributed table, and has almost the same syntax as PostgreSQL's CREATE TABLE command. The only difference is that the command has a distribute by append clause at the end, which tells the database about the distribution column. Note that this method doesn't in fact 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.

Data Loading

CitusDB 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, CitusDB provides two user defined functions. We describe each of the methods and their usage below.

Bulk load using stage

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

The stage command is used to copy data from a file to a distributed table while handling replication and failures automatically. This command borrows its syntax from the client-side copy command in PostgreSQL. Behind the covers, stage first opens a connection to the master node and fetches candidate worker nodes to create new shards

on. Then, the command connects to these worker nodes, creates at least one shard there, and uploads the data to the shards. The command then replicates these shards on other worker nodes till the replication factor is satisfied, and fetches statistics for these shards. Finally, the command stores the shard metadata with the master node.

CitusDB 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 entries for its behavior. These two entries are called shard_max_size and shard_replication_factor, and they live in postgresql.conf.

1. **shard_max_size :-** shard_max_size 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. **shard_replication_factor :-** This entry determines the number of nodes each shard gets replicated to, and defaults to two nodes. 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 UDFs described below to append to previously created shards.

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, CitusDB has 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 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(), the user 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, the user 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)
```

Please note that it is not recommended to run master_create_empty_shard and master_append_table_to_shard in the same transaction. This is because master_create_empty_shard holds a lock on the shard till the end of transaction and this can lead to reduced concurrency.

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

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, CitusDB 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
 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.

Dropping Tables

CitusDB users can use the standard PostgreSQL DROP TABLE command to remove their append distributed tables. As with regular tables, DROP TABLE removes any indexes, rules, triggers, and constraints that exist for the target table.

Please note that this only removes the table's metadata, but doesn't delete the table's shards from the worker nodes. If you wants to drop the shards before removing the table, you can first delete all shards using the *master_apply_delete_command UDF* as discussed in the previous section and then issue the DROP table command.

```
DROP TABLE github_events CASCADE ;
NOTICE:  drop cascades to distribute by append
NOTICE:  removing only metadata for the distributed table
DETAIL:  shards for this table are not removed
DROP TABLE
```

3.2.4 Hash Distribution

Hash distributed tables in CitusDB can be created with the help of pg_shard. In the next few sections, we describe how users can create and distribute tables using the hash distribution method, and do real time inserts and updates to their data in addition to analytics.

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.

```
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 need to specify the distribution column for the table. This can be done by using the master_create_distributed_table() UDF.

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

This function informs the database 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, 2);
```

This UDF takes two arguments in addition to the table name; shard count and the replication factor. This example would create a total of 16 shards where each shard owns a portion of a hash token space and gets replicated on 2 worker nodes. The shard replicas created on the worker nodes have the same table schema, index, and constraint definitions as the table on the master node. Once all replicas are created, this function saves all distributed metadata on the master node.

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

Inserting Data

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":

INSERT INTO github_events VALUES (2489368389,'WatchEvent','t',28229924,'{"action": "started"}','{"id":
```

When inserting rows into hash distributed tables, it is required that the distribution column of the row being inserted must be specified. To execute the insert query, first the value of the distribution column in the incoming row is hashed.

Then, the database determines the right shard in which the row should go by looking at the corresponding hash ranges of the shards. On the basis of this information, the query is forwarded to the right shard, and the remote insert command is executed on all the replicas of that shard.

Bulk inserts

Sometimes, users may want to bulk load several rows together into their 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 CitusDB installation.

Before invoking the script, we set the PATH environment variable to include the CitusDB bin directory.

```
export PATH=/opt/citusdb/4.0/bin/:$PATH
```

Next, 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.

```
/opt/citusdb/4.0/bin/copy_to_distributed_table -C github_events-2015-01-01.csv github_events
```

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

```
/opt/citusdb/4.0/bin/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*.

Updating and Deleting Data

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

```
UPDATE github_events SET org = NULL WHERE repo_id = 5152285;
DELETE FROM github_events WHERE repo_id = 5152285;
```

Currently, we require 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.

Dropping Tables

CitusDB users can use the standard PostgreSQL DROP TABLE command to remove their hash distributed tables. As with regular tables, DROP TABLE removes any indexes, rules, triggers, and constraints that exist for the target table.

Please note that this only removes the table's definition but not the metadata or the table's shards from the worker nodes. If disk space becomes an issue, users can connect to the worker nodes and manually drop the old shards for their hash partitioned tables.

```
postgres=# DROP TABLE github_events;
DROP TABLE
```

3.2.5 Range Distribution (Manual)

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

The syntax for creating a table using range distribution is very similar to a table using append distribution i.e. with the DISTRIBUTE BY clause.

```
/opt/citusdb/4.0/bin/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
) DISTRIBUTE BY RANGE (repo_id);
```

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 describe you can prepare your github_events data for staging into a range partitioned table distributed by repo_id (as shown above). For that, we first 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, we first 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, we 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, we 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, we 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, we load the files using the stage command.

```
/opt/citusdb/4.0/bin/psql -h localhost -d postgres
\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;
\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, CitusDB can easily prune away unrelated shards and ensure that the query hits only one shard. Also, groupings and orderings on the repo_id can be easily pushed down the worker nodes 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 CitusDB'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 worker nodes 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 at the master node.

With this, we end our discussion of how you can create and modify append, hash and range distributed tables using CitusDB. In the section below, we discuss how you can run your analytics queries. Please note that the sections below are independent of the distribution method chosen.

3.3 Querying

As discussed in the previous sections, CitusDB merely extends the the latest PostgreSQL version. This means that users can use standard PostgreSQL SELECT commands on the master node for querying. CitusDB then parallelizes the SELECT queries involving look-ups, complex selections, groupings and orderings, and JOINs to speed up the query performance. At a high level, CitusDB partitions the SELECT query into smaller query fragments, assigns these query fragments to worker nodes, 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 users can run using CitusDB.

3.3.1 Aggregate Functions

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

Count (Distinct) Aggregates

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

To address the common use case of count(distinct) approximations, CitusDB 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. Create the hll extension on all the nodes (the master node and all the worker nodes). The extension already comes installed with the citusdb contrib package.

```
CREATE EXTENSION hll;
```

2. Enable count distinct approximations by setting the 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 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.

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 CitusDB.

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
);
```

The users can call sum(hll_column) to roll up those columns within CitusDB. Please note that these custom aggregates need to be created both on the master node and the worker node.

3.3.2 Limit Pushdown

CitusDB also pushes down the limit clauses to the shards on the worker nodes 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, CitusDB provides an option for network efficient approximate LIMIT clauses.

LIMIT approximations are disabled by default and can be enabled by setting the configuration parameter limit_clause_row_fetch_count. On the basis of this configuration value, CitusDB will limit the number of rows returned by each task for master node aggregation. 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 worker nodes.

```
SET limit_clause_row_fetch_count to 10000;
```

3.3.3 Joins

CitusDB 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, CitusDB 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 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.

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 users want to join the keys in the fact tables (large table) with their corresponding dimension tables (small tables). CitusDB replicates the small table to all nodes where the large table's shards are present. Then, all the joins are performed locally on the worker nodes in parallel. Subsequent join queries that involve the small table then use these cached shards.

Colocated joins

To join two large tables efficiently, it is advised that you distribute them on the same column that you use to join the tables. In this case, the CitusDB master node knows which shards of the tables can have a potential matches with shards of the other tables by looking at the distribution column metadata. This allows CitusDB to prune away shard pairs which cannot produce matching join keys. The joins between remaining shard pairs are executed in parallel on the worker nodes and then the results are returned to the master.

Note: In order to take the most benefit of co-located joins, you should use the hash distribute your tables on the join key with the same number of shards. In this case, all shards will join with exactly one shard of the other table. Also, the shard creation logic will ensure that all shards with the same distribution key ranges are on the same worker nodes. This would mean that no data needs to be transferred between the workers, leading to faster joins.

Repartition joins

In some cases, users may need to join two tables on columns other than the distribution column and irrespective of distribution method. For such cases, CitusDB 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, users should try to distribute their tables by the common join keys whenever possible.

3.3.4 Data Warehousing Queries

We know we have more to do before we can declare CitusDB complete. Still, we have a version which works well for real-time analytics and that many companies rely on in production today. We are continuously working to increase SQL coverage to better support data warehousing use-cases. In the mean-time, since CitusDB is based on 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 feature 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 node. In such cases, it is possible to run data warehousing queries in two steps by storing the results of the inner queries in regular PostgreSQL tables on the master node. Then, the next step can be executed on the master node like a regular PostgreSQL query.

For example, currently CitusDB does not have out of the box support for window functions. 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;
```

We 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 complex subqueries or outer joins.

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 the database.

3.3.5 Query Performance

CitusDB parallelizes incoming queries by breaking the incoming SQL query into multiple fragment queries ("tasks") which run independently on the worker node shards in parallel. This allows CitusDB 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, users 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 processing times versus PostgreSQL on a single server.

CitusDB 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 worker nodes in parallel. As discussed above, 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.

CitusDB's distributed executor then takes these individual query fragments and sends them to worker nodes. 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 worker nodes, the second phase of query optimization kicks in. The worker nodes 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 CitusDB. 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 *Performance Tuning* section of the documentation.

3.4 PostgreSQL extensions

As CitusDB is based on PostgreSQL and each node in a CitusDB cluster is a nearly vanilla PostgreSQL (9.4), you can directly use Postgresql extensions such as hstore, hll, or PostGIS with CitusDB. The only difference is that you would need to run the create extension command on both the master and the worker nodes 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 CitusDB.

With this, we conclude our discussion around the usage of CitusDB. To learn more about the commands, UDFs or configuration parameters, you can visit the *Reference* section of our documentation. To continue learning more about CitusDB and its features, including database internals and cluster administration, you can visit our *Admin Guide*. If you cannot find documentation about a feature you are looking for, please contact us at engage@citusdata.com.

CHAPTER

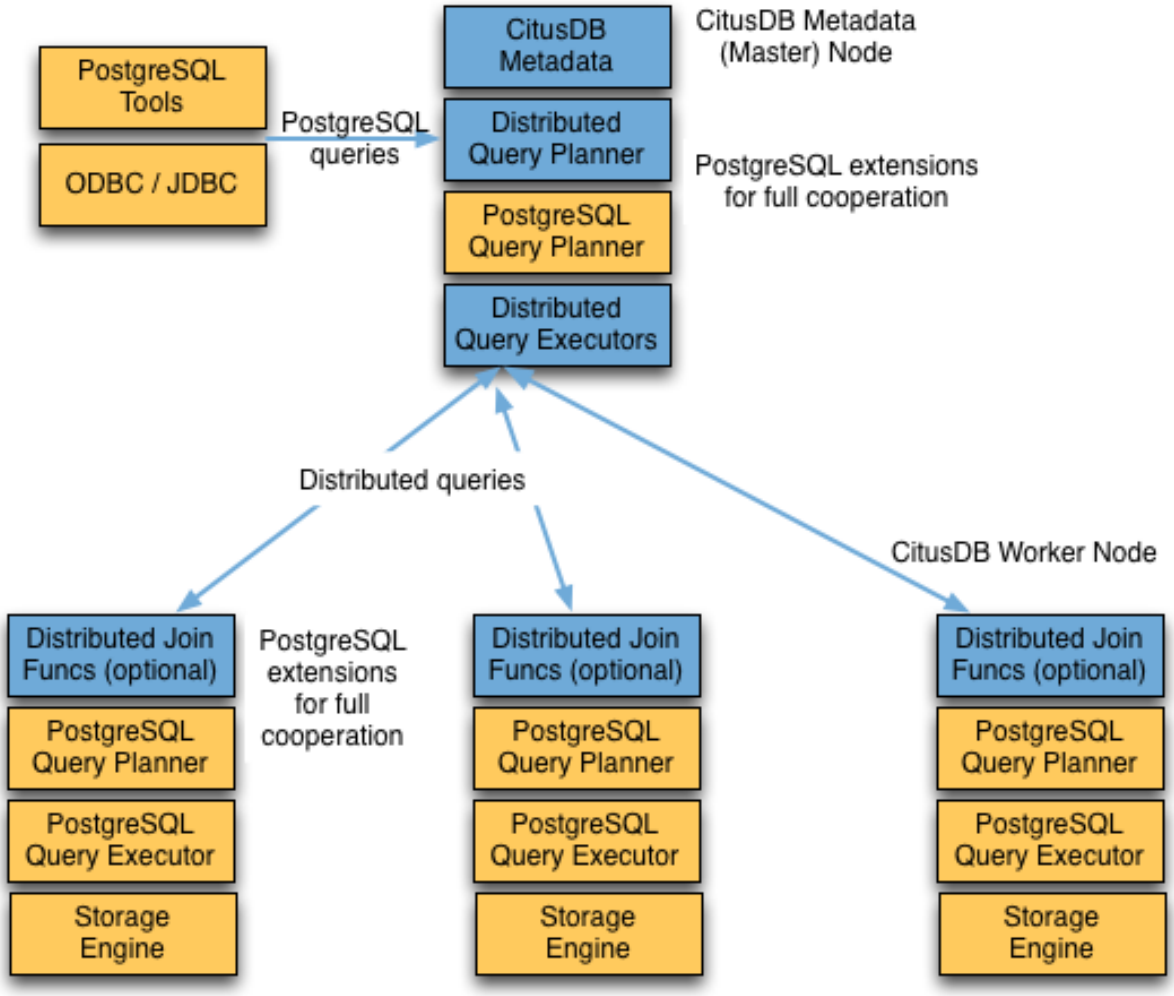
FOUR

ADMIN GUIDE

This section of our documentation describes how you can administer your CitusDB cluster. This includes tuning your cluster for high performance, adding / removing nodes, and upgrading your existing CitusDB installation to a newer version. We first begin by describing CitusDB's distributed query processing pipeline.

4.1 CitusDB Query Processing

CitusDB consists of a master node and multiple worker nodes. The data is sharded and replicated on the worker nodes while the master node stores only metadata about these shards. All queries issued to the cluster are executed via the master node. The master node partitions the query into smaller query fragments where each query fragment can be run independently on a shard. The master node then assigns the query fragments to worker nodes, 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.



CitusDB'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.

4.1.1 Distributed Query Planner

CitusDB'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 node and the worker query fragments which run on individual shards on the worker nodes. The planner then assigns these query fragments to the worker nodes such that all their resources are used efficiently. After this step, the distributed query plan is passed on to the distributed executor for execution.

For INSERT, UPDATE and DELETE queries, CitusDB requires that the query 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 hashes the partition column in the incoming row and looks at hash tokens for the shards in 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.

4.1.2 Distributed Query Executor

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

CitusDB has 3 different executor types - real time, task tracker and routing. The first two are used for SELECT queries while the routing executor is used for handling INSERT, UPDATE, and DELETE queries. We briefly discuss the executors below.

Real-time Executor

The real-time executor is the default executor used by CitusDB. It is well suited for queries which require quick responses like single key lookups, filters, aggregations and colocated joins. The real time executor opens one connection per shard to the worker nodes and sends all fragment queries to them. It then fetches the results from each fragment query, merges them, and gives them 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 worker nodes to avoid overwhelming them too many tasks. One can typically increase the file descriptor limit on modern OSes to avoid throttling, and change CitusDB 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.

Task Tracker Executor

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

Each task tracker daemon on the worker node also makes sure to execute at most 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 worker nodes.

Routing Executor

The routing executor is used by CitusDB for handling INSERT, UPDATE and DELETE queries. This executor assigns the incoming query to the worker node which have the target shard. The query is then handled by the PostgreSQL server on the worker node. In case a modification fails on a shard replica, the executor marks the corresponding shard replica as invalid in order to maintain data consistency.

4.1.3 PostgreSQL planner and executor

Once the distributed executor sends the query fragments to the worker nodes, they are processed like regular PostgreSQL queries. The PostgreSQL planner on that worker node 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 node for final aggregation.

4.2 Performance Tuning

In this section, we describe how users can tune their CitusDB cluster to get maximum performance for their queries. We begin by explaining how choosing the right distribution column and method affects performance. We then describe how you can first tune your database for high performance on one node 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.

4.2.1 Table Distribution and Shards

The first step while creating a CitusDB table is choosing the right distribution column and distribution method. CitusDB supports both append and hash based distribution; and both are better suited to certain use cases. Also, choosing the right distribution column helps CitusDB to 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, users 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, CitusDB 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 CitusDB 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 worker nodes and hence are more efficient.

In addition, CitusDB 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, CitusDB 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 CitusDB 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, CitusDB stores min / max ranges of the partition column in each shard, which allows for more efficient range queries on the partition 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.

4.2.2 PostgreSQL tuning

CitusDB partitions an incoming query into fragment queries, and sends them to the worker nodes for parallel processing. The worker nodes then apply PostgreSQL's standard planning and execution logic for these queries. So, the first step in tuning CitusDB is tuning the PostgreSQL configuration parameters on the worker nodes for high performance.

This step involves loading a small portion of the data into regular PostgreSQL tables on the worker nodes. 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 for the database. 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_segments and checkpoint_timeout settings. Also, depending on the reliability requirements of your application, you can choose to change fsync or synchronous_commit values.

4.2.3 Scaling Out Performance

Once you have achieved the desired performance on a single shard, you can set similar configuration parameters on all your worker nodes. As CitusDB 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 CitusDB. 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 node. CitusDB 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 CitusDB will prune away unrelated shards if the query has filters on the partition key. So, creating more shards than the number of cores might also be beneficial so that you can achieve greater parallelism even after shard pruning.

4.2.4 Distributed Query Performance Tuning

Once you have distributed your data across the cluster, with each worker node 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.

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.

An important performance tuning parameter in context of SELECT query performance is remote_task_check_interval. The master node assigns tasks to workers nodes, 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 node contact the workers more, incurring a higher overhead.

CitusDB has 2 different executor types for running SELECT queries. The desired executor can be selected by setting the task_executor_type configuration parameter. For shorter, high performance queries, users should try to use the real-time executor as much as possible. This is because the real time executor has a simpler architecture and a lower operational overhead. The task tracker executor is well suited for long running, complex queries which require dynamically repartitioning and shuffling data across worker nodes.

Other than the above, there are two configuration parameters which can be useful in cases where approximations produce meaningful results. These two parameters are the limit_clause_row_fetch_count and 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*.

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 CitusDB query planner assigns tasks to worker nodes based on shard locations. The algorithm used while making these assignments can be chosen by setting the 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 worker nodes. This policy is the default and works well in most of the cases. The **round-robin** policy assigns tasks to worker nodes 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 node. 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 worker nodes or between worker nodes and the master. CitusDB 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, binary_master_copy_format and 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 the worker node resources. 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 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 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 max_running_tasks_per_node without much concern.

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

4.3 Cluster Management

In this section, we discuss how you can manage your CitusDB cluster. This includes adding and removing nodes, dealing with node failures and upgrading your CitusDB cluster to a newer version. For moving shards across newly added nodes or replicating shards on failed nodes, users can use the shard rebalancer extension.

The shard rebalancer extension comes installed with the CitusDB contrib package. It mainly provides two functions for rebalancing / re-replicating shards for a distributed table. We discuss both the functions as and when relevant in the sections below. To learn more about these functions, their arguments and usage, you can visit the *Cluster Management Functions* reference.

4.3.1 Scaling out your cluster

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

Adding a worker node

CitusDB 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 node.

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 on the master node to cause it to reload its configuration.

```
select pg_reload_conf();
```

After this point, CitusDB will automatically start assigning new shards to that node. If you want to move existing shards to the newly added node, you can use the rebalance_table_shards UDF. This UDF will move the shards of the given table to make them evenly distributed among the worker nodes.

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

Note: You need to have the shard rebalancer extension created to use this UDF.

```
CREATE EXTENSION shard_rebalancer;
```

Adding a master node

The CitusDB master node only stores metadata about the table shards and does not store any data. This means that all the computation is pushed down to the worker nodes and the master node does only final aggregations on the result of

the workers. Therefore, it is not very likely that the master node becomes a bottleneck for read performance. Also, it is easy to boost up the master node by shifting to a more powerful machine.

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

4.3.2 Dealing With Node Failures

In this sub-section, we discuss how you can deal with node failures without incurring any downtime on your CitusDB cluster. We first discuss how CitusDB handles worker node failures automatically by maintaining multiple replicas of the data. We also 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 node.

Worker Node Failures

CitusDB can easily tolerate worker node failures because of its logical sharding-based architecture. While loading data, CitusDB allows you to specify the replication factor to provide desired availability for your data. In face of worker node failures, CitusDB automatically switches to these replicas to serve your queries and also provides options to bring back your cluster to the same level of availability. It also issues warnings like below on the master node 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 node from the pg_worker_list.conf file in the data directory.

```
vi /opt/citusdb/4.0/data/pg_worker_list.conf
```

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

```
SELECT pg_reload_conf();
```

After this step, CitusDB will stop assigning tasks or storing data on the failed node. Then, you can log into the failed worker node and inspect the cause of the failure. Depending on the type of failure, you can take actions as described below.

Temporary Node Failures

Temporary node failures can occur due to high load, maintenance work on the node, or due to network connectivity issues. As these failures are for a short duration, the primary concern in such cases is to be able to generate query responses even in face of failures.

If the node failure you are seeing is temporary, you can simply remove the worker node from pg_worker_list.conf (as discussed above). CitusDB will then automatically tackle these failures by re-routing the work to healthy nodes. If CitusDB is not able to connect a worker node, it automatically assigns that task to another worker node having a copy of that shard. If the failure occurs mid-query, CitusDB does not re-run the whole query but assigns only the failed tasks / 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.

Permanent Node Failures / Node decommissioning

If you realize that the node failure is permanent, CitusDB will continue to deal with the node failure by re-routing queries to the active nodes in the cluster. However, this may not be ideal for the long-term and in such cases, users may desire to retain the same level of replication so that their application can tolerate more failures. To do this, users can re-replicate the shards using the replicate_table_shards UDF after removing the failed / decommissioned node from pg_worker_list.conf. The replicate_table_shards() function replicates the shards of a table so they all reach the configured replication factor.

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

Note: You need to have the shard rebalancer extension installed to use this UDF.

```
CREATE EXTENSION shard_rebalancer;
```

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 node* section.

Master Node Failures

The CitusDB master node 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 node failures.

1. **Use PostgreSQL streaming replication:** You can use PostgreSQL's streaming replication feature to create a hot standby of the master node. 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. CitusDB'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.

4.3.3 Upgrading CitusDB

This section describes how you can upgrade your existing CitusDB installation to a later version. The upgrade process depends on the version from which you are upgrading. If you are upgrading between minor versions (i.e. 4.0 to 4.1), then the upgrade process is simpler as minor version upgrades in CitusDB are binary compatible. If you are upgrading between major CitusDB versions, then you can use the pg_upgrade utility. In the sections below, we describe the upgrade process for both these cases.

Minor Version Upgrades

Minor version upgrades in CitusDB are binary compatible. This means that you do not have to run pg_upgrade to upgrade between them. You can simply download and install the binaries for the new version and restart your server to upgrade to the new version.

Please note that these steps need to be run on all nodes in the cluster.

1. Download and install CitusDB packages for the new version

After downloading the new CitusDB package from our downloads page, you can install it using the appropriate package manager for your operating system.

For rpm based packages:

```
sudo rpm --install citusdb-4.0.1-1.x86_64.rpm
```

For debian packages

```
sudo dpkg --install citusdb-4.0.1-1.amd64.deb
```

2. Restart the CitusDB server

Once you have installed the packages, you can restart the server and it will automatically start using the binaries of the newer version.

```
/opt/citusdb/4.0/bin/pg_ctl -D /opt/citusdb/4.0/data -l logfile restart
```

Major Version Upgrades

If you are upgrading from CitusDB 3.0 to CitusDB 4.0, then 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. Apart from running pg_upgrade, there are 3 manual steps to be accounted for in order to update CitusDB.

1. Copy over pg_dist_* catalog tables.
2. Copy over pg_foreign_file/cached.
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 node and they are explicitly marked as such.

1. Download and install CitusDB 4.0 on the node having the to-be-upgraded 3.0 data directory

You can first download the new packages from our Downloads page. Then, you can install it using the appropriate command for your operating system.

For rpm based packages:

```
sudo rpm --install citusdb-4.0.1-1.x86_64.rpm
sudo rpm --install citusdb-contrib-4.0.1-1.x86_64.rpm
```

For debian packages:

```
sudo dpkg --install citusdb-4.0.1-1.amd64.deb
sudo dpkg --install citusdb-contrib-4.0.1-1.amd64.deb
```

Note that the 4.0 package will install at /opt/citusdb/4.0.

2. Setup environment variables for the data directories

```
export PGDATA4_0=/opt/citusdb/4.0/data
export PGDATA3_0=/opt/citusdb/3.0/data
```

3. Stop loading data on to that node

If you are upgrading the master node, then you should stop all data-loading/appending and staging before copying out the metadata. If data-loading continues after step 5 below, then the metadata will be out of date.

4. Copy out pg_dist catalog metadata from the 3.0 server (Only needed for master node)

```
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 4.0

```
/opt/citusdb/4.0/bin/initdb $PGDATA4_0
```

You can ignore this step if you are using the standard data directory which CitusDB creates by default while installing the packages.

6. Check upgrade compatibility

```
/opt/citusdb/4.0/bin/pg_upgrade -b /opt/citusdb/3.0/bin/ -B /opt/citusdb/4.0/bin/ -d $PGDATA3_0 -D
$PGDATA4_0
```

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 3.0 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 3.0 server

```
/opt/citusdb/3.0/bin/pg_ctl stop -D $PGDATA3_0
```

8. Run pg_upgrade, removing the –check flag

```
/opt/citusdb/4.0/bin/pg_upgrade -b /opt/citusdb/3.0/bin/ -B /opt/citusdb/4.0/bin/ -d $PGDATA3_0 -D
$PGDATA4_0
```

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

```
cp $PGDATA3_0/pg_worker_list.conf $PGDATA4_0/pg_worker_list.conf
```

10. Re-do changes to config settings in postgresql.conf and pg_hba.conf in the 4.0 data directory

- listen_addresses
- shard_max_size, shard_replication_factor etc
- performance tuning parameters
- enabling connections

11. Copy over the cached foreign data

```
cp -R $PGDATA3_0/pg_foreign_file/* $PGDATA4_0/pg_foreign_file/
```

12. Start the new 4.0 server

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

13. Copy over pg_dist catalog tables to the new server using the 4.0 psql client (Only needed for master-node)

```
/opt/citusdb/4.0/bin/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';
```

14. Restart the sequence pg_dist_shardid_seq (Only needed for master-node)

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

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 partitioned 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.

15. 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 3.0 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 3.0 / 4.0 mode. To do so, you can first upgrade the master node. Then, you can upgrade the worker nodes one at a time. This way you can upgrade the cluster with no downtime.

4.4 Transitioning From PostgreSQL to CitusDB

CitusDB can be used as a drop in replacement for PostgreSQL without making any changes at the application layer. Since CitusDB extends PostgreSQL, users can benefit from new PostgreSQL features and maintain compatibility with existing PostgreSQL tools.

Users can easily migrate from an existing PostgreSQL installation to CitusDB by copying out their data and then using the stage command or the copy script to load in their data. CitusDB also provides the master_append_table_to_shard UDF for users who want to append their PostgreSQL tables into their distributed tables. Then, you can also create all your extensions, operators, user defined functions, and custom data types on your CitusDB cluster just as you would do with PostgreSQL.

Please get in touch with us at engage@citusdata.com if you want to scale out your existing PostgreSQL installation with CitusDB.

With this, we end our discussion regarding CitusDB administration. You can read more about the commands, user defined functions and configuration parameters in the *Reference* section.

CHAPTER

FIVE

REFERENCE

The entries in this reference are meant to provide in reasonable length a formal summary about their respective subjects. More information about the use of CitusDB in narrative, tutorial, or example form, can be found in other parts of the documentation.

5.1 CitusDB SQL Language Reference

CitusDB uses SQL as its query language. As CitusDB provides distributed functionality by extending PostgreSQL, it is compatible with all PostgreSQL constructs. This means that users can use all the tools and features that come with the rich and extensible PostgreSQL ecosystem. 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 9.4 documentation.

To learn about the new features in PostgreSQL 9.4 on which the current CitusDB version is based, you can see the PostgreSQL 9.4 release notes.

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

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

5.2 User Defined Functions Reference

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

5.2.1 Append Distributed Tables

master_create_empty_shard

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

Arguments

table_name: Name of the 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)
```

master_append_table_to_shard

The master_append_table_to_shard() function can be used to append a PostgreSQL table's contents to a given shard. Behind the covers, the function connects to each of the worker nodes which have a replica of the shard and appends the contents of the table to each of them. Then, the function updates metadata for the shard replicas 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 replica of the shard, the function will return successfully. It will also mark any replica to which the append failed as INACTIVE so that any future queries do not consider that shard and issue a warning. If the append fails for all replicas, 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)
```

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 node. If the function is able to successfully delete a replica of the shard, then the metadata for it is deleted. If a particular replica could not be deleted, then it is marked as TO DELETE. The replicas 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 replicas.

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)
```

5.2.2 Hash Distributed Tables

master_create_distributed_table

The master_create_distributed_table() function is used to create a hash distributed table in CitusDB. This function takes in a table name and the distribution column and inserts metadata to mark the table as a hash distributed table.

Arguments

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

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

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');
```

master_create_worker_shards

The master_create_worker_shards() function creates a specified number of shards with the desired replication factor for a hash distributed table. Behind the covers, the function first divides the hash token space spanning between -2 Billion and 2 Billion equally among the desired number of shards and assigns a portion of the space to each shard. The function then connects to the worker nodes and creates shards on them until the replication factor is satisfied for each shard. Once all replicas are created, this function saves all distributed metadata on the master node.

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 worker nodes.

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

5.2.3 Metadata / Configuration Information

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)
```

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 column, replication count, maximum shard size and the shard placement policy for that table. Behind the covers, this function queries CitusDB 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_key: Distribution column / partition key 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_key    | part_replica_count | part_max_size | part_placement_policy
---------------+-------------------+---------------+--------------------+---------------+-----------------------
       115988  | t                 | created_hour |                  2 |    1073741824 |
(1 row)
```

5.2.4 Cluster Management Functions

rebalance_table_shards

The rebalance_table_shards() function moves shards of the given table to make them evenly distributed among the worker nodes. 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 each shard 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 to rebalance

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}');
```

replicate_table_shards

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 to replicate.

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);
```

5.3 Metadata Tables Reference

CitusDB divides each distributed table into multiple logical shards based on the distribution column. The master node 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.

5.3.1 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 :varlev

(1 row)
```

5.3.2 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 partitioned tables, these statistics correspond to min / max values of the distribution column. In case of hash partitioned 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
--------------+---------+--------------+------------+---------------+---------------
       488843 |  102065 | t            |            | 27            | 14995004
       488843 |  102066 | t            |            | 15001035      | 25269705
       488843 |  102067 | t            |            | 25273785      | 28570113
       488843 |  102068 | t            |            | 28570150      | 28678869
```

```
(4 rows)
```

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)

5.3.3 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 partitioned tables, the size of the shard placement on the worker node in bytes. For hash partitioned 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)
```

Shard Placement States

CitusDB 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 or a modification (INSERT, UPDATE or DELETE) 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 CitusDB 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.

5.4 Configuration Reference

There are various configuration parameters that affect the behaviour of CitusDB. These include both standard PostgreSQL parameters and CitusDB 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 CitusDB specific configuration parameters. These parameters can be set similar to PostgreSQL parameters by modifying postgresql.conf or by using the SET command.

5.4.1 Node configuration

pg_worker_list.conf

The CitusDB master node 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 node. To add this information, you need to append the DNS names and port numbers on which the workers are listening 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 lists worker-101 and worker-102 as worker nodes on the master-100 node.

```
master-100# vi /opt/citusdb/3.0/data/pg_worker_list.conf
# HOSTNAME      [PORT]  [RACK]
worker-101
worker-102
```

max_worker_nodes_tracked (integer)

CitusDB 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.

5.4.2 Data Loading

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 node. 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.

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 node.

shard_placement_policy (enum)

Sets the policy to use when choosing nodes for placing newly created shards. When using the stage command, the master node needs to choose the worker nodes on which it will place the new shards. This configuration value is applicable on the master node 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.

5.4.3 Planner Configuration

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 node.

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 node.

count_distinct_error_rate (floating point)

CitusDB calculates 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 node.

task_assignment_policy (enum)

Sets the policy to use when assigning tasks to worker nodes. The master node assigns tasks to worker nodes 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 worker nodes.
- **round-robin:** The round-robin policy assigns tasks to worker nodes 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 node 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 node.

5.4.4 Intermediate Data Transfer Format

binary_worker_copy_format (boolean)

Use the binary copy format to transfer intermediate data between worker nodes. During large table joins, CitusDB may have to dynamically repartition and shuffle data between different worker nodes. 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 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 this change to take effect.

binary_master_copy_format (boolean)

Use the binary copy format to transfer data between master node and the workers. When running distributed queries, the worker nodes transfer their intermediate results to the master node 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 node.

5.4.5 Executor Configuration

pg_shard.use_citusdb_select_logic (boolean)

Informs the database that CitusDB select logic is to be used for a hash partitioned table. Hash partitioned tables in CitusDB are created using pg_shard. By default, pg_shard uses a simple executor which is more suited for single key lookups. Setting this parameter to true allows users to use the CitusDB executor logic for handling complex queries efficiently. This parameter can be set at runtime and it effective on the master node.

remote_task_check_interval (integer)

Sets the frequency at which CitusDB checks for job statuses and defaults to 10ms. The master node assigns tasks to workers nodes, 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 node 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.

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 node contact each worker node more frequently, which is an overhead you may not want to pay in this case.

task_executor_type (enum)

CitusDB has 2 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:

- **real-time:** The real-time executor is the default executor and is well suited for queries which require quick responses.
- **task-tracker:** The task-tracker executor is well suited for long running, complex queries and for efficient resource management.

This parameter can be set at run-time and is effective on the master node. For high performance queries requiring sub-second responses, users should try to use the real-time executor as it has a simpler architecture and lower operational overhead. On the other hand, the task tracker executor is well suited for long running, complex queries which require dynamically repartitioning and shuffling data across worker nodes.

For more details about the executors, you can visit the *Distributed Query Executor* section of our documentation.

Real-time executor configuration

The CitusDB 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 node 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.

Task tracker executor configuration

task_tracker_active (boolean)

The task tracker background process runs on every worker node, and manages the execution of tasks assigned to it. This configuration entry activates the task tracker and is set to on by default. This parameter can only be set at server start and is effective on both - the master and worker nodes. This parameter indicates if you want to start the task tracker process or not. If you set this value to off, the task tracker daemon will not start, and you will not be able to use the task-tracker executor type.

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 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 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.

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 worker nodes.

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, CitusDB 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.

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.

partition_buffer_size (integer)

Sets the buffer size to use for partition operations and defaults to 8MB. CitusDB 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 worker nodes.