Citus Documentation

Release 6.0.1

Citus Data

Apr 07, 2022

About Citus

Welcome to the documentation for Citus 6.0! Citus horizontally scales PostgreSQL across commodity servers using sharding and replication. Its query engine parallelizes incoming SQL queries across these servers to enable real-time responses on large datasets.

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

CHAPTER 1

What is Citus?

Citus horizontally scales PostgreSQL across multiple machines using sharding and replication. Its query engine parallelizes incoming SQL queries across these servers to enable human real-time (less than a second) responses on large datasets.

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

1.1 When to Use Citus

There are two situations where Citus particularly excels: real-time analytics and multi-tenant applications.

1.1.1 Real-Time Analytics

Citus supports real-time queries over large datasets. Commonly these queries occur in rapidly growing event systems or systems with time series data. Example use cases include:

- Analytic dashboards with subsecond response times
- Exploratory queries on unfolding events
- Large dataset archival and reporting
- Analyzing sessions with funnel, segmentation, and cohort queries

Citus' benefits here are its ability to parallelize query execution and scale linearly with the number of worker databases in a cluster.

For concrete examples check out our customer use cases.

1.1.2 Multi-Tenant Applications

Another Citus use case is managing the data for multi-tenant applications. These are applications where a single database cluster serves multiple tenants (typically companies), each of whose data is private from the other tenants.

All tenants share a common schema and Citus distributes their data across shards. Citus routes individual tenant queries to the appropriate shard, each of which acts like a standalone database with full-featured SQL support.

This allows you to scale out your tenants across several machines and CPU cores, adding more memory and processing power for parallelism. Sharing a schema and cluster infrastructure among multiple tenants also uses hardware efficiently and reduces maintenance costs compared with a one-tenant-per-database instance model.

1.2 Considerations for Use

Citus extends PostgreSQL with distributed functionality, but it is not a drop-in replacement that scales out all workloads. A performant Citus cluster involves thinking about the data model, tooling, and choice of SQL features used.

A good way to think about tools and SQL features is the following: if your workload aligns with use-cases noted in the *When to Use Citus* section and you happen to run into an unsupported tool or query, then there's usually a good workaround.

1.3 When Citus is Inappropriate

Workloads which require a large flow of information between worker nodes generally do not work as well. For instance:

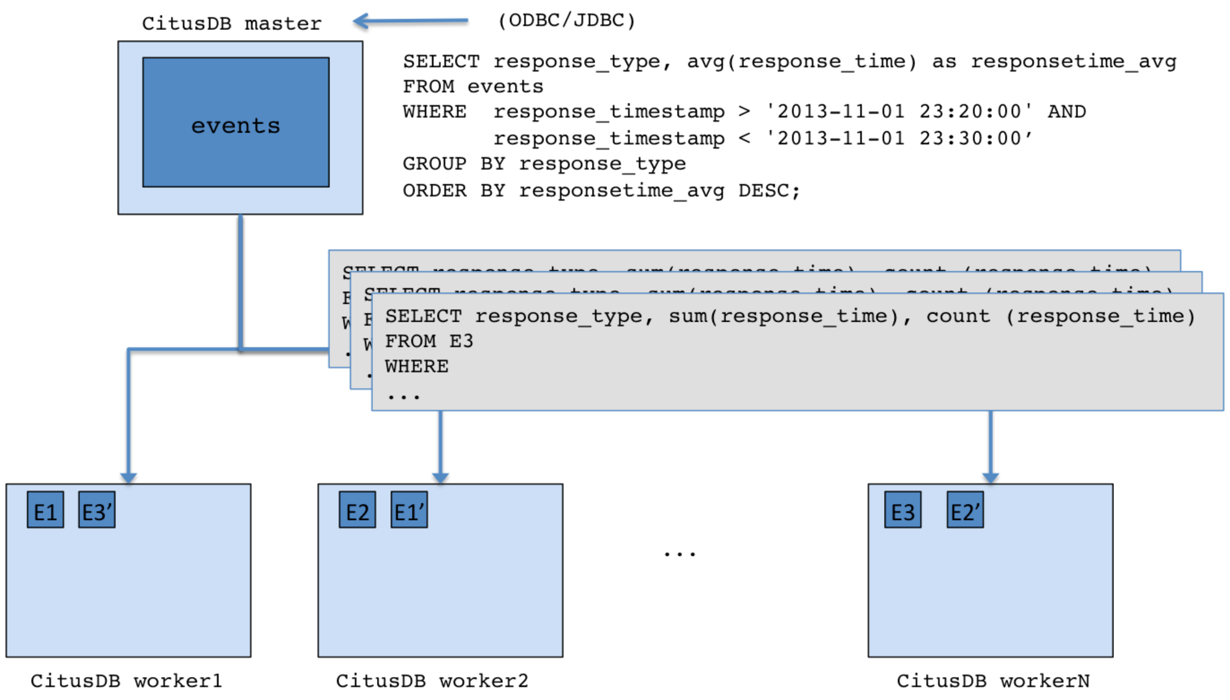
- Traditional data warehousing with long, free-form SQL
- Many distributed transactions across multiple shards
- Queries that return data-heavy ETL results rather than summaries

These constraints come from the fact that Citus operates across many nodes (as compared to a single node database), giving you easy horizontal scaling as well as high availability.

CHAPTER 2

Architecture

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



```
SELECT response_type, avg(response_time) as responsetime_avg
FROM events
WHERE  response_timestamp > '2013-11-01 23:20:00' AND
       response_timestamp < '2013-11-01 23:30:00'
GROUP BY response_type
ORDER BY responsetime_avg DESC;
```

```
SELECT response_type, sum(response_time), count (response_time)
FROM E3
WHERE
...
```

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

2.1 Master / Worker Nodes

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

2.2 Logical Sharding

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

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

2.3 Metadata Tables

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

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

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

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

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

(continues on next page)

(continued from previous page)

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

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

2.4 Query Processing

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

2.5 Failure Handling

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

CHAPTER 3

Hash-Distributed Data

3.1 Start Demo Cluster

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

1. Download the package

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

```
cd citus-tutorial
```

2. Initialize the cluster

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

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

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

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

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

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

3. Start the master and worker

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

Let's start the databases:

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

And initialize them:

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

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

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

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

```
bin/psql -p 9700 -c "SELECT * from master_add_node('localhost', 9701);"
```

3.2 Ingest Data

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

Now that your cluster is running, open a prompt to the master instance:

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

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

This time we're going to make two tables.

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

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

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

```
SET citus.shard_replication_factor = 1;

SELECT create_distributed_table('wikipedia_changes', 'editor');
SELECT create_distributed_table('wikipedia_editors', 'editor');
```

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

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

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

```
# - in a new terminal -

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

This should keep running and aggregating data on the users who are editting right now.

3.3 Run Queries

Let's run some queries! If you run any of these queries multiple times you'll see the results update as more data is ingested. Returning to our existing psql terminal we can ask some simple questions, such as finding edits which were made by bots:

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

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

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

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

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

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

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

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

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

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

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

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

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

CHAPTER 4

Requirements

Citus works with modern 64-bit Linux and most Unix based operating systems. Citus 6.0 requires PostgreSQL 9.5 or later versions.

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

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

CHAPTER 5

Single-Machine Cluster

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

5.1 OS X

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

1. Install PostgreSQL 9.6 and the Citus extension

Use our Homebrew package to extend PostgreSQL with Citus.

```
brew install citus
```

2. Initialize the Cluster

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

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

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

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

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

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

3. Start the master and workers

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

Let's start the databases:

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

And initialize them:

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

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

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

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

```
psql -p 9700 -c "SELECT * from master_add_node('localhost', 9701);"
psql -p 9700 -c "SELECT * from master_add_node('localhost', 9702);"
```

4. Verify that installation has succeeded

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

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

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

5.2 Fedora, CentOS, or Red Hat

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

1. Install PostgreSQL 9.6 and the Citus extension

```
# Add Citus repository for package manager
curl https://install.citusdata.com/community/rpm.sh | sudo bash

# install Citus extension
sudo yum install -y citus60_96
```

2. Initialize the Cluster

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

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

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

# include path to postgres binaries
export PATH=$PATH:/usr/pgsql-9.6/bin

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

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

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

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

3. Start the master and workers

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

Let’s start the databases:

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

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

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

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

```
psql -p 9700 -c "SELECT * from master_add_node('localhost', 9701);"
psql -p 9700 -c "SELECT * from master_add_node('localhost', 9702);"
```

4. Verify that installation has succeeded

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

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

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

5.3 Ubuntu or Debian

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

1. Install PostgreSQL 9.6 and the Citus extension

```
# Add Citus repository for package manager
curl https://install.citusdata.com/community/deb.sh | sudo bash

# install the server and initialize db
sudo apt-get -y install postgresql-9.6-citus-6.0
```

2. Initialize the Cluster

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

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

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

# include path to postgres binaries
export PATH=$PATH:/usr/lib/postgresql/9.6/bin

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

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

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

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

3. Start the master and workers

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

Let's start the databases:

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

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

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

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

```
psql -p 9700 -c "SELECT * from master_add_node('localhost', 9701);"
psql -p 9700 -c "SELECT * from master_add_node('localhost', 9702);"
```

4. Verify that installation has succeeded

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

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

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

5.4 Docker

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

1. Install docker and docker-compose

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

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

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

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

2. Start the Citus Cluster

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

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

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

3. Verify that installation has succeeded

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

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

Then run this query:

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

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

CHAPTER 6

Multi-Machine Cluster

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

6.1 Amazon Web Services

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

6.1.1 Managed Citus Cloud Deployment

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

6.1.2 Manual CloudFormation Deployment

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

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

Introduction

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

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

1. Start a Citus cluster

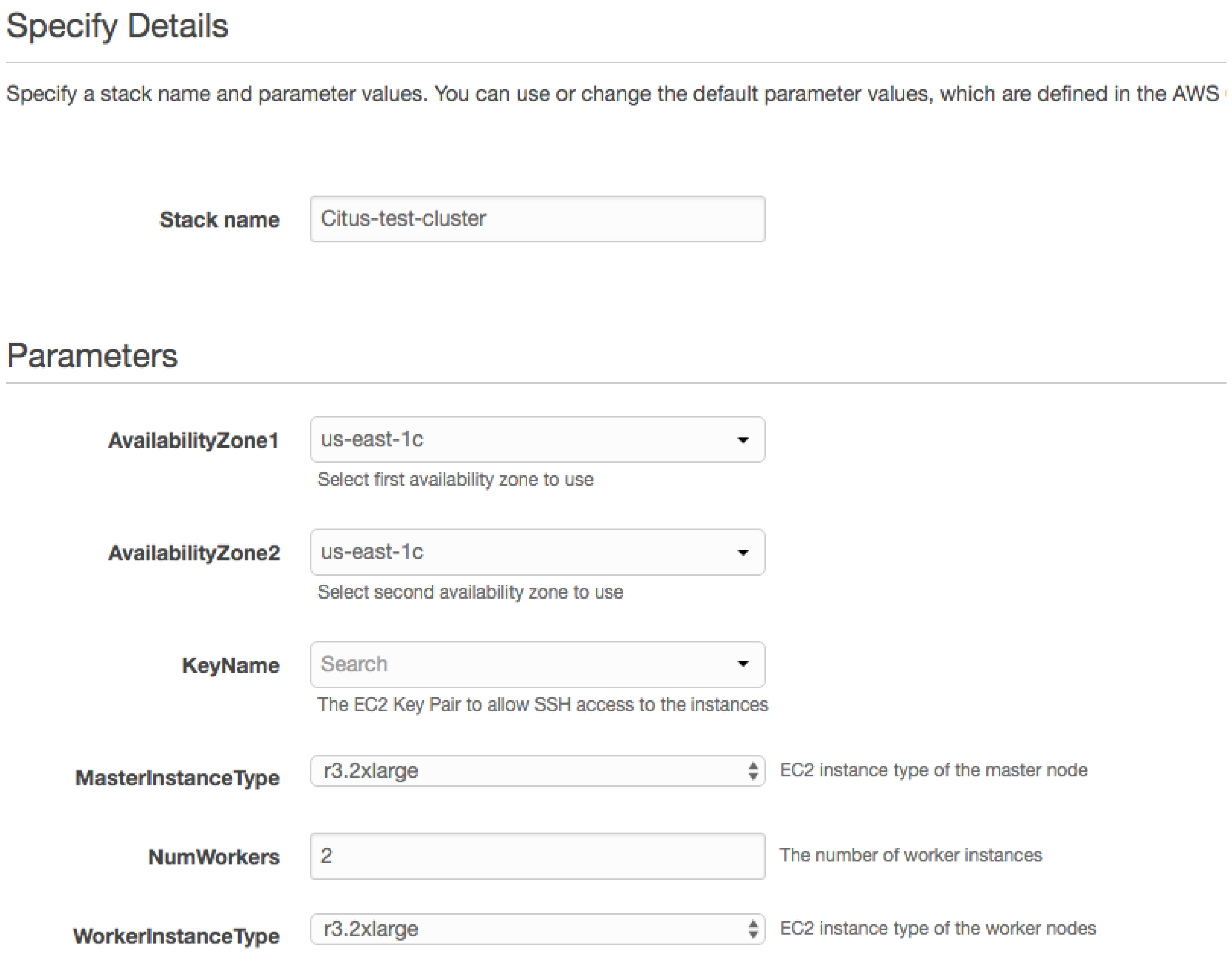
Note: You might need to login to AWS at this step if you aren't already logged in.

2. Select Citus template

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

3. Fill the form with details about your cluster

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



Specify Details

Specify a stack name and parameter values. You can use or change the default parameter values, which are defined in the AWS

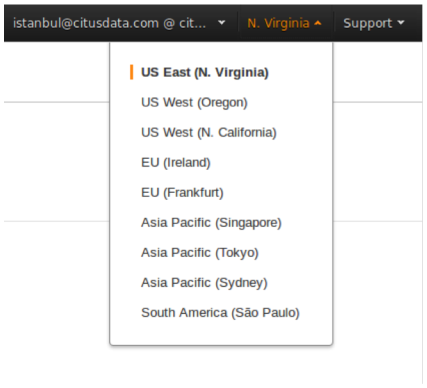
Stack name	Citus-test-cluster	

Parameters

AvailabilityZone1	us-east-1c	Select first availability zone to use
AvailabilityZone2	us-east-1c	Select second availability zone to use
KeyName	Search	The EC2 Key Pair to allow SSH access to the instances
MasterInstanceType	r3.2xlarge	EC2 instance type of the master node
NumWorkers	2	The number of worker instances
WorkerInstanceType	r3.2xlarge	EC2 instance type of the worker nodes

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

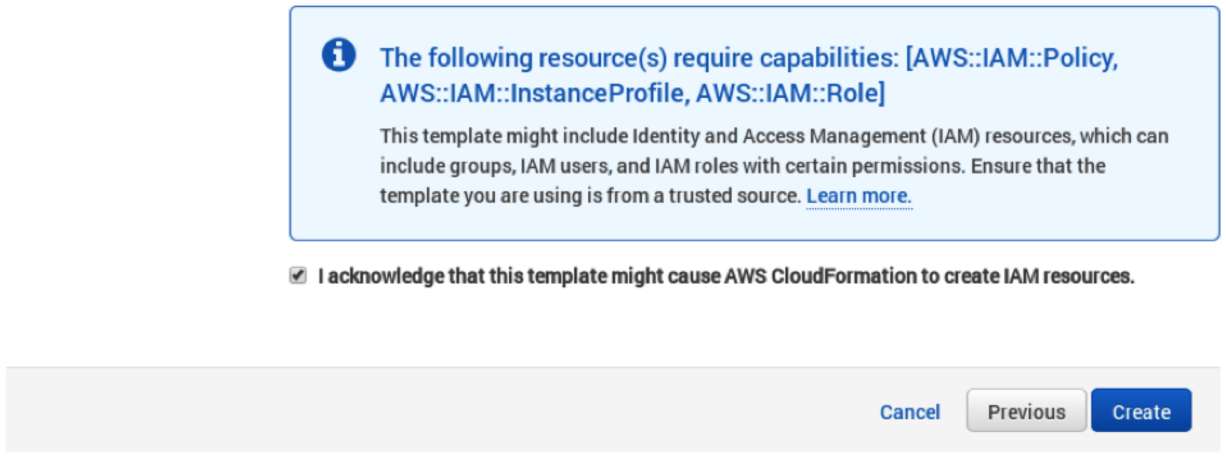
Note: If you want to launch a cluster in a region other than US East, you can update the region in the top-right corner of the AWS console as shown below.



4. Acknowledge IAM capabilities

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

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



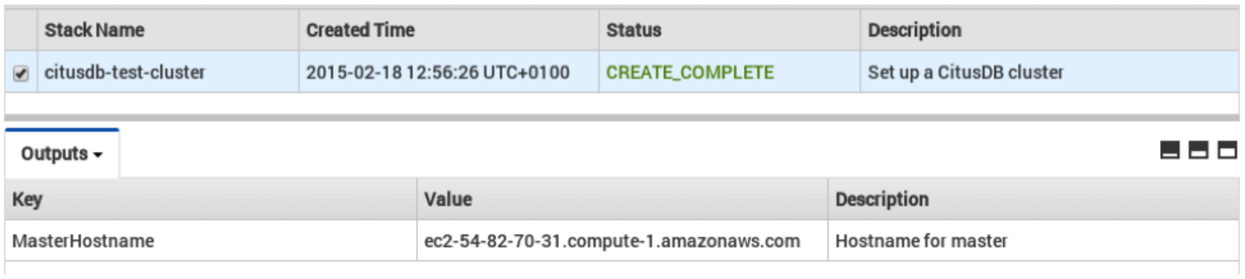
The following resource(s) require capabilities: [AWS::IAM::Policy, AWS::IAM::InstanceProfile, AWS::IAM::Role]

This template might include Identity and Access Management (IAM) resources, which can include groups, IAM users, and IAM roles with certain permissions. Ensure that the template you are using is from a trusted source. Learn more.

[✓] I acknowledge that this template might cause AWS CloudFormation to create IAM resources.

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.



	Stack Name	Created Time	Status	Description
✓	citusdb-test-cluster	2015-02-18 12:56:26 UTC+0100	CREATE_COMPLETE	Set up a CitusDB cluster

Key	Value	Description
MasterHostname	ec2-54-82-70-31.compute-1.amazonaws.com	Hostname for master

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

Troubleshooting:

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

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

6. Login to the cluster

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

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

7. Ready to use the cluster

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

```
/usr/pgsql-9.6/bin/psql -h localhost -d postgres
select * from master_get_active_worker_nodes();
```

8. Cluster notes

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

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

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

```
/usr/pgsql-9.6/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.

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

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

6.2.1 Steps to be executed on all nodes

1. Add repository

```
# Add Citus repository for package manager
curl https://install.citusdata.com/community/rpm.sh | sudo bash
```

2. Install PostgreSQL + Citus and initialize a database

```
# install PostgreSQL with Citus extension
sudo yum install -y citus60_96
# initialize system database (using RHEL 6 vs 7 method as necessary)
sudo service postgresql-9.6 initdb || sudo /usr/pgsql-9.6/bin/postgresql96-setup␣
↪initdb
# preload citus extension
echo "shared_preload_libraries = 'citus'" | sudo tee -a /var/lib/pgsql/9.6/data/
↪postgresql.conf
```

PostgreSQL adds version-specific binaries in */usr/pgsql-9.6/bin*, but you'll usually just need psql, whose latest version is added to your path, and managing the server itself can be done with the *service* command.

3. Configure connection and authentication

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

```
sudo vi /var/lib/pgsql/9.6/data/postgresql.conf
```

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

```
sudo vi /var/lib/pgsql/9.6/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

# Also allow the host unrestricted access to connect to itself
host    all             all             127.0.0.1/32            trust
host    all             all             ::1/128                 trust
```

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

4. Start database servers, create Citus extension

```
# start the db server
sudo service postgresql-9.6 restart
# and make it start automatically when computer does
sudo chkconfig postgresql-9.6 on
```

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

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

6.2.2 Steps to be executed on the master node

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

1. Add worker node information

We need to inform the master about its workers. To add this information, we call a UDF which adds the node information to the pg_dist_node catalog table, which the master uses to get the list of worker nodes. For our example, we assume that there are two workers (named worker-101, worker-102). Add the workers' DNS names (or IP addresses) and server ports to the table.

```
sudo -i -u postgres psql -c "SELECT * from master_add_node('worker-101', 5432);"
sudo -i -u postgres psql -c "SELECT * from master_add_node('worker-102', 5432);"
```

2. 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 we added to the pg_dist_node table above.

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

Ready to use Citus

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

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

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

6.3 Multi-node setup on Ubuntu or Debian

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

6.3.1 Steps to be executed on all nodes

1. Add repository

```
# Add Citus repository for package manager
curl https://install.citusdata.com/community/deb.sh | sudo bash
```

2. Install PostgreSQL + Citus and initialize a database

```
# install the server and initialize db
sudo apt-get -y install postgresql-9.6-citus-6.0

# preload citus extension
sudo pg_conftool 9.6 main set shared_preload_libraries citus
```

This installs centralized configuration in */etc/postgresql/9.6/main*, and creates a database in */var/lib/postgresql/9.6/main*.

3. Configure connection and authentication

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

```
sudo pg_conftool 9.6 main set listen_addresses '*'
```

```
sudo vi /etc/postgresql/9.6/main/pg_hba.conf
```

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

# Also allow the host unrestricted access to connect to itself
host    all             all             127.0.0.1/32            trust
host    all             all             ::1/128                 trust
```

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

4. Start database servers, create Citus extension

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

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

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

6.3.2 Steps to be executed on the master node

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

1. Add worker node information

We need to inform the master about its workers. To add this information, we call a UDF which adds the node information to the pg_dist_node catalog table. For our example, we assume that there are two workers (named worker-101, worker-102). Add the workers' DNS names (or IP addresses) and server ports to the table.

```
sudo -i -u postgres psql -c "SELECT * from master_add_node('worker-101', 5432);"
sudo -i -u postgres psql -c "SELECT * from master_add_node('worker-102', 5432);"
```

2. 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 we added to the pg_dist_node table above.

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

Ready to use Citus

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

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

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

Distributed data modeling refers to choosing how to distribute information across nodes in a multi-machine database cluster and query it efficiently. There are common use cases for a distributed database with well understood design tradeoffs. It will be helpful for you to identify whether your application falls into one of these categories in order to know what features and performance to expect.

Citus uses a column in each table to determine how to allocate its rows among the available shards. In particular, as data is loaded into the table, Citus uses the *distribution column* as a hash key to allocate each row to a shard.

The database administrator picks the distribution column of each table. Thus the main task in distributed data modeling is choosing the best division of tables and their distribution columns to fit the queries required by an application.

CHAPTER 7

Determining the Data Model

As explained in *When to Use Citus*, there are two common use cases for Citus. The first is building a **multi-tenant application**. This use case works best for B2B applications that serve other companies, accounts, or organizations. For example, this application could be a website which hosts store-fronts for other businesses, a digital marketing solution, or a sales automation tool. Applications like these want to continue scaling whether they have hundreds or thousands of tenants. (Horizontal scaling with the multi-tenant architecture imposes no hard tenant limit.) Additionally, Citus' sharding allows individual nodes to house more than one tenant which improves hardware utilization.

The multi-tenant model as implemented in Citus allows applications to scale with minimal changes. This data model provides the performance characteristics of relational databases at scale. It also provides familiar benefits that come with relational databases, such as transactions, constraints, and joins. Once you follow the multi-tenant data model, it is easy to adjust a changing application while staying performant. Citus stores your data within the same relational database, so you can easily change your table schema by creating indices or adding new columns.

There are characteristics to look for in queries and schemas to determine whether the multi-tenant data model is appropriate. Typical queries in this model relate to a single tenant rather than joining information across tenants. This includes OLTP workloads for serving web clients, and OLAP workloads that serve per-tenant analytical queries. Having dozens or hundreds of tables in your database schema is also an indicator for the multi-tenant data model.

The second common Citus use case is **real-time analytics**. The choice between the real-time and multi-tenant models depends on the needs of the application. The real-time model allows the database to ingest a large amount of incoming data and summarize it in "human real-time," which means in less than a second. Examples include making dashboards for data from the internet of things, or from web traffic. In this use case applications want massive parallelism, coordinating hundreds of cores for fast results to numerical, statistical, or counting queries.

The real-time architecture usually has few tables, often centering around a big table of device-, site- or user-events. It deals with high volume reads and writes, with relatively simple but computationally intensive lookups.

If your situation resembles either of these cases then the next step is to decide how to shard your data in a Citus cluster. As explained in *Architecture*, Citus assigns table rows to shards according to the hashed value of the table's distribution column. The database administrator's choice of distribution columns needs to match the access patterns of typical queries to ensure performance.

CHAPTER 8

Distributing by Tenant ID

The multi-tenant architecture uses a form of hierarchical database modeling to distribute queries across nodes in the distributed cluster. The top of the data hierarchy is known as the *tenant id*, and needs to be stored in a column on each table. Citus inspects queres to see which tenant id they involve and routes the query to a single physical node for processing, specifically the node which holds the data shard associated with the tenant id. Running a query with all relevant data placed on the same node is called *co-location*.

The first step is identifying what constitutes a tenant in your app. Common instances include company, account, organization, or customer. The column name will be something like `company_id` or `customer_id`. Examine each of your queries and ask yourself: would it work if it had additional WHERE clauses to restrict all tables involved to rows with the same tenant id? Queries in the multi-tenant model are usually scoped to a tenant, for instance queries on sales or inventory would be scoped within a certain store.

If you're migrating an existing database to the Citus multi-tenant architecture then some of your tables may lack a column for the application-specific tenant id. You will need to add one and fill it with the correct values. This will denormalize your tables slightly. For more details and a concrete example of backfilling the tenant id, see our guide to *Multi-Tenant Migration*.

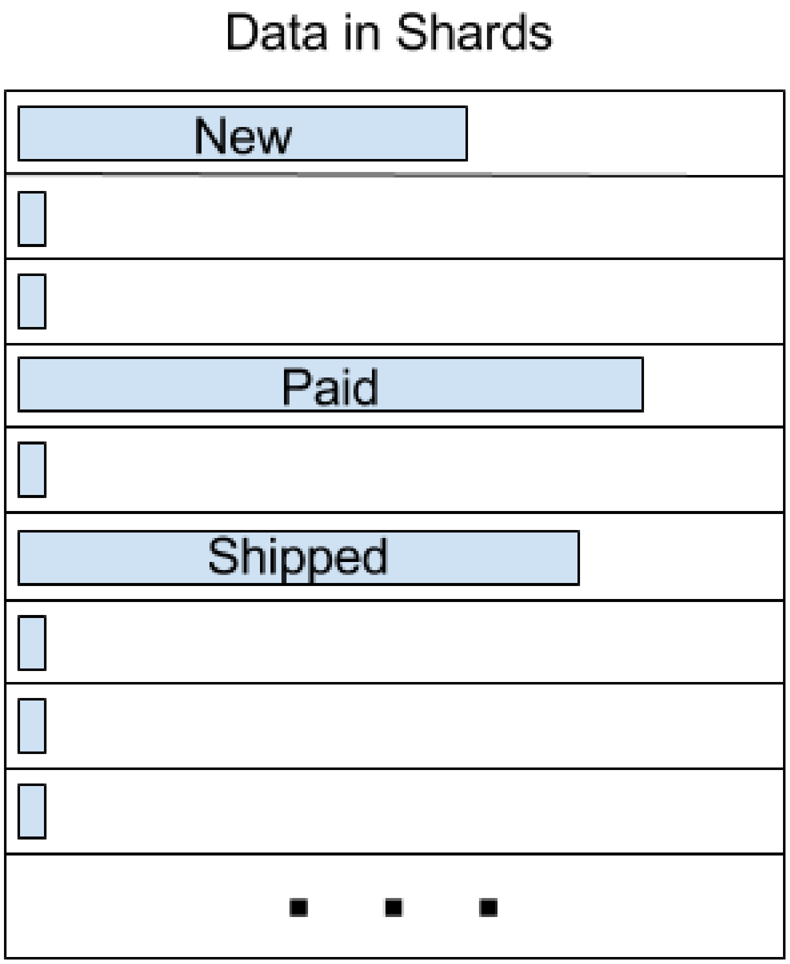
CHAPTER 9

Distributing by Entity ID

While the multi-tenant architecture introduces a hierarchical structure and uses data co-location to parallelize queries between tenants, real-time architectures depend on specific distribution properties of their data to achieve highly parallel processing. We use “entity id” as a term for distribution columns in the real-time model, as opposed to tenant ids in the multi-tenant model. Typical entites are users, hosts, or devices.

Real-time queries typically ask for numeric aggregates grouped by date or category. Citus sends these queries to each shard for partial results and assembles the final answer on the coordinator node. Queries run fastest when as many nodes contribute as possible, and when no individual node bottlenecks.

The more evenly a choice of entity id distributes data to shards the better. At the least the column should have a high cardinality. For comparison, a “status” field on an order table is a poor choice of distribution column because it assumes at most a few values. These values will not be able to take advantage of a cluster with many shards. The row placement will skew into a small handful of shards:



Of columns having high cardinality, it is good additionally to choose those that are frequently used in group-by clauses or as join keys. Distributing by join keys co-locates the joined tables and greatly improves join speed. Real-time schemas usually have few tables, and are generally centered around a big table of quantitative events.

9.1 Typical Real-Time Schemas

9.1.1 Events Table

In this scenario we ingest high volume sensor measurement events into a single table and distribute it across Citus by the `device_id` of the sensor. Every time the sensor makes a measurement we save that as a single event row with measurement details in a jsonb column for flexibility.

```
CREATE TABLE events (
  device_id bigint NOT NULL,
  event_id uuid NOT NULL,
  event_time timestamptz NOT NULL,
  event_type int NOT NULL,
  payload jsonb,
  PRIMARY KEY (device_id, event_id)
);
CREATE INDEX ON events USING BRIN (event_time);
```

(continues on next page)

(continued from previous page)

```

SELECT create_distributed_table('events', 'device_id');
```

Any query that restricts to a given device is routed directly to a worker node for processing. We call this a *single-shard* query. Here is one to get the ten most recent events:

```
SELECT event_time, payload
  FROM events
  WHERE device_id = 298
  ORDER BY event_time DESC
  LIMIT 10;
```

To take advantage of massive parallelism we can run a *cross-shard* query. For instance, we can find the min, max, and average temperatures per minute across all sensors in the last ten minutes (assuming the json payload includes a `temp` value). We can scale this query to any number of devices by adding worker nodes to the Citus cluster.

```
SELECT minute,
  min(temperature)::decimal(10,1) AS min_temperature,
  avg(temperature)::decimal(10,1) AS avg_temperature,
  max(temperature)::decimal(10,1) AS max_temperature
FROM (
  SELECT date_trunc('minute', event_time) AS minute,
         (payload->>'temp')::float AS temperature
  FROM events
  WHERE event_t1me >= now() - interval '10 minutes'
) ev
GROUP BY minute
ORDER BY minute ASC;
```

9.1.2 Events with Roll-Ups

The previous example calculates statistics at runtime, doing possible recalculation between queries. Another approach is precalculating aggregates. This avoids recalculating raw event data and results in even faster queries. For example, a web analytics dashboard might want a count of views per page per day. The raw events data table looks like this:

```
CREATE TABLE page_views (
  page_id int PRIMARY KEY,
  host_ip inet,
  view_time timestamp default now()
);
CREATE INDEX view_time_idx ON page_views USING BRIN (view_time);

SELECT create_distributed_table('page_views', 'page_id');
```

We will precompute the daily view count in this summary table:

```
CREATE TABLE daily_page_views (
  day date,
  page_id int,
  view_count bigint,
  PRIMARY KEY (day, page_id)
);

SELECT create_distributed_table('daily_page_views', 'page_id');
```

Precomputing aggregates is called *roll-up*. Notice that distributing both tables by `page_id` co-locates their data per-page. Any aggregate functions grouped per page can run in parallel, and this includes aggregates in roll-ups. We can use PostgreSQL UPSERT to create and update rollups, like this (the SQL below takes a parameter for the lower bound timestamp):

```
INSERT INTO daily_page_views (day, page_id, view_count)
SELECT view_time::date AS day, page_id, count(*) AS view_count
FROM page_views
WHERE view_time >= $1
GROUP BY view_time::date, page_id
ON CONFLICT (day, page_id) DO UPDATE SET
  view_count = daily_page_views.view_count + EXCLUDED.view_count;
```

9.1.3 Events and Entities

Behavioral analytics seeks to understand users, from the website/product features they use to how they progress through funnels, to the effectiveness of marketing campaigns. Doing analysis tends to involve unforeseen factors which are uncovered by iterative experiments. It is hard to know initially what information about user activity will be relevant to future experiments, so analysts generally try to record everything they can. Using a distributed database like Citus allows them to query the accumulated data flexibly and quickly.

Let's look at a simplified example. Whereas the previous examples dealt with a single events table (possibly augmented with precomputed rollups), this one uses two main tables: users and their events. In particular, Wikipedia editors and their changes:

```
CREATE TABLE wikipedia_editors (
  editor TEXT UNIQUE,
  bot BOOLEAN,

  edit_count INT,
  added_chars INT,
  removed_chars INT,

  first_seen TIMESTAMPTZ,
  last_seen TIMESTAMPTZ
);

CREATE TABLE wikipedia_changes (
  editor TEXT,
  time TIMESTAMP WITH TIME ZONE,

  wiki TEXT,
  title TEXT,

  comment TEXT,
  minor BOOLEAN,
  type TEXT,

  old_length INT,
  new_length INT
);

SELECT create_distributed_table('wikipedia_editors', 'editor');
SELECT create_distributed_table('wikipedia_changes', 'editor');
```

These tables can be populated by the Wikipedia API, and we can distribute them in Citus by the `editor` column. Notice that this is a text column. Citus' hash distribution uses PostgreSQL hashing which supports a number of data types.

A co-located JOIN between editors and changes allows aggregates not only by editor, but by properties of an editor. For instance we can count the difference between the number of newly created pages by bot vs human. The grouping and counting is performed on worker nodes in parallel and the final results are merged on the coordinator node.

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

9.1.4 Events and Reference Tables

We've already seen how every row in a distributed table is stored on a shard. However for small tables there is a trick to achieve a kind of universal colocation. We can choose to place all its rows into a single shard but replicate that shard to every worker node. It introduces storage and update costs of course, but this can be more than counterbalanced by the performance gains of read queries.

We call tables replicated to all nodes *reference tables.* They usually provide metadata about items in a larger table and are reminiscent of what data warehousing calls dimension tables.

CHAPTER 10

Creating Distributed Tables (DDL)

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

```
export PATH=/usr/lib/postgresql/9.6/:$PATH
```

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

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

10.1 Creating And Distributing Tables

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

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

Next, you can use the create_distributed_table() function to specify the table distribution column and create the worker shards.

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

This function informs Citus that the github_events table should be distributed on the repo_id column (by hashing the column value). The function also creates shards on the worker nodes using the citus.shard_count and citus.shard_replication_factor configuration values.

This example would create a total of citus.shard_count number of shards where each shard owns a portion of a hash token space and gets replicated based on the default citus.shard_replication_factor configuration value. The shard replicas created on the worker have the same table schema, index, and constraint definitions as the table on the master. Once the replicas are created, this function saves all distributed metadata on the master.

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

You are now ready to insert data into the distributed table and run queries on it. You can also learn more about the UDF used in this section in the *User Defined Functions Reference* of our documentation.

10.2 Dropping Tables

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

```
DROP TABLE github_events;
```

CHAPTER 11

Ingesting, Modifying Data (DML)

The following code snippets use the distributed tables example dataset, see *Creating Distributed Tables (DDL)*.

11.1 Inserting Data

11.1.1 Single row inserts

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

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

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

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

11.1.2 Bulk loading

Sometimes, you may want to bulk load several rows together into your distributed tables. To bulk load data from a file, you can directly use PostgreSQL's \COPY command.

For example:

```
\COPY github_events FROM 'github_events-2015-01-01-0.csv' WITH (format CSV)
```

Note: There is no notion of snapshot isolation across shards, which means that a multi-shard SELECT that runs concurrently with a COPY might see it committed on some shards, but not on others. If the user is storing events data, he may occasionally observe small gaps in recent data. It is up to applications to deal with this if it is a problem (e.g. exclude the most recent data from queries, or use some lock).

If COPY fails to open a connection for a shard placement then it behaves in the same way as INSERT, namely to mark the placement(s) as inactive unless there are no more active placements. If any other failure occurs after connecting, the transaction is rolled back and thus no metadata changes are made.

11.2 Single-Shard Updates and Deletion

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

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

Currently, Citus requires that standard UPDATE or DELETE statements involve 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. To update or delete across shards see the section below.

11.3 Cross-Shard Updates and Deletion

The most flexible way to modify or delete rows throughout a Citus cluster is the master_modify_multiple_shards command. It takes a regular SQL statement as argument and runs it on all workers:

```
SELECT master_modify_multiple_shards(
  'DELETE FROM github_events WHERE repo_id IN (24509048, 24509049)');
```

This uses a two-phase commit to remove or update data safely everywhere. Unlike the standard UPDATE statement, Citus allows it to operate on more than one shard. To learn more about the function, its arguments and its usage, please visit the *User Defined Functions Reference* section of our documentation.

11.4 Maximizing Write Performance

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

CHAPTER 12

Querying Distributed Tables (SQL)

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

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

12.1 Aggregate Functions

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

12.1.1 Count (Distinct) Aggregates

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

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

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

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

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

(2) Create the hll extension on all the PostgreSQL instances

```
CREATE EXTENSION hll;
```

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

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

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

12.1.2 HyperLogLog Column

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

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

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

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

12.2 Limit Pushdown

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

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

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

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

12.3 Joins

Citus supports equi-JOINs between any number of tables irrespective of their size and distribution method. The query planner chooses the optimal join method and join order based on the statistics gathered from the distributed tables. It evaluates several possible join orders and creates a join plan which requires minimum data to be transferred across network.

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

12.3.1 Broadcast joins

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

12.3.2 Colocated joins

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

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

12.3.3 Repartition joins

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

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

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

12.4 Query Performance

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

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

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

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

CHAPTER 13

PostgreSQL extensions

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

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

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

Migrating an existing relational store to Citus sometimes requires adjusting the schema and queries for optimal performance. Citus extends PostgreSQL with distributed functionality, but it is not a drop-in replacement that scales out all workloads. A performant Citus cluster involves thinking about the data model, tooling, and choice of SQL features used.

Migration tactics differ between the two main Citus use cases of multi-tenant applications and real-time analytics. The former requires fewer data model changes so we'll begin there.

CHAPTER 14

Multi-tenant Data Model

Citus is well suited to hosting B2B multi-tenant application data. In this model application tenants share a Citus cluster and a schema. Each tenant's table data is stored in a shard determined by a configurable tenant id column. Citus pushes queries down to run directly on the relevant tenant shard in the cluster, spreading out the computation. Once queries are routed this way they can be executed without concern for the rest of the cluster. These queries can use the full features of SQL, including joins and transactions, without running into the inherent limitations of a distributed system.

This section will explore how to model for the multi-tenant scenario, including necessary adjustments to the schema and queries.

14.1 Schema Migration

Transitioning from a standalone database instance to a sharded multi-tenant system requires identifying and modifying three types of tables which we may term *per-tenant*, *reference*, and *global*. The distinction hinges on whether the tables have (or reference) a column serving as tenant id. The concept of tenant id depends on the application and who exactly are considered its tenants.

Consider an example multi-tenant application similar to Etsy or Shopify where each tenant is a store. Here's a portion of a simplified schema:

In our example each store is a natural tenant. This is because storefronts benefit from dedicated processing power for their customer data, and stores do not need to access each other's sales or inventory. The tenant id is in this case the store id. We want to distribute data in the cluster in such a way that rows from the above tables in our schema reside on the same node whenever the rows share a store id.

The first step is preparing the tables for distribution. Citus requires that primary keys contain the distribution column, so we must modify the primary keys of these tables and make them compound including a store id. Making primary keys compound will require modifying the corresponding foreign keys as well.

In our example the stores and products tables are already in perfect shape. The orders table needs slight modification: updating the primary and foreign keys to include store_id. The line_items table needs the biggest change. Being normalized, it lacks a store id. We must add that column, and include it in the primary key constraint.

Here are SQL commands to accomplish these changes:

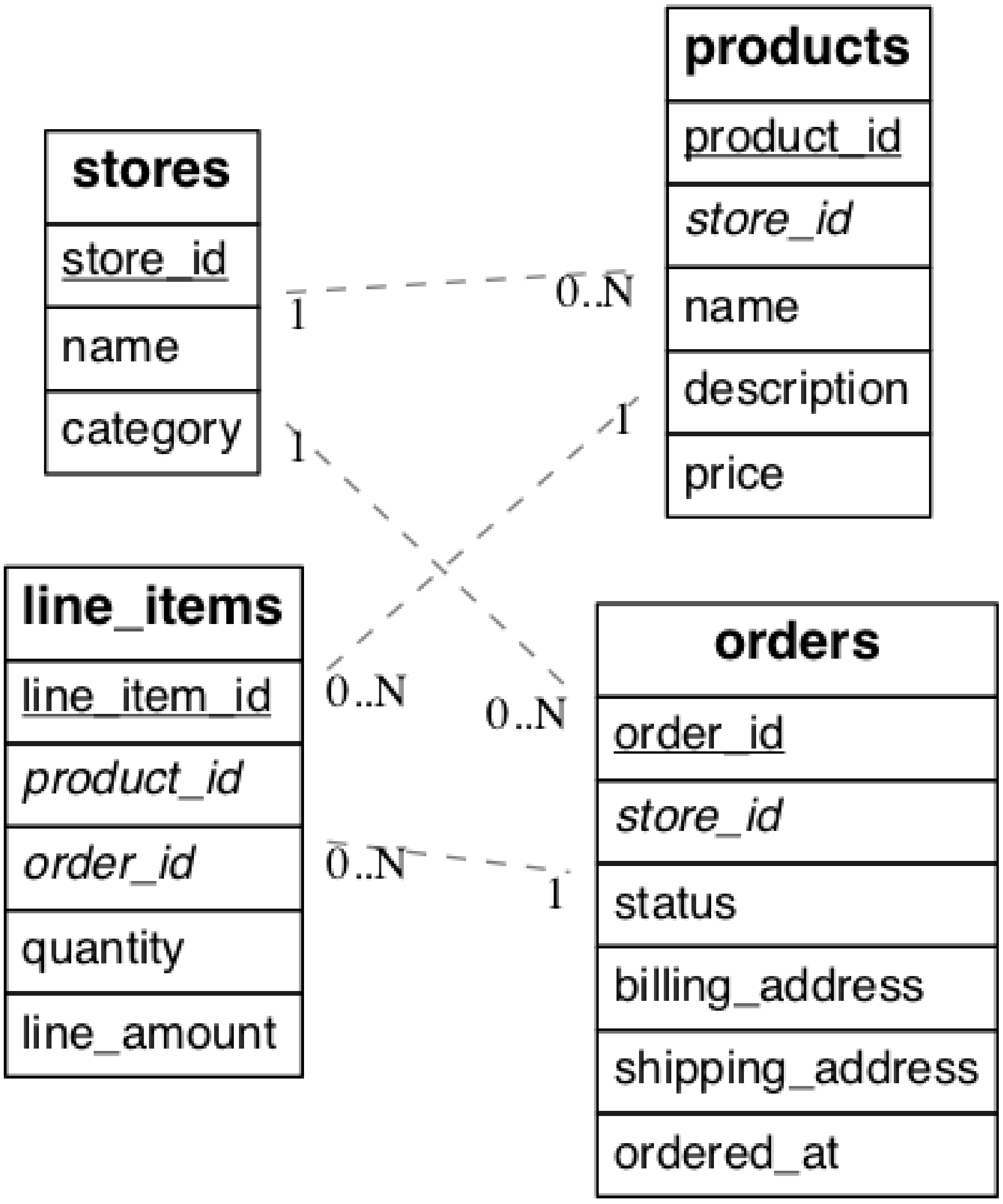


Fig. 1: (Underlined items are primary keys, italicized items are foreign keys.)

```
BEGIN;

-- denormalize line_items by including store_id

ALTER TABLE line_items ADD COLUMN store_id uuid;

-- drop simple primary keys (cascades to foreign keys)

ALTER TABLE products   DROP CONSTRAINT products_pkey CASCADE;
ALTER TABLE orders     DROP CONSTRAINT orders_pkey CASCADE;
ALTER TABLE line_items DROP CONSTRAINT line_items_pkey CASCADE;

-- recreate primary keys to include would-be distribution column

ALTER TABLE products   ADD PRIMARY KEY (store_id, product_id);
ALTER TABLE orders     ADD PRIMARY KEY (store_id, order_id);
ALTER TABLE line_items ADD PRIMARY KEY (store_id, line_item_id);

-- recreate foreign keys to include would-be distribution column

ALTER TABLE line_items ADD CONSTRAINT line_items_store_fkey
  FOREIGN KEY (store_id) REFERENCES stores (store_id);
ALTER TABLE line_items ADD CONSTRAINT line_items_product_fkey
  FOREIGN KEY (store_id, product_id) REFERENCES products (store_id, product_id);
ALTER TABLE line_items ADD CONSTRAINT line_items_order_fkey
  FOREIGN KEY (store_id, order_id) REFERENCES orders (store_id, order_id);

COMMIT;
```

When the job is complete our schema will look like this:

We call the tables considered so far *per-tenant* because querying them for our use case requires information for only one tenant per query. Their rows are distributed across the cluster according to the hashed values of their tenant ids.

There are other types of tables to consider during a transition to Citus. Some are system-wide tables such as information about site administrators. We call them *global* tables and they do not participate in join queries with the per-tenant tables and may remain on the Citus coordinator node unmodified.

Another kind of table are those which join with per-tenant tables but which aren't naturally specific to any one tenant. We call them *reference* tables. Two examples are shipping regions and product categories. We advise that you add a tenant id to these tables and duplicate the original rows, once for each tenant. This ensures that reference data is co-located with per-tenant data and quickly accessible to queries.

14.2 Backfilling Tenant ID

Once the schema is updated and the per-tenant and reference tables are distributed across the cluster it's time to copy data from the original database into Citus. Most per-tenant tables can be copied directly from source tables. However line_items was denormalized with the addition of the store_id column. We have to "backfill" the correct values into this column.

We join orders and line_items to output the data we need including the backfilled store_id column. The results can go into a file for later import into Citus.

```
-- This query gets line item information along with matching store_id values.
-- You can save the result to a file for later import into Citus.

```

(continues on next page)

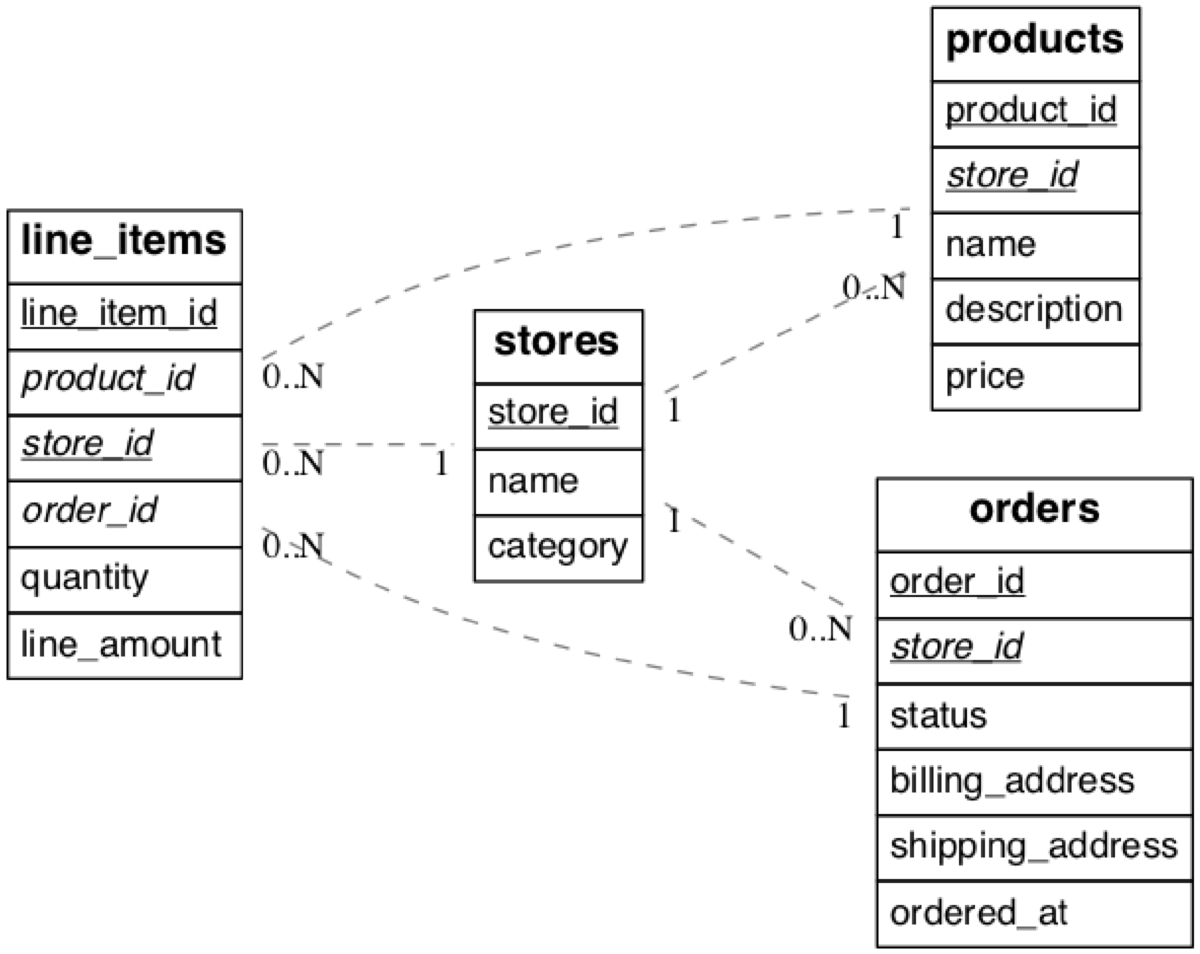


Fig. 2: (Underlined items are primary keys, italicized items are foreign keys.)

(continued from previous page)

```
SELECT orders.store_id AS store_id, line_items.*
  FROM line_items, orders
 WHERE line_items.order_id = orders.order_id
```

To learn how to ingest datasets such as the one generated above into a Citus cluster, see *Ingesting, Modifying Data (DML)*.

14.3 Query Migration

To execute queries efficiently for a specific tenant Citus needs to route them to the appropriate node and run them there. Thus every query must identify which tenant it involves. For simple select, update, and delete queries this means that the *where* clause must filter by tenant id.

Suppose we want to get the details for an order. It used to suffice to filter by order_id. However once orders are distributed by store_id we must include that in the where filter as well.

```
-- before
SELECT * FROM orders WHERE order_id = 123;

-- after
SELECT * FROM orders WHERE order_id = 123 AND store_id = 42;
```

Likewise insert statements must always include a value for the tenant id column. Citus inspects that value for routing the insert command.

When joining tables make sure to filter by tenant id. For instance here is how to inspect how many awesome wool pants a given store has sold:

```
-- One way is to include store_id in the join and also
-- filter by it in one of the queries

SELECT sum(l.quantity)
  FROM line_items l
 INNER JOIN products p
    ON l.product_id = p.product_id
   AND l.store_id = p.store_id
 WHERE p.name='Awesome Wool Pants'
   AND l.store_id='8c69aa0d-3f13-4440-86ca-443566c1fc75'

-- Equivalently you omit store_id from the join condition
-- but filter both tables by it. This may be useful if
-- building the query in an ORM

SELECT sum(l.quantity)
  FROM line_items l
 INNER JOIN products p ON l.product_id = p.product_id
 WHERE p.name='Awesome Wool Pants'
   AND l.store_id='8c69aa0d-3f13-4440-86ca-443566c1fc75'
   AND p.store_id='8c69aa0d-3f13-4440-86ca-443566c1fc75'
```

CHAPTER 15

Real-Time Analytics Data Model

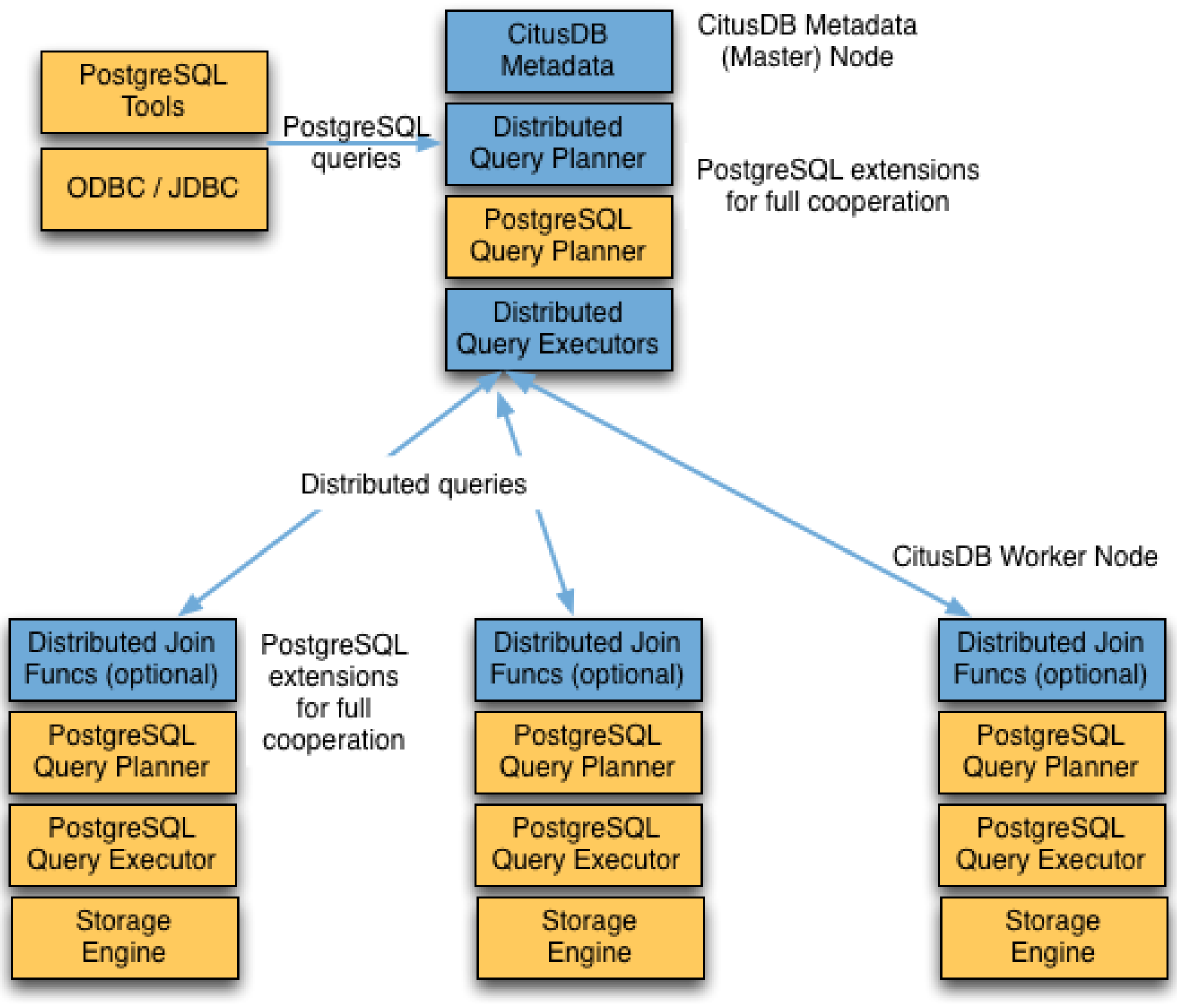
In this model multiple worker nodes calculate aggregate data in parallel for applications such as analytic dashboards. This scenario requires greater interaction between Citus nodes than the multi-tenant case and the transition from a standalone database varies more per application.

In general you can distribute the tables from an existing schema by following the advice in *Query Performance Tuning*. This will provide a baseline from which you can measure and interatively improve performance. For more migration guidance please contact us.

CHAPTER 16

Citus Query Processing

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


PostgreSQL Tools
ODBC / JDBC
PostgreSQL queries
CitusDB Metadata
Distributed Query Planner
PostgreSQL Query Planner
Distributed Query Executors
CitusDB Metadata (Master) Node
PostgreSQL extensions for full cooperation
Distributed queries
CitusDB Worker Node
Distributed Join Funcs (optional)
PostgreSQL Query Planner
PostgreSQL Query Executor
Storage Engine
PostgreSQL extensions for full cooperation

Citus's query processing pipeline involves the two components:

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

We discuss them in greater detail in the subsequent sections.

16.1 Distributed Query Planner

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

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

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

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

16.2 Distributed Query Executor

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

Citus has two executor types - real time and task tracker. The former is useful for handling simple key-value lookups and INSERT, UPDATE, and DELETE queries, while the task tracker is better suited for larger SELECT queries.

16.2.1 Real-time Executor

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

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

Furthermore, when the real time executor detects simple INSERT, UPDATE or DELETE queries it assigns the incoming query to the worker which has the target shard. The query is then handled by the worker PostgreSQL server and the results are returned back to the user. In case a modification fails on a shard replica, the executor marks the corresponding shard replica as invalid in order to maintain data consistency.

16.2.2 Task Tracker Executor

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

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

16.3 PostgreSQL planner and executor

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

CHAPTER 17

Scaling Out Data Ingestion

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

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

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

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

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

-- Enable timing to see reponse times
\timing on

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

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

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

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

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

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

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

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

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

17.3 Bulk Copy (100-200k/s)

Hash distributed tables support COPY from the Citus master for bulk ingestion, which can achieve much higher ingestion rates than regular INSERT statements.

COPY can be used to load data directly from an application using COPY .. FROM STDIN, or from a file on the server or program executed on the server.

```
COPY counters FROM STDIN WITH (FORMAT CSV);
```

In psql, the \COPY command can be used to load data from the local machine. The \COPY command actually sends a COPY .. FROM STDIN command to the server before sending the local data, as would an application that loads data directly.

```
psql -c "\COPY counters FROM 'counters-20160304.csv' (FORMAT CSV)"
```

A very powerful feature of COPY for hash distributed tables is that it asynchronously copies data to the workers over many parallel connections, one for each shard placement. This means that data can be ingested using multiple workers and multiple cores in parallel. Especially when there are expensive indexes such as a GIN, this can lead to major performance boosts over ingesting into a regular PostgreSQL table.

Note: To avoid opening too many connections to the workers. We recommend only running only one COPY command on a hash distributed table at a time. In practice, running more than two at a time rarely results in performance benefits. An exception is when all the data in the ingested file has a specific partition key value, which goes into a single shard. COPY will only open connections to shards when necessary.

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

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

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

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

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

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

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

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

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

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

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

CHAPTER 18

Query Performance Tuning

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

18.1 Table Distribution and Shards

The first step while creating a distributed table is choosing the right distribution column. This helps Citus push down several operations directly to the worker shards and prune away unrelated shards which lead to significant query speedups.

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

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

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

18.2 PostgreSQL tuning

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

Tuning the parameters is a matter of experimentation and often takes several attempts to achieve acceptable performance. Thus it's best to load only a small portion of your data when tuning to make each iteration go faster.

To begin the tuning process create a Citus cluster and load data in it. From the master node, run the EXPLAIN command on representative queries to inspect performance. Citus extends the EXPLAIN command to provide information about distributed query execution. The EXPLAIN output shows how each worker processes the query and also a little about how the master node combines their results.

Here is an example of explaining the plan for a particular query in *one of* our example tutorials.

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

Distributed Query into pg_merge_job_0005
  Executor: Real-Time
  Task Count: 16
  Tasks Shown: One of 16
  ->  Task
    Node: host=localhost port=9701 dbname=postgres
    ->  Limit  (cost=0.15..6.87 rows=10 width=32)
      ->  Nested Loop  (cost=0.15..131.12 rows=195 width=32)
        ->  Seq Scan on wikipedia_changes_102024 c  (cost=0.00..13.90 rows=390␣
↪width=64)
        ->  Index Scan using wikipedia_editors_editor_key_102008 on wikipedia_editors_
↪102008 e  (cost=0.15..0.29 rows=1 width=32)
          Index Cond: (editor = c.editor)
          Filter: (bot IS TRUE)
Master Query
  ->  Limit  (cost=0.00..0.00 rows=0 width=0)
    ->  Seq Scan on pg_merge_job_0005  (cost=0.00..0.00 rows=0 width=0)
(15 rows)
```

This tells you several things. To begin with there are sixteen shards, and we're using the real-time Citus executor setting:

```
Distributed Query into pg_merge_job_0005
  Executor: Real-Time
  Task Count: 16
```

Next it picks one of the workers to and shows you more about how the query behaves there. It indicates the host, port, and database so you can connect to the worker directly if desired:

```
Tasks Shown: One of 16
->  Task
  Node: host=localhost port=9701 dbname=postgres
```

Distributed EXPLAIN next shows the results of running a normal PostgreSQL EXPLAIN on that worker for the fragment query:

```
->  Limit  (cost=0.15..6.87 rows=10 width=32)
  ->  Nested Loop  (cost=0.15..131.12 rows=195 width=32)
    ->  Seq Scan on wikipedia_changes_102024 c  (cost=0.00..13.90 rows=390 width=64)
    ->  Index Scan using wikipedia_editors_editor_key_102008 on wikipedia_editors_
↪102008 e  (cost=0.15..0.29 rows=1 width=32)
      Index Cond: (editor = c.editor)
      Filter: (bot IS TRUE)
```

You can now connect to the worker at 'localhost', port '9701' and tune query performance for the shard wikipedia_changes_102024 using standard PostgreSQL techniques. As you make changes run EXPLAIN again from the master or right on the worker.

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

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

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

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

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

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

Once you have tuned a worker to your satisfaction you will have to manually apply those changes to the other workers as well. To verify that they are all behaving properly, set this configuration variable on the master:

```
SET citus.explain_all_tasks = 1;
```

This will cause EXPLAIN to show the the query plan for all tasks, not just one.

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

Distributed Query into pg_merge_job_0003
  Executor: Real-Time
  Task Count: 16
  Tasks Shown: All
  ->  Task
    Node: host=localhost port=9701 dbname=postgres
    ->  Limit  (cost=0.15..6.87 rows=10 width=32)
      ->  Nested Loop  (cost=0.15..131.12 rows=195 width=32)
        ->  Seq Scan on wikipedia_changes_102024 c  (cost=0.00..13.90 rows=390␣
↪width=64)
        ->  Index Scan using wikipedia_editors_editor_key_102008 on wikipedia_editors_
↪102008 e  (cost=0.15..0.29 rows=1 width=32)
```

(continues on next page)

(continued from previous page)

```
          Index Cond: (editor = c.editor)
          Filter: (bot IS TRUE)
  ->  Task
    Node: host=localhost port=9702 dbname=postgres
    ->  Limit  (cost=0.15..6.87 rows=10 width=32)
      ->  Nested Loop  (cost=0.15..131.12 rows=195 width=32)
        ->  Seq Scan on wikipedia_changes_102025 c  (cost=0.00..13.90 rows=390␣
↪width=64)
        ->  Index Scan using wikipedia_editors_editor_key_102009 on wikipedia_editors_
↪102009 e  (cost=0.15..0.29 rows=1 width=32)
          Index Cond: (editor = c.editor)
          Filter: (bot IS TRUE)
  ->  Task
    Node: host=localhost port=9701 dbname=postgres
    ->  Limit  (cost=1.13..2.36 rows=10 width=74)
      ->  Hash Join  (cost=1.13..8.01 rows=56 width=74)
        Hash Cond: (c.editor = e.editor)
        ->  Seq Scan on wikipedia_changes_102036 c  (cost=0.00..5.69 rows=169␣
↪width=83)
        ->  Hash  (cost=1.09..1.09 rows=3 width=12)
          ->  Seq Scan on wikipedia_editors_102020 e  (cost=0.00..1.09 rows=3␣
↪width=12)
            Filter: (bot IS TRUE)
  --
  -- ... repeats for all 16 tasks
  --     alternating between workers one and two
  --     (running in this case locally on ports 9701, 9702)
  --
Master Query
  ->  Limit  (cost=0.00..0.00 rows=0 width=0)
    ->  Seq Scan on pg_merge_job_0003  (cost=0.00..0.00 rows=0 width=0)
```

Differences in worker execution can be caused by tuning configuration differences, uneven data distribution across shards, or hardware differences between the machines. To get more information about the time it takes the query to run on each shard you can use EXPLAIN ANALYZE.

Note: Note that when citus.explain_all_tasks is enabled, EXPLAIN plans are retrieved sequentially, which may take a long time for EXPLAIN ANALYZE. Also a remote EXPLAIN may error out when explaining a broadcast join while the shards for the small table have not yet been fetched. An error message is displayed advising to run the query first.

18.3 Scaling Out Performance

As mentioned, once you have achieved the desired performance for a single shard you can set similar configuration parameters on all your workers. As Citus runs all the fragment queries in parallel across the worker nodes, users can scale out the performance of their queries to be the cumulative of the computing power of all of the CPU cores in the cluster assuming that the data fits in memory.

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

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

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

18.4 Distributed Query Performance Tuning

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

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

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

18.4.1 General

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

Citus has two executor types for running SELECT queries. The desired executor can be selected by setting the citus.task_executor_type configuration parameter. If your use case mainly requires simple key-value lookups or requires sub-second responses to aggregations and joins, you can choose the real-time executor. On the other hand if there are long running queries which require repartitioning and shuffling of data across the workers, then you can switch to the the task tracker executor.

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

18.4.2 Advanced

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

Task Assignment Policy

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

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

Intermediate Data Transfer Format

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

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

Real Time Executor

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

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

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

Task Tracker Executor

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

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

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

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

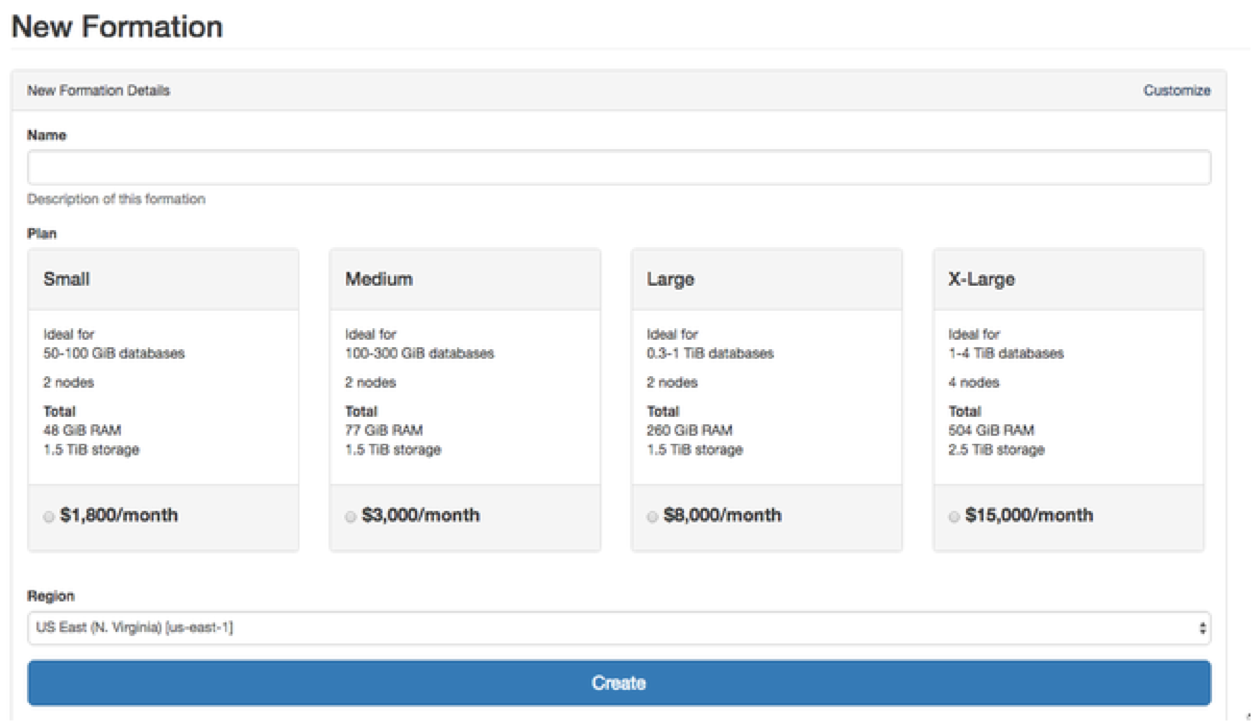
CHAPTER 19

Citus Overview

Citus Cloud is a fully managed hosted version of Citus Enterprise edition on top of AWS. Citus Cloud comes with the benefit of Citus allowing you to easily scale out your memory and processing power, without having to worry about keeping it up and running.

19.1 Provisioning

Once you've created your account at https://console.citusdata.com you can provision your Citus cluster. When you login you'll be at the home of the dashboard, and from here you can click New Formation to begin your formation creation.


New Formation

New Formation Details

Customize

Name

Description of this formation

Plan

Small	Medium	Large	X-Large
Ideal for 50-100 GiB databases	Ideal for 100-300 GiB databases	Ideal for 0.3-1 TiB databases	Ideal for 1-4 TiB databases
2 nodes	2 nodes	2 nodes	4 nodes
Total 48 GiB RAM 1.5 TiB storage	Total 77 GiB RAM 1.5 TiB storage	Total 260 GiB RAM 1.5 TiB storage	Total 504 GiB RAM 2.5 TiB storage
$1,800/month	$3,000/month	$8,000/month	$15,000/month

Region

US East (N. Virginia) [us-east-1]

19.1.1 Configuring Your Plan

Citus Cloud plans vary based on the size of your primary node, size of your distributed nodes, number of distributed nodes and whether you have high availability or not. From within the Citus console you can configure your plan or you can preview what it might look like within the pricing calculator.

The key items you'll care about for each node:

- Storage - All nodes come with 512 GB of storage
- Memory - The memory on each node varies based on the size of node you select
- Cores - The cores on each node varies based on the size of node you select

High Availability

The high availability option on a cluster automatically provisions instance stand-bys. These stand-bys receive streaming updates directly from each of the leader nodes. We continuously monitor the leader nodes to ensure they're available and healthy. In the event of a failure we automatically switch to the stand-bys.

Note that your data is replicated to S3 with and without enabling high availability. This allows disaster recovery and reduces the risk of data loss. Although the data is safe either way, we suggest enabling high availability if you cannot tolerate up to one hour of downtime in the rare occurrence of an instance failure. We do not offer a SLA on uptime.

CHAPTER 20

Features

20.1 Replication model

Citus Cloud runs with a different replication from that outlined in the Citus docs. We leverage PostgreSQL streaming replication rather than Citus replication. This provides superior guarantees about data preservation. Enable HA at provisioning to achieve the equivalent of Citus with replication_factor set to two (the most popular setting for Citus users).

20.2 Continuous protection

Citus Cloud continuously protects the cluster data against hardware failure. To do this we perform backups every twenty-four hours, then stream the write-ahead log (WAL) from PostgreSQL to S3 every 16 MB or 60 seconds, whichever is less. Even without high availability enabled you won't lose any data. In the event of a complete infrastructure failure we'll restore your back-up and replay the WAL to the exact moment before your system crashed.

20.3 High Availability

In addition to continuous protection which is explained above, high availability is available if your application requires less exposure to downtime. We provision stand-bys if you select high availability at provisioning time. This can be for your primary node, or for your distributed nodes.

20.4 Security

20.4.1 Encryption

All data within Citus Cloud is encrypted at rest, including data on the instance as well as all backups for disaster recovery. We also require that you connect to your database with TLS.

20.4.2 Two-Factor Authentication

We support two factor authentication for all Citus accounts. You can enable it from within your Citus Cloud account. We support Google Authenticator and Authy as two primary apps for setting up your two factor authentication.

CHAPTER 21

Logging

21.1 What Is Logged

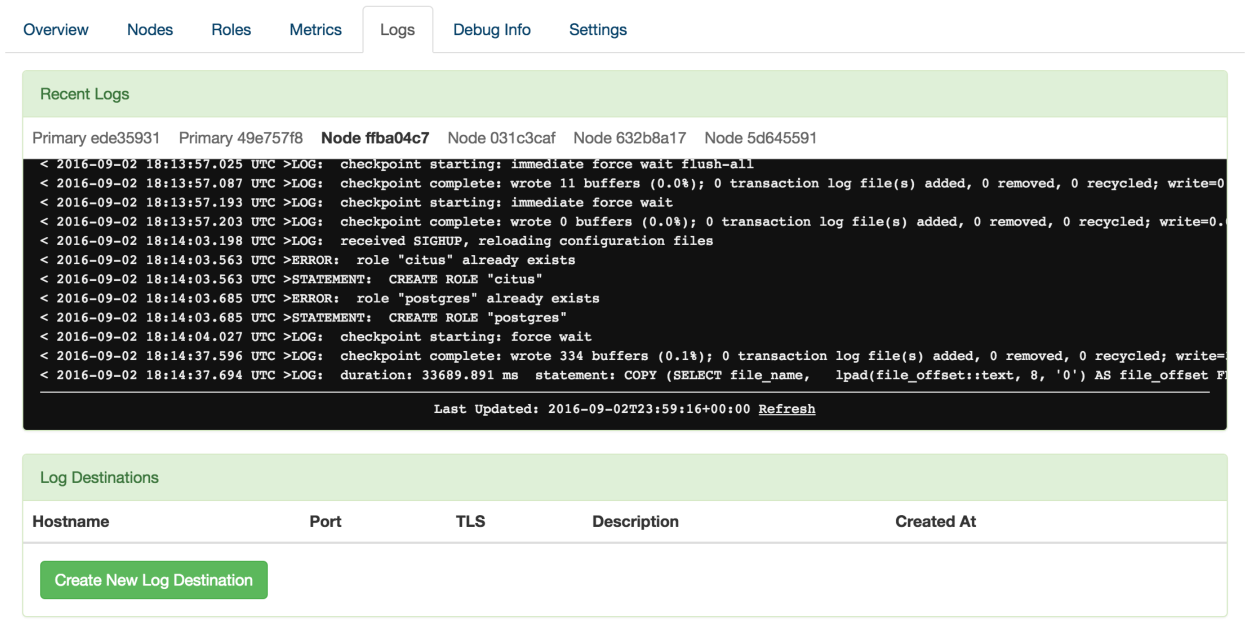
By default, Citus Cloud logs all errors and other useful information that happen on any of the Citus instances and makes it available to you.

The logs will contain the following messages:

- Citus and PostgreSQL errors
- Slow queries that take longer than 30 seconds
- Checkpoint statistics
- Temporary files that are written and bigger than 64 MB
- Autovacuum that takes more than 30 seconds

21.2 Recent Logs

The Citus Cloud dashboard automatically shows you the most recent 100 log lines from each of your servers. You don't need to configure anything to access this information.



```
< 2016-09-02 18:13:57.025 UTC >LOG:  checkpoint starting: immediate force wait flush-all
< 2016-09-02 18:13:57.087 UTC >LOG:  checkpoint complete: wrote 11 buffers (0.0%); 0 transaction log file(s) added, 0 removed, 0 recycled; write=0
< 2016-09-02 18:13:57.193 UTC >LOG:  checkpoint starting: immediate force wait
< 2016-09-02 18:13:57.203 UTC >LOG:  checkpoint complete: wrote 0 buffers (0.0%); 0 transaction log file(s) added, 0 removed, 0 recycled; write=0.
< 2016-09-02 18:14:03.198 UTC >LOG:  received SIGHUP, reloading configuration files
< 2016-09-02 18:14:03.563 UTC >ERROR:  role "citus" already exists
< 2016-09-02 18:14:03.563 UTC >STATEMENT:  CREATE ROLE "citus"
< 2016-09-02 18:14:03.685 UTC >ERROR:  role "postgres" already exists
< 2016-09-02 18:14:03.685 UTC >STATEMENT:  CREATE ROLE "postgres"
< 2016-09-02 18:14:04.027 UTC >LOG:  checkpoint starting: force wait
< 2016-09-02 18:14:37.596 UTC >LOG:  checkpoint complete: wrote 334 buffers (0.1%); 0 transaction log file(s) added, 0 removed, 0 recycled; write=
< 2016-09-02 18:14:37.694 UTC >LOG:  duration: 33689.891 ms  statement: COPY (SELECT file_name,   lpad(file_offset::text, 8, '0') AS file_offset F

                         Last Updated: 2016-09-02T23:59:16+00:00 Refresh
```

Log Destinations

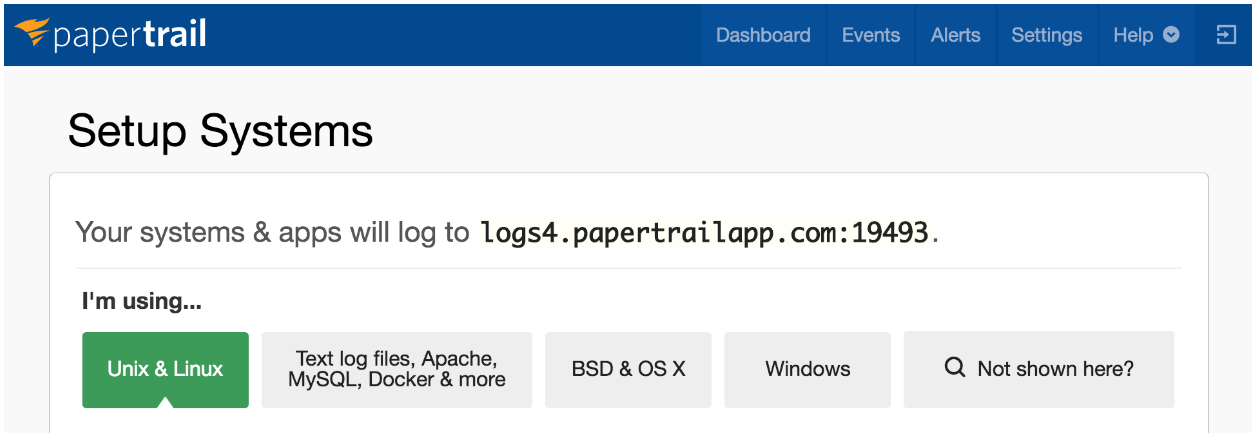
Hostname	Port	TLS	Description	Created At

21.3 External Log Destinations

For anything more than a quick look at your logs, we recommend setting up an external provider as a log destination. Through this method you'll receive all logs using the Syslog protocol, and can analyze and retain them according to your own preferences.

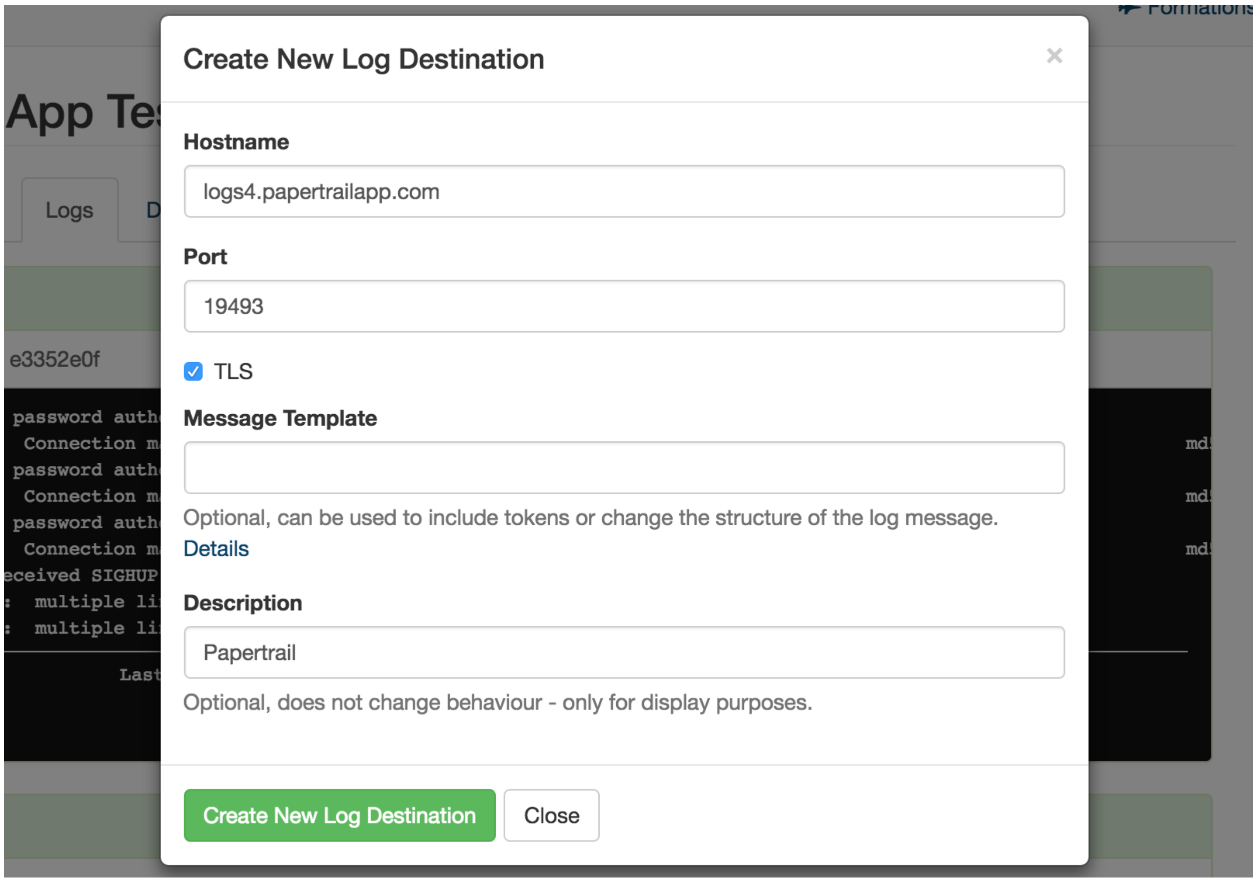
As an example, the process for setting up Papertrail, a common log provider, goes like this:

1. Determine Syslog host and port number (note that Citus only supports providers that can accept incoming Syslog data)



Your systems & apps will log to `logs4.papertrailapp.com:19493`.

2. Add a new log destination in your Citus Cloud dashboard in the "Logs" tab, like this:


Create New Log Destination

Hostname

logs4.papertrailapp.com

Port

19493

✓ TLS

Message Template

Optional, can be used to include tokens or change the structure of the log message.
Details

Description

Papertrail

Optional, does not change behaviour - only for display purposes.

For Papertrail leave Message Template empty. Other providers might require a token to be set - follow the syslog-ng instructions for the provider, if they make them available.

3. After creation, it might take up to 5 minutes for logging to be configured. You'll then see it show up in your favorite provider's dashboard.

21.4 Supported External Providers

We're currently testing with these providers:

- Papertrail
- Splunk Enterprise
- Loggly

We likely also support other providers that can receive syslog. Please reach out if you encounter any issues.

CHAPTER 22

Support and Billing

All Citus Cloud plans come with email support included. Premium support including SLA around response time and phone escalation is available on a contract basis for customers that may need a more premium level of support.

22.1 Support

Email and web based support is available on all Citus Cloud plans. You can open a support inquiry by emailing cloud-support@citusdata.com or clicking on the support icon in the lower right of your screen within Citus Cloud.

22.2 Billing and pricing

Citus Cloud bills on a per minute basis. We bill for a minimum of one hour of usage across all plans. Pricing varies based on the size and configuration of the cluster. A few factors that determine your price are:

- Size of your distributed nodes
- Number of distributed nodes
- Whether you have high availability enabled, both on the primary node and on distributed nodes
- Size of your primary node

You can see pricing of various configurations directly within our pricing calculator.

CHAPTER 23

Real Time Dashboards

Citus provides real-time queries over large datasets. One workload we commonly see at Citus involves powering real-time dashboards of event data.

For example, you could be a cloud services provider helping other businesses monitor their HTTP traffic. Every time one of your clients receives an HTTP request your service receives a log record. You want to ingest all those records and create an HTTP analytics dashboard which gives your clients insights such as the number HTTP errors their sites served. It's important that this data shows up with as little latency as possible so your clients can fix problems with their sites. It's also important for the dashboard to show graphs of historical trends.

Alternatively, maybe you're building an advertising network and want to show clients clickthrough rates on their campaigns. In this example latency is also critical, raw data volume is also high, and both historical and live data are important.

In this technical solution we'll demonstrate how to build part of the first example, but this architecture would work equally well for the second and many other use-cases.

23.1 Running It Yourself

There will be code snippets in this tutorial but they don't specify a complete system. There's a github repo with all the details in one place. If you've followed our installation instructions for running Citus on either a single or multiple machines you're ready to try it out.

23.2 Data Model

The data we're dealing with is an immutable stream of log data. We'll insert directly into Citus but it's also common for this data to first be routed through something like Kafka. Doing so has the usual advantages, and makes it easier to pre-aggregate the data once data volumes become unmanageably high.

We'll use a simple schema for ingesting HTTP event data. This schema serves as an example to demonstrate the overall architecture; a real system might use additional columns.

```
-- this is run on the master
CREATE TABLE http_request (
  site_id INT,
  ingest_time TIMESTAMPTZ DEFAULT now(),

  url TEXT,
  request_country TEXT,
  ip_address TEXT,

  status_code INT,
  response_time_msec INT
);

SELECT create_distributed_table('http_request', 'site_id');
```

When we call *create_distributed_table* we ask Citus to hash-distribute `http_request` using the `site_id` column. That means all the data for a particular site will live in the same shard.

The UDF uses the default configuration values for shard count and replication factor. We recommend *using 2-4x as many shards* as CPU cores in your cluster. Using this many shards lets you rebalance data across your cluster after adding new worker nodes.

Using a replication factor of 2 means every shard is held on multiple workers. When a worker fails the master will prevent downtime by serving queries for that worker's shards using the other replicas.

Note: In Citus Cloud you must use a replication factor of 1 (instead of the 2 used here). As Citus Cloud uses streaming replication to achieve high availability maintaining shard replicas would be redundant.

With this, the system is ready to accept data and serve queries! We've provided a data ingest script you can run to generate example data. Once you've ingested data, you can run dashboard queries such as:

```
SELECT
  date_trunc('minute', ingest_time) as minute,
  COUNT(1) AS request_count,
  SUM(CASE WHEN (status_code between 200 and 299) THEN 1 ELSE 0 END) as success_count,
  SUM(CASE WHEN (status_code between 200 and 299) THEN 0 ELSE 1 END) as error_count,
  SUM(response_time_msec) / COUNT(1) AS average_response_time_msec
FROM http_request
WHERE site_id = 1 AND date_trunc('minute', ingest_time) > now() - interval '5 minutes'
GROUP BY minute;
```

The setup described above works, but has two drawbacks:

- Your HTTP analytics dashboard must go over each row every time it needs to generate a graph. For example, if your clients are interested in trends over the past year, your queries will aggregate every row for the past year from scratch.
- Your storage costs will grow proportionally with the ingest rate and the length of the queryable history. In practice, you may want to keep raw events for a shorter period of time (one month) and look at historical graphs over a longer time period (years).

23.3 Rollups

You can overcome both drawbacks by rolling up the raw data into a pre-aggregated form. Here, we'll aggregate the raw data into a table which stores summaries of 1-minute intervals. In a production system, you would probably also want something like 1-hour and 1-day intervals, these each correspond to zoom-levels in the dashboard. When the user wants request times for the last month the dashboard can simply read and chart the values for each of the last 30 days.

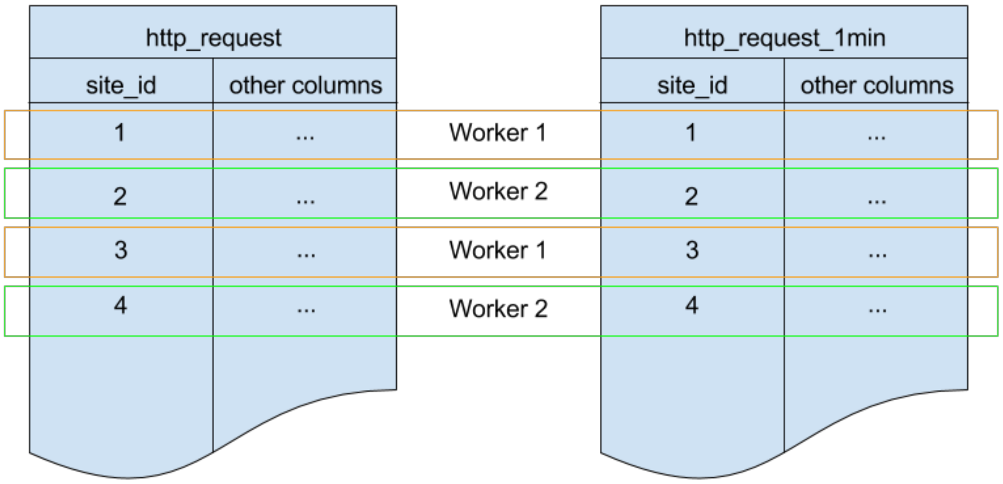
```
CREATE TABLE http_request_1min (
      site_id INT,
      ingest_time TIMESTAMPTZ, -- which minute this row represents

      error_count INT,
      success_count INT,
      request_count INT,
      average_response_time_msec INT,
      CHECK (request_count = error_count + success_count),
      CHECK (ingest_time = date_trunc('minute', ingest_time))
);

SELECT create_distributed_table('http_request_1min', 'site_id');

-- indexes aren't automatically created by Citus
-- this will create the index on all shards
CREATE INDEX http_request_1min_idx ON http_request_1min (site_id, ingest_time);
```

This looks a lot like the previous code block. Most importantly: It also shards on `site_id` and uses the same default configuration for shard count and replication factor. Because all three of those match, there's a 1-to-1 correspondence between `http_request` shards and `http_request_1min` shards, and Citus will place matching shards on the same worker. This is called colocation; it makes queries such as joins faster and our rollups possible.



In order to populate `http_request_1min` we're going to periodically run the equivalent of an INSERT INTO SELECT. We'll run a function on all the workers which runs INSERT INTO SELECT on every matching pair of shards. This is possible because the tables are colocated.

```
-- this function is created on the workers
CREATE FUNCTION rollup_1min(p_source_shard text, p_dest_shard text) RETURNS void
AS $$
BEGIN
  -- the dest shard will have a name like: http_request_1min_204566, where 204566 is␣
↪the
```

(continues on next page)

(continued from previous page)

```
  -- shard id. We lock using that id, to make sure multiple instances of this function
  -- never simultaneously write to the same shard.
  IF pg_try_advisory_xact_lock(29999, split_part(p_dest_shard, '_', 4)::int) = false␣
↪THEN
    -- N.B. make sure the int constant (29999) you use here is unique within your␣
↪system
    RETURN;
  END IF;

  EXECUTE format($insert$
    INSERT INTO %2$I (
      site_id, ingest_time, request_count,
      success_count, error_count, average_response_time_msec
    ) SELECT
      site_id,
      date_trunc('minute', ingest_time) as minute,
      COUNT(1) as request_count,
      SUM(CASE WHEN (status_code between 200 and 299) THEN 1 ELSE 0 END) as success_
↪count,
      SUM(CASE WHEN (status_code between 200 and 299) THEN 0 ELSE 1 END) as error_
↪count,
      SUM(response_time_msec) / COUNT(1) AS average_response_time_msec
    FROM %1$I
    WHERE
      date_trunc('minute', ingest_time)
        > (SELECT COALESCE(max(ingest_time), timestamp '10-10-1901') FROM %2$I)
      AND date_trunc('minute', ingest_time) < date_trunc('minute', now())
    GROUP BY site_id, minute
    ORDER BY minute ASC;
  $insert$, p_source_shard, p_dest_shard);
END;
$$ LANGUAGE 'plpgsql';
```

Inside this function you can see the dashboard query from earlier. It's been wrapped in some machinery which writes the results into `http_request_1min` and allows passing in the name of the shards to read and write from. It also takes out an advisory lock, to ensure there aren't any concurrency bugs where the same rows are written multiple times.

The machinery above which accepts the names of the shards to read and write is necessary because only the master has the metadata required to know what the shard pairs are. It has its own function to figure that out:

```
-- this function is created on the master
CREATE FUNCTION colocated_shard_placements(left_table REGCLASS, right_table REGCLASS)
RETURNS TABLE (left_shard TEXT, right_shard TEXT, nodename TEXT, nodeport BIGINT) AS $
↪$
  SELECT
    a.logicalrelid::regclass||'_'||a.shardid,
    b.logicalrelid::regclass||'_'||b.shardid,
    nodename, nodeport
  FROM pg_dist_shard a
  JOIN pg_dist_shard b USING (shardminvalue)
  JOIN pg_dist_shard_placement p ON (a.shardid = p.shardid)
  WHERE a.logicalrelid = left_table AND b.logicalrelid = right_table;
$$ LANGUAGE 'sql';
```

Using that metadata, every minute it runs a script which calls `rollup_1min` once for each pair of shards:

```
#!/usr/bin/env bash

QUERY=$(cat <<END
  SELECT * FROM colocated_shard_placements(
    'http_request'::regclass, 'http_request_1min'::regclass
  );
END
)

COMMAND="psql -h \$2 -p \$3 -c \"SELECT rollup_1min('\$0', '\$1')\""

psql -tA -F" " -c "$QUERY" | xargs -P32 -n4 sh -c "$COMMAND"
```

Note: There are many ways to make sure the function is called periodically and no answer that works well for every system. If you're able to run cron on the same machine as the master, and assuming you named the above script `run_rollups.sh`, you can do something as simple as this:

```
* * * * * /some/path/run_rollups.sh
```

The dashboard query from earlier is now a lot nicer:

```
SELECT
  request_count, success_count, error_count, average_response_time_msec
FROM http_request_1min
WHERE site_id = 1 AND date_trunc('minute', ingest_time) > date_trunc('minute', now())␣
↪- interval '5 minutes';
```

23.4 Expiring Old Data

The rollups make queries faster, but we still need to expire old data to avoid unbounded storage costs. Once you decide how long you'd like to keep data for each granularity, you could easily write a function to expire old data. In the following example, we decided to keep raw data for one day and 1-minute aggregations for one month.

```
-- another function for the master
CREATE FUNCTION expire_old_request_data() RETURNS void
AS $$
  SET LOCAL citus.all_modification_commutative TO TRUE;
  SELECT master_modify_multiple_shards(
    'DELETE FROM http_request WHERE ingest_time < now() - interval ''1 day'';');
  SELECT master_modify_multiple_shards(
    'DELETE FROM http_request_1min WHERE ingest_time < now() - interval ''1 month'';
↪');
END;
$$ LANGUAGE 'sql';
```

Note: The above function should be called every minute. You could do this by adding a crontab entry on the master node:

```
* * * * * psql -c "SELECT expire_old_request_data();"
```

That's the basic architecture! We provided an architecture that ingests HTTP events and then rolls up these events into their pre-aggregated form. This way, you can both store raw events and also power your analytical dashboards with subsecond queries.

The next sections extend upon the basic architecture and show you how to resolve questions which often appear.

23.5 Approximate Distinct Counts

A common question in http analytics deals with *approximate distinct counts*: How many unique visitors visited your site over the last month? Answering this question exactly requires storing the list of all previously-seen visitors in the rollup tables, a prohibitively large amount of data. A datatype called hyperloglog, or HLL, can answer the query approximately; it takes a surprisingly small amount of space to tell you approximately how many unique elements are in a set you pass it. Its accuracy can be adjusted. We'll use ones which, using only 1280 bytes, will be able to count up to tens of billions of unique visitors with at most 2.2% error.

An equivalent problem appears if you want to run a global query, such as the number of unique ip addresses which visited any of your client's sites over the last month. Without HLLs this query involves shipping lists of ip addresses from the workers to the master for it to deduplicate. That's both a lot of network traffic and a lot of computation. By using HLLs you can greatly improve query speed.

First you must install the hll extension; the github repo has instructions. Next, you have to enable it:

```
-- this part must be run on all nodes
CREATE EXTENSION hll;

-- this part runs on the master
ALTER TABLE http_request_1min ADD COLUMN distinct_ip_addresses hll;
```

When doing our rollups, we can now aggregate sessions into an hll column with queries like this:

```
SELECT
  site_id, date_trunc('minute', ingest_time) as minute,
  hll_add_agg(hll_hash_text(ip_address)) AS distinct_ip_addresses
FROM http_request
WHERE date_trunc('minute', ingest_time) = date_trunc('minute', now())
GROUP BY site_id, minute;
```

Now dashboard queries are a little more complicated, you have to read out the distinct number of ip addresses by calling the `hll_cardinality` function:

```
SELECT
  request_count, success_count, error_count, average_response_time_msec,
  hll_cardinality(distinct_ip_addresses) AS distinct_ip_address_count
FROM http_request_1min
WHERE site_id = 1 AND ingest_time = date_trunc('minute', now());
```

HLLs aren't just faster, they let you do things you couldn't previously. Say we did our rollups, but instead of using HLLs we saved the exact unique counts. This works fine, but you can't answer queries such as "how many distinct sessions were there during this one-week period in the past we've thrown away the raw data for?".

With HLLs, this is easy. You'll first need to inform Citus about the `hll_union_agg` aggregate function and its semantics. You do this by running the following:

```
-- this should be run on the workers and master
CREATE AGGREGATE sum (hll)
(
```

(continues on next page)

(continued from previous page)

```
  sfunc = hll_union_trans,
  stype = internal,
  finalfunc = hll_pack
);
```

Now, when you call SUM over a collection of HLLs, PostgreSQL will return the HLL for us. You can then compute distinct ip counts over a time period with the following query:

```
SELECT
  hll_cardinality(SUM(distinct_ip_addresses))
FROM http_request_1min
WHERE ingest_time BETWEEN timestamp '06-01-2016' AND '06-28-2016';
```

You can find more information on HLLs in the project's GitHub repository.

23.6 Unstructured Data with JSONB

Citus works well with Postgres' built-in support for unstructured data types. To demonstrate this, let's keep track of the number of visitors which came from each country. Using a semi-structure data type saves you from needing to add a column for every individual country and ending up with rows that have hundreds of sparsely filled columns. We have a blog post explaining which format to use for your semi-structured data. The post recommends JSONB, here we'll demonstrate how to incorporate JSONB columns into your data model.

First, add the new column to our rollup table:

```
ALTER TABLE http_request_1min ADD COLUMN country_counters JSONB;
```

Next, include it in the rollups by adding a clause like this to the rollup function:

```
SELECT
  site_id, minute,
  jsonb_object_agg(request_country, country_count)
FROM (
  SELECT
    site_id, date_trunc('minute', ingest_time) AS minute,
    request_country,
    count(1) AS country_count
  FROM http_request
  GROUP BY site_id, minute, request_country
) AS subquery
GROUP BY site_id, minute;
```

Now, if you want to get the number of requests which came from america in your dashboard, your can modify the dashboard query to look like this:

```
SELECT
  request_count, success_count, error_count, average_response_time_msec,
  country_counters->'USA' AS american_visitors
FROM http_request_1min
WHERE site_id = 1 AND ingest_time = date_trunc('minute', now());
```

23.7 Resources

This article shows a complete system to give you an idea of what building a non-trivial application with Citus looks like. Again, there's a github repo with all the scripts mentioned here.

CHAPTER 24

Cluster Management

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

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

24.1 Scaling out your cluster

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

24.1.1 Adding a worker

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

To add a new node to the cluster, you first need to add the DNS name or IP address of that node and port (on which PostgreSQL is running) in the pg_dist_node catalog table. You can do so using the *master_add_node* UDF. Example:

```
SELECT * from master_add_node('node-name', 5432);
```

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

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

24.1.2 Adding a master

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

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

24.2 Dealing With Node Failures

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

24.2.1 Worker Node Failures

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

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

On seeing such warnings, the first step would be to log into the failed node and inspect the cause of the failure.

In the meanwhile, Citus will automatically re-route the work to the healthy workers. Also, if Citus is not able to connect to a worker, it will assign that task to another node having a copy of that shard. If the failure occurs mid-query, Citus does not re-run the whole query but assigns only the failed query fragments leading to faster responses in face of failures.

Once the node is brought back up, Citus will automatically continue connecting to it and using the data.

If the node suffers a permanent failure then you may wish to retain the same level of replication so that your application can tolerate more failures. To make this simpler, Citus enterprise provides a replicate_table_shards UDF which can be called after. This function copies the shards of a table across the healthy nodes so they all reach the configured replication factor.

To remove a permanently failed node from the list of workers, you should first mark all shard placements on that node as invalid (if they are not already so) using the following query:

```
UPDATE pg_dist_shard_placement set shardstate = 3 where nodename = 'bad-node-name'␣
↪and nodeport = 5432;
```

Then, you can remove the node using master_remove_node, as shown below:

```
select master_remove_node('bad-node-name', 5432);
```

If you want to add a new node to the cluster to replace the failed node, you can follow the instructions described in the *Adding a worker* section. Finally, you can use the function provided in Citus Enterprise to replicate the invalid shards and maintain the replication factor.

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

24.2.2 Master Node Failures

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

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

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

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

CHAPTER 25

Upgrading Citus

25.1 Upgrading Citus Versions

Upgrading the Citus version requires first obtaining the new Citus extension and then installing it in each of your database instances. The first step varies by operating system.

25.1.1 Step 1. Update Citus Package

OS X

```
brew update
brew upgrade citus
```

Ubuntu or Debian

```
sudo apt-get update
sudo apt-get upgrade postgresql-9.5-citus
```

Fedora, CentOS, or Red Hat

```
sudo yum update citus_95
```

25.1.2 Step 2. Apply Update in DB

Restart PostgreSQL:

```
pg_ctl -D /data/base -l logfile restart
```

```
# after restarting postgres
psql -c "ALTER EXTENSION citus UPDATE;"
```

(continues on next page)

(continued from previous page)

```
psql -c "\dx"
# you should see a newer Citus 6.0 version in the list
```

That's all it takes! No further steps are necessary after updating the extension on all database instances in your cluster.

25.2 Upgrading PostgreSQL version from 9.5 to 9.6

Citus v6.0 is compatible with PostgreSQL 9.5.x and 9.6.x. If you are running Citus on PostgreSQL versions 9.5 and wish to upgrade to version 9.6, Please contact us for upgrade steps.

Note: PostgreSQL 9.6 requires using Citus 6.0.

CHAPTER 26

Citus SQL Language Reference

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

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

To learn more about PostgreSQL and its features, you can visit the PostgreSQL 9.6 documentation.

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

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

CHAPTER 27

SQL Workarounds

Before attempting workarounds consider whether Citus is appropriate for your situation. Citus' current version works well for real-time analytics use cases. We are continuously working to increase SQL coverage to better support other use-cases such as data warehousing queries.

Citus supports most, but not all, SQL statements directly. Its SQL support continues to improve. Also many of the unsupported features have workarounds; below are a number of the most useful.

27.1 Subqueries in WHERE

A common type of query asks for values which appear in designated ways within a table, or aggregations of those values. For instance we might want to find which users caused events of types *A and B* in a table which records *one* user and event record per row:

```
select user_id
  from events
 where event_type = 'A'
   and user_id in (
     select user_id
       from events
      where event_type = 'B'
   )
```

Another example. How many distinct sessions viewed the top twenty-five most visited web pages?

```
select page_id, count(distinct session_id)
  from visits
 where page_id in (
   select page_id
     from visits
    group by page_id
    order by count(*) desc
    limit 25
```

(continues on next page)

(continued from previous page)

```
)
group by page_id;
```

Citus does not allow subqueries in the WHERE clause so we must choose a workaround.

27.1.1 Workaround 1. Generate explicit WHERE-IN expression

Best used when you don't want to complicate code in the application layer.

In this technique we use PL/pgSQL to construct and execute one statement based on the results of another.

```
-- create temporary table with results
do language plpgsql $$
  declare user_ids integer[];
begin
  execute
    'select user_id'
    '  from events'
    ' where event_type = ''B'''
    into user_ids;
  execute format(
    'create temp table results_temp as '
    'select user_id'
    '  from events'
    ' where user_id = any(array[%s])'
    '   and event_type = ''A''',
    array_to_string(user_ids, ','));
end;
$$;

-- read results, remove temp table
select * from results_temp;
drop table results_temp;
```

27.1.2 Workaround 2. Build query in SQL client

Best used for simple cases when the subquery returns limited rows.

Like the previous workaround this one creates an explicit list of values for an IN comparison. This workaround does too, except it does so in the application layer, not in the backend. It works best when there is a short list for the IN clause. For instance the page visits query is a good candidate because it limits its inner query to twenty-five rows.

```
-- first run this
select page_id
  from visits
 group by page_id
 order by count(*) desc
 limit 25;
```

Interpolate the list of ids into a new query

```
-- Notice the explicit list of ids obtained from previous query
-- and added by the application layer
select page_id, count(distinct session_id)
```

(continues on next page)

(continued from previous page)

```
  from visits
 where page_id in (2,3,5,7,13)
group by page_id
```

27.2 SELECT DISTINCT

Citus does not yet support SELECT DISTINCT but you can use GROUP BY for a simple workaround:

```
-- rather than this
-- select distinct col from table;

-- use this
select col from table group by col;
```

27.3 JOIN a local and a distributed table

Attempting to execute a JOIN between a local and a distributed table causes an error:

```
ERROR: cannot plan queries that include both regular and partitioned relations
```

In Citus Community and Enterprise editions there is a workaround. You can replicate the local table to a single shard on every worker and push the join query down to the workers. We do this by defining the table as a 'reference' table using a different table creation API and setting citus.shard_replication_factor to the current number of worker nodes. Suppose we want to join tables *here* and *there*, where *there* is already distributed but *here* is on the master database.

```
-- First get the number of current active worker nodes
SELECT count(1) FROM master_get_active_worker_nodes();

SET citus.shard_replication_factor = <number of nodes>

SELECT create_reference_table('here');
```

This will create a table with a single shard (non-distributed), but will replicate that shard to every node in the cluster. Now Citus will accept a join query between *here* and *there*, and each worker will have all the information it needs to work efficiently.

Note: Citus Cloud uses PostgreSQL replication, not Citus replication, so this technique does not work there.

27.4 Data Warehousing Queries

When queries have restrictive filters (i.e. when very few results need to be transferred to the master) there is a general technique to run unsupported queries in two steps. First store the results of the inner queries in regular PostgreSQL tables on the master. Then the next step can be executed on the master like a regular PostgreSQL query.

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

```
select repo_id, actor->'id', count(*)
  over (partition by repo_id)
  from github_events
 where repo_id = 1 or repo_id = 2;
```

You can re-write the query like below:

Statement 1:

```
create temp table results as (
  select repo_id, actor->'id' as actor_id
    from github_events
   where repo_id = 1 or repo_id = 2
);
```

Statement 2:

```
select repo_id, actor_id, count(*)
  over (partition by repo_id)
  from results;
```

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

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

CHAPTER 28

User Defined Functions Reference

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

28.1 Table and Shard DDL

28.1.1 create_distributed_table

The create_distributed_table() function is used to define a distributed table and create its shards if its a hash-distributed table. This function takes in a table name, the distribution column and an optional distribution method and inserts appropriate metadata to mark the table as distributed. The function defaults to 'hash' distribution if no distribution method is specified. If the table is hash-distributed, the function also creates worker shards based on the shard count and shard replication factor configuration values. This function replaces usage of master_create_distributed_table() followed by master_create_worker_shards().

Arguments

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

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

distribution_method: (Optional) The method according to which the table is to be distributed. Permissible values are append or hash, and defaults to 'hash'.

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 create_distributed_table('github_events', 'repo_id');
```

28.1.2 create_reference_table

The create_reference_table() function is used to define a small reference or dimension table. This function takes in a table name, and creates a distributed table with just one shard, and replication factor equal to the value specified in the citus.shard_replication_factor configuration variable. The distribution column is unimportant since the UDF only creates one shard for the table.

Arguments

table_name: Name of the small dimension or reference table which needs to be distributed.

Return Value

N/A

Example

This example informs the database that the nation table should be defined as a reference table

```
SELECT create_reference_table('nation');
```

28.1.3 master_create_distributed_table

Note: This function is deprecated, and replaced by *create_distributed_table*.

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

Arguments

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

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

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

Return Value

N/A

Example

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

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

28.1.4 master_create_worker_shards

Note: This function is deprecated, and replaced by *create_distributed_table*.

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

Arguments

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

shard_count: Number of shards to create.

replication_factor: Desired replication factor for each shard.

Return Value

N/A

Example

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

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

28.1.5 master_create_empty_shard

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

Arguments

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

Return Value

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

Example

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

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

28.2 Table and Shard DML

28.2.1 master_append_table_to_shard

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

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

Arguments

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

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

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

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

Return Value

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

Example

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

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

28.2.2 master_apply_delete_command

The master_apply_delete_command() function is used to delete shards which match the criteria specified by the delete command. This function deletes a shard only if all rows in the shard match the delete criteria. As the function uses shard metadata to decide whether or not a shard needs to be deleted, it requires the WHERE clause in the DELETE statement to be on the distribution column. If no condition is specified, then all shards of that table are deleted.

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

Arguments

delete_command: valid SQL DELETE command

Return Value

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

Example

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

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

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

28.2.3 master_modify_multiple_shards

The master_modify_multiple_shards() function is used to run a query against all shards which match the criteria specified by the query. As the function uses shard metadata to decide whether or not a shard needs to be updated, it requires the WHERE clause in the query to be on the distribution column. Depending on the value of citus.multi_shard_commit_protocol, the commit can be done in one- or two-phases.

Limitations:

- It cannot be called inside a transaction block
- It must be called with simple operator expressions only

Arguments

modify_query: A simple DELETE or UPDATE query as a string.

Return Value

N/A

Example

```
SELECT master_modify_multiple_shards(
  'DELETE FROM customer_delete_protocol WHERE c_custkey > 500 AND c_custkey < 500');
```

28.3 Metadata / Configuration Information

28.3.1 master_add_node

The master_add_node() function registers a new node addition in the cluster in the Citus metadata table pg_dist_node.

Arguments

node_name: DNS name or IP address of the new node to be added.

node_port: The port on which PostgreSQL is listening on the worker node.

Return Value

A tuple which represents a row from *pg_dist_node* table.

Example

```
select * from master_add_node('new-node', 12345);
 nodeid | groupid | nodename | nodeport | noderack | hasmetadata
--------+---------+----------+----------+----------+-------------
      7 |       7 | new-node |    12345 | default  | f
(1 row)
```

28.3.2 master_remove_node

The master_remove_node() function removes the specified node from the pg_dist_node metadata table. This function will error out if there are existing shard placements on this node in pg_dist_shard_placement. Thus, before using this function, the shards will need to be moved off that node.

Arguments

node_name: DNS name of the node to be removed.

node_port: The port on which PostgreSQL is listening on the worker node.

Return Value

N/A

Example

```
select master_remove_node('new-node', 12345);
 master_remove_node
--------------------

(1 row)
```

28.3.3 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 the pg_dist_node catalog table 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
```

(continues on next page)

(continued from previous page)

```
(3 rows)
```

28.3.4 master_get_table_metadata

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

Arguments

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

Return Value

A tuple containing the following information:

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

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

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

part_key: Distribution column for the table.

part_replica_count: Current shard replication count.

part_max_size: Current maximum shard size in bytes.

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

Example

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

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

28.4 Cluster Management And Repair Functions

28.4.1 master_copy_shard_placement

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

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

Arguments

shard_id: Id of the shard to be repaired.

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

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

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

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

Return Value

N/A

Example

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

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

28.4.2 rebalance_table_shards

Note: The rebalance_table_shards function is a part of Citus Enterprise. Please contact us to obtain this functionality.

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

Arguments

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

threshold: (Optional) A float number between 0.0 and 1.0 which indicates the maximum difference ratio of node utilization from average utilization. For example, specifying 0.1 will cause the shard rebalancer to attempt to balance all nodes to hold the same number of shards ±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}');
```

28.4.3 replicate_table_shards

Note: The replicate_table_shards function is a part of Citus Enterprise. Please contact us to obtain this functionality.

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

Arguments

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

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

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

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

Return Value

N/A

Examples

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

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

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

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

CHAPTER 29

Metadata Tables Reference

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

29.1 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	regclass	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'
partkey	text	Detailed information about the distribution column including column number, type and other relevant information.
colocationid	integer	Colocation group to which this table belongs. Tables in the same group allow colocated joins and distributed rollups among other optimizations. This value references the colocationid column in the pg_dist_colocation table.
repmodel	char	The method used for data replication. The values of this column corresponding to different replication methods are :- citus statement-based replication: 'c' postgresql streaming replication: 's'

```
SELECT * from pg_dist_partition;
 logicalrelid  | partmethod |                                                          ␣
↪partkey                                                        | colocationid |␣
↪repmodel
```

(continues on next page)

(continued from previous page)

```
---------------+------------+---------------------------------------------------------
↪-------------------------------------------------------------+--------------+-----
↪-----
 github_events | h          | {VAR :varno 1 :varattno 4 :vartype 20 :vartypmod -1␣
↪:varcollid 0 :varlevelsup 0 :varnoold 1 :varoattno 4 :location -1} |            2 |␣
↪c
 (1 row)
```

29.2 Shard table

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

Name	Type	Description
logicalrelid	regclass	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.
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 | shardminvalue | shardmaxvalue
---------------+---------+--------------+---------------+---------------
 github_events |  102026 | t            | 268435456     | 402653183
 github_events |  102027 | t            | 402653184     | 536870911
 github_events |  102028 | t            | 536870912     | 671088639
 github_events |  102029 | t            | 671088640     | 805306367
 (4 rows)
```

29.2.1 Shard Storage Types

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

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

29.3 Shard placement table

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

Name	Type	Description
shardid	bigint	Shard identifier associated with this placement. This values references the shardid column in the pg_dist_shard catalog table.
shardstate	int	Describes the state of this placement. Different shard states are discussed in the section below.
shardlength	bigint	For append distributed tables, the size of the shard placement on the worker node in bytes. For hash distributed tables, zero.
nodename	text	Host name or IP address 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.
placementid	bigint	Unique auto-generated identifier for each individual placement.

```
SELECT * from pg_dist_shard_placement;
  shardid | shardstate | shardlength | nodename  | nodeport | placementid
 ---------+------------+-------------+-----------+----------+-------------
   102008 |          1 |           0 | localhost |    12345 |           1
   102008 |          1 |           0 | localhost |    12346 |           2
   102009 |          1 |           0 | localhost |    12346 |           3
   102009 |          1 |           0 | localhost |    12347 |           4
   102010 |          1 |           0 | localhost |    12347 |           5
   102010 |          1 |           0 | localhost |    12345 |           6
   102011 |          1 |           0 | localhost |    12345 |           7
```

29.3.1 Shard Placement States

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

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

29.4 Worker node table

The pg_dist_node table contains information about the worker nodes in the cluster.

Name	Type	Description
nodeid	int	Auto-generated identifier for an individual node.
groupid	int	Identifier used to denote a group of one primary server and zero or more secondary servers, when the streaming replication model is used. By default it is the same as the nodeid.
nodename	text	Host Name or IP Address of the PostgreSQL worker node.
nodeport	int	Port number on which the PostgreSQL worker node is listening.
noderack	text	(Optional) Rack placement information for the worker node.
hasmetadata	boolean	Reserved for internal use.

```
SELECT * from pg_dist_node;
 nodeid | groupid | nodename  | nodeport | noderack | hasmetadata
--------+---------+-----------+----------+----------+-------------
      1 |       1 | localhost |    12345 | default  | f
      2 |       2 | localhost |    12346 | default  | f
      3 |       3 | localhost |    12347 | default  | f
(3 rows)
```

29.5 Colocation group table

The pg_dist_colocation table contains information about which tables' shards should be placed together, or *colocated*. When two tables are in the same colocation group, Citus ensures shards with the same partition values will be placed on the same worker nodes. This enables join optimizations, certain distributed rollups, and foreign key support. Shard colocation is inferred when the shard counts, replication factors, and partition column types all match between two tables; however, a custom colocation group may be specified when creating a distributed table, if so desired.

Name	Type	Description
colocationid	int	Unique identifier for the colocation group this row corresponds to.
shardcount	int	Shard count for all tables in this colocation group
replicationfactor	int	Replication factor for all tables in this colocation group.
distributioncolumntype	oid	The type of the distribution column for all tables in this colocation group.

```
SELECT * from pg_dist_colocation;
  colocationid | shardcount | replicationfactor | distributioncolumntype
 --------------+------------+-------------------+------------------------
             2 |         32 |                 2 |                     20
  (1 row)
```

CHAPTER 30

Configuration Reference

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

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

30.1 Node configuration

30.1.1 citus.max_worker_nodes_tracked (integer)

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

30.2 Data Loading

30.2.1 citus.multi_shard_commit_protocol (enum)

Sets the commit protocol to use when performing COPY on a hash distributed table. On each individual shard placement, the COPY is performed in a transaction block to ensure that no data is ingested if an error occurs during the COPY. However, there is a particular failure case in which the COPY succeeds on all placements, but a (hardware) failure occurs before all transactions commit. This parameter can be used to prevent data loss in that case by choosing between the following commit protocols:

- **1pc:** The transactions in which which COPY is performed on the shard placements is committed in a single round. Data may be lost if a commit fails after COPY succeeds on all placements (rare). This is the default protocol.

- **2pc:** The transactions in which COPY is performed on the shard placements are first prepared using PostgreSQL's two-phase commit and then committed. Failed commits can be manually recovered or aborted using COMMIT PREPARED or ROLLBACK PREPARED, respectively. When using 2pc, max_prepared_transactions should be increased on all the workers, typically to the same value as max_connections.

30.2.2 citus.shard_replication_factor (integer)

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

30.2.3 citus.shard_count (integer)

Sets the shard count for hash-partitioned tables and defaults to 32. This value is used by the *create_distributed_table* UDF when creating hash-partitioned tables. This parameter can be set at run-time and is effective on the master.

30.2.4 citus.shard_max_size (integer)

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

30.3 Planner Configuration

30.3.1 citus.large_table_shard_count (integer)

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

30.3.2 citus.limit_clause_row_fetch_count (integer)

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

30.3.3 citus.count_distinct_error_rate (floating point)

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

30.3.4 citus.task_assignment_policy (enum)

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

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

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

30.4 Intermediate Data Transfer Format

30.4.1 citus.binary_worker_copy_format (boolean)

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

30.4.2 citus.binary_master_copy_format (boolean)

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

30.5 DDL

30.5.1 citus.enable_ddl_propagation (boolean)

Specifies whether to automatically propagate DDL changes from the master to all workers. The default value is true. Because some schema changes require an access exclusive lock on tables and because the automatic propagation applies to all workers sequentially it can make a Citus cluter temporarily less responsive. You may choose to disable this setting and propagate changes manually.

Note: Currently CREATE INDEX and ALTER TABLE are the only DDL changes that Citus propagates automatically.

30.6 Executor Configuration

30.6.1 citus.all_modifications_commutative

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

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

30.6.2 citus.remote_task_check_interval (integer)

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

30.6.3 citus.task_executor_type (enum)

Citus has two 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 optimal when you require fast responses to queries that involve aggregations and colocated joins spanning across multiple shards.
- **task-tracker:** The task-tracker executor is well suited for long running, complex queries which require shuffling of data across worker nodes and efficient resource management.

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

30.6.4 Real-time executor configuration

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

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

max_connections (integer)

Sets the maximum number of concurrent connections to the database server. The default is typically 100 connections, but might be less if your kernel settings will not support it (as determined during initdb). The real time executor maintains an open connection for each shard to which it sends queries. Increasing this configuration parameter will allow the executor to have more concurrent connections and hence handle more shards in parallel. This parameter has to be changed on the workers as well as the master, and can be done only during server start.

max_files_per_process (integer)

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

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

30.6.5 Task tracker executor configuration

citus.task_tracker_delay (integer)

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

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

citus.max_tracked_tasks_per_node (integer)

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

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

citus.max_assign_task_batch_size (integer)

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

citus.max_running_tasks_per_node (integer)

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

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

citus.partition_buffer_size (integer)

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

30.6.6 Explain output

citus.explain_all_tasks (boolean)

By default, Citus shows the output of a single, arbitrary task when running EXPLAIN on a distributed query. In most cases, the explain output will be similar across tasks. Occassionally, some of the tasks will be planned differently or have much higher execution times. In those cases, it can be useful to enable this parameter, after which the EXPLAIN output will include all tasks. This may cause the EXPLAIN to take longer.

CHAPTER 31

Append Distribution

Note: Append distribution is a specialized technique which requires care to use efficiently. Hash distribution is a better choice for most situations.

While Citus' most common use cases involve hash data distribution, it can also distribute timeseries data across a variable number of shards by their order in time. This section provides a short reference to loading, deleting, and maninpulating timeseries data.

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

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

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

31.1 Creating and Distributing Tables

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

```
export PATH=/usr/lib/postgresql/9.6/:$PATH
```

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

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

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

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

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

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

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

31.2 Expiring Data

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

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

The example below deletes those shards from the github_events table which have all rows with created_at >= '2015-01-01 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␣
↪>= ''2015-01-01 00:00:00''');
 master_apply_delete_command
-----------------------------
                           3
(1 row)
```

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

31.3 Deleting Data

The most flexible way to modify or delete rows throughout a Citus cluster is the master_modify_multiple_shards command. It takes a regular SQL statement as argument and runs it on all workers:

```
SELECT master_modify_multiple_shards(
  'DELETE FROM github_events WHERE created_at >= ''2015-01-01 00:00:00''');
```

The function uses a configurable commit protocol to update or delete data safely across multiple shards. Unlike master_apply_delete_command, it works at the row- rather than shard-level to modify or delete all rows that match the condition in the where clause. It deletes rows regardless of whether they comprise an entire shard. To learn more about the function, its arguments and its usage, please visit the *User Defined Functions Reference* section of our documentation.

31.4 Dropping Tables

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

```
DROP TABLE github_events;
```

31.5 Data Loading

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

31.5.1 Bulk load using \copy

The \copy command is used to copy data from a file to a distributed table while handling replication and failures automatically. You can also use the server side COPY command. In the examples, we use the \copy command from psql, which sends a COPY .. FROM STDIN to the server and reads files on the client side, whereas COPY from a file would read the file on the server.

You can use \copy both on the master and from any of the workers. When using it from the worker, you need to add the master_host option. Behind the scenes, \copy first opens a connection to the master using the provided master_host option and uses master_create_empty_shard to create a new shard. Then, the command connects to the workers and copies data into the replicas until the size reaches shard_max_size, at which point another new shard is created. Finally, the command fetches statistics for the shards and updates the metadata.

```
SET citus.shard_max_size TO '64MB';
\copy github_events from 'github_events-2015-01-01-0.csv' WITH (format CSV, master_
↪host 'master-host-101')
```

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

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

(1) **citus.shard_max_size :-** This parameter determines the maximum size of a shard created using \copy, and defaults to 1 GB. If the file is larger than this parameter, \copy will break it up into multiple shards.

(2) **citus.shard_replication_factor :-** This parameter determines the number of nodes each shard gets replicated to, and defaults to two. The ideal value for this parameter depends on the size of the cluster and rate of node failure. For example, you may want to increase the replication factor if you run large clusters and observe node failures on a more frequent basis.

Note: The configuration setting citus.shard_replication_factor can only be set on the master node.

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

Note: There is no notion of snapshot isolation across shards, which means that a multi-shard SELECT that runs concurrently with a COPY might see it committed on some shards, but not on others. If the user is storing events data, he may occasionally observe small gaps in recent data. It is up to applications to deal with this if it is a problem (e.g. exclude the most recent data from queries, or use some lock).

If COPY fails to open a connection for a shard placement then it behaves in the same way as INSERT, namely to mark the placement(s) as inactive unless there are no more active placements. If any other failure occurs after connecting, the transaction is rolled back and thus no metadata changes are made.

31.5.2 Incremental loads by appending to existing shards

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

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

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

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

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

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

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

31.5.3 Increasing data loading performance

The methods described above enable you to achieve high bulk load rates which are sufficient for most use cases. If you require even higher data load rates, you can use the functions described above in several ways and write scripts to better control sharding and data loading. The next section explains how to go even faster.

31.6 Scaling Data Ingestion

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

31.6.1 Master Node Bulk Ingestion (100k/s-200k/s)

To ingest data into an append distributed table, you can use the COPY command, which will create a new shard out of the data you ingest. COPY can break up files larger than the configured citus.shard_max_size into multiple shards. COPY for append distributed tables only opens connections for the new shards, which means it behaves a bit differently than COPY for hash distributed tables, which may open connections for all shards. A COPY for append distributed tables command does not ingest rows in parallel over many connections, but it is safe to run many commands in parallel.

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

-- Add data into a new staging table
\COPY events FROM 'path-to-csv-file' WITH CSV
```

COPY creates new shards every time it is used, which allows many files to be ingested simultaneously, but may cause issues if queries end up involving thousands of shards. An alternative way to ingest data is to append it to existing shards using the master_append_table_to_shard function. To use master_append_table_to_shard, the data needs to be loaded into a staging table and some custom logic to select an appropriate shard is required.

```
-- Prepare a staging table
CREATE TABLE stage_1 (LIKE events);
\COPY stage_1 FROM 'path-to-csv-file WITH CSV

-- In a separate transaction, append the staging table
SELECT master_append_table_to_shard(select_events_shard(), 'stage_1', 'master-node',␣
↪5432);
```

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

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

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

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

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

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

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

To learn more about the master_append_table_to_shard and master_create_empty_shard UDFs, please visit the *User Defined Functions Reference* section of the documentation.

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

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

Append distributed tables support COPY via the worker, by specifying the address of the master in a master_host option, and optionally a master_port option (defaults to 5432). COPY via the workers has the same general properties as COPY via the master, except the initial parsing is not bottlenecked on the master.

```
psql -h worker-node-1 -c "\COPY events FROM 'data.csv' WITH (FORMAT CSV, MASTER_HOST
↪'master-node')"
```

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

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

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

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

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

  IF num_small_shards IS NULL OR num_small_shards < 20 THEN
    SELECT master_create_empty_shard('events') INTO shard_id;
  END IF;

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

A drawback of ingesting into many shards concurrently is that shards may span longer time ranges, which means that queries for a specific time period may involve shards that contain a lot of data outside of that period.

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

31.6.3 Pre-processing Data in Citus

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

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

```
CREATE TABLE github_events
(
```

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

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

For a more complete example, see Interactive Analytics on GitHub Data using PostgreSQL with Citus.

CHAPTER 32

Frequently Asked Questions

32.1 Can I create primary keys on distributed tables?

Currently Citus imposes primary key constraint only if the distribution column is a part of the primary key. This assures that the constraint needs to be checked only on one shard to ensure uniqueness.

32.2 How do I add nodes to an existing Citus cluster?

You can add nodes to a Citus cluster by calling the master_add_node UDF with the hostname (or IP address) and port number of the new node. After adding a node to an existing cluster, it will not contain any data (shards). Citus will start assigning any newly created shards to this node. To rebalance existing shards from the older nodes to the new node, the Citus Enterprise edition provides a shard rebalancer utility. You can find more information about shard rebalancing in the *Cluster Management* section.

32.3 How do I change the shard count for a hash partitioned table?

Optimal shard count is related to the total number of cores on the workers. Citus partitions an incoming query into its fragment queries which run on individual worker shards. Hence the degree of parallelism for each query is governed by the number of shards the query hits. To ensure maximum parallelism, you should create enough shards on each node such that there is at least one shard per CPU core.

We typically recommend creating a high number of initial shards, e.g. 2x or 4x the number of current CPU cores. This allows for future scaling if you add more workers and CPU cores.

32.4 How does Citus handle failure of a worker node?

If a worker node fails during e.g. a SELECT query, jobs involving shards from that server will automatically fail over to replica shards located on healthy hosts. This means intermittent failures will not require restarting potentially long-running analytical queries, so long as the shards involved can be reached on other healthy hosts. You can find more information about Citus' failure handling logic in *Dealing With Node Failures*.

32.5 How does Citus handle failover of the master node?

As the Citus master node is similar to a standard PostgreSQL server, regular PostgreSQL synchronous replication and failover can be used to provide higher availability of the master node. Many of our customers use synchronous replication in this way to add resilience against master node failure. You can find more information about handling *Master Node Failures*.

32.6 How do I ingest the results of a query into a distributed table?

Citus supports the INSERT / SELECT syntax for copying the results of a query on a distributed table into a distributed table, when the tables are colocated.

If your tables are not colocated, or you are using append distribution, there are workarounds you can use (for eg. using COPY to copy data out and then back into the destination table). Please contact us if your use-case demands such ingest workflows.

32.7 Can I join distributed and non-distributed tables together in the same query?

If you want to do joins between small dimension tables (regular Postgres tables) and large tables (distributed), then you could distribute the small tables by creating 1 shard and N (with N being the number of workers) replicas. Citus will then be able to push the join down to the worker nodes. If the local tables you are referring to are large, we generally recommend to distribute the larger tables to reap the benefits of sharding and parallelization which Citus offers. For a deeper discussion into the way Citus handles joins, please see our *Joins* documentation.

32.8 Are there any PostgreSQL features not supported by CitusDB?

Since Citus provides distributed functionality by extending PostgreSQL, it uses the standard PostgreSQL SQL constructs. This includes the support for wide range of data types (including semi-structured data types like jsonb, hstore), full text search, operators and functions, foreign data wrappers, etc.

PostgreSQL has a wide SQL coverage; and Citus does not support that entire spectrum out of the box when querying distributed tables. Some constructs which aren't supported natively for distributed tables are:

- Window Functions
- CTEs
- Set operations
- Transactional semantics for queries that span across multiple shards

It is important to note that you can still run all of those queries on regular PostgreSQL tables in the Citus cluster. As a result, you can address many use cases through a combination of rewriting the queries and/or adding some extensions. We are working on increasing the distributed SQL coverage for Citus to further simplify these queries. So, if you run into an unsupported construct, please contact us and we will do our best to help you out.

32.9 How do I choose the shard count when I hash-partition my data?

Optimal shard count is related to the total number of cores on the workers. Citus partitions an incoming query into its fragment queries which run on individual worker shards. Hence, the degree of parallelism for each query is governed by the number of shards the query hits. To ensure maximum parallelism, you should create enough shards on each node such that there is at least one shard per CPU core.

We typically recommend creating a high number of initial shards, e.g. 2x or 4x the number of current CPU cores. This allows for future scaling if you add more workers and CPU cores.

32.10 How does citus support count(distinct) queries?

Citus can push down count(distinct) entirely down to the worker nodes in certain situations (for example if the distinct is on the distribution column or is grouped by the distribution column in hash-partitioned tables). In other situations, Citus uses the HyperLogLog extension to compute approximate distincts. You can read more details on how to enable approximate *Count (Distinct) Aggregates*.

32.11 In which situations are uniqueness constraints supported on distributed tables?

Citus is able to enforce a primary key or uniqueness constraint only when the constrained columns contain the distribution column. In particular this means that if a single column constitutes the primary key then it has to be the distribution column as well.

This restriction allows Citus to localize a uniqueness check to a single shard and let PostgreSQL on the worker node do the check efficiently.