Citus Documentation

Release 6.2.4

Citus Data

Apr 07, 2022

Welcome to the documentation for Citus 6.2! 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

Citus serves many use cases. Two common ones are scaling multi-tenant (B2B) databases and real-time analytics.

1.1.1 Multi-Tenant Database

Most B2B applications already have the notion of a tenant, customer, or account built into their data model. In this model, the database serves many tenants, each of whose data is separate from other tenants.

Citus provides full SQL coverage for this workload, and enables scaling out your relational database to 100K+ tenants. Citus also adds new features for multi-tenancy. For example, Citus supports tenant isolation to provide performance guarantees for large tenants, and has the concept of reference tables to reduce data duplication across tenants.

These capabilities allow you to scale out your tenants' data across many machines, and easily add more CPU, memory, and disk resources. Further, sharing the same database schema across multiple tenants makes efficient use of hardware resources and simplifies database management.

1.1.2 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.

1.2 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:

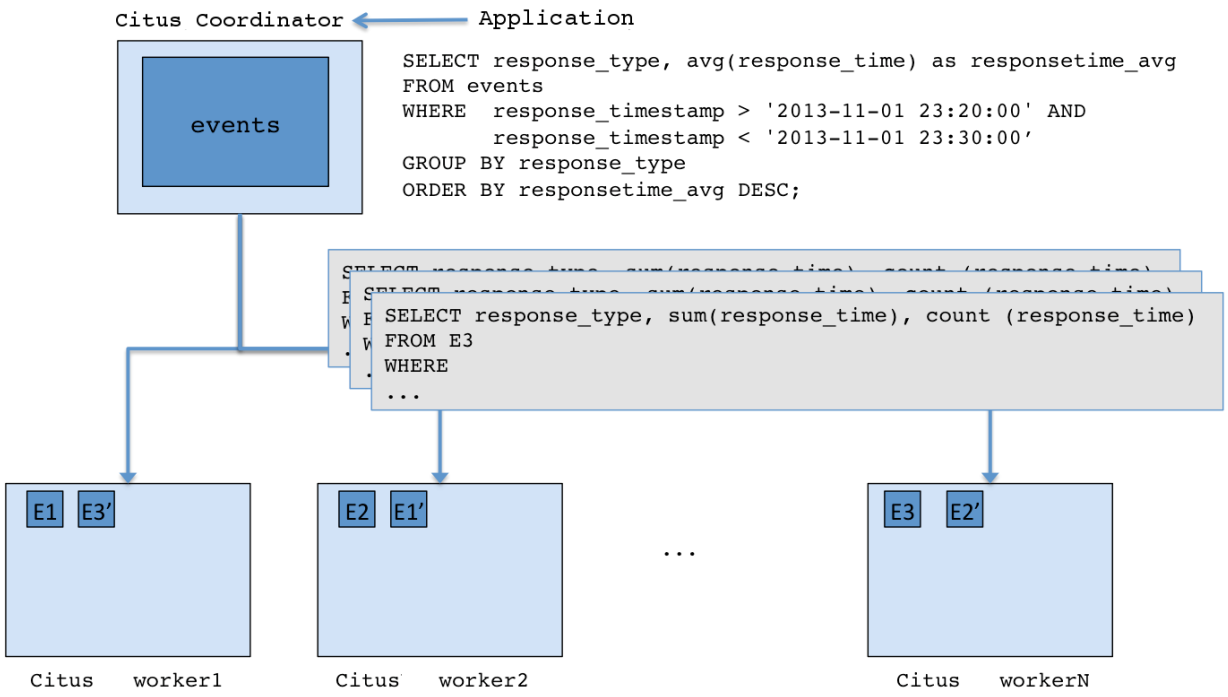
- Traditional data warehousing with long, free-form SQL
- Many distributed transactions across multiple shards
- Queries that return data-heavy ETL results rather than summaries

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.



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

2.1 Coordinator / Worker Nodes

You first choose one of the PostgreSQL instances in the cluster as the Citus coordinator. You then add the DNS names of worker PostgreSQL instances (Citus workers) to a metadata table on the coordinator. From that point on, you interact with the coordinator through standard PostgreSQL interfaces for data loading and querying. All the data is distributed across the workers. The coordinator 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 coordinator 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 coordinator 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 coordinator ever experiences a failure.

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 coordinator 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 coordinator 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 coordinator 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

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 coordinator.
- All the nodes in the cluster to connect to each other over TCP without any interactive authentication.

CHAPTER 4

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.

4.1 Docker (Mac or Linux)

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

1. Install Docker Community Edition and Docker Compose

On Mac:

- Install Docker.
- Start Docker by clicking on the application's icon.

On Linux:

```
curl -sSL https://get.docker.com/ | sh
sudo usermod -aG docker $USER && exec sg docker newgrp `id -gn`
sudo systemctl start docker

sudo curl -sSL https://github.com/docker/compose/releases/download/1.11.2/docker-
↪compose-`uname -s`-`uname -m` -o /usr/local/bin/docker-compose
sudo chmod +x /usr/local/bin/docker-compose
```

The above version of Docker Compose is sufficient for running Citus, or you can install the latest version. **2. Start the Citus Cluster**

Citus uses Docker Compose to run and connect containers holding the database coordinator node, workers, and a persistent data volume. To create a local cluster download our Docker Compose configuration file and run it

```
curl -L https://raw.githubusercontent.com/citusdata/docker/master/docker-compose.yml >
↪ 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.

Note: If you already have PostgreSQL running on your machine you may encounter this error when starting the Docker containers:

```
Error starting userland proxy:
Bind for 0.0.0.0:5432: unexpected error address already in use
```

This is because the "master" (coordinator) service attempts to bind to the standard PostgreSQL port 5432. Simply adjust `docker-compose.yml`. Under the `master` section change the host port from 5432 to 5433 or another non-conflicting number.

```
- ports: ['5432:5432']
+ ports: ['5433:5432']
```

3. Verify that installation has succeeded

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

Once you have the cluster up and running, you can visit our tutorials on *multi-tenant applications* or *real-time analytics* to get started with Citus in minutes.

4. Shut down the cluster when ready

When you wish to stop the docker containers, use Docker Compose:

```
docker-compose -p citus down
```

4.2 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 citus62_96
```

2. Initialize the Cluster

Citus has two kinds of components, the coordinator and the workers. The coordinator 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/coordinator citus/worker1 citus/worker2

# create three normal postgres instances
initdb -D citus/coordinator
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/coordinator/postgresql.conf
echo "shared_preload_libraries = 'citus'" >> citus/worker1/postgresql.conf
echo "shared_preload_libraries = 'citus'" >> citus/worker2/postgresql.conf
```

3. Start the coordinator and workers

We will start the PostgreSQL instances on ports 9700 (for the coordinator) 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/coordinator -o "-p 9700" -l coordinator_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 coordinator 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 coordinator node has picked up the desired worker configuration. First start the psql shell on the coordinator 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.

4.3 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-6.2
```

2. Initialize the Cluster

Citus has two kinds of components, the coordinator and the workers. The coordinator 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/coordinator citus/worker1 citus/worker2

# create three normal postgres instances
initdb -D citus/coordinator
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/coordinator/postgresql.conf
echo "shared_preload_libraries = 'citus'" >> citus/worker1/postgresql.conf
echo "shared_preload_libraries = 'citus'" >> citus/worker2/postgresql.conf
```

3. Start the coordinator and workers

We will start the PostgreSQL instances on ports 9700 (for the coordinator) 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/coordinator -o "-p 9700" -l coordinator_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 coordinator 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 coordinator node has picked up the desired worker configuration. First start the psql shell on the coordinator 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.

CHAPTER 5

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.

5.1 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.

5.1.1 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.

5.1.2 Manual CloudFormation Deployment

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

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

Introduction

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

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

1. Start a Citus cluster

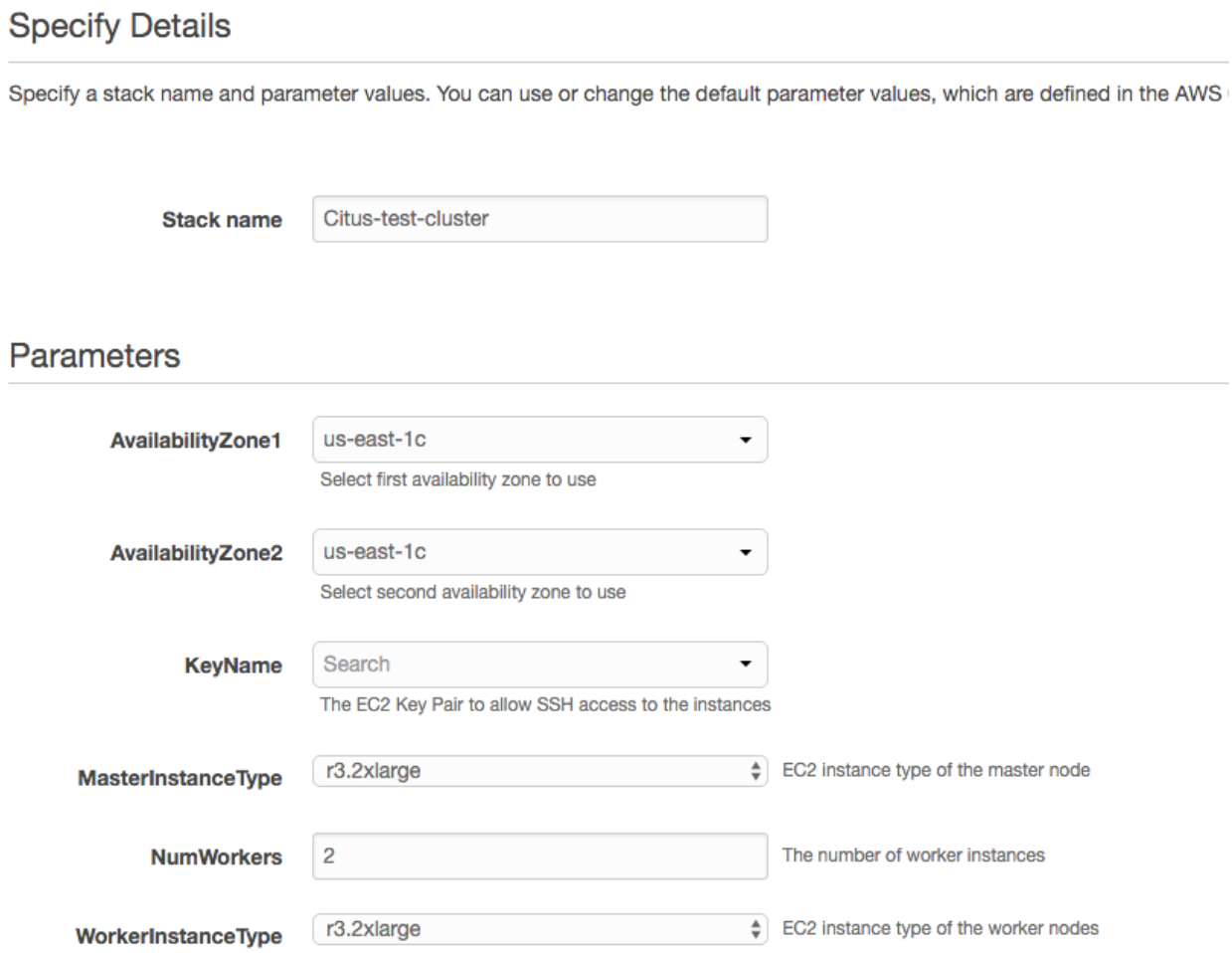
Note: You might need to login to AWS at this step if you aren't already logged in.

2. Select Citus template

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

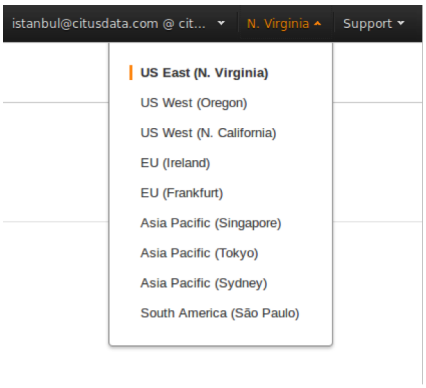
3. Fill the form with details about your cluster

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



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

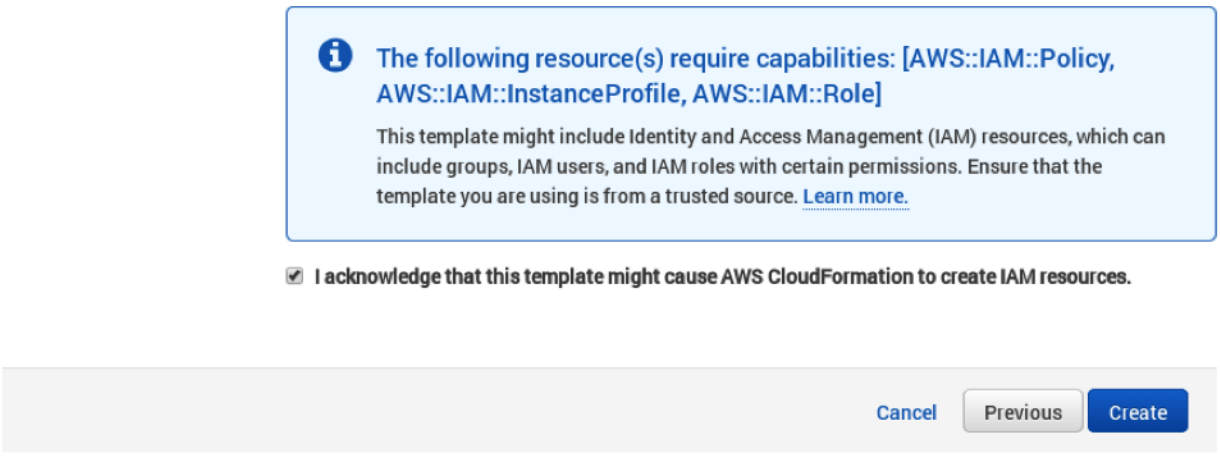
Note: If you want to launch a cluster in a region other than US East, you can update the region in the top-right corner of the AWS console as shown below.



4. Acknowledge IAM capabilities

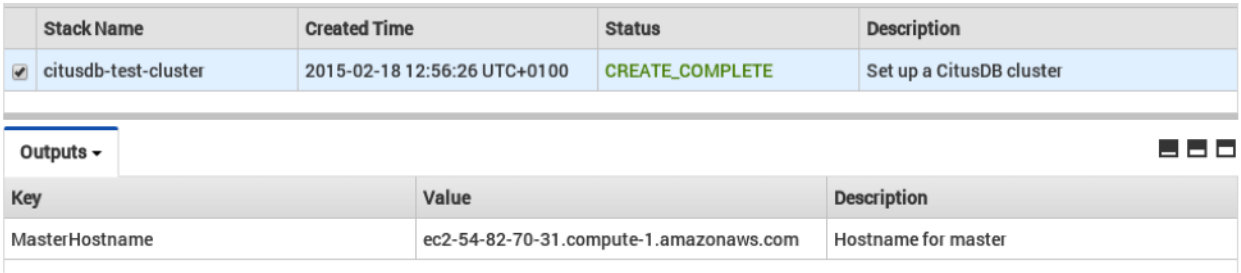
The next screen displays Tags and a few advanced options. For simplicity, we use the default options and click Next.

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



5. Cluster launching

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



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

Troubleshooting:

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

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

6. Login to the cluster

Once the cluster creation completes, you can immediately connect to the coordinator 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 coordinator 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.

5.2 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.

5.2.1 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 citus62_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, see our notes about *Worker Security*. 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;"
```

5.2.2 Steps to be executed on the coordinator node

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

1. Add worker node information

We need to inform the coordinator 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 coordinator 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 coordinator 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
```

5.3 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.

5.3.1 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-6.2

# 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, see our notes about *Worker Security*. 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;"
```

5.3.2 Steps to be executed on the coordinator node

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

1. Add worker node information

We need to inform the coordinator 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 coordinator 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
```

CHAPTER 6

Multi-tenant Applications

In this tutorial, we will use a sample ad analytics dataset to demonstrate how you can use Citus to power your multi-tenant application.

Note: This tutorial assumes that you already have Citus installed and running. If you don't have Citus running, you can:

- Provision a cluster using Citus Cloud, or
- Setup Citus locally using *Docker (Mac or Linux)*.

6.1 Data model and sample data

We will demo building the database for an ad-analytics app which companies can use to view, change, analyze and manage their ads and campaigns (see an example app). Such an application has good characteristics of a typical multi-tenant system. Data from different tenants is stored in a central database, and each tenant has an isolated view of their own data.

We will use three Postgres tables to represent this data. To get started, you will need to download sample data for these tables:

```
curl https://examples.citusdata.com/tutorial/companies.csv > companies.csv
curl https://examples.citusdata.com/tutorial/campaigns.csv > campaigns.csv
curl https://examples.citusdata.com/tutorial/ads.csv > ads.csv
```

If you are using Docker, you should use the `docker cp` command to copy the files into the Docker container.

```
docker cp companies.csv citus_master:.
docker cp campaigns.csv citus_master:.
docker cp ads.csv citus_master:.
```

6.2 Creating tables

To start, you can first connect to the Citus co-ordinator using psql.

If you are using Citus Cloud, you can connect by specifying the connection string (URL in the formation details):

```
psql connection-string
```

Please note that certain shells may require you to quote the connection string when connecting to Citus Cloud. For example, `psql "connection-string"`.

If you are using Docker, you can connect by running psql with the docker exec command:

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

Then, you can create the tables by using standard PostgreSQL `CREATE TABLE` commands.

```
CREATE TABLE companies (
    id bigint NOT NULL,
    name text NOT NULL,
    image_url text,
    created_at timestamp without time zone NOT NULL,
    updated_at timestamp without time zone NOT NULL
);

CREATE TABLE campaigns (
    id bigint NOT NULL,
    company_id bigint NOT NULL,
    name text NOT NULL,
    cost_model text NOT NULL,
    state text NOT NULL,
    monthly_budget bigint,
    blacklisted_site_urls text[],
    created_at timestamp without time zone NOT NULL,
    updated_at timestamp without time zone NOT NULL
);

CREATE TABLE ads (
    id bigint NOT NULL,
    company_id bigint NOT NULL,
    campaign_id bigint NOT NULL,
    name text NOT NULL,
    image_url text,
    target_url text,
    impressions_count bigint DEFAULT 0,
    clicks_count bigint DEFAULT 0,
    created_at timestamp without time zone NOT NULL,
    updated_at timestamp without time zone NOT NULL
);
```

Next, you can create primary key indexes on each of the tables just like you would do in PostgreSQL

```
ALTER TABLE companies ADD PRIMARY KEY (id);
ALTER TABLE campaigns ADD PRIMARY KEY (id, company_id);
ALTER TABLE ads ADD PRIMARY KEY (id, company_id);
```

6.3 Distributing tables and loading data

We will now go ahead and tell Citus to distribute these tables across the different nodes we have in the cluster. To do so, you can run `create_distributed_table` and specify the table you want to shard and the column you want to shard on. In this case, we will shard all the tables on the `company_id`.

```
SELECT create_distributed_table('companies', 'id');
SELECT create_distributed_table('campaigns', 'company_id');
SELECT create_distributed_table('ads', 'company_id');
```

Sharding all tables on the company identifier allows Citus to *colocate* the tables together and allow for features like primary keys, foreign keys and complex joins across your cluster. You can learn more about the benefits of this approach here.

Then, you can go ahead and load the data we downloaded into the tables using the standard PostgreSQL `\COPY` command. Please make sure that you specify the correct file path if you downloaded the file to some other location.

```
\copy companies from 'companies.csv' with csv;
\copy campaigns from 'campaigns.csv' with csv;
\copy ads from 'ads.csv' with csv;
```

6.4 Running queries

Now that we have loaded data into the tables, let's go ahead and run some queries. Citus supports standard `INSERT`, `UPDATE` and `DELETE` commands for inserting and modifying rows in a distributed table which is the typical way of interaction for a user-facing application.

For example, you can insert a new company by running:

```
INSERT INTO companies VALUES (5000, 'New Company', 'https://randomurl/image.png',␣
↪now(), now());
```

If you want to double the budget for all the campaigns of a company, you can run an UPDATE command:

```
UPDATE campaigns
SET monthly_budget = monthly_budget*2
WHERE company_id = 5;
```

Another example of such an operation would be to run transactions which span multiple tables. Let's say you want to delete a campaign and all its associated ads, you could do it atomically by running.

```
BEGIN;
DELETE from campaigns where id = 46 AND company_id = 5;
DELETE from ads where campaign_id = 46 AND company_id = 5;
COMMIT;
```

Other than transactional operations, you can also run analytics queries on this data using standard SQL. One interesting query for a company to run would be to see details about its campaigns with maximum budget.

```
SELECT name, cost_model, state, monthly_budget
FROM campaigns
WHERE company_id = 5
ORDER BY monthly_budget DESC
LIMIT 10;
```

We can also run a join query across multiple tables to see information about running campaigns which receive the most clicks and impressions.

```
SELECT campaigns.id, campaigns.name, campaigns.monthly_budget,
       sum(impressions_count) as total_impressions, sum(clicks_count) as total_clicks
FROM ads, campaigns
```

```
WHERE ads.company_id = campaigns.company_id
AND campaigns.company_id = 5
AND campaigns.state = 'running'
GROUP BY campaigns.id, campaigns.name, campaigns.monthly_budget
ORDER BY total_impressions, total_clicks;
```

With this, we come to the end of our tutorial on using Citus to power a simple multi-tenant application. As a next step, you can look at the *Distributing by Tenant ID* section to see how you can model your own data for multi-tenancy.

CHAPTER 7

Real-time Analytics

In this tutorial, we will demonstrate how you can use Citus to ingest events data and run analytical queries on that data in human real-time. For that, we will use a sample Github events dataset.

Note: This tutorial assumes that you already have Citus installed and running. If you don't have Citus running, you can:

- Provision a cluster using Citus Cloud, or
- Setup Citus locally using *Docker (Mac or Linux)*.

7.1 Data model and sample data

We will demo building the database for a real-time analytics application. This application will insert large volumes of events data and enable analytical queries on that data with sub-second latencies. In our example, we're going to work with the Github events dataset. This dataset includes all public events on Github, such as commits, forks, new issues, and comments on these issues.

We will use two Postgres tables to represent this data. To get started, you will need to download sample data for these tables:

```
curl https://examples.citusdata.com/tutorial/users.csv > users.csv
curl https://examples.citusdata.com/tutorial/events.csv > events.csv
```

If you are using Docker, you should use the `docker cp` command to copy the files into the Docker container.

```
docker cp users.csv citus_master:.
docker cp events.csv citus_master:.
```

7.2 Creating tables

To start, you can first connect to the Citus coordinator using psql.

If you are using Citus Cloud, you can connect by specifying the connection string (URL in the formation details):

```
psql connection-string
```

Please note that certain shells may require you to quote the connection string when connecting to Citus Cloud. For example, `psql "connection-string"`.

If you are using Docker, you can connect by running psql with the docker exec command:

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

Then, you can create the tables by using standard PostgreSQL `CREATE TABLE` commands.

```
CREATE TABLE github_events
(
    event_id bigint,
    event_type text,
    event_public boolean,
    repo_id bigint,
    payload jsonb,
    repo jsonb,
    user_id bigint,
    org jsonb,
    created_at timestamp
);

CREATE TABLE github_users
(
    user_id bigint,
    url text,
    login text,
    avatar_url text,
    gravatar_id text,
    display_login text
);
```

Next, you can create indexes on events data just like you would do in PostgreSQL. In this example, we're also going to create a `GIN` index to make querying on `jsonb` fields faster.

```
CREATE INDEX event_type_index ON github_events (event_type);
CREATE INDEX payload_index ON github_events USING GIN (payload jsonb_path_ops);
```

7.3 Distributing tables and loading data

We will now go ahead and tell Citus to distribute these tables across the nodes in the cluster. To do so, you can run `create_distributed_table` and specify the table you want to shard and the column you want to shard on. In this case, we will shard all the tables on `user_id`.

```
SELECT create_distributed_table('github_users', 'user_id');
SELECT create_distributed_table('github_events', 'user_id');
```

Sharding all tables on the user identifier allows Citus to *colocate* these tables together, and allows for efficient joins and distributed roll-ups. You can learn more about the benefits of this approach here.

Then, you can go ahead and load the data we downloaded into the tables using the standard PostgreSQL `\COPY` command. Please make sure that you specify the correct file path if you downloaded the file to a different location.

```
\copy github_users from 'users.csv' with csv;
\copy github_events from 'events.csv' with csv;
```

7.4 Running queries

Now that we have loaded data into the tables, let's go ahead and run some queries. First, let's check how many users we have in our distributed database.

```
SELECT count(*) FROM github_users;
```

Now, let's analyze Github push events in our data. We will first compute the number of commits per minute by using the number of distinct commits in each push event.

```
SELECT date_trunc('minute', created_at) AS minute,
       sum((payload->>'distinct_size')::int) AS num_commits
FROM github_events
WHERE event_type = 'PushEvent'
GROUP BY minute
ORDER BY minute;
```

We also have a users table. We can also easily join the users with events, and find the top ten users who created the most repositories.

```
SELECT login, count(*)
FROM github_events ge
JOIN github_users gu
ON ge.user_id = gu.user_id
WHERE event_type = 'CreateEvent' AND payload @> '{"ref_type": "repository"}'
GROUP BY login
ORDER BY count(*) DESC LIMIT 10;
```

Citus also supports standard `INSERT`, `UPDATE`, and `DELETE` commands for ingesting and modifying data. For example, you can update a user's display login by running the following command:

```
UPDATE github_users SET display_login = 'no1youknow' WHERE user_id = 24305673;
```

With this, we come to the end of our tutorial. As a next step, you can look at the *Distributing by Entity ID* section to see how you can model your own data and power real-time analytical applications. 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 8

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.

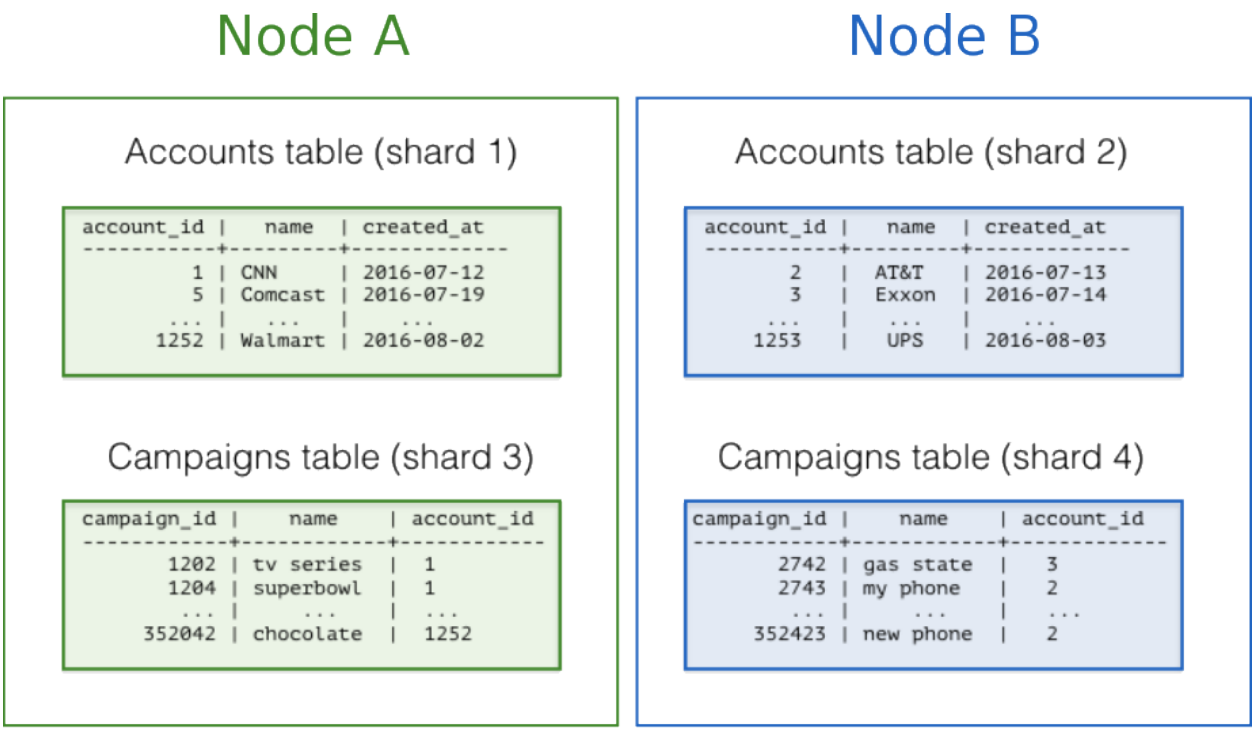
If your situation resembles either of these cases then the next step is to decide how to shard your data in a Citus cluster. As explained in *Architecture*, Citus assigns table rows to shards according to the hashed value of the table's distribution column. The database administrator's choice of distribution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

CHAPTER 9

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 queries to see which tenant id they involve and routes the query to a single worker 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 *Table Co-Location*.

The following diagram illustrates co-location in the multi-tenant data model. It contains two tables, Accounts and Campaigns, each distributed by `account_id`. The shaded boxes represent shards, each of whose color represents which worker node contains it. Green shards are stored together on one worker node, and blue on another. Notice how a join query between Accounts and Campaigns would have all the necessary data together on one node when restricting both tables to the same account_id.



To apply this design in your own schema the first step is identifying what constitutes a tenant in your application. 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*.

9.1 Typical Multi-Tenant Schema

Most SaaS applications already have the notion of tenancy built into their data model. In the following, we will look at an example schema from the online advertising space. In this example, a web advertising platform has tenants that it refers to as accounts. Each account holds and tracks advertising clicks across various campaigns.

```
CREATE TABLE accounts (
  id bigint,
  name text NOT NULL,
  image_url text NOT NULL,

  PRIMARY KEY (id)
);

CREATE TABLE ads (
  id bigint,
  account_id bigint,
  campaign_id bigint,
  name text NOT NULL,
  image_url text NOT NULL,
  target_url text NOT NULL,
  impressions_count bigint DEFAULT 0 NOT NULL,
  clicks_count bigint DEFAULT 0 NOT NULL,

  PRIMARY KEY (account_id, id),
  FOREIGN KEY (account_id) REFERENCES accounts
);

CREATE TABLE clicks (
  id bigint,
  account_id bigint,
  ad_id bigint,
  clicked_at timestamp without time zone NOT NULL,
  site_url text NOT NULL,
  cost_per_click_usd numeric(20,10),
  user_ip inet NOT NULL,
  user_data jsonb NOT NULL,

  PRIMARY KEY (account_id, id),
  FOREIGN KEY (account_id) REFERENCES accounts,
  FOREIGN KEY (account_id, ad_id) REFERENCES ads (account_id, id)
);

SELECT create_distributed_table('accounts',  'id');
SELECT create_distributed_table('ads',       'account_id');
SELECT create_distributed_table('clicks',    'account_id');
```

Notice how the primary and foreign keys always contain the tenant id (in this case `account_id`). Often this re-

quires them to be compound keys. Enforcing key constraints is generally difficult in distributed databases. For Citus, the inclusion of the tenant id allows the database to push DML down to single nodes and successfully enforce the constraint.

Queries including a tenant id enable more than just key constraints. Such queries enjoy full SQL coverage in Citus, including JOINs, transactions, grouping, and aggregates. In the multi-tenant architecture, SQL queries that filter by tenant id work without modification, combining the familiarity of PostgreSQL with the power of horizontal scaling for large numbers of tenants.

Let's look at example queries that span some of these capabilities. First an analytical query to count newly arriving clicks per campaign for an arbitrary account, say account id=9700. Citus pushes this query down to the node containing tenant 9700 and executes it all in one place. Notice the tenant id is included in the join conditions.

```
SELECT ads.campaign_id, COUNT(*)
  FROM ads
  JOIN clicks c
    ON (ads.id = ad_id AND ads.account_id = c.account_id)
 WHERE ads.account_id = 9700
   AND clicked_at > now()::date
 GROUP BY ads.campaign_id;
```

What's more, Citus gives full ACID guarantees for single-tenant DML. The following query transactionally removes the record of a click (id = 12995) and decrements the click count cache for its associated ad. Notice we include a filter for `account_id` on all the statements to ensure they affect the same tenant.

```
BEGIN;

-- get the ad id for later update
SELECT ad_id
  FROM clicks
 WHERE id = 12995
   AND account_id = 9700;

-- delete the click
DELETE FROM clicks
 WHERE id = 12995
   AND account_id = 9700;

-- decrement the ad click count for the ad we previously found
UPDATE ads
   SET clicks_count = clicks_count - 1
 WHERE id = <the ad id>
   AND account_id = 9700;

COMMIT;
```

We've seen some of the benefits of Citus for single-tenant queries, but it can also run and parallelize many kinds of queries across tenants, including aggregates. For instance, we can request the total clicks for ads by account:

```
SELECT account_id, sum(clicks_count) AS total_clicks
  FROM ads GROUP BY account_id
ORDER BY total_clicks DESC;
```

There is even a way to run DML statements on multiple tenants. As long as the update statement references data local to its own tenant it can be applied simultaneously to all tenants with a helper UDF called `master_modify_multiple_shards`. Here is an example of modifying all image urls to use secure connections.

```
SELECT master_modify_multiple_shards(
  'UPDATE ads SET image_url = replace(image_url, ''http:'', ''https:'')'
);
```

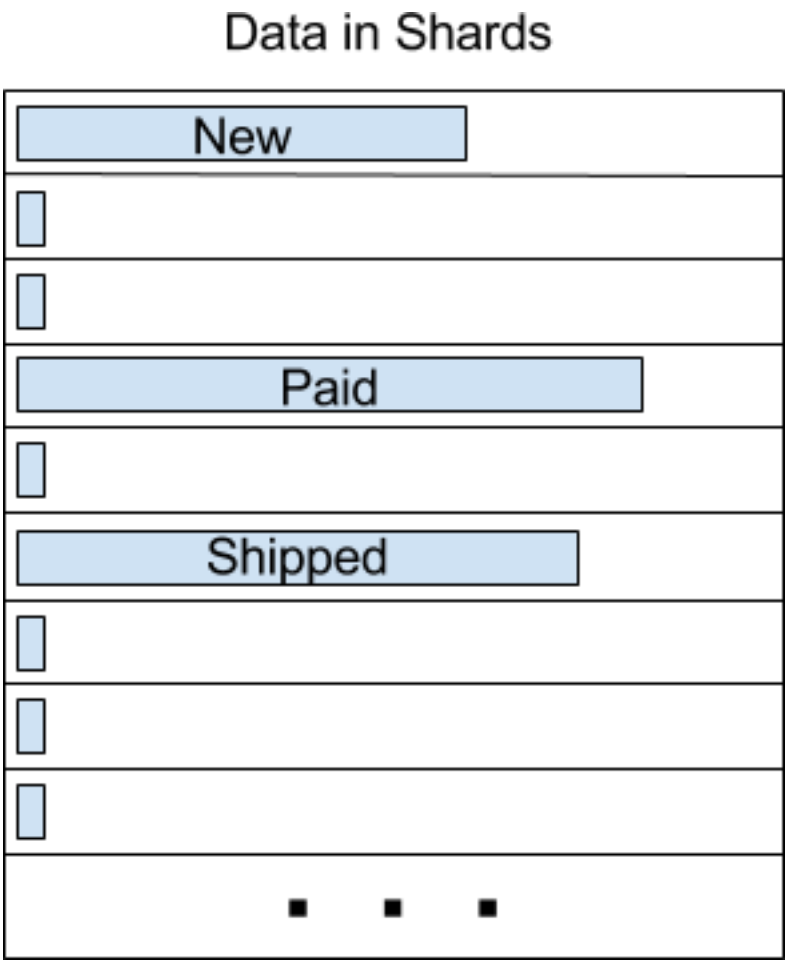
CHAPTER 10

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.

10.1 Typical Real-Time Schemas

10.1.1 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);

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_time >= now() - interval '10 minutes'
) ev
GROUP BY minute
ORDER BY minute ASC;
```

10.1.2 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;
```

10.1.3 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.

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

10.1.4 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 *co-location*. 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 11

Table Co-Location

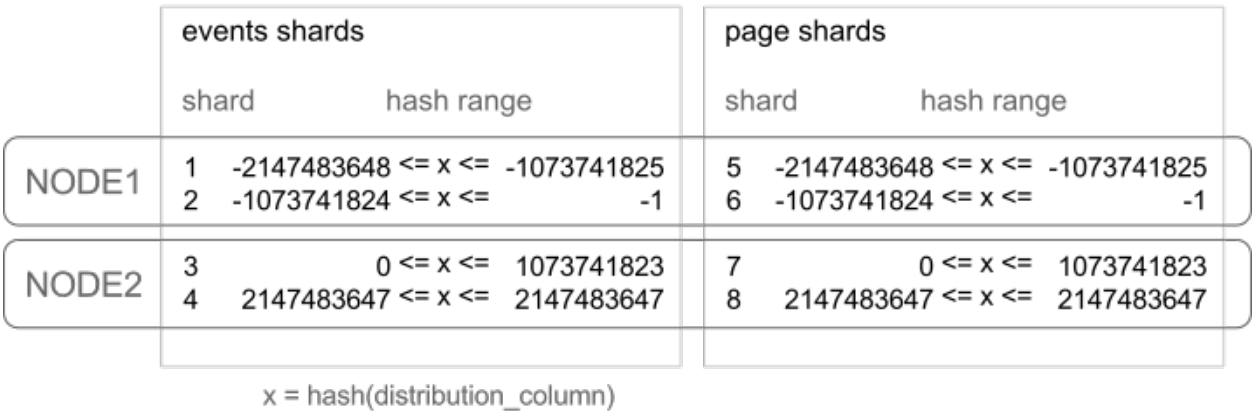
Relational databases are the first choice of data store for many applications due to their enormous flexibility and reliability. Historically the one criticism of relational databases is that they can run on only a single machine, which creates inherent limitations when data storage needs outpace server improvements. The solution to rapidly scaling databases is to distribute them, but this creates a performance problem of its own: relational operations such as joins then need to cross the network boundary. Co-location is the practice of dividing data tactically, where one keeps related information on the same machines to enable efficient relational operations, but takes advantage of the horizontal scalability for the whole dataset.

The principle of data co-location is that all tables in the database have a common distribution column and are sharded across machines in the same way, such that rows with the same distribution column value are always on the same machine, even across different tables. As long as the distribution column provides a meaningful grouping of data, relational operations can be performed within the groups.

11.1 Data co-location in Citus for hash-distributed tables

The Citus extension for PostgreSQL is unique in being able to form a distributed database of databases. Every node in a Citus cluster is a fully functional PostgreSQL database and Citus adds the experience of a single homogenous database on top. While it does not provide the full functionality of PostgreSQL in a distributed way, in many cases it can take full advantage of features offered by PostgreSQL on a single machine through co-location, including full SQL support, transactions and foreign keys.

In Citus a row is stored in a shard if the hash of the value in the distribution column falls within the shard's hash range. To ensure co-location, shards with the same hash range are always placed on the same node even after rebalance operations, such that equal distribution column values are always on the same node across tables.



A distribution column that we've found to work well in practice is tenant ID in multi-tenant applications. For example, SaaS applications typically have many tenants, but every query they make is specific to a particular tenant. While one option is providing a database or schema for every tenant, it is often costly and impractical as there can be many operations that span across users (data loading, migrations, aggregations, analytics, schema changes, backups, etc). That becomes harder to manage as the number of tenants grows.

11.2 A practical example of co-location

Consider the following tables which might be part of a multi-tenant web analytics SaaS:

```
CREATE TABLE event (
  event_id bigint,
  tenant_id int,
  page_id int,
  payload jsonb,
  primary key (tenant_id, event_id)
);

CREATE TABLE page (
  page_id int,
  tenant_id int,
  path text,
  primary key (tenant_id, page_id)
);
```

Now we want to answer queries that may be issued by a customer-facing dashboard, such as: "Return the number of visits in the past week for all pages starting with '/blog' in tenant six."

11.3 Using Regular PostgreSQL Tables

If our data was in a single PostgreSQL node, we could easily express our query using the rich set of relational operations offered by SQL:

```
SELECT page_id, count(event_id)
FROM
  page
LEFT JOIN  (
  SELECT * FROM event
  WHERE (payload->>'time')::timestamptz >= now() - interval '1 week'
) recent
USING (tenant_id, page_id)
WHERE tenant_id = 6 AND path LIKE '/blog%'
GROUP BY page_id;
```

As long the working set for this query fits in memory, this is an appropriate solution for many application since it offers maximum flexibility. However, even if you don't need to scale yet, it can be useful to consider the implications of scaling out on your data model.

11.4 Distributing tables by ID

As the number of tenants and the data stored for each tenant grows, query times will typically go up as the working set no longer fits in memory or CPU becomes a bottleneck. In this case, we can shard the data across many nodes using

Citus. The first and most important choice we need to make when sharding is the distribution column. Let's start with a naive choice of using `event_id` for the event table and `page_id` for the `page` table:

```
-- naively use event_id and page_id as distribution columns
SELECT create_distributed_table('event', 'event_id');
SELECT create_distributed_table('page', 'page_id');
```

Given that the data is dispersed across different workers, we cannot simply perform a join as we would on a single PostgreSQL node. Instead, we will need to issue two queries:

Across all shards of the page table (Q1):

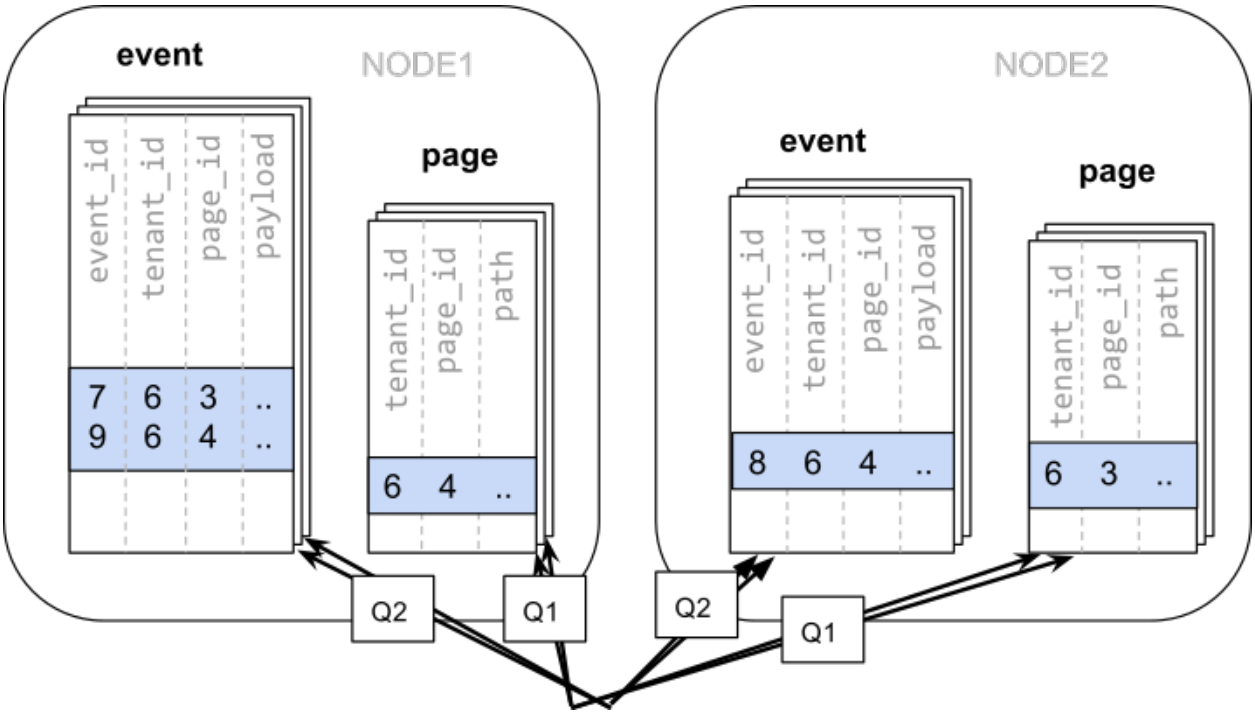
```
SELECT page_id FROM page WHERE path LIKE '/blog%' AND tenant_id = 6;
```

Across all shards of the event table (Q2):

```
SELECT page_id, count(*) AS count
FROM event
WHERE page_id IN (ARRAY[...page IDs from first query...]) AND
      tenant_id = 6 AND
      payload->>'time' >= now() - interval '1 week'
GROUP BY page_id ORDER BY count DESC LIMIT 10;
```

Afterwards, the results from the two steps need to be combined by the application.

The data required to answer the query is scattered across the shards on the different nodes and each of those shards will need to be queried:



In this case the data distribution creates substantial drawbacks:

- Overhead from querying each shard, running multiple queries
- Overhead of Q1 returning many rows to the client
- Q2 becoming very large

- The need to write queries in multiple steps, combine results, requires changes in the application

A potential upside of the relevant data being dispersed is that the queries can be parallelised, which Citus will do. However, this is only beneficial if the amount of work that the query does is substantially greater than the overhead of querying many shards. It's generally better to avoid doing such heavy lifting directly from the application, for example by pre-aggregating the data.

11.5 Distributing tables by tenant

Looking at our query again, we can see that all the rows that the query needs have one dimension in common: `tenant_id`. The dashboard will only ever query for a tenant's own data. That means that if data for the same tenant are always co-located on a single PostgreSQL node, our original query could be answered in a single step by that node by performing a join on `tenant_id` and `page_id`.

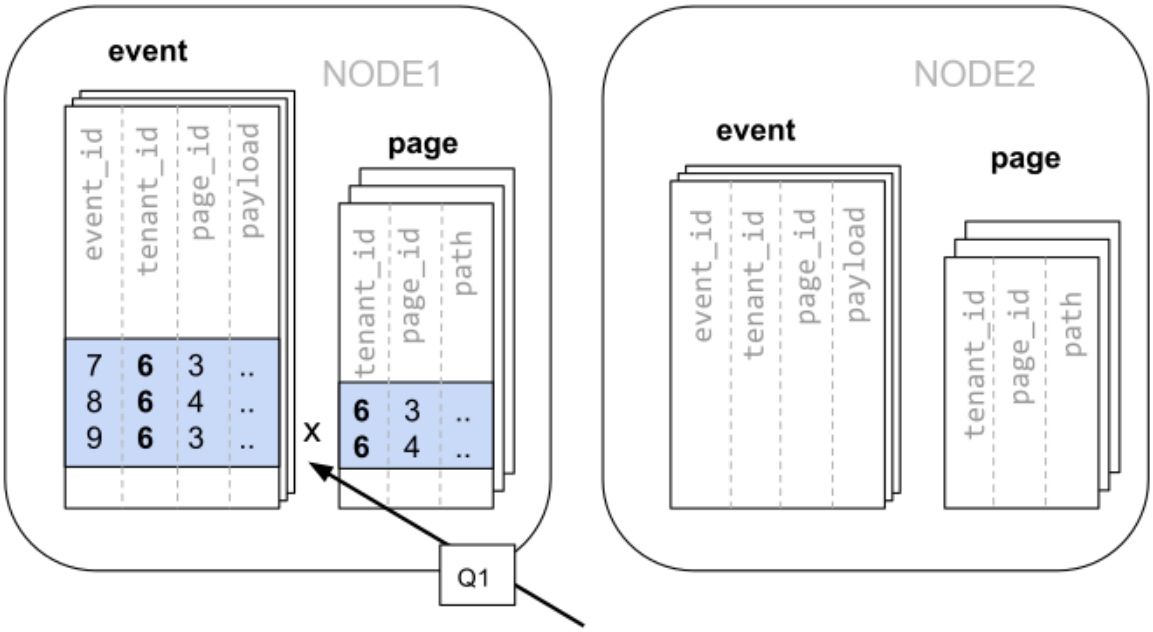
In Citus, rows with the same distribution column value are guaranteed to be on the same node. Each shard in a distributed table effectively has a set of co-located shards from other distributed tables that contain the same distribution column values (data for the same tenant). Starting over, we can create our tables with `tenant_id` as the distribution column.

```
-- co-locate tables by using a common distribution column
SELECT create_distributed_table('event', 'tenant_id');
SELECT create_distributed_table('page', 'tenant_id', colocate_with => 'event');
```

In this case, Citus can answer the same query that you would run on a single PostgreSQL node without modification (Q1):

```
SELECT page_id, count(event_id)
FROM
  page
LEFT JOIN  (
  SELECT * FROM event
  WHERE (payload->>'time')::timestamptz >= now() - interval '1 week'
) recent
USING (tenant_id, page_id)
WHERE tenant_id = 6 AND path LIKE '/blog%'
GROUP BY page_id;
```

Because of the tenantid filter and join on tenantid, Citus knows that the entire query can be answered using the set of co-located shards that contain the data for that particular tenant, and the PostgreSQL node can answer the query in a single step, which enables full SQL support.



In some cases, queries and table schemas will require minor modifications to ensure that the tenant_id is always included in unique constraints and join conditions. However, this is usually a straightforward change, and the extensive rewrite that would be required without having co-location is avoided.

While the example above queries just one node because there is a specific tenant_id = 6 filter, co-location also allows us to efficiently perform distributed joins on tenant_id across all nodes, be it with SQL limitations.

11.6 Co-location means better feature support

The full list of Citus features that are unlocked by co-location are:

- Full SQL support for queries on a single set of co-located shards
- Multi-statement transaction support for modifications on a single set of co-located shards
- Aggregation through INSERT..SELECT
- Foreign keys
- Distributed outer joins

Data co-location is a powerful technique for providing both horizontal scale and supporting relational data models. The cost of migrating or building applications using a distributed database that enables relational operations through co-location is often substantially lower than moving to a restrictive data model (e.g. NoSQL) and unlike a single-node database it can scale out with the size of your business. For more information about migrating an existing database see *Transitioning to a Multi-Tenant Data Model*.

CHAPTER 12

Creating and Modifying Distributed Tables (DDL)

12.1 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.

```
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 coordinator. Once the replicas are created, this function saves all distributed metadata on the coordinator.

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 *Citus Utility Function Reference* of our documentation.

12.1.1 Reference Tables

The above method distributes tables into multiple horizontal shards, but it's also possible to distribute tables into a single shard and replicate it to every worker node. Tables distributed this way are called *reference tables*. They are typically small non-partitioned tables which we want to locally join with other tables on any worker. One US-centric example is information about states.

```
-- a reference table

CREATE TABLE states (
  code char(2) PRIMARY KEY,
  full_name text NOT NULL,
  general_sales_tax numeric(4,3)
);

-- distribute it to all workers

SELECT create_reference_table('states');
```

Other queries, such as one calculating tax for a shopping cart, can join on the `states` table with no network overhead.

In addition to distributing a table as a single replicated shard, the `create_reference_table` UDF marks it as a reference table in the Citus metadata tables. Citus automatically performs two-phase commits (2PC) for modifications to tables marked this way, which provides strong consistency guarantees.

If you have an existing distributed table which has a shard count of one, you can upgrade it to be a recognized reference table by running

```
SELECT upgrade_to_reference_table('table_name');
```

12.1.2 Distributing Coordinator Data

If an existing PostgreSQL database is converted into the coordinator node for a Citus cluster, the data in its tables can be distributed efficiently and with minimal interruption to an application.

The `create_distributed_table` function described earlier works on both empty and non-empty tables, and for the latter automatically distributes table rows throughout the cluster. You will know if it does this by the presence of the message, "NOTICE: Copying data from local table..." For example:

```
CREATE TABLE series AS SELECT i FROM generate_series(1,1000000) i;
SELECT create_distributed_table('series', 'i');
NOTICE:  Copying data from local table...
 create_distributed_table
 --------------------------

 (1 row)
```

Writes on the table are blocked while the data is migrated, and pending writes are handled as distributed queries once the function commits. (If the function fails then the queries become local again.) Reads can continue as normal and will become distributed queries once the function commits.

Note: When distributing a number of tables with foreign keys between them, it's best to drop the foreign keys before running `create_distributed_table` and recreating them after distributing the tables. Foreign keys cannot always be enforced when one table is distributed and the other is not.

When migrating data from an external database, such as from Amazon RDS to Citus Cloud, first create the Citus distributed tables via `create_distributed_table`, then copy the data into the table.

12.2 Co-Locating Tables

Co-location is the practice of dividing data tactically, keeping related information on the same machines to enable efficient relational operations, while taking advantage of the horizontal scalability for the whole dataset. For more information and examples see *Table Co-Location*.

Tables are co-located in groups. To manually control a table's co-location group assignment use the optional `colocate_with` parameter of `create_distributed_table`. If you don't care about a table's co-location then omit this parameter. It defaults to the value `'default'`, which groups the table with any other default co-location table having the same distribution column type, shard count, and replication factor.

```
-- these tables are implicitly co-located by using the same
-- distribution column type and shard count with the default
-- co-location group

SELECT create_distributed_table('A', 'some_int_col');
SELECT create_distributed_table('B', 'other_int_col');
```

If you would prefer a table to be in its own co-location group, specify `'none'`.

```
-- not co-located with other tables

SELECT create_distributed_table('A', 'foo', colocate_with => 'none');
```

To co-locate a number of tables, distribute one and then put the others into its co-location group. For example:

```
-- distribute stores
SELECT create_distributed_table('stores', 'store_id');

-- add to the same group as stores
SELECT create_distributed_table('orders', 'store_id', colocate_with => 'stores');
SELECT create_distributed_table('products', 'store_id', colocate_with => 'stores');
```

Information about co-location groups is stored in the *pg_dist_colocation* table, while *pg_dist_partition* reveals which tables are assigned to which groups.

12.2.1 Upgrading from Citus 5.x

Starting with Citus 6.0, we made co-location a first-class concept, and started tracking tables' assignment to co-location groups in pg_dist_colocation. Since Citus 5.x didn't have this concept, tables created with Citus 5 were not explicitly marked as co-located in metadata, even when the tables were physically co-located.

Since Citus uses co-location metadata information for query optimization and pushdown, it becomes critical to inform Citus of this co-location for previously created tables. To fix the metadata, simply mark the tables as co-located:

```
-- Assume that stores, products and line_items were created in a Citus 5.x database.

-- Put products and line_items into store's co-location group
SELECT mark_tables_colocated('stores', ARRAY['products', 'line_items']);
```

This function requires the tables to be distributed with the same method, column type, number of shards, and replication method. It doesn't re-shard or physically move data, it merely updates Citus metadata.

12.3 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;
```

12.4 Modifying Tables

Citus automatically propagates many kinds of DDL statements, which means that modifying a distributed table on the coordinator node will update shards on the workers too. Other DDL statements require manual propagation, and certain others are prohibited such as those which would modify a distribution column. Attempting to run DDL that is ineligible for automatic propagation will raise an error and leave tables on the coordinator node unchanged. Additionally, some constraints like primary keys and uniqueness can only be applied prior to distributing a table.

By default Citus performs DDL with a one-phase commit protocol. For greater safety you can enable two-phase commits by setting

```
SET citus.multi_shard_commit_protocol = '2pc';
```

Here is a reference of the categories of DDL statements which propagate. Note that automatic propagation can be enabled or disabled with a *configuration parameter*.

12.4.1 Adding/Modifying Columns

Citus propagates most ALTER TABLE commands automatically. Adding columns or changing their default values work as they would in a single-machine PostgreSQL database:

```
-- Adding a column

ALTER TABLE products ADD COLUMN description text;

-- Changing default value

ALTER TABLE products ALTER COLUMN price SET DEFAULT 7.77;
```

Significant changes to an existing column are fine too, except for those applying to the *distribution column*. This column determines how table data distributes through the Citus cluster and cannot be modified in a way that would change data distribution.

```
-- Cannot be executed against a distribution column

-- Removing a column

ALTER TABLE products DROP COLUMN description;

-- Changing column data type

ALTER TABLE products ALTER COLUMN price TYPE numeric(10,2);

-- Renaming a column
```

```
ALTER TABLE products RENAME COLUMN product_no TO product_number;
```

12.4.2 Adding/Removing Constraints

Using Citus allows you to continue to enjoy the safety of a relational database, including database constraints (see the PostgreSQL docs). Due to the nature of distributed systems, Citus will not cross-reference uniqueness constraints or referential integrity between worker nodes. Foreign keys must always be declared between *colocated tables*. To do this, use compound foreign keys that include the distribution column.

This example, excerpted from a *Typical Multi-Tenant Schema*, shows how to create primary and foreign keys on distributed tables.

```
--
-- Adding a primary key
-- --------------------

-- Ultimately we'll distribute these tables on the account id, so the
-- ads and clicks tables use compound keys to include it.

ALTER TABLE accounts ADD PRIMARY KEY (id);
ALTER TABLE ads ADD PRIMARY KEY (account_id, id);
ALTER TABLE clicks ADD PRIMARY KEY (account_id, id);

-- Next distribute the tables
-- (primary keys must be created prior to distribution)

SELECT create_distributed_table('accounts',  'id');
SELECT create_distributed_table('ads',       'account_id');
SELECT create_distributed_table('clicks',    'account_id');

--
-- Adding foreign keys
-- -------------------

-- Note that this can happen before or after distribution, as long as
-- there exists a uniqueness constraint on the target column(s) which
-- can only be enforced before distribution.

ALTER TABLE ads ADD CONSTRAINT ads_account_fk
  FOREIGN KEY (account_id) REFERENCES accounts (id);
ALTER TABLE clicks ADD CONSTRAINT clicks_account_fk
  FOREIGN KEY (account_id) REFERENCES accounts (id);
```

Uniqueness constraints, like primary keys, must be added prior to table distribution.

```
-- Suppose we want every ad to use a unique image. Notice we can
-- enforce it only per account when we distribute by account id.

ALTER TABLE ads ADD CONSTRAINT ads_unique_image
  UNIQUE (account_id, image_url);
```

Not-null constraints can always be applied because they require no lookups between workers.

```
ALTER TABLE ads ALTER COLUMN image_url SET NOT NULL;
```

12.4.3 Adding/Removing Indices

Citus supports adding and removing indices:

```
-- Adding an index

CREATE INDEX clicked_at_idx ON clicks USING BRIN (clicked_at);

-- Removing an index

DROP INDEX clicked_at_idx;
```

Adding an index takes a write lock, which can be undesirable in a multi-tenant "system-of-record." To minimize application downtime, create the index concurrently instead. This method requires more total work than a standard index build and takes significantly longer to complete. However, since it allows normal operations to continue while the index is built, this method is useful for adding new indexes in a production environment.

```
-- Adding an index without locking table writes

CREATE INDEX CONCURRENTLY clicked_at_idx ON clicks USING BRIN (clicked_at);
```

12.4.4 Manual Modification

Currently other DDL commands are not auto-propagated, however you can propagate the changes manually using this general four-step outline:

1. Begin a transaction and take an ACCESS EXCLUSIVE lock on coordinator node against the table in question.
2. In a separate connection, connect to each worker node and apply the operation to all shards.
3. Disable DDL propagation on the coordinator and run the DDL command there.
4. Commit the transaction (which will release the lock).

Contact us for guidance about the process, we have internal tools which can make it easier.

CHAPTER 13

Ingesting, Modifying Data (DML)

The following code snippets use the Github events example, see *Creating and Modifying Distributed Tables (DDL)*.

13.1 Inserting Data

13.1.1 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.

13.1.2 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.

First download our example github_events 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
```

Then, you can copy the data using psql:

```
\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.

13.1.3 Distributed Aggregations

Applications like event data pipelines and real-time dashboards require sub-second queries on large volumes of data. One way to make these queries fast is by calculating and saving aggregates ahead of time. This is called "rolling up" the data and it avoids the cost of processing raw data at run-time. As an extra benefit, rolling up timeseries data into hourly or daily statistics can also save space. Old data may be deleted when its full details are no longer needed and aggregates suffice.

For example, here is a distributed table for tracking page views by url:

```
CREATE TABLE page_views (
  site_id int,
  url text,
  host_ip inet,
  view_time timestamp default now(),

  PRIMARY KEY (site_id, url)
);

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

Once the table is populated with data, we can run an aggregate query to count page views per URL per day, restricting to a given site and year.

```
-- how many views per url per day on site 5?
SELECT view_time::date AS day, site_id, url, count(*) AS view_count
  FROM page_views
  WHERE site_id = 5 AND
    view_time >= date '2016-01-01' AND view_time < date '2017-01-01'
  GROUP BY view_time::date, site_id, url;
```

The setup described above works, but has two drawbacks. First, when you repeatedly execute the aggregate query, it must go over each related row and recompute the results for the entire data set. If you're using this query to render a dashboard, it's faster to save the aggregated results in a daily page views table and query that table. Second, storage costs will grow proportionally with data volumes and the length of queryable history. In practice, you may want to keep raw events for a short time period and look at historical graphs over a longer time window.

To receive those benefits, we can create a `daily_page_views` table to store the daily statistics.

```
CREATE TABLE daily_page_views (
  site_id int,
  day date,
  url text,
```

```
  view_count bigint,
  PRIMARY KEY (site_id, day, url)
);

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

In this example, we distributed both `page_views` and `daily_page_views` on the `site_id` column. This ensures that data corresponding to a particular site will be *co-located* on the same node. Keeping the two tables' rows together on each node minimizes network traffic between nodes and enables highly parallel execution.

Once we create this new distributed table, we can then run `INSERT INTO ... SELECT` to roll up raw page views into the aggregated table. In the following, we aggregate page views each day. Citus users often wait for a certain time period after the end of day to run a query like this, to accommodate late arriving data.

```
-- roll up yesterday's data
INSERT INTO daily_page_views (day, site_id, url, view_count)
  SELECT view_time::date AS day, site_id, url, count(*) AS view_count
  FROM page_views
  WHERE view_time >= date '2017-01-01' AND view_time < date '2017-01-02'
  GROUP BY view_time::date, site_id, url;

-- now the results are available right out of the table
SELECT day, site_id, url, view_count
  FROM daily_page_views
  WHERE site_id = 5 AND
    day >= date '2016-01-01' AND day < date '2017-01-01';
```

It's worth noting that for `INSERT INTO ... SELECT` to work on distributed tables, Citus requires the source and destination table to be co-located. In summary:

- The tables queried and inserted are distributed by analogous columns
- The select query includes the distribution column
- The insert statement includes the distribution column

The rollup query above aggregates data from the previous day and inserts it into `daily_page_views`. Running the query once each day means that no rollup tables rows need to be updated, because the new day's data does not affect previous rows.

The situation changes when dealing with late arriving data, or running the rollup query more than once per day. If any new rows match days already in the rollup table, the matching counts should increase. PostgreSQL can handle this situation with "ON CONFLICT," which is its technique for doing upserts. Here is an example.

```
-- roll up from a given date onward,
-- updating daily page views when necessary
INSERT INTO daily_page_views (day, site_id, url, view_count)
  SELECT view_time::date AS day, site_id, url, count(*) AS view_count
  FROM page_views
  WHERE view_time >= date '2017-01-01'
  GROUP BY view_time::date, site_id, url
  ON CONFLICT (day, url, site_id) DO UPDATE SET
    view_count = daily_page_views.view_count + EXCLUDED.view_count;
```

13.2 Single-Shard Updates and Deletion

You can 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.

13.3 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 *Citus Utility Function Reference* section of our documentation.

13.4 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 14

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 coordinator 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.

14.1 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 coordinator then combines the results from the workers to produce the final output.

14.1.1 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 coordinator.

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 coordinator 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.

14.1.2 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 coordinator and the workers.

14.2 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 coordinator. 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;
```

14.3 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.

14.3.1 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.

14.3.2 Co-located 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 coordinator 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 coordinator.

Note: In order to benefit most from co-located 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.

14.3.3 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, co-located 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.

14.4 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 15

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 coordinator 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 16

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.

16.1 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:

```
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)
```

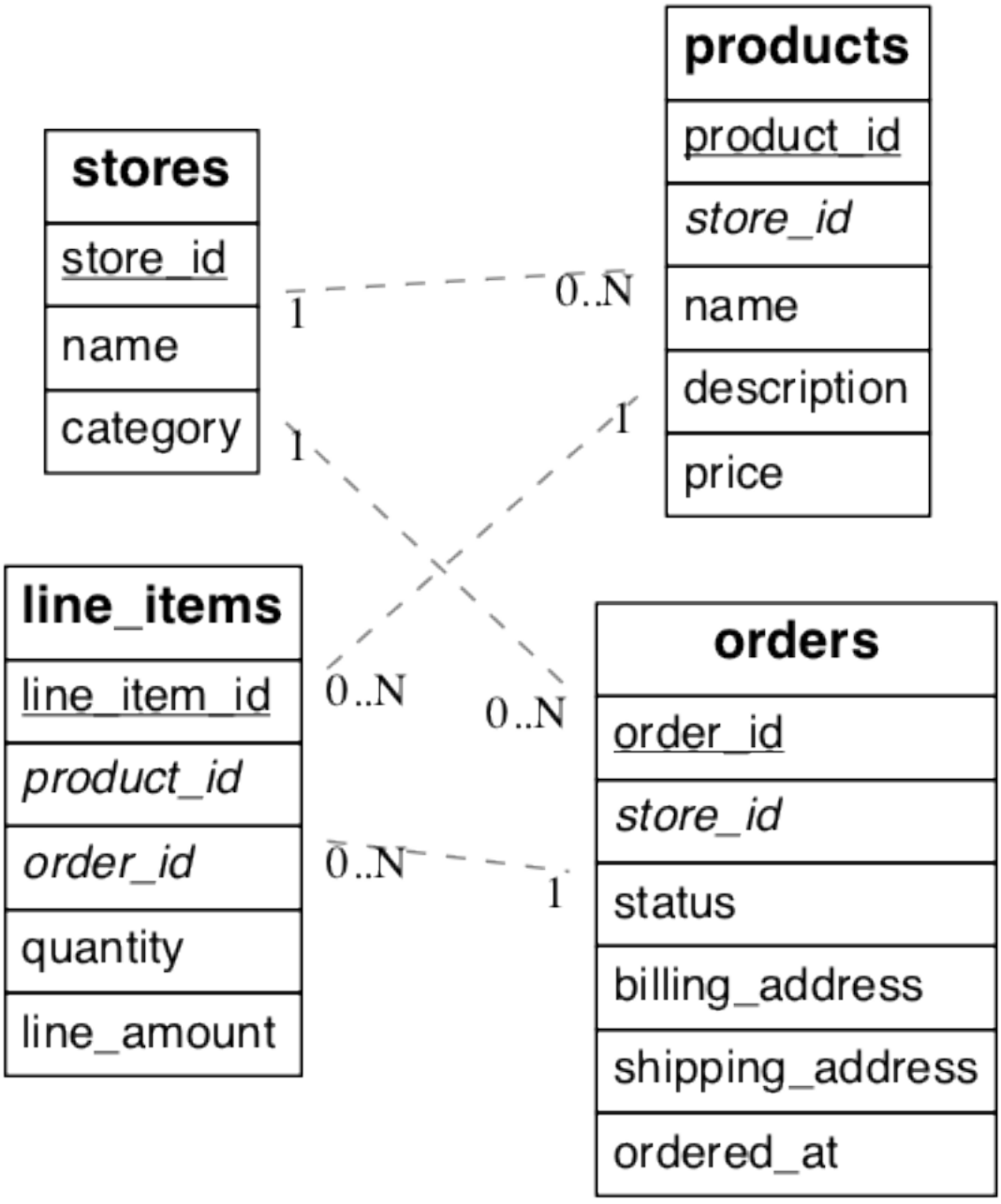


Fig. 16.1: (Underlined items are primary keys, italicized items are foreign keys.)

```
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.

16.2 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.

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)*.

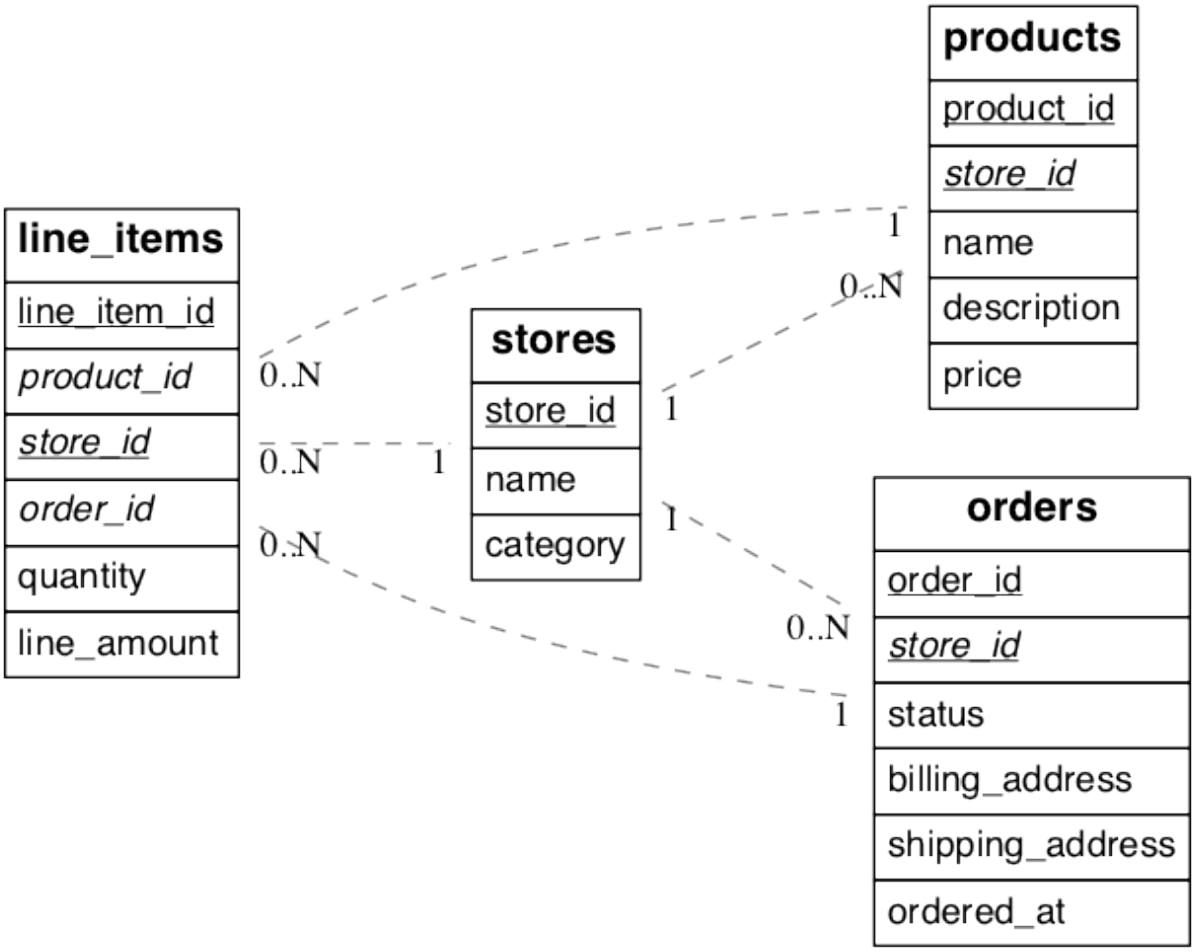


Fig. 16.2: (Underlined items are primary keys, italicized items are foreign keys.)

16.3 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'
```

16.3.1 Validating Query Migration

With large and complex application code-bases, certain queries generated by the application can often be overlooked, and thus won't have a tenant_id filter on them. Citus' parallel executor will still execute these queries successfully, and so, during testing, these queries remain hidden since the application still works fine. However, if a query doesn't contain the tenant_id filter, Citus' executor will hit every shard in parallel, but only one will return any data. This consumes resources needlessly, and may exhibit itself as a problem only when one moves to a higher-throughput production environment.

To prevent encoutering such issues only after launching in production, one can set a config value to log queries which hit more than one shard. In a properly configured and migrated multi-tenant application, each query should only hit one shard at a time.

During testing, one can configure the following:

```
SET citus.multi_task_query_log_level = 'error';
```

Citus will then error out if it encounters queries which are going to hit more than one shard. Erroring out during testing allows the application developer to find and migrate such queries.

During a production launch, one can configure the same setting to warn, instead of error out:

```
SET citus.multi_task_query_log_level = 'warning';
```

The *configuration parameter section* has more info on supported values for this setting.

16.4 App Migration

16.4.1 Ruby on Rails

Above, we discussed the framework-agnostic database changes required for using Citus in the multi-tenant use case. This section investigates specifically how to migrate multi-tenant Rails applications to a Citus storage backend. We'll use the activerecord-multi-tenant Ruby gem for easier scale-out.

This Ruby gem has evolved from our experience working with customers scaling out their multi-tenant apps. It patches some restrictions that ActiveRecord and Rails currently have when it comes to automatic query building. It is based on the excellent acts_as_tenant library, and extends it for the particular use-case of a distributed multi-tenant database like Citus.

Preparing to scale-out a multi-tenant application

Initially you'll often start out with all tenants placed on a single database node, and using a framework like Ruby on Rails and ActiveRecord to load the data for a given tenant when you serve a web request that returns the tenant's data.

ActiveRecord makes a few assumptions about the data storage that limit your scale-out options. In particular, ActiveRecord introduces a pattern where you normalize data and split it into many distinct models each identified by a single `id` column, with multiple `belongs_to` relationships that tie objects back to a tenant or customer:

```
# typical pattern with multiple belongs_to relationships

class Customer < ActiveRecord::Base
  has_many :sites
end
class Site < ActiveRecord::Base
  belongs_to :customer
  has_many :page_views
end
class PageView < ActiveRecord::Base
  belongs_to :site
end
```

The tricky thing with this pattern is that in order to find all page views for a customer, you'll have to query for all of a customer's sites first. This becomes a problem once you start sharding data, and in particular when you run UPDATE or DELETE queries on nested models like page views in this example.

There are a few steps you can take today, to make scaling out easier in the future:

1. Introduce a column for the tenant_id on every record that belongs to a tenant

In order to scale out a multi-tenant model, it's essential you can locate all records that belong to a tenant quickly. The easiest way to achieve this is to simply add a `tenant_id` column (or "customer_id" column, etc) on every object that belongs to a tenant, and backfilling your existing data to have this column set correctly.

When you move to a distributed multi-tenant database like Citus in the future, this will be a required step - but if you've done this before, you can simply COPY over your data, without doing any additional data modification.

2. Use UNIQUE constraints which include the tenant_id

Unique and foreign-key constraints on values other than the tenant_id will present a problem in any distributed system, since it's difficult to make sure that no two nodes accept the same unique value. Enforcing the constraint would require expensive scans of the data across all nodes.

To solve this problem, for the models which are logically related to a store (the tenant for our app), you should add store_id to the constraints, effectively scoping objects unique inside a given store. This helps add the concept of tenancy to your models, thereby making the multi-tenant system more robust.

For example, Rails creates a primary key by default, that only includes the `id` of the record:

```
Indexes:
    "page_views_pkey" PRIMARY KEY, btree (id)
```

You should modify that primary key to also include the tenant_id:

```
ALTER TABLE page_views DROP CONSTRAINT page_views_pkey;
ALTER TABLE page_views ADD PRIMARY KEY(id, customer_id);
```

An exception to this rule might be an email or username column on a users table (unless you give each tenant their own login page), which is why, once you scale out, we typically recommend these to be split out from your distributed tables and placed as a local table on the Citus coordinator node.

3. Include the tenant_id in all queries, even when you can locate an object using its own object_id

The easiest way to run a typical SQL query in a distributed system without restrictions is to always access data that lives on a single node, determined by the tenant you are accessing.

For this reason, once you use a distributed system like Citus, we recommend you always specify both the tenant_id and an object's own ID for queries, so the coordinator can locate your data quickly, and can route the query to a single shard - instead of going to each shard in the system individually and asking the shard whether it knows the given object_id.

Updating the Rails Application

You can get started by including `gem 'activerecord-multi-tenant'` into your Gemfile, running `bundle install`, and then annotating your ActiveRecord models like this:

```
class PageView < ActiveRecord::Base
  multi_tenant :customer
  # ...
end
```

In this case `customer` is the tenant model, and your `page_views` table needs to have a `customer_id` column that references the customer the page view belongs to.

The activerecord-multi-tenant Gem aims to make it easier to implement the above data changes in a typical Rails application.

As mentioned in the beginning, by adding `multi_tenant :customer` annotations to your models, the library automatically takes care of including the tenant_id with all queries.

In order for that to work, you'll always need to specify which tenant you are accessing, either by specifying it on a per-request basis:

```
class ApplicationController < ActionController::Base
  # Opt-into the "set_current_tenant" controller helpers by specifying this:
  set_current_tenant_through_filter

  before_filter :set_customer_as_tenant

  def set_customer_as_tenant
    customer = Customer.find(session[:current_customer_id])
    set_current_tenant(customer) # Set the tenant
  end
end
```

Or by wrapping your code in a block, e.g. for background and maintenance tasks:

```
customer = Customer.find(session[:current_customer_id])
# ...
MultiTenant.with(customer) do
  site = Site.find(params[:site_id])

  # Modifications automatically include tenant_id
  site.update! last_accessed_at: Time.now

  # Queries also include tenant_id automatically
  site.page_views.count
end
```

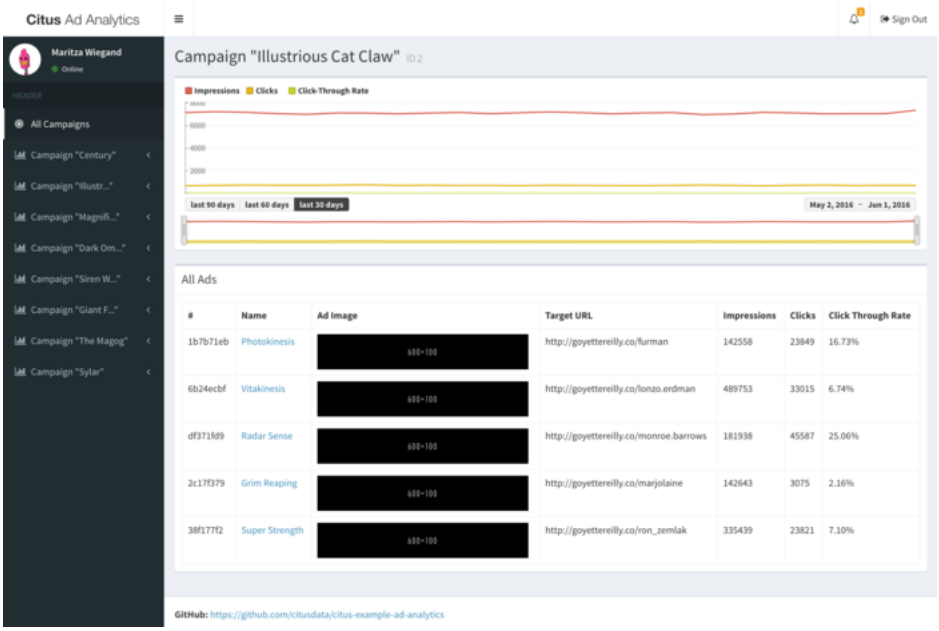
Once you are ready to use a distributed multi-tenant database like Citus, all you need is a few adjustments to your migrations, and you're good to go:

```
class InitialTables < ActiveRecord::Migration
  def up
    create_table :page_views, partition_key: :customer_id do |t|
      t.references :customer, null: false
      t.references :site, null: false

      t.text :url, null: false
      ...
      t.timestamps null: false
    end
    create_distributed_table :page_views, :account_id
  end

  def down
    drop_table :page_views
  end
end
```

Note the `partition_key: :customer_id`, something that's added to Rails' `create_table` by our library, which ensures that the primary key includes the tenant_id column, as well as `create_distributed_table` which enables Citus to scale out the data to multiple nodes.

Example Application

If you are interested in a more complete example, check out our reference app that showcases a simplified sample SaaS application for ad analytics.



As you can see in the screenshot, most data is associated to the currently logged in customer - even though this is complex analytical data, all data is accessed in the context of a single customer or tenant.

16.4.2 Django

At the start of this section we discussed the framework-agnostic database changes required for using Citus in the multi-tenant use case. This section investigates specifically how to migrate multi-tenant Django applications to a Citus storage backend.

Preparing to scale-out a multi-tenant application

Initially you'll often start with all tenants placed on a single database node, and using a framework like Django to load the data for a given tenant when you serve a web request that returns the tenant's data.

Django's typical conventions make a few assumptions about the data storage that limit scale-out options. In particular, the ORM introduces a pattern where you normalize data and split it into many distinct models each identified by a single `id` column (usually added implicitly by the ORM). For instance, consider this simplified model:

```
from django.utils import timezone
from django.db import models

class Store(models.Model):
    name = models.CharField(max_length=255)
    url = models.URLField()

class Product(models.Model):
    name = models.CharField(max_length=255)
    description = models.TextField()
    price = models.DecimalField(max_digits=6, decimal_places=2),
```

```
    quantity = models.IntegerField()
    store = models.ForeignKey(Store)

class Purchase(models.Model):
    ordered_at = models.DateTimeField(default=timezone.now)
    billing_address = models.TextField()
    shipping_address = models.TextField()

    product = models.ForeignKey(Product)
```

The tricky thing with this pattern is that in order to find all purchases for a store, you'll have to query for all of a store's products first. This becomes a problem once you start sharding data, and in particular when you run UPDATE or DELETE queries on nested models like purchases in this example.

1. Introduce a column for the store_id on every record that belongs to a store

In order to scale out a multi-tenant model, it's essential that you can locate all records that belong to a store quickly. The easiest way to achieve this is to simply add a `store_id` column on every object that belongs to a store, and backfill your existing data to have this column set correctly.

2. Use UNIQUE constraints which include the store_id

Unique and foreign-key constraints on values other than the tenant_id will present a problem in any distributed system, since it's difficult to make sure that no two nodes accept the same unique value. Enforcing the constraint would require expensive scans of the data across all nodes.

To solve this problem, for the models which are logically related to a store (the tenant for our app), you should add store_id to the constraints, effectively scoping objects unique inside a given store. This helps add the concept of tenancy to your models, thereby making the multi-tenant system more robust.

Let's begin by adjusting our model definitions and have Django generate a new migration for the two changes discussed.

```
from django.utils import timezone
from django.db import models

class Store(models.Model):
  name = models.CharField(max_length=255)
  url = models.URLField()

class Product(models.Model):
  name = models.CharField(max_length=255)
  description = models.TextField()
  price = models.DecimalField(max_digits=6, decimal_places=2),
  quantity = models.IntegerField()
  store = models.ForeignKey(Store)

  class Meta(object):                  # added
    unique_together = ["id", "store"]  #

class Purchase(models.Model):
  ordered_at = models.DateTimeField(default=timezone.now)
  billing_address = models.TextField()
  shipping_address = models.TextField()

  product = models.ForeignKey(
    Product,
    db_constraint=False                # added
  )
```

```
  store = models.ForeignKey(Store)      # added

  class Meta(object):                   # added
    unique_together = ["id", "store"]   #
```

Create a migration to reflect the change: `./manage.py makemigrations`.

Next we need some custom migrations to adapt the existing key structure in the database for compatibility with Citus. To keep these migrations separate from the ones for the ordinary application, we'll make a new citus application in the same Django project.

```
# Make a new sub-application in the project
django-admin startapp citus
```

Edit `appname/settings.py` and add `'citus'` to the array `INSTALLED_APPS`.

Next we'll add a custom migration to remove simple primary keys which will become composite: `./manage.py makemigrations citus --empty --name remove_simple_pk`. Edit the result to look like this:

```
from __future__ import unicode_literals
from django.db import migrations

class Migration(migrations.Migration):
  dependencies = [
    ('appname', '<name of latest migration>')
  ]

  operations = [
    # Django considers "id" the primary key of these tables, but
    # the database mustn't, because the primary key will be composite
    migrations.RunSQL(
      "ALTER TABLE mtdjango_product DROP CONSTRAINT mtdjango_product_pkey;",
      "ALTER TABLE mtdjango_product ADD CONSTRAINT mtdjango_product_pkey PRIMARY KEY␣
↪(store_id, id)"
    ),
    migrations.RunSQL(
      "ALTER TABLE mtdjango_purchase DROP CONSTRAINT mtdjango_purchase_pkey;",
      "ALTER TABLE mtdjango_purchase ADD CONSTRAINT mtdjango_purchase_pkey PRIMARY␣
↪KEY (store_id, id)"
    ),
  ]
```

Next, we'll make one to tell Citus to mark tables for distribution. `./manage.py makemigrations citus --empty --name distribute_tables`. Edit the result to look like this:

```
from __future__ import unicode_literals
from django.db import migrations

class Migration(migrations.Migration):
  dependencies = [
    # leave this as it was generated
  ]

  operations = [
    migrations.RunSQL(
      "SELECT create_distributed_table('mtdjango_store','id')"
    ),
    migrations.RunSQL(
```

```
      "SELECT create_distributed_table('mtdjango_product','store_id')"
    ),
    migrations.RunSQL(
      "SELECT create_distributed_table('mtdjango_purchase','store_id')"
    ),
  ]
```

Finally, we'll establish a composite foreign key. `./manage.py makemigrations citus --empty --name composite_fk`.

```
from __future__ import unicode_literals
from django.db import migrations

class Migration(migrations.Migration):
  dependencies = [
    # leave this as it was generated
  ]

  operations = [
    migrations.RunSQL(
      """
          ALTER TABLE mtdjango_purchase
          ADD CONSTRAINT mtdjango_purchase_product_fk
          FOREIGN KEY (store_id, product_id)
          REFERENCES mtdjango_product (store_id, id)
          ON DELETE CASCADE;
      """,
      "ALTER TABLE mtdjango_purchase DROP CONSTRAINT mtdjango_purchase_product_fk"
    ),
  ]
```

Apply the migrations by running `./manage.py migrate`.

At this point the Django application models are ready to work with a Citus backend. You can continue by importing data to the new system and modifying controllers as necessary to deal with the model changes.

Updating the Django Application

To simplify queries in the Django application, Citus has developed a Python library called django-multitenant (still in beta as of this writing). Include `django-multitenant` in the `requirements.txt` package file for your project, and then modify your models.

First, include the library in models.py:

```
from django_multitenant import *
```

Next, change the base class for each model from `models.Model` to `TenantModel`, and add a property specifying the name of the tenant id. For instance, to continue the earlier example:

```
class Store(TenantModel):
  tenant_id = 'id'
  # ...

class Product(TenantModel):
  tenant_id = 'store_id'
  # ...
```

```
class Purchase(TenantModel):
  tenant_id = 'store_id'
  # ...
```

No extra database migration is necessary beyond the steps in the previous section. The library allows application code to easily scope queries to a single tenant. It automatically adds the correct SQL filters to all statements, including fetching objects through relations.

For instance:

```
# set the current tenant to the first store
s = Store.objects.all()[0]
set_current_tenant(s)

# now this count query applies only to Products for that store
Product.objects.count()

# Find purchases for risky products in the current store
Purchase.objects.filter(product__description='Dangerous Toy')
```

In the context of an application controller, the current tenant object can be stored as a SESSION variable when a user logs in, and controller actions can `set_current_tenant` to this value.

CHAPTER 17

Real-Time Analytics Data Model

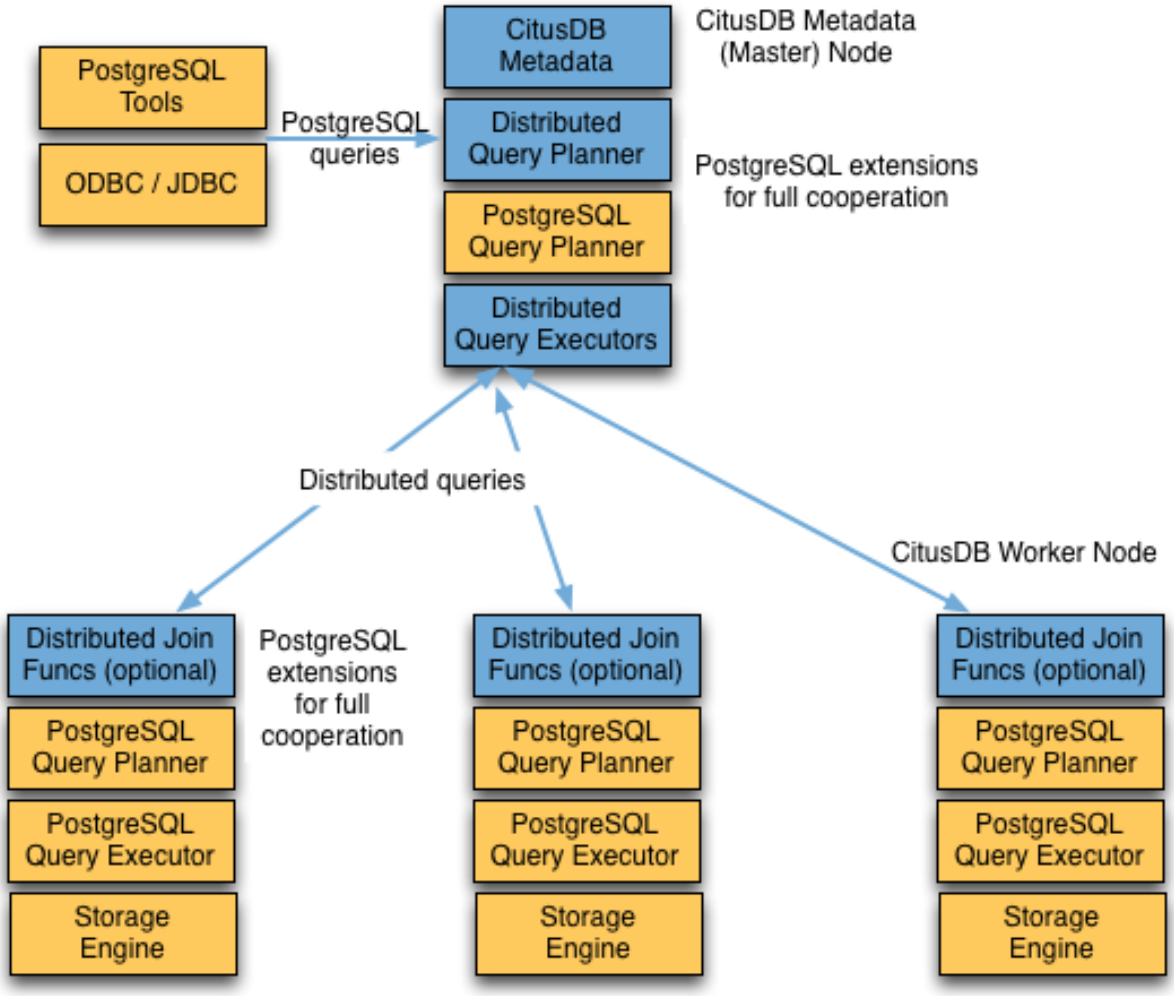
In this model multiple worker nodes calculate aggregate data in parallel for applications such as analytic dashboards. This scenario requires greater interaction between Citus nodes than the multi-tenant case and the transition from a standalone database varies more per application.

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 interactively improve performance. For more migration guidance please contact us.

CHAPTER 18

Citus Query Processing

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



Citus's query processing pipeline involves the two components:

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

We discuss them in greater detail in the subsequent sections.

18.1 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 coordinator query which runs on the coordinator 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.

18.2 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.

18.2.1 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 co-located 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.

18.2.2 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 coordinator 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.

18.3 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 coordinator for final aggregation.

CHAPTER 19

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 application integration, throughput, and latency. In this section, we discuss different approaches to data ingestion, and provide guidelines for expected throughput and latency numbers.

19.1 Real-time Insert and Updates

On the Citus coordinator, you can perform INSERT, INSERT .. ON CONFLICT, UPDATE, and DELETE commands directly on distributed tables. When you issue one of these commands, the changes are immediately visible to the user.

When you run an INSERT (or another ingest command), Citus first finds the right shard placements based on the value in the distribution column. Citus then connects to the worker nodes storing the shard placements, and performs an INSERT on each of them. From the perspective of the user, the INSERT takes several milliseconds to process because of the network latency to worker nodes. The Citus coordinator node however can process concurrent INSERTs to reach high throughputs.

19.1.1 Insert Throughput

To measure data ingest rates with Citus, we use a standard tool called pgbench and provide repeatable benchmarking steps.

We also used these steps to run pgbench across different Citus Cloud formations on AWS and observed the following ingest rates for transactional INSERT statements. For these benchmark results, we used the default configuration for Citus Cloud formations, and set pgbench's concurrent thread count to 64 and client count to 256. We didn't apply any optimizations to improve performance numbers; and you can get higher ingest ratios by tuning your database setup.

Coordinator Node	Worker Nodes	Latency (ms)	Transactions per sec
2 cores - 7.5GB RAM	2 * (1 core - 15GB RAM)	28.5	9,000
4 cores - 15GB RAM	2 * (1 core - 15GB RAM)	15.3	16,600
8 cores - 30GB RAM	2 * (1 core - 15GB RAM)	15.2	16,700
8 cores - 30GB RAM	4 * (1 core - 15GB RAM)	8.6	29,600

We have three observations that follow from these benchmark numbers. First, the top row shows performance numbers for an entry level Citus cluster with one c4.xlarge (two physical cores) as the coordinator and two r4.large (one physical core each) as worker nodes. This basic cluster can deliver 9K INSERTs per second, or 775 million transactional INSERT statements per day.

Second, a more powerful Citus cluster that has about four times the CPU capacity can deliver 30K INSERTs per second, or 2.75 billion INSERT statements per day.

Third, across all data ingest benchmarks, the network latency combined with the number of concurrent connections PostgreSQL can efficiently handle, becomes the performance bottleneck. In a production environment with hundreds of tables and indexes, this bottleneck will likely shift to a different resource.

19.1.2 Update Througput

To measure UPDATE throughputs with Citus, we used the same benchmarking steps and ran pgbench across different Citus Cloud formations on AWS.

Coordinator Node	Worker Nodes	Latency (ms)	Transactions per sec
2 cores - 7.5GB RAM	2 * (1 core - 15GB RAM)	25.0	10,200
4 cores - 15GB RAM	2 * (1 core - 15GB RAM)	19.6	13,000
8 cores - 30GB RAM	2 * (1 core - 15GB RAM)	20.3	12,600
8 cores - 30GB RAM	4 * (1 core - 15GB RAM)	10.7	23,900

These benchmark numbers show that Citus's UPDATE throughput is slightly lower than those of INSERTs. This is because pgbench creates a primary key index for UPDATE statements and an UPDATE incurs more work on the worker nodes. It's also worth noting two additional differences between INSERT and UPDATEs.

First, UPDATE statements cause bloat in the database and VACUUM needs to run regularly to clean up this bloat. In Citus, since VACUUM runs in parallel across worker nodes, your workloads are less likely to be impacted by VACUUM.

Second, these benchmark numbers show UPDATE throughput for standard Citus deployments. If you're on the Citus community edition, using statement-based replication, and you increased the default replication factor to 2, you're going to observe notably lower UPDATE throughputs. For this particular setting, Citus comes with additional configuration (citus.all_modifications_commutative) that may increase UPDATE ratios.

19.1.3 Insert and Update: Throughput Checklist

When you're running the above pgbench benchmarks on a moderately sized Citus cluster, you can generally expect 10K-50K INSERTs per second. This translates to approximately 1 to 4 billion INSERTs per day. If you aren't observing these throughputs numbers, remember the following checklist:

- Check the network latency between your application and your database. High latencies will impact your write throughput.
- Ingest data using concurrent threads. If the roundtrip latency during an INSERT is 4ms, you can process 250 INSERTs/second over one thread. If you run 100 concurrent threads, you will see your write throughput increase with the number of threads.
- Check whether the nodes in your cluster have CPU or disk bottlenecks. Ingested data passes through the coordinator node, so check whether your coordinator is bottlenecked on CPU.
- Avoid closing connections between INSERT statements. This avoids the overhead of connection setup.
- Remember that column size will affect insert speed. Rows with big JSON blobs will take longer than those with small columns like integers.

19.1.4 Insert and Update: Latency

The benefit of running INSERT or UPDATE commands, compared to issuing bulk COPY commands, is that changes are immediately visible to other queries. When you issue an INSERT or UPDATE command, the Citus coordinator node directly routes this command to related worker node(s). The coordinator node also keeps connections to the workers open within the same session, which means subsequent commands will see lower response times.

```
-- Set up a distributed table that keeps account history information
CREATE TABLE pgbench_history (tid int, bid int, aid int, delta int, mtime timestamp);
SELECT create_distributed_table('pgbench_history', 'aid');

-- Enable timing to see reponse times
\timing on

-- First INSERT requires connection set-up, second will be faster
INSERT INTO pgbench_history VALUES (10, 1, 10000, -5000, CURRENT_TIMESTAMP); -- Time:␣
↪10.314 ms
INSERT INTO pgbench_history VALUES (10, 1, 22000, 5000, CURRENT_TIMESTAP); -- Time: 3.
↪132 ms
```

19.2 Bulk Copy (250K - 2M/s)

Distributed tables support COPY from the Citus coordinator for bulk ingestion, which can achieve much higher ingestion rates than INSERT statements.

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

```
COPY pgbench_history 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 pgbench_history FROM 'pgbench_history-2016-03-04.csv' (FORMAT CSV)"
```

A powerful feature of COPY for 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.

From a throughput standpoint, you can expect data ingest ratios of 250K - 2M rows per second when using COPY. To learn more about COPY performance across different scenarios, please refer to the following blog post.

Note: To avoid opening too many connections to worker nodes, we recommend running only two COPY commands on a distributed table at a time. In practice, running more than four 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.

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

Masterless Citus (Citus MX) builds on the Citus extension. It gives you the ability to query and write to distributed tables from any node, which allows you to horizontally scale out your write-throughput using PostgreSQL. It also removes the need to interact with a primary node in a Citus cluster for data ingest or queries.

Citus MX is currently available in private beta on Citus Cloud. For more information see *MX (Beta)*.

CHAPTER 20

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.

20.1 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.

20.2 PostgreSQL tuning

The Citus coordinator 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 coordinator 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 coordinator node combines their results.

Here is an example of explaining the plan for a particular example query.

```
EXPLAIN
 SELECT date_trunc('minute', created_at) AS minute,
        sum((payload->>'distinct_size')::int) AS num_commits
   FROM github_events
  WHERE event_type = 'PushEvent'
  GROUP BY minute
  ORDER BY minute;
```

```
Sort  (cost=0.00..0.00 rows=0 width=0)
  Sort Key: minute
  ->  HashAggregate  (cost=0.00..0.00 rows=0 width=0)
    Group Key: minute
    ->  Custom Scan (Citus Real-Time)  (cost=0.00..0.00 rows=0 width=0)
      Task Count: 32
      Tasks Shown: One of 32
      ->  Task
        Node: host=localhost port=5433 dbname=postgres
        ->  HashAggregate  (cost=93.42..98.36 rows=395 width=16)
          Group Key: date_trunc('minute'::text, created_at)
          ->  Seq Scan on github_events_102042 github_events  (cost=0.00..88.20␣
↪rows=418 width=503)
            Filter: (event_type = 'PushEvent'::text)
(13 rows)
```

This tells you several things. To begin with there are thirty-two shards, and the planner chose the Citus real-time executor to execute this query:

```
->  Custom Scan (Citus Real-Time)  (cost=0.00..0.00 rows=0 width=0)
  Task Count: 32
```

Next it picks one of the workers 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 32
->  Task
  Node: host=localhost port=5433 dbname=postgres
```

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

```
->  HashAggregate  (cost=93.42..98.36 rows=395 width=16)
  Group Key: date_trunc('minute'::text, created_at)
  ->  Seq Scan on github_events_102042 github_events  (cost=0.00..88.20 rows=418␣
↪width=503)
    Filter: (event_type = 'PushEvent'::text)
```

You can now connect to the worker at 'localhost', port '5433' and tune query performance for the shard github_events_102042 using standard PostgreSQL techniques. As you make changes run EXPLAIN again from the coordinator 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 coordinator:

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

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

```
EXPLAIN
 SELECT date_trunc('minute', created_at) AS minute,
        sum((payload->>'distinct_size')::int) AS num_commits
   FROM github_events
  WHERE event_type = 'PushEvent'
  GROUP BY minute
  ORDER BY minute;
```

```
 Sort  (cost=0.00..0.00 rows=0 width=0)
   Sort Key: minute
   ->  HashAggregate  (cost=0.00..0.00 rows=0 width=0)
     Group Key: minute
     ->  Custom Scan (Citus Real-Time)  (cost=0.00..0.00 rows=0 width=0)
       Task Count: 32
       Tasks Shown: All
       ->  Task
         Node: host=localhost port=5433 dbname=postgres
         ->  HashAggregate  (cost=93.42..98.36 rows=395 width=16)
           Group Key: date_trunc('minute'::text, created_at)
           ->  Seq Scan on github_events_102042 github_events  (cost=0.00..88.20␣
↪rows=418 width=503)
             Filter: (event_type = 'PushEvent'::text)
       ->  Task
         Node: host=localhost port=5434 dbname=postgres
         ->  HashAggregate  (cost=103.21..108.57 rows=429 width=16)
```

```
           Group Key: date_trunc('minute'::text, created_at)
           ->  Seq Scan on github_events_102043 github_events  (cost=0.00..97.47␣
↪rows=459 width=492)
             Filter: (event_type = 'PushEvent'::text)
      --
      -- ... repeats for all 32 tasks
      --     alternating between workers one and two
      --     (running in this case locally on ports 5433, 5434)
      --

(199 rows)
```

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.

20.3 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.

20.4 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 coordinator 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 coordinator 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.

20.4.1 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 coordinator 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*.

20.4.2 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 coordinator. 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 coordinator 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.

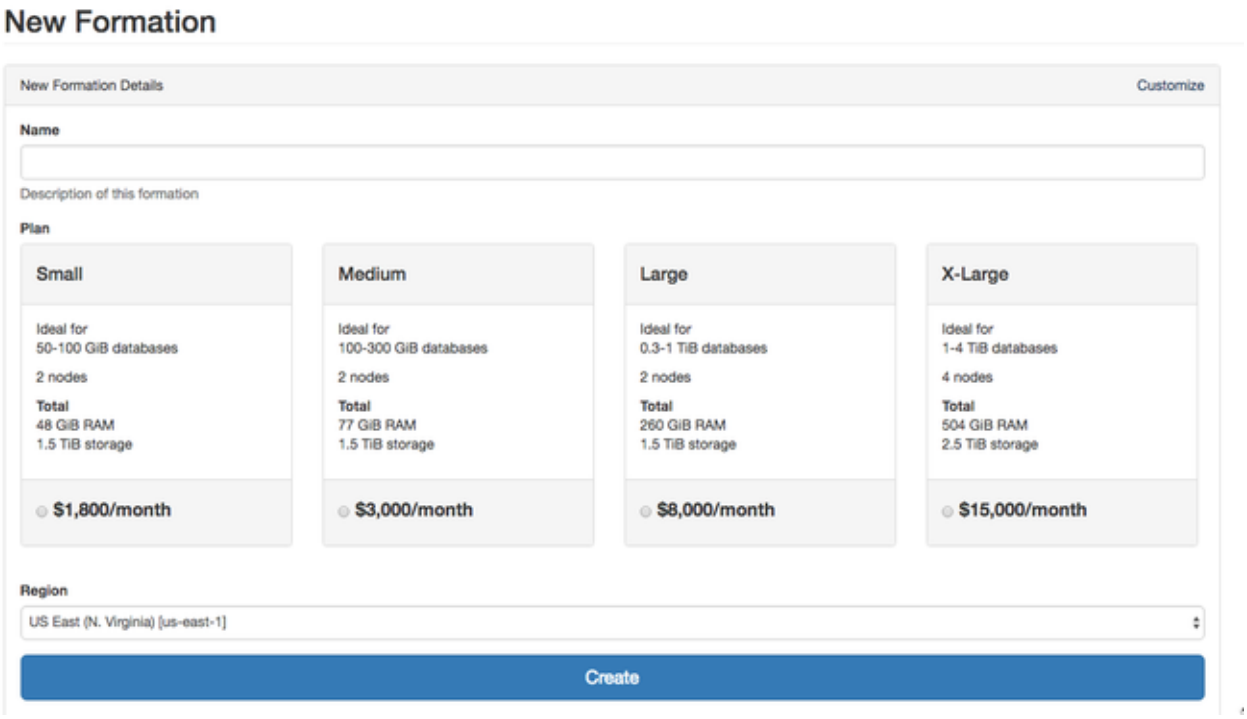
CHAPTER 21

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.

21.1 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.



21.1.1 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

CHAPTER 22

High-Availability

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.

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.

Let's examine these concepts in greater detail.

22.1 Introduction to High Availability and Disaster Recovery

In the real world, insurance is used to manage risk when a natural disaster such as a hurricane or flood strikes. In the database world, there are two critical methods of insurance. High Availability (HA) replicates the latest database version virtually instantly. Disaster Recovery (DR) offers continuous protection by saving every database change, allowing database restoration to any point in time.

In what follows, we'll dig deeper as to what disaster recovery and high availability are, as well as how we've implemented them for Citus Cloud.

22.1.1 What is High Availability and Disaster Recovery?

High availability and disaster recovery are both forms of data backups that are mutually exclusive and inter-related. The difference between them, is that HA has a secondary reader database replica (often referred to as stand-by or follower) ready to take over at any moment, but DR just writes to cold storage (in the case of Amazon that's S3) and has latency in the time for the main database to recover data.

22.1.2 Overview of High Availability

For HA, any data that is written to a primary database called the Writer is instantly replicated onto a secondary database called the Reader in real-time, through a stream called a WAL or Write-Ahead-Log.

To ensure HA is functioning properly, Citus Cloud runs health checks every 30 seconds. If the primary fails and data can't be accessed after six consecutive attempts, a failover is initiated. This means the primary node will be replaced by the standby node and a new standby will be created.

22.1.3 Overview of Disaster Recovery

For DR, read-only data is replayed from colder storage. On AWS this is from S3, and for Postgres this is downloaded in 16 MB pieces. On Citus Cloud this happens via WAL-E, using precisely the same procedure as creating a new standby for HA. WAL-E is an open source tool initially developed by our team, for archiving PostgreSQL WAL (Write Ahead Log) files quickly, continuously and with a low operational burden.

This means we can restore your database by fetching the base backup and replaying all of the WAL files on a fresh install in the event of hardware failure, data corruption or other failure modes

On Citus Cloud prior to kicking off the DR recovery process, the AWS EC2 instance is automatically restarted. This process usually takes 7±2 minutes. If it restarts without any issues, the setup remains the same. If the EC2 instance fails to restart, a new instance is created. This happens at a rate of at least 30MB/second, so 512GB of data would take around 5 hours.

22.1.4 How High Availability and Disaster Recovery fit together

While some may be familiar many are not acutely aware of the relationship between HA and DR.

Although it's technically possible to have one without the other, they are unified in that the HA streaming replication and DR archiving transmit the same bytes.

For HA the primary "writer" database replicates data through streaming replication to the secondary "reader" database. For DR, the same data is read from S3. In both cases, the "reader" database downloads the WAL and applies it incrementally.

Since DR and HA gets regularly used for upgrades and side-grades, the DR system is maintained with care. We ourselves rely on it for releasing new production features.

22.1.5 Disaster Recovery takes a little extra work but gives greater reliability

You might think that if HA provides virtually instant backup reliability, so 'Why bother with DR?' There are some compelling reasons to use DR in conjunction with HA including cost, reliability and control.

From a cost efficiency perspective, since HA based on EBS and EC2 is a mirrored database, you have to pay for every layer of redundancy added. However, DR archives in S3 are often 10-30% of the monthly cost of a database instance. And with Citus Cloud the S3 cost is already covered for you in the standard price of your cluster.

From reliability perspective, S3 has proven to be up to a thousand times more reliable than EBS and EC2, though a more reasonable range is ten to a hundred times. S3 archives also have the advantage of immediate restoration, even while teams try to figure out what's going on. Conversely, sometimes EBS volume availability can be down for hours with uncertainty it will completely restore.

From a control perspective, using DR means a standby database can be created while reducing the impact on the primary database. It also has the capability of being able to recover a database from a previous version.

22.1.6 Trade-offs between latency and reliability

There is a long history of trade-offs between latency and reliability, dating back to when the gold standard for backups were on spools of tape.

Writing data to and then reading data from S3 offers latencies that are 100 to 1,000 times longer than streaming bytes between two computers as seen in streaming replication. However, S3's availability and durability are both in excess of ten times better than an EBS volume.

On the other hand, the throughput of S3 is excellent: with parallelism, and without downstream bottlenecks, one can achieve multi-gigabit throughput in backup and WAL reading and writing.

22.1.7 How High Availability and Disaster Recovery is used for crash recovery

When many customers entrust your company with their data, it is your duty to keep it safe under all circumstances. So when the most severe database crashes strike, you need to be ready to recover.

Our team is battle-hardened from years of experience as the original Heroku Postgres team, managing over 1.5 million databases. Running at that scale with constant risks of failure, meant that it was important to automate recovery processes.

Such crashes are a nightmare. But crash recovery is a way to make sure you sleep well at night by making sure none of your or your customers data is lost and your downtime is minimal.

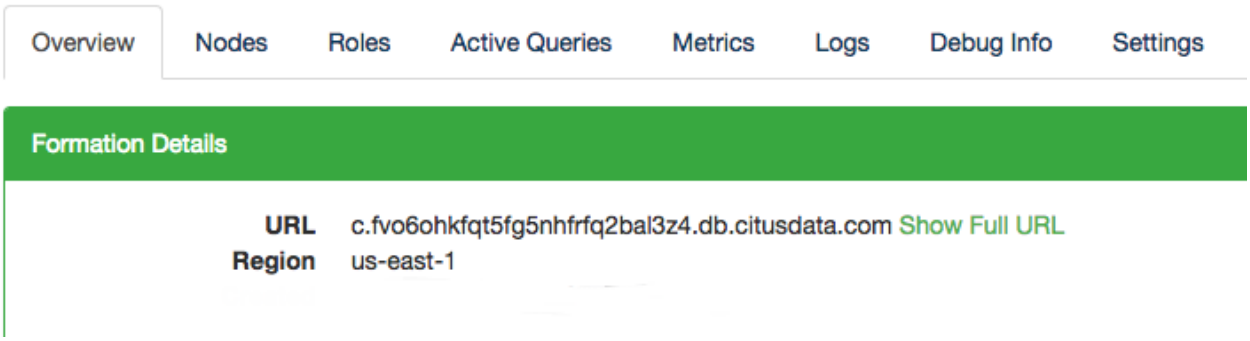
CHAPTER 23

Connection and Security

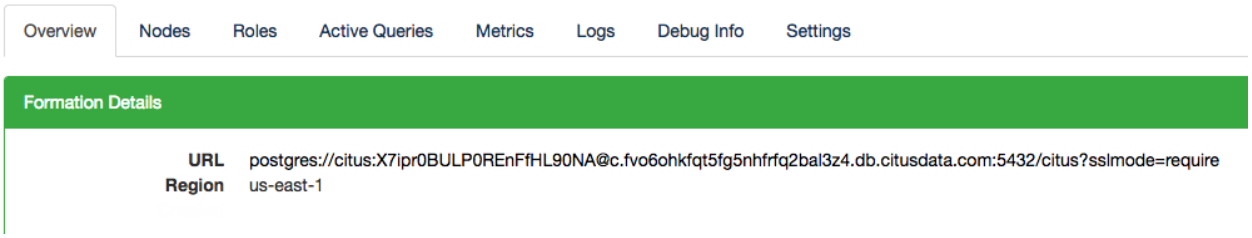
Applications connect to Citus the same way they would PostgreSQL, using a connection URI. This is a string which includes network and authentication information, and has the form:

```
postgresql://[user[:password]@][host][:port][/dbname][?param1=value1&...]
```

The connection string for each Cloud Formation is provided on the Overview tab in Citus Console.



By default the URL displays only the hostname of the connection, but the full URL is available by clicking the "Show Full URL" link.



For security Citus Cloud accepts only SSL connections, which is why the URL contains the `?sslmode=require` parameter. To avoid a man-in-the-middle attack, you can also verify that the server certificate is correct. Download the official Citus Cloud certificate and refer to it in connection string parameters:

```
?sslrootcert=/location/to/citus.crt&sslmode=verify-full
```

The string may need to be quoted in your shell to preserve the ampersand.

Note: Database clients must support SSL to connect to Citus Cloud. In particular `psql` needs to be compiled `--with-openssl` if building PostgreSQL from source.

A coordinator node on Citus Cloud has a hard limit of three hundred simultaneous active connections to limit memory consumption. If more connections are required, change the port in the connection URL from 5432 to 6432. This will connect to PgBouncer rather than directly to the coordinator, allowing up to roughly two thousand simultaneous connections. The coordinator can still only process three hundred at a time, but more can connect and PgBouncer will queue them.

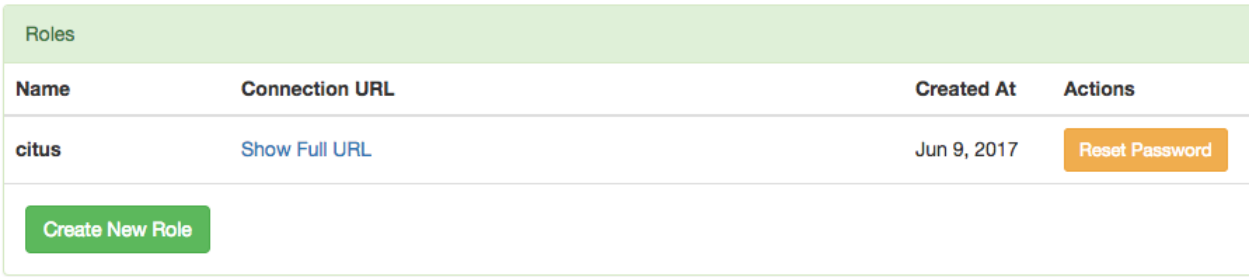
To measure the number of active connections at a given time, run:

```
SELECT COUNT(*)
  FROM pg_stat_activity
 WHERE state <> 'idle';
```

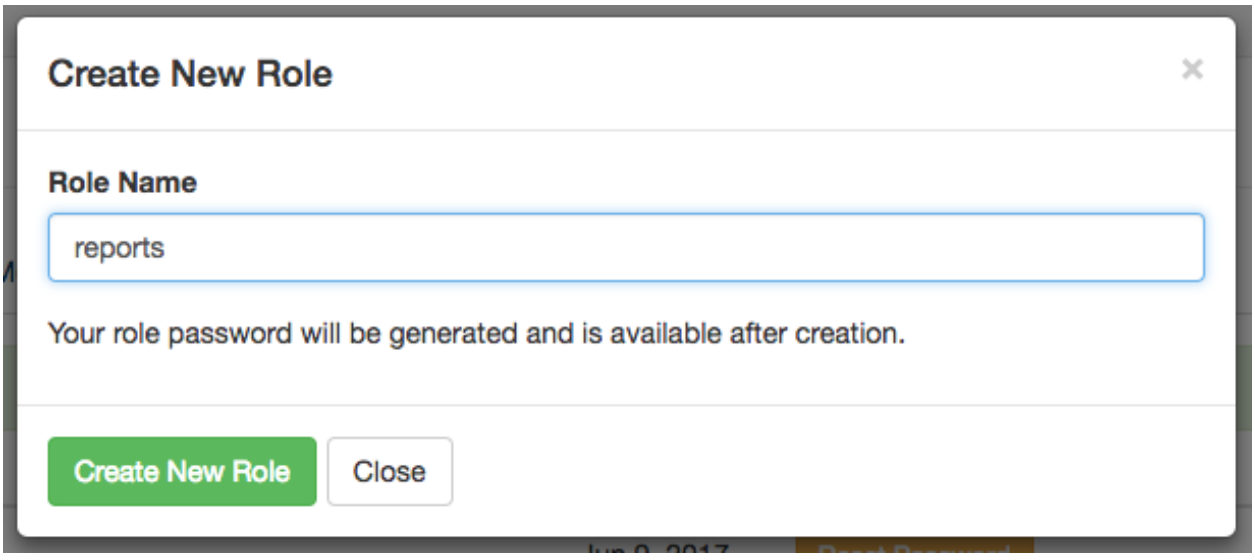
23.1 Users and Permissions

As we saw above, every new Citus Cloud formation includes a user account called `citus`. This account is great for creating tables and other DDL, but it has too much power for certain applications.

We'll want to create new roles for specialized purposes. For instance, a user with read-only access is perfect for a web/reporting tool. The Cloud console allows us to create a new user, and will set up a new password automatically. Go to the "Roles" tab and click "Create New Role."



It pops up a dialog where we will fill in the role name, which we can call `reports`.



After creating the role on a fresh formation, there will be three roles:

```
-[ RECORD 1 ]-----------------------------------------
| Role name  | citus                                  |
```

```
| Attributes |                                                                  |
| Member of  | {reports}                                                        |
-[ RECORD 2 ]----------------------------------------
| Role name  | postgres                                                         |
| Attributes | Superuser, Create role, Create DB, Replication, Bypass RLS |
| Member of  | {}                                                               |
-[ RECORD 3 ]----------------------------------------
| Role name  | reports                                                          |
| Attributes |                                                                  |
| Member of  | {}                                                               |
-------------------------------------------------
```

The new `reports` role starts with no privileges, except "usage" on the public schema, meaning the ability to get a list of the tables etc inside. We have to specifically grant the role extra permissions to database objects. For instance, to allow read-only access to `mytable`, connect to Citus as the `citus` user with the connection string provided in the Cloud console and issue this command:

```
-- run as the citus user

GRANT SELECT ON mytable TO reports;
```

You can confirm the privileges by consulting the information schema:

```
SELECT grantee, privilege_type
  FROM information_schema.role_table_grants
 WHERE table_name = 'mytable';
```

```
-----------------
| grantee | privilege_type |
-----------------
| citus   | INSERT         |
| citus   | SELECT         |
| citus   | UPDATE         |
| citus   | DELETE         |
| citus   | TRUNCATE       |
| citus   | REFERENCES     |
| citus   | TRIGGER        |
| reports | SELECT         |
-----------------
```

The PostgreSQL documentation has more detailed information about types of privileges you can GRANT on database objects.

23.1.1 Granting Privileges in Bulk

Citus propagates single-table GRANT statements through the entire cluster, making them apply on all worker nodes. However GRANTs that are system-wide (e.g. for all tables in a schema) need to be applied individually to every data node using a Citus helper function.

```
-- applies to the coordinator node
GRANT SELECT ON ALL TABLES IN SCHEMA public TO reports;

-- make it apply to workers as well
SELECT run_command_on_workers(
  'GRANT SELECT ON ALL TABLES IN SCHEMA public TO reports;'
);
```

23.2 Cloud Security

23.2.1 Encryption

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

23.2.2 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.

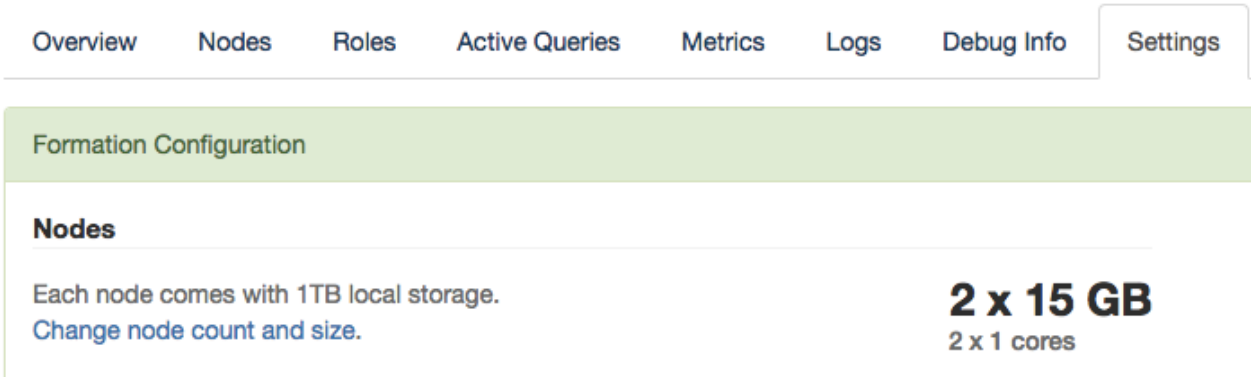
CHAPTER 24

Scaling

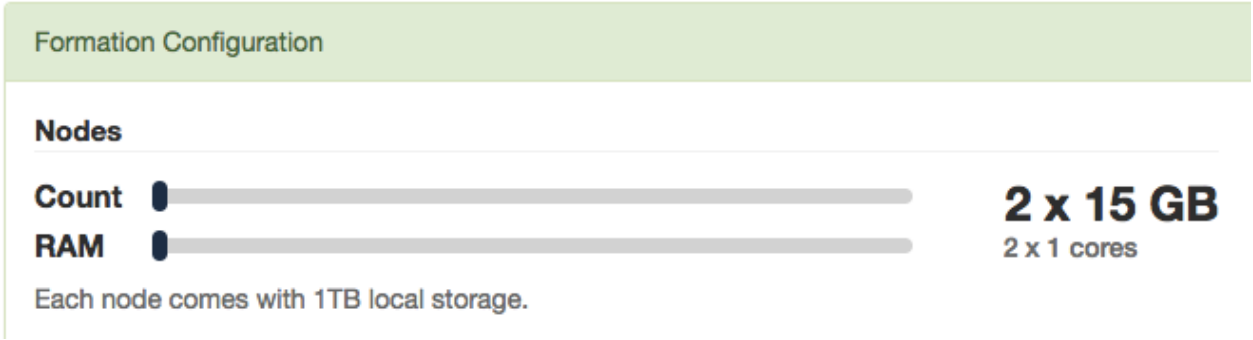
Citus Cloud provides self-service scaling to deal with increased load. The web interface makes it easy to either add new worker nodes or increase existing nodes' memory and CPU capacity.

For most cases either approach to scaling is fine and will improve performance. However there are times you may want to choose one over the other. If the cluster is reaching its disk limit then adding more nodes is the best choice. Alternately, if there is still a lot of headroom on disk but performance is suffering, then scaling node RAM and processing power is the way to go.

Both adjustments are available in the formation configuration panel of the settings tab:

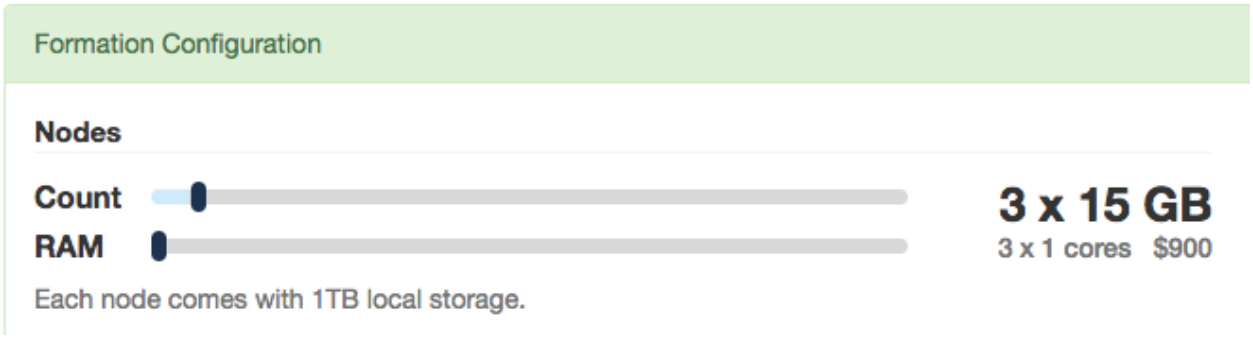


Clicking the node change link provides two sliders:

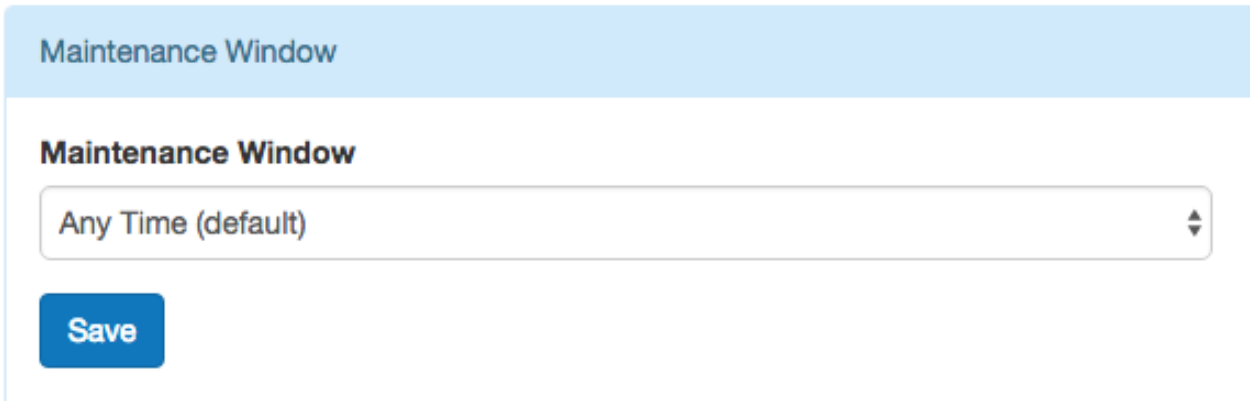


The first slider, **count**, scales out the cluster by adding new nodes. The second, **RAM**, scales it up by changing the instance size (RAM and CPU cores) of existing nodes.

For example, just drag the slider for node count:

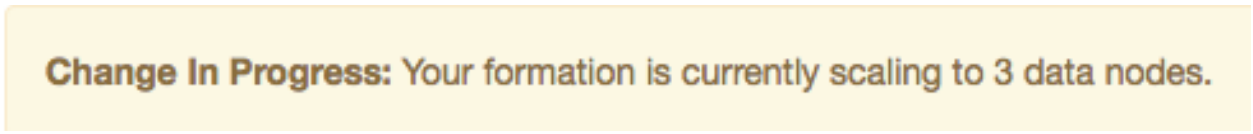


After you adjust the sliders and accept the changes, Citus Cloud begins applying the changes. Increasing the number of nodes will begin immediately, whereas increasing node instance size will wait for a time in the user-specified maintenance window.

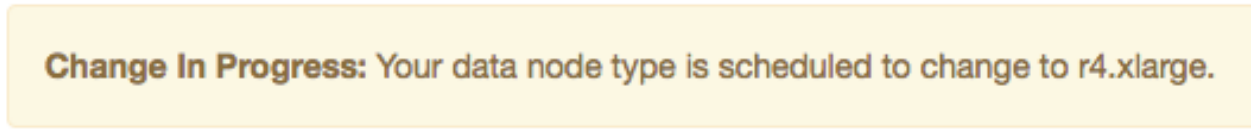


Citus Cloud will display a popup message in the console while scaling actions have begun or are scheduled. The message will disappear when the action completes.

For instance, when adding nodes:



Or when waiting for node resize to begin in the maintenance window:



24.1 Scaling Up (increasing node size)

Resizing node size works by creating a PostgreSQL follower for each node, where the followers are provisioned with the desired amount of RAM and CPU cores. It takes an average of forty minutes per hundred gigabytes of data for the primary nodes' data to be fully synchronized on the followers. After the synchronization is complete, Citus Cloud does a quick switchover from the existing primary nodes to their followers which takes about two minutes. The creation and switchover process uses the same well-tested replication mechanism that powers Cloud's *High-Availability* feature. During the switchover period clients may experience errors and have to retry queries, especially cross-tenant queries hitting multiple nodes.

24.2 Scaling Out (adding new nodes)

Node addition completes in five to ten minutes, which is faster than node resizing because the new nodes are created without data. To take advantage of the new nodes you still must adjust manually rebalance the shards, meaning move some shards from existing nodes to the new ones.

Citus does not automatically rebalance on node creation because shard rebalancing takes locks on rows whose shards are being moved, degrading write performance for other database clients. The slowdown isn't terribly severe because Citus moves data a shard (or a group of colocated shards) at a time while inserts to other shards can continue normally. However, for maximum control, the choice of when to run the shard rebalancer is left to the database administrator.

To start the shard rebalance, connect to the cluster coordinator node with psql and run:

```
SELECT rebalance_table_shards('distributed_table_name');
```

Citus will output the progress as it moves each shard.

Note: The `rebalance_table_shards` function rebalances all tables in the *colocation group* of the table named in its argument. Thus you do not have to call it for every single table, just call it on a representative table from each colocation group.

CHAPTER 25

Logging

25.1 What Is Logged

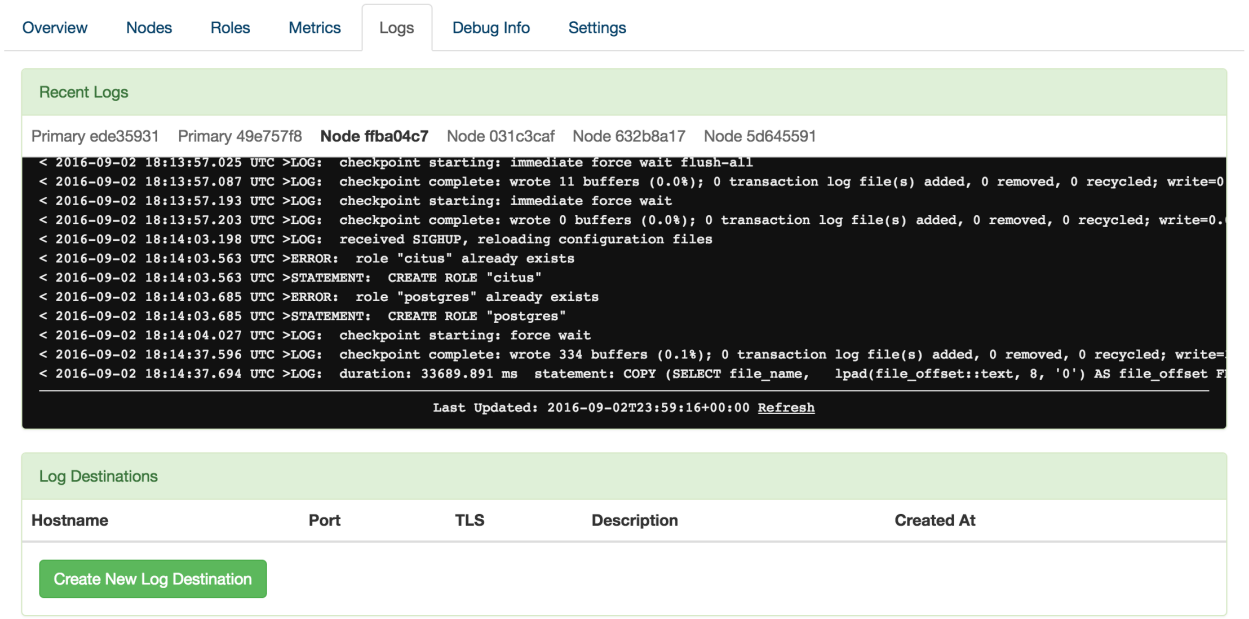
By default, Citus Cloud logs all errors and other useful information that happen on any of the Citus instances and makes it available to you.

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

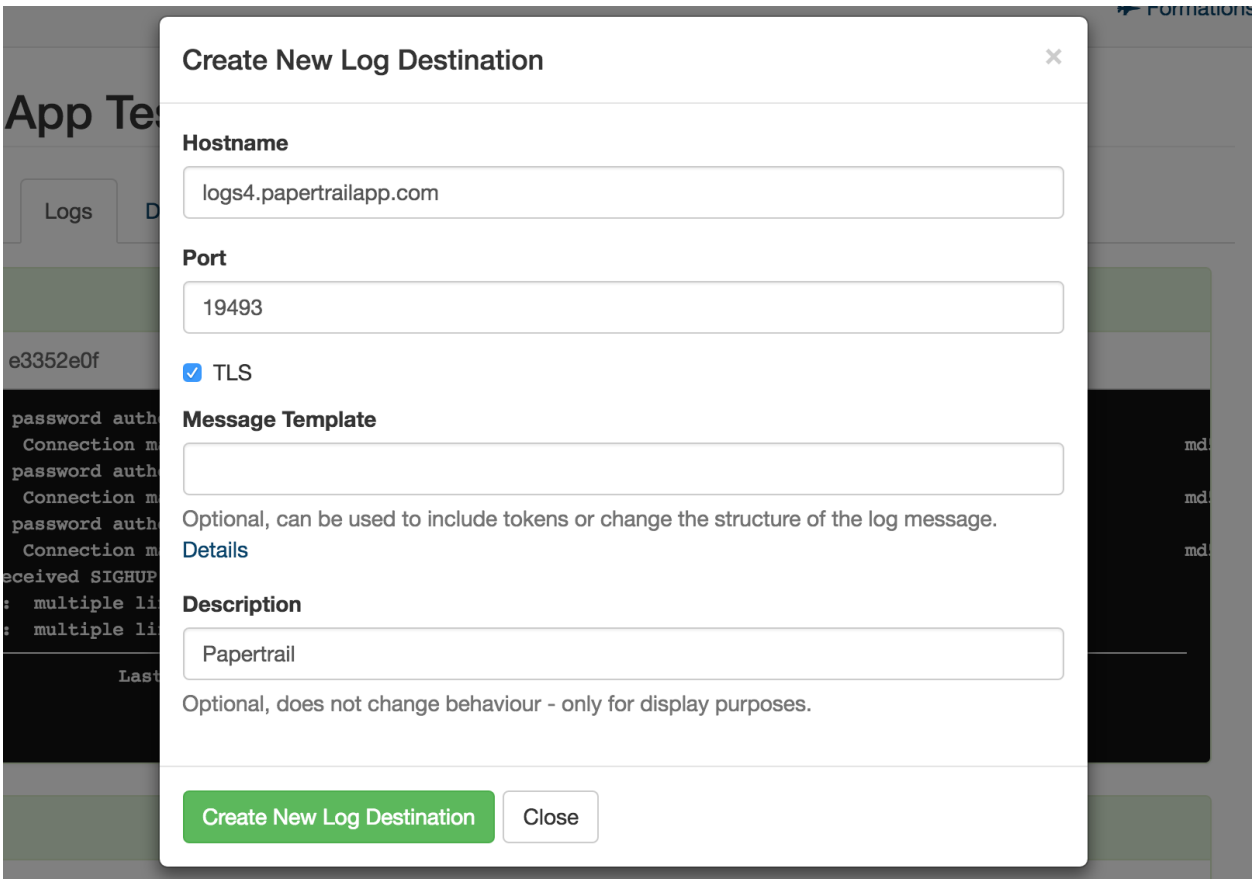
25.2 Recent Logs

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.



25.3 External Log Destinations

For anything more than a quick look at your logs, we recommend setting up an external provider. Through this method the logs will transmit to a dedicated logging service and you can analyze and retain them according to your own preferences. To use an external provider, create a new logging destination in the Citus Cloud console. For instance, here is the new destination dialog filled in with Papertrail settings:



Note that after creation, it might take up to five minutes for logging preferences to be applied. You'll then see logs show up in your chosen provider's dashboard.

The settings to use differ per provider. In the following tables we list settings verified to work for a number of popular providers.

25.3.1 Verified Provider Settings

Replace `<token>` with the custom token listed in the provider's web site. This token tells them the logs are yours.

Papertrail

Hostname	logs4.papertrailapp.com
Port	19493
TLS	Yes
Protocol	IETF Syslog
Message Template	

Loggly

Hostname	logs-01.loggly.com
Port	514
TLS	No
Protocol	BSD Syslog over TCP
Message Template	<${PRI}>1 ${ISODATE} ${HOST} ${PROGRAM} ${PID} ${MSGID} [<token>@41058 tag=\"CITUS\"] $MSG\n

Sumologic

Hostname	syslog.collection.us2.sumologic.com
Port	6514
TLS	Yes
Protocol	IETF Syslog
Message Template	<${PRI}>1 ${ISODATE} ${HOST} ${PROGRAM} ${PID} ${MSGID} [<token>@41123] $MSG\n

Logentries

Hostname	data.logentries.com
Port	80
TLS	No
Protocol	IETF Syslog
Message Template	<token> $ISODATE $HOST $MSG\n

Other

We support other providers that can receive syslog via the BSD or IETF protocols. Internally Citus Cloud uses syslog-ng, so check your providers configuration documentation for syslog-ng settings.

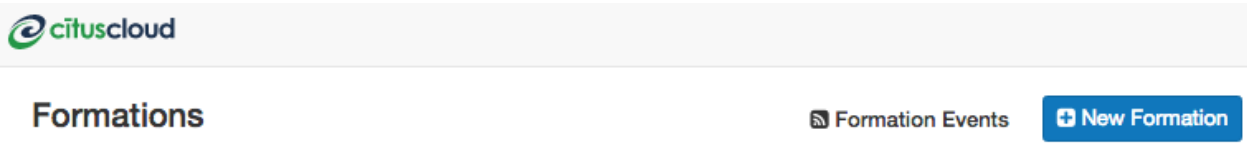
Please reach out if you encounter any issues.

CHAPTER 26

Monitoring

To monitor events in the life of a formation with outside tools via a standard format, we offer RSS feeds per organization. You can use a feed reader or RSS Slack integration (e.g. on an `#ops` channel) to keep up to date.

On the upper right of the "Formations" list in the Cloud console, follow the "Formation Events" link to the RSS feed.



The feed includes entries for three types of events, each with the following details:

Server Unavailable

This is a notification of connectivity problems such as hardware failure.

- Formation name
- Formation url
- Server

Failover Scheduled

For planned upgrades, or when operating a formation without high availability that experiences a failure, this event will appear to indicate a future planned failover event.

- Formation name
- Formation url
- Leader
- Failover at

For planned failovers, "failover at" will usually match your maintenance window. Note that the failover might happen at this point or shortly thereafter, once a follower is available and has caught up to the primary database.

Failover

Failovers happen to address hardware failure, as mentioned, and also for other reasons such as performing system software upgrades, or transferring data to a server with better hardware.

- Formation name
- Formation url
- Leader

- Situation
- Follower

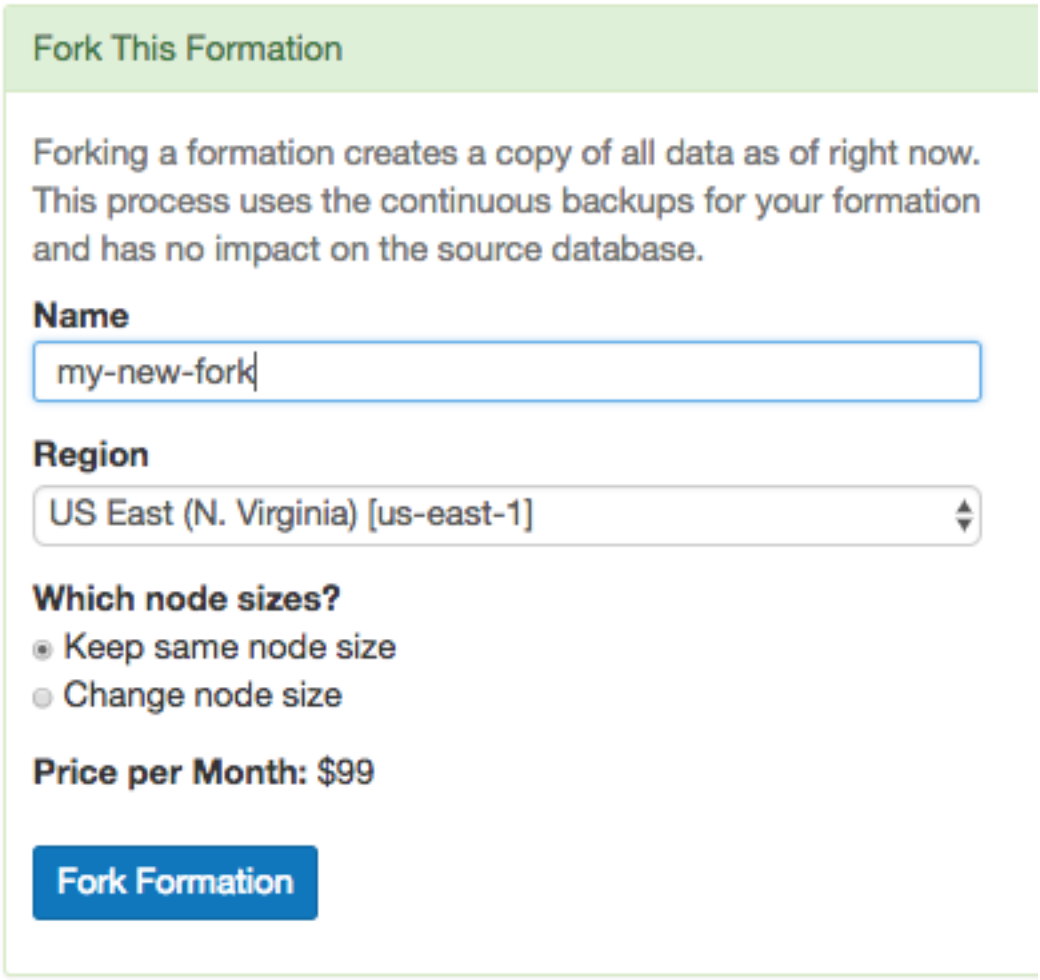
CHAPTER 27

Forking

Forking a Citus Cloud formation makes a copy of the cluster's data at an instant in time and produces a new formation in precisely that state. It allows you to change, query, or generally experiment with production data in a separate protected environment. Fork creation runs quickly, and you can do it as often as you want without causing any extra load on the original cluster. This is because forking doesn't query the cluster, rather it taps into the write-ahead logs for each database in the formation.

27.1 How to Fork a Formation

Citus Cloud makes forking easy. The control panel for each formation has a "Fork" tab. Go there and enter the name, region, and node sizing information for the destination cluster.



Shortly after you click "Fork Formation," the new formation will appear in the Cloud console. It runs on separate hardware and your database can connect to it in the *usual way*.

27.2 When is it Useful

A fork is a great place to do experiments. Do you think that denormalizing a table might speed things up? What about creating a roll-up table for a dashboard? How can you persuade your manager that you need more RAM in the coordinator node rather than in the workers? You could prove yourself if only you could try your idea on the production data.

In such cases, what you need is a temporary copy of the production database. But it would take forever to copy, say, 500GB of data to a new formation. Not to mention that making the copy would slow down the production database. Copying the database in the old fashioned way is not a good idea.

However a Citus fork is different. Forking fetches write-ahead log data from S3 and has zero effect on the production load. You can apply your experiments to the fork and destroy it when you're done.

Another use of forking is to enable complex analytical queries. Sometimes data analysts want to have access to live production data for complex queries that would take hours. What's more, they sometimes want to bend the data: denormalize tables, create aggregations, create an extra index or even pull all the data onto one machine.

Obviously, it is not a good idea to let anyone play with a production database. You can instead create a fork and give

it to whomever wants to play with real data. You can re-create a fork every month to update your analytics results.

27.3 How it Works Internally

Citus is an extension of PostgreSQL and can thus leverage all the features of the underlying database. Forking is actually a special form of point-in-time recovery (PITR) into a new database where the recovery time is the time the fork is initiated. The two features relevant for PITR are:

- Base Backups
- Write-Ahead Log (WAL) Shipping

About every twenty-four hours Cloud calls pg_basebackup to make a new base backup, which is just an archive of the PostgreSQL data directory. Cloud also continuously ships the database write-ahead logs (WAL) to Amazon S3 with WAL-E.

Base backups and WAL archives are all that is needed to restore the database to some specific point in time. To do so, we start an instance of the database on the base backup taken most recently before the desired restoration point. The new PostgreSQL instances, upon entering recovery mode, will start playing WAL segments up to the target point. After the recovery instances reach the specified target, they will be available for use as a regular database.

A Citus formation is a group of PostgreSQL instances that work together. To restore the formation we simply need to restore all nodes in the cluster to the same point in time. We perform that operation on each node and, once done, we update metadata in the coordinator node to tell it that this new cluster has branched off from your original.

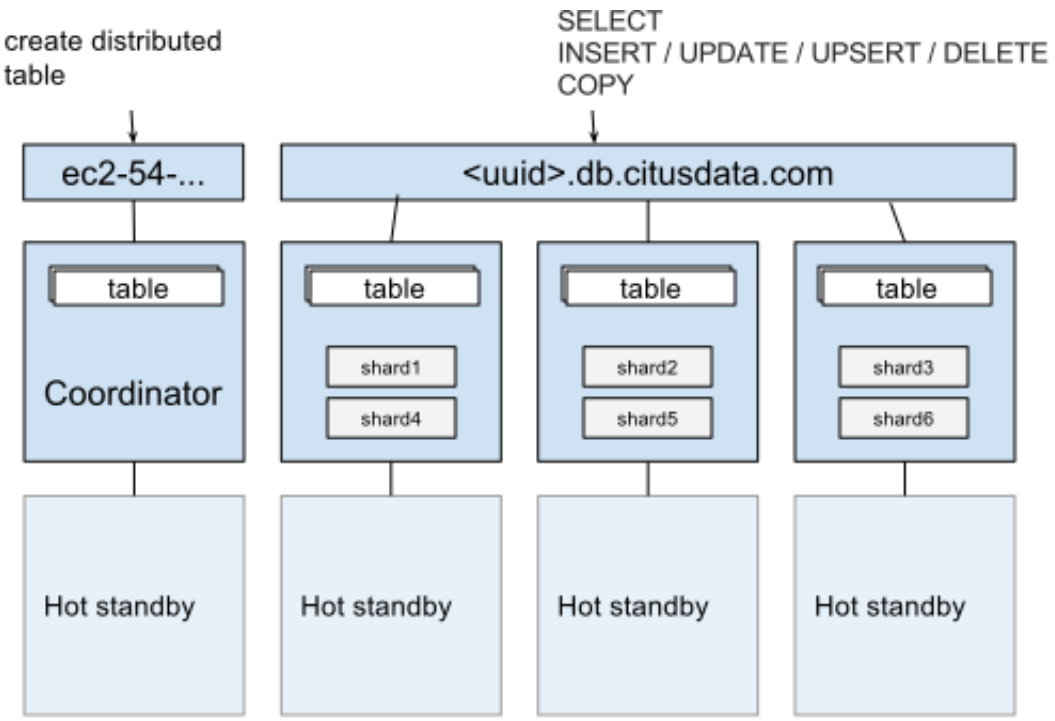
CHAPTER 28

MX (Beta)

Citus MX is a new version of Citus that adds the ability to use hash-distributed tables from any node in a Citus cluster, which allows you to scale out your query throughput by opening many connections across all the nodes. This is particularly useful for performing small reads and writes at a very high rate in a way that scales horizontally. Citus MX is currently available in private beta on Citus Cloud.

28.1 Architecture

In the Citus MX architecture, all nodes are PostgreSQL servers running the Citus extension. One node is acting as coordinator and the others as data nodes, each node also has a hot standby that automatically takes over in case of failure. The coordinator is the authoritative source of metadata for the cluster and data nodes store the actual data in shards. Distributed tables can only be created, altered, or dropped via the coordinator, but can be queried from any node. When making changes to a table (e.g. adding a column), the metadata for the distributed tables is propagated to the workers using PostgreSQL's built-in 2PC mechanism and distributed locks. This ensures that the metadata is always consistent such that every node can run distributed queries in a reliable way.

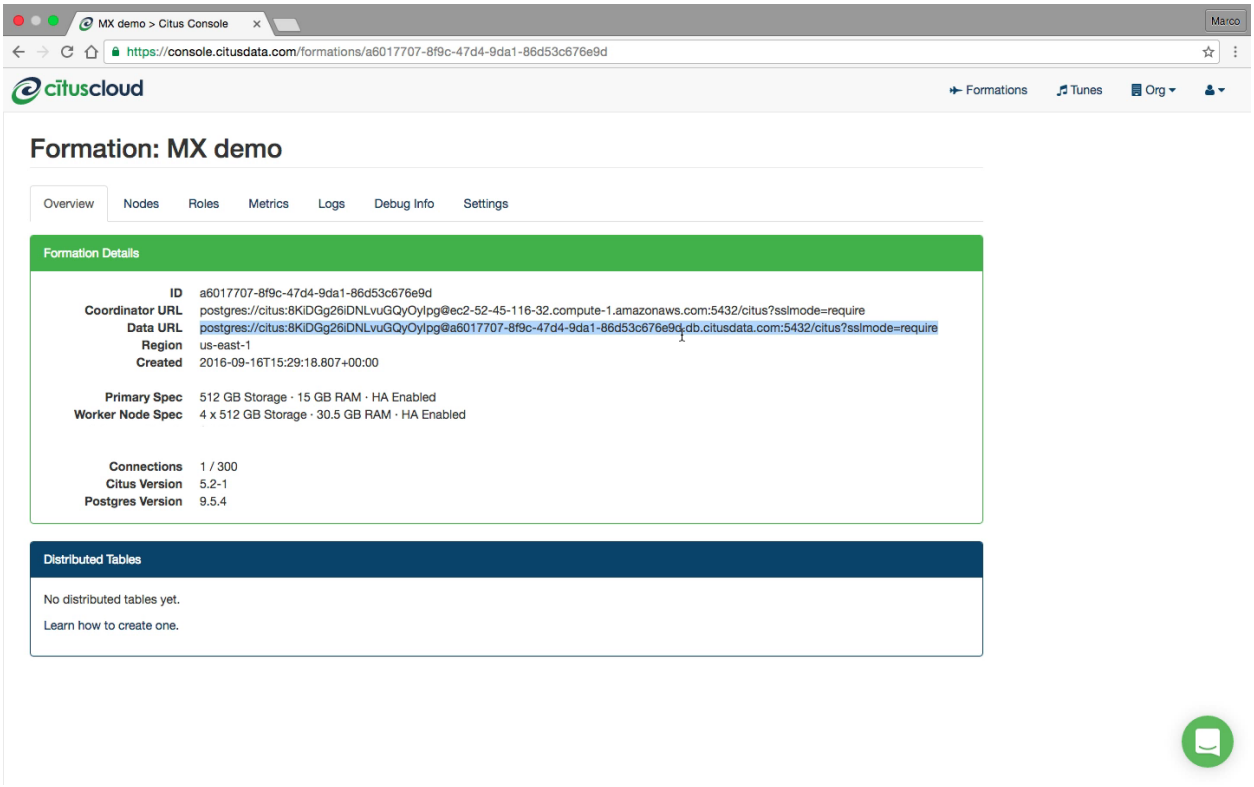


Citus MX Architecture

Citus MX uses PostgreSQL's own streaming replication, which allows a higher rate of writes on the shards as well as removing the need to perform all writes through a single leader node to ensure linearizability and consistency. From the Citus perspective, there is now only a single replica of each shard and it does not have to keep multiple replicas in sync, since streaming replication handles that. In the background, we monitor every node and automatically fail over to a hot standby in case of a failure.

28.2 Data Access

In Citus MX you can access your database in one of two ways: Either through the coordinator which allows you to create or change distributed tables, or via the data URL, which routes you to one of the data nodes on which you can perform regular queries on the distributed tables. These are also the nodes that hold the shards, the regular PostgreSQL tables in which the data is stored.



Supported operations on the coordinator are: Create/drop distributed table, shard rebalancer, DDL, DML, SELECT, COPY.

Supported operations on the data URL are: DML, SELECT, COPY.

If you connect to the data URL using *psql* and run *\d*, then you will see all the distributed tables and some of the shards. Importantly, distributed tables are the same from all nodes, so it does not matter to which node you are routed when using the data URL when querying distributed tables. When performing a query on a distributed table, the right shard is determined based on the filter conditions and the query is forwarded to the node that stores the shard. If a query spans all the shards, it is parallelised across all the nodes.

For some advanced usages, you may want to perform operations on shards directly (e.g. add triggers). In that case, you can connect to each individual worker node rather than using the data URL. You can find the worker nodes hostnames by running *SELECT * FROM master_get_active_worker_nodes()* from any node and use the same credentials as the data URL.

A typical way of using MX is to manually set up tables via the coordinator and then making all queries via the data URL. An alternative way is to use the coordinator as your main application back-end, and use the data URL for data ingestion. The latter is useful if you also need to use some local PostgreSQL tables. We find both approaches to be viable in a production setting.

28.3 Scaling Out a Raw Events Table

A common source of high volume writes are various types of sensors reporting back measurements. This can include software-based sensors such as network telemetry, mobile devices, or hardware sensors in Internet-of-things applications. Below we give an example of how to set-up a write-scalable events table in Citus MX.

Since Citus is an PostgreSQL extension, you can use all the latest PostgreSQL 9.5 features, including JSONB and BRIN indexes. When sensors can generate different types of events, JSONB can be useful to represent different data structures. Brin indexes allow you to index data that is ordered by time in a compact way.

To create a distributed events table with a JSONB column and a BRIN index, we can run the following commands:

```
$ psql postgres://citus:pw@coordinator-host:5432/citus?sslmode=require
```

```
CREATE TABLE events (
  device_id bigint not null,
  event_id uuid not null default uuid_generate_v4(),
  event_time timestamp not null default now(),
  event_type int not null default 0,
  payload jsonb,
  primary key (device_id, event_id)
);
CREATE INDEX event_time_idx ON events USING BRIN (event_time);
SELECT create_distributed_table('events', 'device_id');
```

Once the distributed table is created, we can immediately start using it via the data URL and writes done on one node will immediately be visible from all the other nodes in a consistent way.

```
$ psql postgres://citus:pw@data-url:5432/citus?sslmode=require
```

```
citus=> INSERT INTO events (device_id, payload)
VALUES (12, '{"temp":"12.8","unit":"C"}');

Time: 3.674 ms
```

SELECT queries that filter by a specific device_id are particularly fast, because Citus can route them directly to a single worker and execute them on a single shard.

```
$ psql postgres://citus:pw@data-url:5432/citus?sslmode=require
```

```
citus=> SELECT event_id, event_time, payload FROM events WHERE device_id = 12 ORDER␣
↪BY event_time DESC LIMIT 10;

Time: 4.212 ms
```

As with regular Citus, you can also run analytical queries which are parallelized across the cluster:

```
citus=>
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_time >= now() - interval '10 minutes'
) ev
GROUP BY minute ORDER BY minute ASC;


Time: 554.565
```

The ability to perform analytical SQL queries combined with high volume data ingestion uniquely positions Citus for real-time analytics applications.

An important aspect to consider is that horizontally scaling out your processing power ensures that indexes don't necessarily become an ingestion bottleneck as your application grows. PostgreSQL has very powerful indexing capabilities and with the ability to scale out you can almost always get the desired read- and write-performance.

28.4 Limitations Compared to Citus

All Citus 6.2 features are supported in Citus MX with the following exceptions:

Append-distributed tables currently cannot be made available from workers. They can still be used in the traditional way, with queries going through the coordinator. However, append-distributed tables already allowed you to *Bulk Copy (250K - 2M/s)*.

When performing writes on a hash-distributed table with a bigserial column via the data URL, sequence numbers are no longer monotonic, but instead have the form <16-bit unique node ID><48-bit local sequence number> to ensure uniqueness. The coordinator node always has node ID 0, meaning it will generate sequence numbers as normal. Serial types smaller than bigserial cannot be used in distributed tables.

CHAPTER 29

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.

29.1 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.

29.2 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 30

Multi-tenant Applications

Contents

Estimated read time: 30 minutes

If you're building a Software-as-a-service (SaaS) application, you probably already have the notion of tenancy built into your data model. Typically, most information relates to tenants / customers / accounts and the database tables capture this natural relation.

For SaaS applications, each tenant's data can be stored together in a single database instance and kept isolated and invisible from other tenants. This is efficient in three ways. First application improvements apply to all clients. Second, sharing a database between tenants uses hardware efficiently. Last, it is much simpler to manage a single database for all tenants than a different database server for each tenant.

However, a single relational database instance has traditionally had trouble scaling to the volume of data needed for a large multi-tenant application. Developers were forced to relinquish the benefits of the relational model when data exceeded the capacity of a single database node.

Citus allows users to write multi-tenant applications as if they are connecting to a single PostgreSQL database, when in fact the database is a horizontally scalable cluster of machines. Client code requires minimal modifications and can continue to use full SQL capabilities.

This guide takes a sample multi-tenant application and describes how to model it for scalability with Citus. Along the

way we examine typical challenges for multi-tenant applications like isolating tenants from noisy neighbors, scaling hardware to accommodate more data, and storing data that differs across tenants. PostgreSQL and Citus provide all the tools needed to handle these challenges, so let's get building.

30.1 Let's Make an App – Ad Analytics

We'll build the back-end for an application that tracks online advertising performance and provides an analytics dashboard on top. It's a natural fit for a multi-tenant application because user requests for data concern one (their own) company at a time. Code for the full example application is available on Github.

Let's start by considering a simplified schema for this application. The application must keep track of multiple companies, each of which runs advertising campaigns. Campaigns have many ads, and each ad has associated records of its clicks and impressions.

Here is the example schema. We'll make some minor changes later, which allow us to effectively distribute and isolate the data in a distributed environment.

```
CREATE TABLE companies (
  id bigserial PRIMARY KEY,
  name text NOT NULL,
  image_url text,
  created_at timestamp without time zone NOT NULL,
  updated_at timestamp without time zone NOT NULL
);

CREATE TABLE campaigns (
  id bigserial PRIMARY KEY,
  company_id bigint REFERENCES companies (id),
  name text NOT NULL,
  cost_model text NOT NULL,
  state text NOT NULL,
  monthly_budget bigint,
  blacklisted_site_urls text[],
  created_at timestamp without time zone NOT NULL,
  updated_at timestamp without time zone NOT NULL
);

CREATE TABLE ads (
  id bigserial PRIMARY KEY,
  campaign_id bigint REFERENCES campaigns (id),
  name text NOT NULL,
  image_url text,
  target_url text,
  impressions_count bigint DEFAULT 0,
  clicks_count bigint DEFAULT 0,
  created_at timestamp without time zone NOT NULL,
  updated_at timestamp without time zone NOT NULL
);

CREATE TABLE clicks (
  id bigserial PRIMARY KEY,
  ad_id bigint REFERENCES ads (id),
  clicked_at timestamp without time zone NOT NULL,
  site_url text NOT NULL,
  cost_per_click_usd numeric(20,10),
  user_ip inet NOT NULL,
  user_data jsonb NOT NULL
```

```
);

CREATE TABLE impressions (
  id bigserial PRIMARY KEY,
  ad_id bigint REFERENCES ads (id),
  seen_at timestamp without time zone NOT NULL,
  site_url text NOT NULL,
  cost_per_impression_usd numeric(20,10),
  user_ip inet NOT NULL,
  user_data jsonb NOT NULL
);
```

There are modifications we can make to the schema which will give it a performance boost in a distributed environment like Citus. To see how, we must become familiar with how Citus distributes data and executes queries.

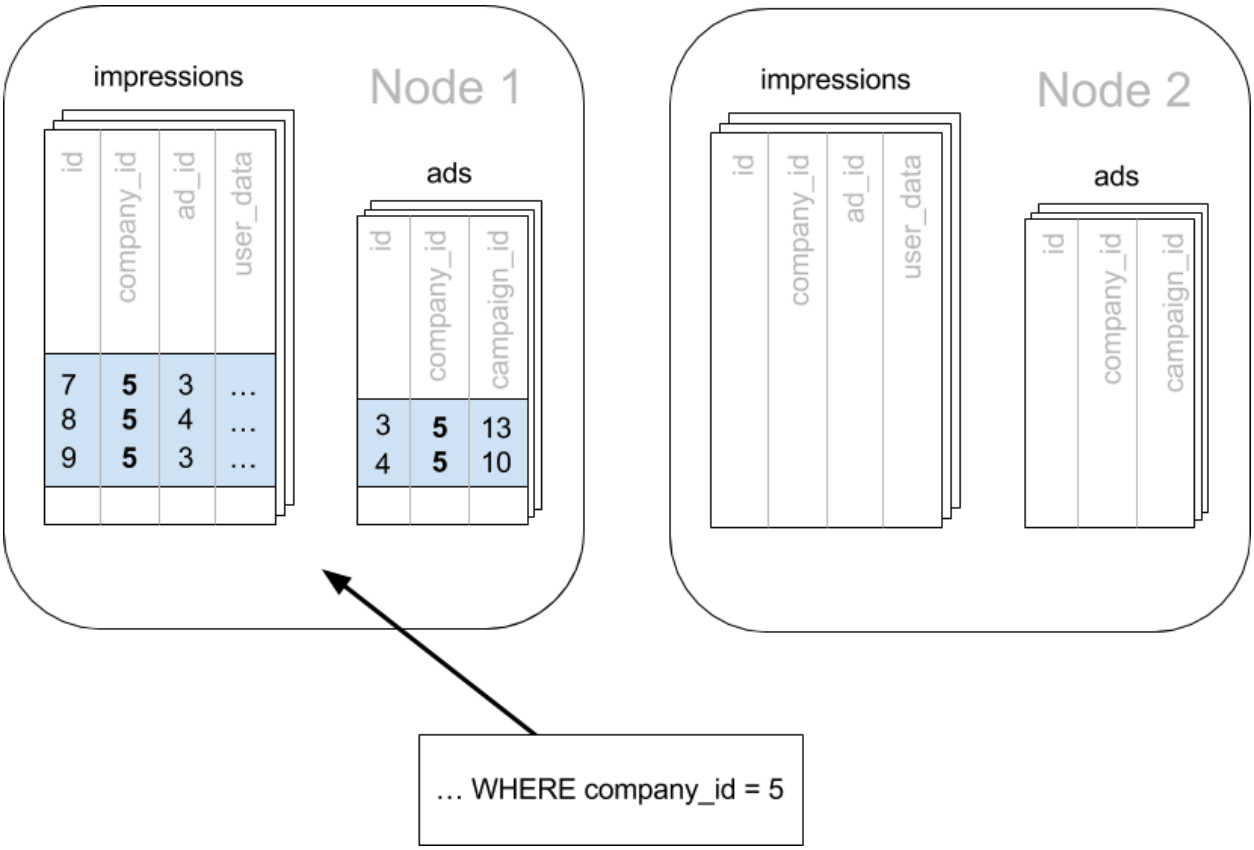
30.2 Scaling the Relational Data Model

The relational data model is great for applications. It protects data integrity, allows flexible queries, and accommodates changing data. Traditionally the only problem was that relational databases weren't considered capable of scaling to the workloads needed for big SaaS applications. Developers had to put up with NoSQL databases – or a collection of backend services – to reach that size.

With Citus you can keep your data model *and* make it scale. Citus appears to applications as a single PostgreSQL database, but it internally routes queries to an adjustable number of physical servers (nodes) which can process requests in parallel.

Multi-tenant applications have a nice property that we can take advantage of: queries usually always request information for one tenant at a time, not a mix of tenants. For instance, when a salesperson is searching prospect information in a CRM, the search results are specific to his employer; other businesses' leads and notes are not included.

Because application queries are restricted to a single tenant, such as a store or company, one approach for making multi-tenant application queries fast is to store *all* data for a given tenant on the same node. This minimizes network overhead between the nodes and allows Citus to support all your application's joins, key constraints and transactions efficiently. With this, you can scale across multiple nodes without having to totally re-write or re-architect your application.



We do this in Citus by making sure every table in our schema has a column to clearly mark which tenant owns which rows. In the ad analytics application the tenants are companies, so we must ensure all tables have a `company_id` column.

We can tell Citus to use this column to read and write rows to the same node when the rows are marked for the same company. In Citus' terminology `company_id` will be the *distribution column*, which you can learn more about in *Distributed Data Modeling*.

30.3 Preparing Tables and Ingesting Data

In the previous section we identified the correct distribution column for our multi-tenant application: the company id. Even in a single-machine database it can be useful to denormalize tables with the addition of company id, whether it be for row-level security or for additional indexing. The extra benefit, as we saw, is that including the extra column helps for multi-machine scaling as well.

The schema we have created so far uses a separate `id` column as primary key for each table. Citus requires that primary and foreign key constraints include the distribution column. This requirement makes enforcing these constraints much more efficient in a distributed environment as only a single node has to be checked to guarantee them.

In SQL, this requirement translates to making primary and foreign keys composite by including `company_id`. This is compatible with the multi-tenant case because what we really need there is to ensure uniqueness on a per-tenant basis.

Putting it all together, here are the changes which prepare the tables for distribution by `company_id`.

```
CREATE TABLE companies (
  id bigserial PRIMARY KEY,
```

```
  name text NOT NULL,
  image_url text,
  created_at timestamp without time zone NOT NULL,
  updated_at timestamp without time zone NOT NULL
);

CREATE TABLE campaigns (
  id bigserial,       -- was: PRIMARY KEY
  company_id bigint REFERENCES companies (id),
  name text NOT NULL,
  cost_model text NOT NULL,
  state text NOT NULL,
  monthly_budget bigint,
  blacklisted_site_urls text[],
  created_at timestamp without time zone NOT NULL,
  updated_at timestamp without time zone NOT NULL,
  PRIMARY KEY (company_id, id) -- added
);

CREATE TABLE ads (
  id bigserial,       -- was: PRIMARY KEY
  company_id bigint,  -- added
  campaign_id bigint, -- was: REFERENCES campaigns (id)
  name text NOT NULL,
  image_url text,
  target_url text,
  impressions_count bigint DEFAULT 0,
  clicks_count bigint DEFAULT 0,
  created_at timestamp without time zone NOT NULL,
  updated_at timestamp without time zone NOT NULL,
  PRIMARY KEY (company_id, id),         -- added
  FOREIGN KEY (company_id, campaign_id) -- added
    REFERENCES campaigns (company_id, id)
);

CREATE TABLE clicks (
  id bigserial,        -- was: PRIMARY KEY
  company_id bigint,   -- added
  ad_id bigint,        -- was: REFERENCES ads (id),
  clicked_at timestamp without time zone NOT NULL,
  site_url text NOT NULL,
  cost_per_click_usd numeric(20,10),
  user_ip inet NOT NULL,
  user_data jsonb NOT NULL,
  PRIMARY KEY (company_id, id),      -- added
  FOREIGN KEY (company_id, ad_id)    -- added
    REFERENCES ads (company_id, id)
);

CREATE TABLE impressions (
  id bigserial,         -- was: PRIMARY KEY
  company_id bigint,    -- added
  ad_id bigint,         -- was: REFERENCES ads (id),
  seen_at timestamp without time zone NOT NULL,
  site_url text NOT NULL,
  cost_per_impression_usd numeric(20,10),
  user_ip inet NOT NULL,
  user_data jsonb NOT NULL,
```

```
  PRIMARY KEY (company_id, id),        -- added
  FOREIGN KEY (company_id, ad_id)      -- added
    REFERENCES ads (company_id, id)
);
```

You can learn more about migrating your own data model in *multi-tenant schema migration*.

30.3.1 Try it Yourself

Note: This guide is designed so you can follow along in your own Citus database. Use one of these alternatives to spin up a database:

- Run Citus locally using *Docker (Mac or Linux)*, or
- Provision a cluster using Citus Cloud

You'll run the SQL commands using psql:

- **Docker**: `docker exec -it citus_master psql -U postgres`
- **Cloud**: `psql "connection-string"` where the connection string for your formation is available in the Cloud Console.

In either case psql will be connected to the coordinator node for the cluster.

At this point feel free to follow along in your own Citus cluster by downloading and executing the SQL to create the schema. Once the schema is ready, we can tell Citus to create shards on the workers. From the coordinator node, run:

```
SELECT create_distributed_table('companies',    'id');
SELECT create_distributed_table('campaigns',    'company_id');
SELECT create_distributed_table('ads',          'company_id');
SELECT create_distributed_table('clicks',       'company_id');
SELECT create_distributed_table('impressions', 'company_id');
```

The *create_distributed_table* function informs Citus that a table should be distributed among nodes and that future incoming queries to those tables should be planned for distributed execution. The function also creates shards for the table on worker nodes, which are low-level units of data storage Citus uses to assign data to nodes.

The next step is loading sample data into the cluster from the command line.

```
# download and ingest datasets from the shell

for dataset in companies campaigns ads clicks impressions geo_ips; do
  curl -O https://examples.citusdata.com/mt_ref_arch/${dataset}.csv
done
```

Note: **If you are using Docker,** you should use the `docker cp` command to copy the files into the Docker container.

```
for dataset in companies campaigns ads clicks impressions geo_ips; do
  docker cp ${dataset}.csv citus_master:.
done
```

Being an extension of PostgreSQL, Citus supports bulk loading with the COPY command. Use it to ingest the data you downloaded, and make sure that you specify the correct file path if you downloaded the file to some other location. Back inside psql run this:

```
\copy companies from 'companies.csv' with csv
\copy campaigns from 'campaigns.csv' with csv
\copy ads from 'ads.csv' with csv
\copy clicks from 'clicks.csv' with csv
\copy impressions from 'impressions.csv' with csv
```

30.4 Integrating Applications

Here's the good news: once you have made the slight schema modification outlined earlier, your application can scale with very little work. You'll just connect the app to Citus and let the database take care of keeping the queries fast and the data safe.

Any application queries or update statements which include a filter on `company_id` will continue to work exactly as they are. As mentioned earlier, this kind of filter is common in multi-tenant apps. When using an Object-Relational Mapper (ORM) you can recognize these queries by methods such as `where` or `filter`.

ActiveRecord:

```
Impression.where(company_id: 5).count
```

Django:

```
Impression.objects.filter(company_id=5).count()
```

Basically when the resulting SQL executed in the database contains a `WHERE company_id = :value` clause on every table (including tables in JOIN queries), then Citus will recognize that the query should be routed to a single node and execute it there as it is. This makes sure that all SQL functionality is available. The node is an ordinary PostgreSQL server after all.

Also, to make it even simpler, you can use our activerecord-multi-tenant library for Rails (one for Django is in progress as well) which will automatically add these filters to all your queries, even the complicated ones. Check out our *App Migration* section for details.

This guide is framework-agnostic, so we'll point out some Citus features using SQL. Use your imagination for how these statements would be expressed in your language of choice.

Here is a simple query and update operating on a single tenant.

```
-- campaigns with highest budget

SELECT name, cost_model, state, monthly_budget
  FROM campaigns
 WHERE company_id = 5
 ORDER BY monthly_budget DESC
 LIMIT 10;

-- double the budgets!

UPDATE campaigns
   SET monthly_budget = monthly_budget*2
 WHERE company_id = 5;
```

A common pain point for users scaling applications with NoSQL databases is the lack of transactions and joins. However, transactions work as you'd expect them to in Citus:

```
-- transactionally reallocate campaign budget money

BEGIN;

UPDATE campaigns
   SET monthly_budget = monthly_budget + 1000
 WHERE company_id = 5
   AND id = 40;

UPDATE campaigns
   SET monthly_budget = monthly_budget - 1000
 WHERE company_id = 5
   AND id = 41;

COMMIT;
```

As a final demo of SQL support, we have a query which includes aggregates and window functions and it works the same in Citus as it does in PostgreSQL. The query ranks the ads in each campaign by the count of their impressions.

```
SELECT a.campaign_id,
       RANK() OVER (
         PARTITION BY a.campaign_id
         ORDER BY a.campaign_id, count(*) desc
       ), count(*) as n_impressions, a.id
  FROM ads as a,
       impressions as i
 WHERE a.company_id = 5
   AND i.company_id = a.company_id
   AND i.ad_id      = a.id
GROUP BY a.campaign_id, a.id
ORDER BY a.campaign_id, n_impressions desc;
```

In short when queries are scoped to a tenant then inserts, updates, deletes, complex SQL, and transactions all work as expected.

30.5 Sharing Data Between Tenants

Up until now all tables have been distributed by `company_id`, but sometimes there is data that can be shared by all tenants, and doesn't "belong" to any tenant in particular. For instance, all companies using this example ad platform might want to get geographical information for their audience based on IP addresses. In a single machine database this could be accomplished by a lookup table for geo-ip, like the following. (A real table would probably use PostGIS but bear with the simplified example.)

```
CREATE TABLE geo_ips (
  addrs cidr NOT NULL PRIMARY KEY,
  latlon point NOT NULL
    CHECK (-90  <= latlon[0] AND latlon[0] <= 90 AND
           -180 <= latlon[1] AND latlon[1] <= 180)
);
CREATE INDEX ON geo_ips USING gist (addrs inet_ops);
```

To use this table efficiently in a distributed setup, we need to find a way to co-locate the `geo_ips` table with clicks for not just one – but every – company. That way, no network traffic need be incurred at query time. We do this in Citus by designating `geo_ips` as a *reference table*.

```
-- Make synchronized copies of geo_ips on all workers

SELECT create_reference_table('geo_ips');
```

Reference tables are replicated across all worker nodes, and Citus automatically keeps them in sync during modifications. Notice that we call *create_reference_table* rather than `create_distributed_table`.

Now that `geo_ips` is established as a reference table, load it with example data:

```
\copy geo_ips from 'geo_ips.csv' with csv
```

Now joining clicks with this table can execute efficiently. We can ask, for example, the locations of everyone who clicked on ad 290.

```
SELECT c.id, clicked_at, latlon
  FROM geo_ips, clicks c
 WHERE addrs >> c.user_ip
   AND c.company_id = 5
   AND c.ad_id = 290;
```

30.6 Online Changes to the Schema

Another challenge with multi-tenant systems is keeping the schemas for all the tenants in sync. Any schema change needs to be consistently reflected across all the tenants. In Citus, you can simply use standard PostgreSQL DDL commands to change the schema of your tables, and Citus will propagate them from the coordinator node to the workers using a two-phase commit protocol.

For example, the advertisements in this application could use a text caption. We can add a column to the table by issuing the standard SQL on the coordinator:

```
ALTER TABLE ads
  ADD COLUMN caption text;
```

This updates all the workers as well. Once this command finishes, the Citus cluster will accept queries that read or write data in the new `caption` column.

For a fuller explanation of how DDL commands propagate through the cluster, see *Modifying Tables*.

30.7 When Data Differs Across Tenants

Given that all tenants share a common schema and hardware infrastructure, how can we accommodate tenants which want to store information not needed by others? For example, one of the tenant applications using our advertising database may want to store tracking cookie information with clicks, whereas another tenant may care about browser agents. Traditionally databases using a shared schema approach for multi-tenancy have resorted to creating a fixed number of pre-allocated "custom" columns, or having external "extension tables." However PostgreSQL provides a much easier way with its unstructured column types, notably JSONB.

Notice that our schema already has a JSONB field in `clicks` called `user_data`. Each tenant can use it for flexible storage.

Suppose company five includes information in the field to track whether the user is on a mobile device. The company can query to find who clicks more, mobile or traditional visitors:

```
SELECT
  user_data->>'is_mobile' AS is_mobile,
  count(*) AS count
FROM clicks
WHERE company_id = 5
GROUP BY user_data->>'is_mobile'
ORDER BY count DESC;
```

The database administrator can even create a partial index to improve speed for an individual tenant's query patterns. Here is one to improve company 5's filters for clicks from users on mobile devices:

```
CREATE INDEX click_user_data_is_mobile
ON clicks ((user_data->>'is_mobile'))
WHERE company_id = 5;
```

Additionally, PostgreSQL supports GIN indices on JSONB. Creating a GIN index on a JSONB column will create an index on every key and value within that JSON document. This speeds up a number of JSONB operators such as `?`, `?|`, and `?&`.

```
CREATE INDEX click_user_data
ON clicks USING gin (user_data);

-- this speeds up queries like, "which clicks have
-- the is_mobile key present in user_data?"

SELECT id
  FROM clicks
 WHERE user_data ? 'is_mobile'
   AND company_id = 5;
```

30.8 Scaling Hardware Resources

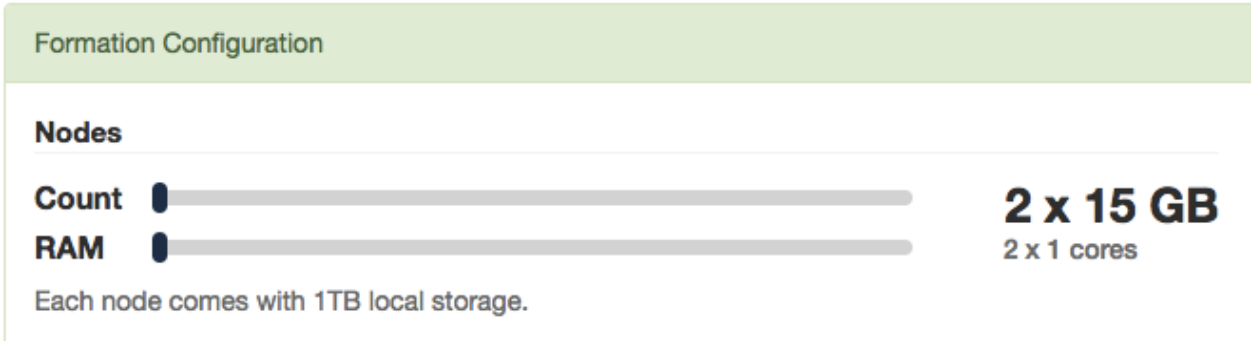
Note: This section uses features available only in Citus Cloud and Citus Enterprise. Also, please note that these features are available in Citus Cloud across all plans except for the "Dev Plan".

Multi-tenant databases should be designed for future scale as business grows or tenants want to store more data. Citus can scale out easily by adding new machines without having to make any changes or take application downtime.

Being able to rebalance data in the Citus cluster allows you to grow your data size or number of customers and improve performance on demand. Adding new machines allows you to keep data in memory even when it is much larger than what a single machine can store.

Also, if data increases for only a few large tenants, then you can isolate those particular tenants to separate nodes for better performance.

To scale out your Citus cluster, first add a new worker node to it. On Citus Cloud, you can use the slider present in the "Settings" tab, sliding it to add the required number of nodes. Alternately, if you run your own Citus installation, you can add nodes manually with the *master_add_node* UDF.



Once you add the node it will be available in the system. However at this point no tenants are stored on it and Citus will not yet run any queries there. To move your existing data, you can ask Citus to rebalance the data. This operation moves bundles of rows called shards between the currently active nodes to attempt to equalize the amount of data on each node.

```
SELECT rebalance_table_shards('companies');
```

Rebalancing preserves *Table Co-Location*, which means we can tell Citus to rebalance the companies table and it will take the hint and rebalance the other tables which are distributed by company_id. Also, applications do not need to undergo downtime during shard rebalancing. Read requests continue seamlessly, and writes are locked only when they affect shards which are currently in flight.

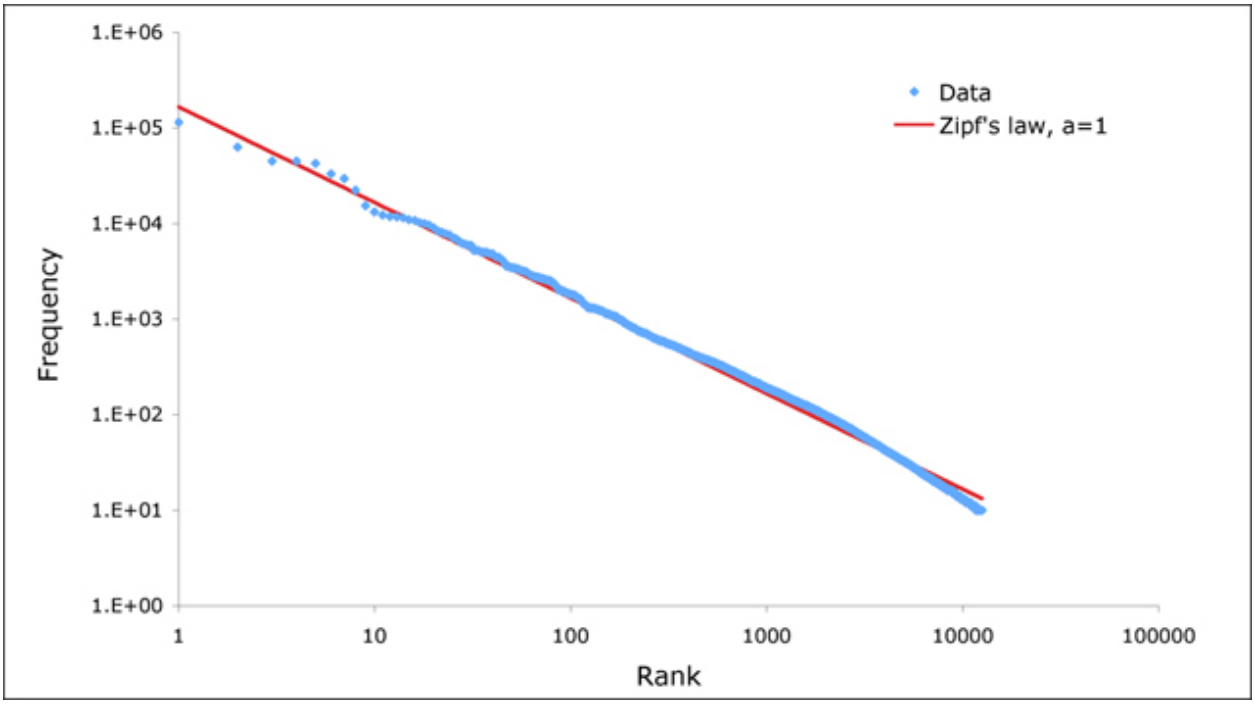
You can learn more about how shard rebalancing works here: *Scaling Out (adding new nodes)*.

30.9 Dealing with Big Tenants

Note: This section uses features available only in Citus Cloud and Citus Enterprise.

The previous section describes a general-purpose way to scale a cluster as the number of tenants increases. However, users often have two questions. The first is what will happen to their largest tenant if it grows too big. The second is what are the performance implications of hosting a large tenant together with small ones on a single worker node, and what can be done about it.

Regarding the first question, investigating data from large SaaS sites reveals that as the number of tenants increases, the size of tenant data typically tends to follow a Zipfian distribution.



For instance, in a database of 100 tenants, the largest is predicted to account for about 20% of the data. In a more realistic example for a large SaaS company, if there are 10k tenants, the largest will account for around 2% of the data. Even at 10TB of data, the largest tenant will require 200GB, which can pretty easily fit on a single node.

Another question is regarding performance when large and small tenants are on the same node. Standard shard rebalancing will improve overall performance but it may or may not improve the mixing of large and small tenants. The rebalancer simply distributes shards to equalize storage usage on nodes, without examining which tenants are allocated on each shard.

To improve resource allocation and make guarantees of tenant QoS it is worthwhile to move large tenants to dedicated nodes. Citus provides the tools to do this.

In our case, let's imagine that our old friend company id=5 is very large. We can isolate the data for this tenant in two steps. We'll present the commands here, and you can consult *Tenant Isolation* to learn more about them.

First sequester the tenant's data into a bundle (called a shard) suitable to move. The CASCADE option also applies this change to the rest of our tables distributed by `company_id`.

```
SELECT isolate_tenant_to_new_shard(
  'companies', 5, 'CASCADE'
);
```

The output is the shard id dedicated to hold `company_id=5`:

```
--------------------
| isolate_tenant_to_new_shard |
--------------------
|                      102240 |
--------------------
```

Next we move the data across the network to a new dedicated node. Create a new node as described in the previous section. Take note of its hostname as shown in the Nodes tab of the Cloud Console.

```
-- find the node currently holding the new shard
```

```
SELECT nodename, nodeport
  FROM pg_dist_shard_placement
 WHERE shardid = 102240;

-- move the shard to your choice of worker (it will also move the
-- other shards created with the CASCADE option)

SELECT master_move_shard_placement(
  102240,
  'source_host', source_port,
  'dest_host', dest_port);
```

You can confirm the shard movement by querying *pg_dist_shard_placement* again.

30.10 Where to Go From Here

With this, you now know how to use Citus to power your multi-tenant application for scalability. If you have an existing schema and want to migrate it for Citus, see *Multi-Tenant Transitioning*.

To adjust a front-end application, specifically Ruby on Rails, read *App Migration*. Finally, try *Citus Cloud*, the easiest way to manage a Citus cluster, available with discounted developer plan pricing.

CHAPTER 31

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 section 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.

31.1 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.

31.2 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 coordinator

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. 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.

Note: Citus Cloud uses streaming replication to achieve high availability and thus maintaining shard replicas would be redundant. In any production environment where streaming replication is unavailable, you should set `citus.shard_replication_factor` to 2 or higher for fault tolerance.

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).

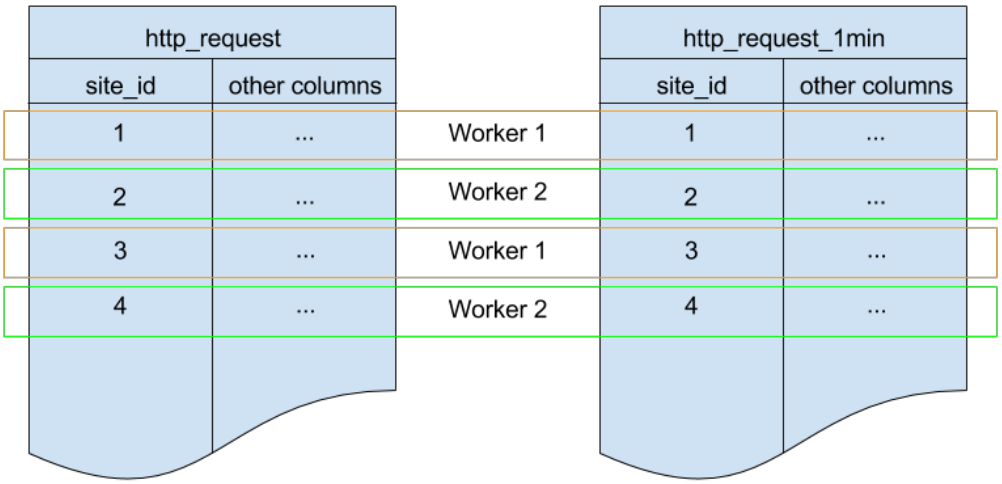
31.3 Rollups

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 *co-location*; 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 co-located.

```
-- 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
  -- 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 coordinator 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 coordinator
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 coordinator, 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';
```

31.4 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 coordinator
CREATE OR REPLACE FUNCTION expire_old_request_data() RETURNS void
AS $$
BEGIN
  SET citus.all_modification_commutative TO TRUE;
  PERFORM master_modify_multiple_shards(
            'DELETE FROM http_request WHERE ingest_time < now() - interval ''1 day''';
↪');
  PERFORM master_modify_multiple_shards(
            'DELETE FROM http_request_1min WHERE ingest_time < now() - interval ''1␣
↪month''';');
END;
$$ LANGUAGE 'plpgsql';
```

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

31.5 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 coordinator 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 coordinator
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 coordinator
CREATE AGGREGATE sum (hll)
(
  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.

31.6 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());
```

31.7 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 32

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.

32.1 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.

32.1.1 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');
```

32.1.2 Adding a coordinator

The Citus coordinator 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 coordinator does only final aggregations on the result of the workers. Therefore, it is not very likely that the coordinator becomes a bottleneck for read performance. Also, it is easy to boost up the coordinator by shifting to a more powerful machine.

However, in some write heavy use cases where the coordinator becomes a performance bottleneck, users can add another coordinator. 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 coordinator and scale out performance. If your setup requires you to use multiple coordinators, please contact us.

32.2 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 coordinator.

32.2.1 Worker Node Failures

Citus supports two modes of replication, allowing it to tolerate worker-node failures. In the first model, we use PostgreSQL's streaming replication to replicate the entire worker-node as-is. In the second model, Citus can replicate data modification statements, thus replicating shards across different worker nodes. They have different advantages depending on the workload and use-case as discussed below:

1. **PostgreSQL streaming replication.** This option is best for heavy OLTP workloads. It replicates entire worker nodes by continuously streaming their WAL records to a standby. You can configure streaming replication on-premise yourself by consulting the PostgreSQL replication documentation or use *Citus Cloud* which is pre-configured for replication and high-availability.

2. **Citus shard replication.** This option is best suited for an append-only workload. Citus replicates shards across different nodes by automatically replicating DML statements and managing consistency. If a node goes down, the co-ordinator node will continue to serve queries by routing the work to the replicas seamlessly. To enable shard replication simply set `SET citus.shard_replication_factor = 2;` (or higher) before distributing data to the cluster.

32.2.2 Coordinator Node Failures

The Citus coordinator 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 coordinator failures.

1. **Use PostgreSQL streaming replication:** You can use PostgreSQL's streaming replication feature to create a hot standby of the coordinator. Then, if the primary coordinator 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.

32.3 Tenant Isolation

Note: Tenant isolation is a feature of **Citus Enterprise Edition** and *Citus Cloud* only.

Citus places table rows into worker shards based on the hashed value of the rows' distribution column. Multiple distribution column values often fall into the same shard. In the Citus multi-tenant use case this means that tenants often share shards.

However sharing shards can cause resource contention when tenants differ drastically in size. This is a common situation for systems with a large number of tenants – we have observed that the size of tenant data tend to follow a Zipfian distribution as the number of tenants increases. This means there are a few very large tenants, and many smaller ones. To improve resource allocation and make guarantees of tenant QoS it is worthwhile to move large tenants to dedicated nodes.

Citus Enterprise Edition and *Citus Cloud* provide the tools to isolate a tenant on a specific node. This happens in two phases: 1) isolating the tenant's data to a new dedicated shard, then 2) moving the shard to the desired node. To understand the process it helps to know precisely how rows of data are assigned to shards.

Every shard is marked in Citus metadata with the range of hashed values it contains (more info in the reference for *pg_dist_shard*). The Citus UDF `isolate_tenant_to_new_shard(table_name,tenant_id)` moves a tenant into a dedicated shard in three steps:

1. Creates a new shard for `table_name` which (a) includes rows whose distribution column has value `tenant_id` and (b) excludes all other rows.
2. Moves the relevant rows from their current shard to the new shard.
3. Splits the old shard into two with hash ranges that abut the excision above and below.

Furthermore, the UDF takes a `CASCADE` option which isolates the tenant rows of not just `table_name` but of all tables *co-located* with it. Here is an example:

```
-- This query creates an isolated shard for the given tenant_id and
-- returns the new shard id.

-- General form:

SELECT isolate_tenant_to_new_shard('table_name', tenant_id);

-- Specific example:

SELECT isolate_tenant_to_new_shard('lineitem', 135);

-- If the given table has co-located tables, the query above errors out and
-- advises to use the CASCADE option

SELECT isolate_tenant_to_new_shard('lineitem', 135, 'CASCADE');
```

Output:

```
--------------------
| isolate_tenant_to_new_shard |
--------------------
|                      102240 |
--------------------
```

The new shard(s) are created on the same node as the shard(s) from which the tenant was removed. For true hardware isolation they can be moved to a separate node in the Citus cluster. As mentioned, the `isolate_tenant_to_new_shard` function returns the newly created shard id, and this id can be used to move the shard:

```
-- find the node currently holding the new shard
SELECT nodename, nodeport
  FROM pg_dist_shard_placement
```

```
 WHERE shardid = 102240;

-- list the available worker nodes that could hold the shard
SELECT * FROM master_get_active_worker_nodes();

-- move the shard to your choice of worker
-- (it will also move any shards created with the CASCADE option)
SELECT master_move_shard_placement(
  102240,
  'source_host', source_port,
  'dest_host', dest_port);
```

Note that `master_move_shard_placement` will also move any shards which are co-located with the specified one, to preserve their co-location.

32.4 Running a Query on All Workers

Broadcasting a statement for execution on all workers is useful for viewing properties of entire worker databases or creating UDFs uniformly throughout the cluster. For example:

```
-- Make a UDF available on all workers
SELECT run_command_on_workers($cmd$ CREATE FUNCTION ...; $cmd$);

-- List the work_mem setting of each worker database
SELECT run_command_on_workers($cmd$ SHOW work_mem; $cmd$);
```

The `run_command_on_workers` function can run only queries which return a single column and single row.

32.5 Worker Security

For your convenience getting started, our multi-node installation instructions direct you to set up the `pg_hba.conf` on the workers with its authentication method set to "trust" for local network connections. However you might desire more security.

To require that all connections supply a hashed password, update the PostgreSQL `pg_hba.conf` on every worker node with something like this:

```
# Require password 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              md5

# Require passwords when the host connects to itself as well
host    all             all             127.0.0.1/32            md5
host    all             all             ::1/128                 md5
```

The coordinator node needs to know roles' passwords in order to communicate with the workers. Add a .pgpass file to the postgres user's home directory, with a line for each combination of worker address and role:

```
hostname:port:database:username:password
```

Sometimes workers need to connect to one another, such as during *repartition joins*. Thus each worker node requires a copy of the .pgpass file as well.

32.6 Diagnostics

32.6.1 Finding which shard contains data for a specific tenant

The rows of a distributed table are grouped into shards, and each shard is placed on a worker node in the Citus cluster. In the multi-tenant Citus use case we can determine which worker node contains the rows for a specific tenant by putting together two pieces of information: the *shard id* associated with the tenant id, and the shard placements on workers. The two can be retrieved together in a single query. Suppose our multi-tenant application's tenants and are stores, and we want to find which worker node holds the data for Gap.com (id=4, suppose).

To find the worker node holding the data for store id=4, ask for the placement of rows whose distribution column has value 4:

```
SELECT *
  FROM pg_dist_shard_placement
 WHERE shardid = (
   SELECT get_shard_id_for_distribution_column('stores', 4)
 );
```

The output contains the host and port of the worker database.

```
---------------------------------------------
| shardid | shardstate | shardlength | nodename  | nodeport | placementid |
---------------------------------------------
|  102009 |          1 |           0 | localhost |     5433 |           2 |
---------------------------------------------
```

32.6.2 Finding the distribution column for a table

Each distributed table in Citus has a "distribution column." For more information about what this is and how it works, see *Distributed Data Modeling*. There are many situations where it is important to know which column it is. Some operations require joining or filtering on the distribution column, and you may encounter error messages with hints like, "add a filter to the distribution column."

The `pg_dist_*` tables on the coordinator node contain diverse metadata about the distributed database. In particular `pg_dist_partition` holds information about the distribution column (formerly called *partition* column) for each table. You can use a convenient utility function to look up the distribution column name from the low-level details in the metadata. Here's an example and its output:

```
-- create example table

CREATE TABLE products (
  store_id bigint,
  product_id bigint,
  name text,
  price money,

  CONSTRAINT products_pkey PRIMARY KEY (store_id, product_id)
);

-- pick store_id as distribution column

SELECT create_distributed_table('products', 'store_id');

-- get distribution column name for products table
```

```
SELECT column_to_column_name(logicalrelid, partkey) AS dist_col_name
  FROM pg_dist_partition
 WHERE logicalrelid='products'::regclass;
```

Output:

```
-----------
| dist_col_name |
-----------
| store_id      |
-----------
```

CHAPTER 33

Table Management

33.1 Determining Table and Relation Size

The usual way to find table sizes in PostgreSQL, `pg_total_relation_size`, drastically under-reports the size of distributed tables. All this function does on a Citus cluster is reveal the size of tables on the coordinator node. In reality the data in distributed tables lives on the worker nodes (in shards), not on the coordinator. A true measure of distributed table size is obtained as a sum of shard sizes. Citus provides helper functions to query this information.

UDF	Returns
citus_relation_size(relation_name)	• Size of actual data in table (the "main fork"). • A relation can be the name of a table or an index.
citus_table_size(relation_name)	• citus_relation_size plus: – size of free space map – size of visibility map
citus_total_relation_size(relation_name)	• citus_table_size plus: – size of indices

These functions are analogous to three of the standard PostgreSQL object size functions, with the additional note that

- They work only when `citus.shard_replication_factor` = 1.
- If they can't connect to a node, they error out.

Here is an example of using one of the helper functions to list the sizes of all distributed tables:

```
SELECT logicalrelid AS name,
       pg_size_pretty(citus_table_size(logicalrelid)) AS size
  FROM pg_dist_partition;
```

Output:

```
---------------
|     name      | size  |
---------------
| github_users  | 39 MB |
| github_events | 37 MB |
---------------
```

33.2 Vacuuming Distributed Tables

In PostgreSQL (and other MVCC databases), an UPDATE or DELETE of a row does not immediately remove the old version of the row. The accumulation of outdated rows is called bloat and must be cleaned to avoid decreased query performance and unbounded growth of disk space requirements. PostgreSQL runs a process called the auto-vacuum daemon that periodically vacuums (aka removes) outdated rows.

It's not just user queries which scale in a distributed database, vacuuming does too. In PostgreSQL big busy tables have great potential to bloat, both from lower sensitivity to PostgreSQL's vacuum scale factor parameter, and generally because of the extent of their row churn. Splitting a table into distributed shards means both that individual shards are smaller tables and that auto-vacuum workers can parallelize over different parts of the table on different machines. Ordinarily auto-vacuum can only run one worker per table.

Due to the above, auto-vacuum operations on a Citus cluster are probably good enough for most cases. However for tables with particular workloads, or companies with certain "safe" hours to schedule a vacuum, it might make more sense to manually vacuum a table rather than leaving all the work to auto-vacuum.

To vacuum a table, simply run this on the coordinator node:

```
VACUUM my_distributed_table;
```

Using vacuum against a distributed table will send a vacuum command to every one of that table's placements (one connection per placement). This is done in parallel. All options are supported (including the `column_list` parameter) except for `VERBOSE`. The vacuum command also runs on the coordinator, and does so before any workers nodes are notified. Note that unqualified vacuum commands (i.e. those without a table specified) do not propagate to worker nodes.

33.3 Analyzing Distributed Tables

PostgreSQL's ANALYZE command collects statistics about the contents of tables in the database. Subsequently, the query planner uses these statistics to help determine the most efficient execution plans for queries.

The auto-vacuum daemon, discussed in the previous section, will automatically issue ANALYZE commands whenever the content of a table has changed sufficiently. The daemon schedules ANALYZE strictly as a function of the number of rows inserted or updated; it has no knowledge of whether that will lead to meaningful statistical changes. Administrators might prefer to manually schedule ANALYZE operations instead, to coincide with statistically meaningful table changes.

To analyze a table, run this on the coordinator node:

```
ANALYZE my_distributed_table;
```

Citus propagates the ANALYZE command to all worker node placements.

CHAPTER 34

Upgrading Citus

34.1 Upgrading Citus Versions

Citus adheres to semantic versioning with patch-, minor-, and major-versions. The upgrade process differs for each, requiring more effort for bigger version jumps.

Upgrading the Citus version requires first obtaining the new Citus extension and then installing it in each of your database instances. Citus uses separate packages for each minor version to ensure that running a default package upgrade will provide bug fixes but never break anything. Let's start by examining patch upgrades, the easiest kind.

34.1.1 Patch Version Upgrade

To upgrade a Citus version to its latest patch, issue a standard upgrade command for your package manager. Assuming version 6.2 is currently installed:

Ubuntu or Debian

```
sudo apt-get update
sudo apt-get install --only-upgrade postgresql-9.6-citus-6.2
sudo service postgresql restart
```

Fedora, CentOS, or Red Hat

```
sudo yum update citus62_96
sudo service postgresql-9.6 restart
```

34.1.2 Major and Minor Version Upgrades

Major and minor version upgrades follow the same steps, but be careful: major upgrades can make backward-incompatible changes in the Citus API. It is best to review the Citus changelog before a major upgrade and look for any changes which may cause problems for your application.

Each major and minor version of Citus is published as a package with a separate name. Installing a newer package will automatically remove the older version. Here is how to upgrade from 6.1 to 6.2 for instance:

Step 1. Update Citus Package

Ubuntu or Debian

```
sudo apt-get update
sudo apt-get install postgresql-9.6-citus-6.2
sudo service postgresql restart
```

Fedora, CentOS, or Red Hat

```
# Fedora, CentOS, or Red Hat
sudo yum swap citus61_96 citus62_96
sudo service postgresql-9.6 restart
```

Step 2. Apply Update in DB

After installing the new package and restarting the database, run the extension upgrade script.

```
# you must restart PostgreSQL before running this
psql -c 'ALTER EXTENSION citus UPDATE;'

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

Note: During a major version upgrade, from the moment of yum installing a new version, Citus will refuse to run distributed queries until the server is restarted and ALTER EXTENSION is executed. This is to protect your data, as Citus object and function definitions are specific to a version. After a yum install you should (a) restart and (b) run alter extension. In rare cases if you experience an error with upgrades, you can disable this check via the *citus.enable_version_checks* configuration parameter. You can also contact us providing information about the error, so we can help debug the issue.

Step 3. (upgrade from 5.x only) Add Co-Location Metadata

When doing a major upgrade from Citus 5.x be sure to create metadata for your implicit table co-location. Read more about that in the *Upgrading from Citus 5.x* section of the co-location page.

34.2 Upgrading PostgreSQL version from 9.5 to 9.6

Note: PostgreSQL 9.6 requires using Citus 6.0 or greater. To upgrade PostgreSQL with an older version of Citus, first upgrade Citus as explained in *Major and Minor Version Upgrades*.

Record the following paths before you start (your actual paths may be different than those below):

Existing data directory (e.g. /opt/pgsql/9.5/data) `export OLD_PG_DATA=/opt/pgsql/9.5/data`

Existing PostgreSQL installation path (e.g. /usr/pgsql-9.5) `export OLD_PG_PATH=/usr/pgsql-9.5`

New data directory after upgrade `export NEW_PG_DATA=/opt/pgsql/9.6/data`

New PostgreSQL installation path `export NEW_PG_PATH=/usr/pgsql-9.6`

34.2.1 On the Coordinator Node

1. If using Citus v5.x follow the *previous steps* to install Citus 6.0 onto the existing postgresql 9.5 server.
2. Back up Citus metadata in the old server.

```
CREATE TABLE public.pg_dist_partition AS SELECT * FROM pg_catalog.pg_dist_
↪partition;
CREATE TABLE public.pg_dist_shard AS SELECT * FROM pg_catalog.pg_dist_shard;
CREATE TABLE public.pg_dist_shard_placement AS SELECT * FROM pg_catalog.pg_
↪dist_shard_placement;
CREATE TABLE public.pg_dist_node AS SELECT * FROM pg_catalog.pg_dist_node;
CREATE TABLE public.pg_dist_local_group AS SELECT * FROM pg_catalog.pg_dist_
↪local_group;
CREATE TABLE public.pg_dist_transaction AS SELECT * FROM pg_catalog.pg_dist_
↪transaction;
CREATE TABLE public.pg_dist_colocation AS SELECT * FROM pg_catalog.pg_dist_
↪colocation;
```

3. Configure the new database instance to use Citus.

- Include Citus as a shared preload library in postgresql.conf:

```
shared_preload_libraries = 'citus'
```

- **DO NOT CREATE** Citus extension yet

4. Stop the old and new servers.
5. Check upgrade compatibility.

```
$NEW_PG_PATH/bin/pg_upgrade -b $OLD_PG_PATH/bin/ -B $NEW_PG_PATH/bin/ \
                            -d $OLD_PG_DATA -D $NEW_PG_DATA --check
```

You should see a "Clusters are compatible" message. If you do not, fix any errors before proceeding. Please ensure that

- `NEW_PG_DATA` contains an empty database initialized by new PostgreSQL version
- The Citus extension **IS NOT** created

6. Perform the upgrade (like before but without the `--check` option).

```
$NEW_PG_PATH/bin/pg_upgrade -b $OLD_PG_PATH/bin/ -B $NEW_PG_PATH/bin/ \
                            -d $OLD_PG_DATA -D $NEW_PG_DATA
```

7. Start the new server.
8. Restore metadata.

```
INSERT INTO pg_catalog.pg_dist_partition SELECT * FROM public.pg_dist_
↪partition;
INSERT INTO pg_catalog.pg_dist_shard SELECT * FROM public.pg_dist_shard;
INSERT INTO pg_catalog.pg_dist_shard_placement SELECT * FROM public.pg_dist_
↪shard_placement;
INSERT INTO pg_catalog.pg_dist_node SELECT * FROM public.pg_dist_node;
TRUNCATE TABLE pg_catalog.pg_dist_local_group;
INSERT INTO pg_catalog.pg_dist_local_group SELECT * FROM public.pg_dist_
↪local_group;
```

```
INSERT INTO pg_catalog.pg_dist_transaction SELECT * FROM public.pg_dist_
↪transaction;
INSERT INTO pg_catalog.pg_dist_colocation SELECT * FROM public.pg_dist_
↪colocation;
```

9. Drop temporary metadata tables.

```
DROP TABLE public.pg_dist_partition;
DROP TABLE public.pg_dist_shard;
DROP TABLE public.pg_dist_shard_placement;
DROP TABLE public.pg_dist_node;
DROP TABLE public.pg_dist_local_group;
DROP TABLE public.pg_dist_transaction;
DROP TABLE public.pg_dist_colocation;
```

10. Restart sequences.

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

SELECT setval('pg_catalog.pg_dist_groupid_seq', (SELECT MAX(groupid)+1 AS␣
↪max_group_id FROM pg_dist_node), false);

SELECT setval('pg_catalog.pg_dist_node_nodeid_seq', (SELECT MAX(nodeid)+1 AS␣
↪max_node_id FROM pg_dist_node), false);

SELECT setval('pg_catalog.pg_dist_shard_placement_placementid_seq', (SELECT␣
↪MAX(placementid)+1 AS max_placement_id FROM pg_dist_shard_placement),␣
↪false);

SELECT setval('pg_catalog.pg_dist_colocationid_seq', (SELECT␣
↪MAX(colocationid)+1 AS max_colocation_id FROM pg_dist_colocation), false);
```

11. Register triggers.

```
CREATE OR REPLACE FUNCTION create_truncate_trigger(table_name regclass)␣
↪RETURNS void LANGUAGE plpgsql as $$
DECLARE
  command  text;
  trigger_name text;

BEGIN
  trigger_name := 'truncate_trigger_' || table_name::oid;
  command := 'create trigger ' || trigger_name || ' after truncate on ' ||␣
↪table_name || ' execute procedure pg_catalog.citus_truncate_trigger()';
  execute command;
  command := 'update pg_trigger set tgisinternal = true where tgname
 = ' || quote_literal(trigger_name);
  execute command;
END;
$$;

SELECT create_truncate_trigger(logicalrelid) FROM pg_dist_partition ;

DROP FUNCTION create_truncate_trigger(regclass);
```

12. Set dependencies.

```
INSERT INTO
  pg_depend
SELECT
  'pg_class'::regclass::oid as classid,
  p.logicalrelid::regclass::oid as objid,
  0 as objsubid,
  'pg_extension'::regclass::oid as refclassid,
  (select oid from pg_extension where extname = 'citus') as refobjid,
  0 as refobjsubid ,
  'n' as deptype
FROM
  pg_dist_partition p;
```

34.2.2 On Worker Nodes

1. Install Citus 6.0 onto existing PostgreSQL 9.5 server as outlined in *Major and Minor Version Upgrades*.
2. Stop the old and new servers.
3. Check upgrade compatibility to PostgreSQL 9.6.

```
$NEW_PG_PATH/bin/pg_upgrade -b $OLD_PG_PATH/bin/ -B $NEW_PG_PATH/bin/ \
                            -d $OLD_PG_DATA -D $NEW_PG_DATA --check
```

   You should see a "Clusters are compatible" message. If you do not, fix any errors before proceeding. Please ensure that

   - `NEW_PG_DATA` contains an empty database initialized by new PostgreSQL version
   - The Citus extension **IS NOT** created

4. Perform the upgrade (like before but without the `--check` option).

```
$NEW_PG_PATH/bin/pg_upgrade -b $OLD_PG_PATH/bin/ -B $NEW_PG_PATH/bin/ \
                            -d $OLD_PG_DATA -D $NEW_PG_DATA
```

5. Start the new server.

CHAPTER 35

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 36

SQL Workarounds

Before attempting workarounds consider whether Citus is appropriate for your situation. Citus' current version works well for *real-time analytics and multi-tenant use cases.*

Citus supports all SQL statements in the multi-tenant use-case. For real-time analytics use-cases, with queries which span across nodes, Citus supports a subset of SQL statements. We are continuously working to increase SQL coverage to better support other use-cases such as data warehousing queries. Also many of the unsupported features have workarounds; below are a number of the most useful.

36.1 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
 )
 group by page_id;
```

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

36.1.1 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;
```

36.1.2 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)
  from visits
 where page_id in (2,3,5,7,13)
group by page_id
```

36.2 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;
```

36.3 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
```

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. Suppose we want to join tables *here* and *there*, where *there* is already distributed but *here* is on the coordinator database.

```
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.

36.4 Data Warehousing Queries

When queries have restrictive filters (i.e. when very few results need to be transferred to the coordinator) 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 coordinator. Then the next step can be executed on the coordinator 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 37

Citus Utility Function 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.

37.1 Table and Shard DDL

37.1.1 create_distributed_table

The create_distributed_table() function is used to define a distributed table and create its shards if it's 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. If the table contains any rows, they are automatically distributed to worker nodes.

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'.

colocate_with: (Optional) include current table in the co-location group of another table. By default tables are co-located when they are distributed by columns of the same type, have the same shard count, and have the same replication factor. Possible values for `colocate_with` are `default`, `none` to start a new co-location group, or the name of another table to co-locate with that table. (See *Co-Locating Tables*.)

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');

-- alternatively, to be more explicit:
SELECT create_distributed_table('github_events', 'repo_id',
                                colocate_with => 'github_repo');
```

For more examples, see *Creating and Modifying Distributed Tables (DDL)*.

37.1.2 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, replicated to every worker node. 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');
```

37.1.3 upgrade_to_reference_table

The upgrade_to_reference_table() function takes an existing distributed table which has a shard count of one, and upgrades it to be a recognized reference table. After calling this function, the table will be as if it had been created with *create_reference_table*.

Arguments

table_name: Name of the distributed table (having shard count = 1) which will be distributed as a reference table.

Return Value

N/A

Example

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

```
SELECT upgrade_to_reference_table('nation');
```

37.1.4 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');
```

37.1.5 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 coordinator.

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);
```

37.1.6 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 coordinator 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)
```

37.2 Table and Shard DML

37.2.1 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)
```

37.2.2 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 coordinator. 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)
```

37.2.3 master_modify_multiple_shards

The master_modify_multiple_shards() function is used to run data modification statements which could span multiple shards. 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');
```

37.3 Metadata / Configuration Information

37.3.1 master_add_node

The master_add_node() function registers a new node addition in the cluster in the Citus metadata table pg_dist_node. It also copies reference tables to the new 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 | isactive
--------+---------+----------+----------+----------+-------------+----------
      7 |       7 | new-node |    12345 | default  | f           | t
(1 row)
```

37.3.2 master_add_inactive_node

The `master_add_inactive_node` function, similar to *master_add_node*, registers a new node in `pg_dist_node`. However it marks the new node as inactive, meaning no shards will be placed there. Also it does *not* copy reference tables to the new 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_inactive_node('new-node', 12345);
 nodeid | groupid | nodename | nodeport | noderack | hasmetadata | isactive
--------+---------+----------+----------+----------+-------------+----------
      7 |       7 | new-node |    12345 | default  | f           | f
(1 row)
```

37.3.3 master_activate_node

The `master_activate_node` function marks a node as active in the Citus metadata table `pg_dist_node` and copies reference tables to the node. Useful for nodes added via *master_add_inactive_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_activate_node('new-node', 12345);
 nodeid | groupid | nodename | nodeport | noderack | hasmetadata | isactive
--------+---------+----------+----------+----------+-------------+----------
      7 |       7 | new-node |    12345 | default  | f           | t
(1 row)
```

37.3.4 master_disable_node

The `master_disable_node` function is the opposite of `master_activate_node`. It marks a node as inactive in the Citus metadata table `pg_dist_node`, removing it from the cluster temporarily. The function also deletes all reference table placements from the disabled node. To reactivate the node, just run `master_activate_node` again.

Arguments

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

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

Return Value

N/A

Example

```
select * from master_disable_node('new-node', 12345);
```

37.3.5 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)
```

37.3.6 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

(3 rows)
```

37.3.7 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)
```

37.3.8 get_shard_id_for_distribution_column

Citus assigns every row of a distributed table to a shard based on the value of the row's distribution column and the table's method of distribution. In most cases the precise mapping is a low-level detail that the database administrator can ignore. However it can be useful to determine a row's shard, either for manual database maintenance tasks or just to satisfy curiosity. The `get_shard_id_for_distribution_column` function provides this info for hash- and range-distributed tables as well as reference tables. It does not work for the append distribution.

Arguments

table_name: The distributed table.

distribution_value: The value of the distribution column.

Return Value

The shard id Citus associates with the distribution column value for the given table.

Example

```
SELECT get_shard_id_for_distribution_column('my_table', 4);

 get_shard_id_for_distribution_column
--------------------------------------
                               540007
(1 row)
```

37.3.9 column_to_column_name

Translates the `partkey` column of `pg_dist_partition` into a textual column name. This is useful to determine the distribution column of a distributed table.

For a more detailed discussion, see *Finding the distribution column for a table*.

Arguments

table_name: The distributed table.

column_var_text: The value of `partkey` in the `pg_dist_partition` table.

Return Value

The name of `table_name`'s distribution column.

Example

```
-- get distribution column name for products table

SELECT column_to_column_name(logicalrelid, partkey) AS dist_col_name
  FROM pg_dist_partition
 WHERE logicalrelid='products'::regclass;
```

Output:

```
----------
| dist_col_name |
----------
| company_id    |
----------
```

37.3.10 citus_relation_size

Get the disk space used by all the shards of the specified distributed table. This includes the size of the "main fork," but excludes the visibility map and free space map for the shards.

Arguments

logicalrelid: the name of a distributed table.

Return Value

Size in bytes as a bigint.

Example

```
SELECT pg_size_pretty(citus_relation_size('github_events'));
```

```
pg_size_pretty
--------------
23 MB
```

37.3.11 citus_table_size

Get the disk space used by all the shards of the specified distributed table, excluding indexes (but including TOAST, free space map, and visibility map).

Arguments

logicalrelid: the name of a distributed table.

Return Value

Size in bytes as a bigint.

Example

```
SELECT pg_size_pretty(citus_table_size('github_events'));
```

```
pg_size_pretty
--------------
37 MB
```

37.3.12 citus_total_relation_size

Get the total disk space used by the all the shards of the specified distributed table, including all indexes and TOAST data.

Arguments

logicalrelid: the name of a distributed table.

Return Value

Size in bytes as a bigint.

Example

```
SELECT pg_size_pretty(citus_total_relation_size('github_events'));
```

```
pg_size_pretty
--------------
73 MB
```

37.4 Cluster Management And Repair Functions

37.4.1 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 coordinator. 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);
```

37.4.2 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 $\pm 10\%$. Specifically, the shard rebalancer will try to converge utilization of all worker nodes to the (1 - threshold) * average_utilization ... (1 + threshold) * average_utilization range.

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

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

Return Value

N/A

Example

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

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

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

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

37.4.3 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);
```

37.4.4 isolate_tenant_to_new_shard

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

This function creates a new shard to hold rows with a specific single value in the distribution column. It is especially handy for the multi-tenant Citus use case, where a large tenant can be placed alone on its own shard and ultimately its own physical node.

For a more in-depth discussion, see *Tenant Isolation*.

Arguments

table_name: The name of the table to get a new shard.

tenant_id: The value of the distribution column which will be assigned to the new shard.

cascade_option: (Optional) When set to "CASCADE," also isolates a shard from all tables in the current table's *Co-Locating Tables*.

Return Value

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

Examples

Create a new shard to hold the lineitems for tenant 135:

```
SELECT isolate_tenant_to_new_shard('lineitem', 135);
```

```
--------------------
| isolate_tenant_to_new_shard |
--------------------
|                      102240 |
--------------------
```

CHAPTER 38

Metadata Tables Reference

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

38.1 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	Co-location group to which this table belongs. Tables in the same group allow co-located 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
---------------+------------+------------------------------------------------------------------
↪----------------------------------------------------------------------+--------------+-----
↪-----
 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)
```

38.2 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)
```

38.2.1 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)

38.3 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
```

38.3.1 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.

38.4 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.
isactive	boolean	Whether the node is active accepting shard placements.

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

38.5 Co-location group table

The pg_dist_colocation table contains information about which tables' shards should be placed together, or *co-located*. When two tables are in the same co-location 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 co-location is inferred when the shard counts, replication factors, and partition column types all match between two

tables; however, a custom co-location group may be specified when creating a distributed table, if so desired.

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

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

CHAPTER 39

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.

39.1 General configuration

39.1.1 citus.max_worker_nodes_tracked (integer)

Citus tracks worker nodes' locations and their membership in a shared hash table on the coordinator 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 coordinator node.

39.1.2 citus.enable_version_checks (bool)

Upgrading Citus version requires a server restart (to pick up the new shared-library), as well as running an ALTER EXTENSION UPDATE command. The failure to execute both steps could potentially cause errors or crashes. Citus thus validates the version of the code and that of the extension match, and errors out if they don't.

This value defaults to true, and is effective on the coordinator. In rare cases, complex upgrade processes may require setting this parameter to false, thus disabling the check.

39.2 Data Loading

39.2.1 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.

39.2.2 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 1. This parameter can be set at run-time and is effective on the coordinator. 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.

39.2.3 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 coordinator.

39.2.4 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 coordinator.

39.3 Planner Configuration

39.3.1 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 coordinator.

39.3.2 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 coordinator.

39.3.3 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 coordinator.

39.3.4 citus.task_assignment_policy (enum)

Sets the policy to use when assigning tasks to workers. The coordinator 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 coordinator.

39.4 Intermediate Data Transfer Format

39.4.1 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.

39.4.2 citus.binary_master_copy_format (boolean)

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

39.5 DDL

39.5.1 citus.enable_ddl_propagation (boolean)

Specifies whether to automatically propagate DDL changes from the coordinator 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: For a list of DDL propagation support, see *Modifying Tables*.

39.6 Executor Configuration

39.6.1 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 coordinator.

39.6.2 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 coordinator 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 coordinator and can be set at runtime.

39.6.3 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 co-located 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 coordinator. For more details about the executors, you can visit the *Distributed Query Executor* section of our documentation.

39.6.4 citus.multi_task_query_log_level (enum)

Sets a log-level for any query which generates more than one task (i.e. which hits more than one shard). This is useful during a multi-tenant application migration, as you can choose to error or warn for such queries, to find them and add a tenant_id filter to them. This parameter can be set at runtime and is effective on the coordinator. The default value for this parameter is 'off'.

The supported values for this enum are:

- **off:** Turn off logging any queries which generate multiple tasks (i.e. span multiple shards)
- **debug:** Logs statement at DEBUG severity level.
- **log:** Logs statement at LOG severity level.
- **notice:** Logs statement at NOTICE severity level.
- **warning:** Logs statement at WARNING severity level.
- **error:** Logs statement at ERROR severity level.

Note that it may be useful to use `error` or `warning` during testing, and a lower log-level like `notice` or `log` during actual production deployment.

39.6.5 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 coordinator, 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 coordinator, 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.

39.6.6 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 coordinator 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 coordinator.

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.

39.6.7 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 40

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 co-located joins and for queries involving equality based filters on the distribution column.

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

40.1 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.

40.2 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 *Citus Utility Function 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.

40.3 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 *Citus Utility Function Reference* section of our documentation.

40.4 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;
```

40.5 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.

40.5.1 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 coordinator 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 coordinator 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 'coordinator-host')
```

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 one. Set it to two if you want Citus to replicate data automatically and provide fault tolerance. You may want to increase the factor even higher 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 coordinator 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.

40.5.2 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 *Citus Utility Function Reference* section of the documentation.

40.5.3 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.

40.6 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.

40.6.1 Coordinator 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', 'coordinator-
↪host', 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 *Citus Utility Function Reference* section of the documentation.

40.6.2 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 coordinator 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 coordinator, except the initial parsing is not bottlenecked on the coordinator.

```
psql -h worker-host-n -c "\COPY events FROM 'data.csv' WITH (FORMAT CSV, MASTER_HOST
↪'coordinator-host')"
```

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 coordinator. An example of staging a file via a worker using psql is as follows:

```
stage_table=$(psql -tA -h worker-host-n -c "SELECT 'stage_'||nextval('stage_id_
↪sequence')")
psql -h worker-host-n -c "CREATE TABLE $stage_table (time timestamp, data jsonb)"
psql -h worker-host-n -c "\COPY $stage_table FROM 'data.csv' WITH CSV"
psql -h coordinator-host -c "SELECT master_append_table_to_shard(choose_underutilized_
↪shard(), '$stage_table', 'worker-host-n', 5432)"
psql -h worker-host-n -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.

40.6.3 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
(
    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 41

Frequently Asked Questions

41.1 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.

41.2 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.

41.3 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.

41.4 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*.

41.5 How does Citus handle failover of the coordinator node?

As the Citus coordinator node is similar to a standard PostgreSQL server, regular PostgreSQL synchronous replication and failover can be used to provide higher availability of the coordinator node. Many of our customers use synchronous replication in this way to add resilience against coordinator node failure. You can find more information about handling *Coordinator Node Failures*.

41.6 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 *co-located*.

If your tables are not co-located, 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.

41.7 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 can distribute the small tables as "reference tables." This creates a single shard replicated across all worker nodes. 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, see *Reference Tables* and our *Joins* documentation.

41.8 Are there any PostgreSQL features not supported by Citus?

Since Citus provides distributed functionality by extending PostgreSQL, it uses the standard PostgreSQL SQL constructs. It provides full SQL support for queries which access a single node in the database cluster. These queries are common, for instance, in multi-tenant applications where different nodes store different tenants (see *When to Use Citus*).

Other queries which, by contrast, combine data from multiple nodes, do not support the entire spectrum of PostgreSQL features. However they still enjoy broad SQL coverage, including semi-structured data types (like jsonb, hstore), full text search, operators, functions, and foreign data wrappers. Note that the following constructs aren't supported natively for cross-node queries:

- Window Functions
- CTEs
- Set operations
- Transactional semantics for queries that span across multiple shards

41.9 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.

41.10 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*.

41.11 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.

41.12 Which shard contains data for a particular tenant?

Citus provides UDFs and metadata tables to determine the mapping of a distribution column value to a particular shard, and the shard placement on a worker node. See *Finding which shard contains data for a specific tenant* for more details.

41.13 I forgot the distribution column of a table, how do I find it?

The Citus coordinator node metadata tables contain this information. See *Finding the distribution column for a table*.

41.14 Why does pg_relation_size report zero bytes for a distributed table?

The data in distributed tables lives on the worker nodes (in shards), not on the coordinator. A true measure of distributed table size is obtained as a sum of shard sizes. Citus provides helper functions to query this information. See *Determining Table and Relation Size* to learn more.

41.15 Can I run Citus on Heroku or Amazon RDS?

At this time Heroku and Amazon do not support running Citus directly on top of Heroku PostgreSQL or Amazon RDS. It is up to them if/when they enable the Citus extension. If you are looking for something similar, Citus Cloud is our

database-as-a-service which we fully manage for you. It runs on top of AWS (like both RDS and Heroku PostgreSQL) and should provide a very similar product experience, with the addition of Citus' horizontal scaling.

41.16 Can I shard by schema on Citus for multi-tenant applications?

It turns out that while storing each tenant's information in a separate schema can be an attractive way to start when dealing with tenants, it leads to problems down the road. In Citus we partition by the tenant_id, and a shard can contain data from several tenants. To learn more about the reason for this design, see our article Lessons learned from PostgreSQL schema sharding.

41.17 How does cstore_fdw work with Citus?

Citus treats cstore_fdw tables just like regular PostgreSQL tables. When cstore_fdw is used with Citus, each logical shard is created as a foreign cstore_fdw table instead of a regular PostgreSQL table. If your cstore_fdw use case is suitable for the distributed nature of Citus (e.g. large dataset archival and reporting), the two can be used to provide a powerful tool which combines query parallelization, seamless sharding and HA benefits of Citus with superior compression and I/O utilization of cstore_fdw.

41.18 What happened to pg_shard?

The pg_shard extension is deprecated and no longer supported.

Starting with the open-source release of Citus v5.x, pg_shard's codebase has been merged into Citus to offer you a unified solution which provides the advanced distributed query planning previously only enjoyed by CitusDB customers while preserving the simple and transparent sharding and real-time writes and reads pg_shard brought to the PostgreSQL ecosystem. Our flagship product, Citus, provides a superset of the functionality of pg_shard and we have migration steps to help existing users to perform a drop-in replacement. Please contact us for more information.