
Citus Documentation

Release 8.0

Citus Data

Dec 20, 2018

1	What is Citus?	3
1.1	How Far Can Citus Scale?	3
2	When to Use Citus	5
2.1	Multi-Tenant Database	5
2.2	Real-Time Analytics	5
2.3	Considerations for Use	6
2.4	When Citus is Inappropriate	6
3	Quick Tutorials	7
3.1	Multi-tenant Applications	7
3.2	Real-time Analytics	10
4	Single-Machine Cluster	13
4.1	Docker (Mac or Linux)	13
4.2	Ubuntu or Debian	14
4.3	Fedora, CentOS, or Red Hat	16
5	Multi-Machine Cluster	19
5.1	Ubuntu or Debian	19
5.2	Fedora, CentOS, or Red Hat	21
5.3	AWS CloudFormation	22
6	Deploy on Citus Cloud	27
7	Multi-tenant Applications	29
7.1	Let's Make an App – Ad Analytics	30
7.2	Scaling the Relational Data Model	31
7.3	Preparing Tables and Ingesting Data	32
7.4	Integrating Applications	35
7.5	Sharing Data Between Tenants	36
7.6	Online Changes to the Schema	37
7.7	When Data Differs Across Tenants	37
7.8	Scaling Hardware Resources	38
7.9	Dealing with Big Tenants	39
7.10	Where to Go From Here	41
8	Real-Time Dashboards	43
8.1	Data Model	43

8.2	Rollups	45
8.3	Expiring Old Data	47
8.4	Approximate Distinct Counts	47
8.5	Unstructured Data with JSONB	48
9	Timeseries Data	51
9.1	Scaling Timeseries Data on Citus	52
9.2	Automating Partition Creation	53
10	Concepts	57
10.1	Nodes	57
10.2	Distributed Data	57
11	Citus MX	61
11.1	Data Access	62
11.2	Scaling Out a Raw Events Table	63
11.3	MX Limitations	64
12	Determining Application Type	65
12.1	At a Glance	65
12.2	Examples and Characteristics	65
13	Choosing Distribution Column	67
13.1	Multi-Tenant Apps	67
13.2	Real-Time Apps	68
13.3	Timeseries Data	69
13.4	Table Co-Location	69
14	Migrating an Existing App	75
14.1	Identify Distribution Strategy	75
14.2	Prepare Tables for Migration	76
14.3	Prepare Application for Migration	79
14.4	Migrate Data	81
15	SQL Reference	85
15.1	Creating and Modifying Distributed Tables (DDL)	85
15.2	Ingesting, Modifying Data (DML)	91
15.3	Caching Aggregations with Rollups	92
15.4	Querying Distributed Tables (SQL)	95
15.5	Query Processing	100
15.6	Manual Query Propagation	105
15.7	SQL Support and Workarounds	108
16	Citus API	111
16.1	Citus Utility Functions	111
16.2	Citus Tables and Views	131
16.3	Configuration Reference	143
16.4	Append Distribution	151
17	External Integrations	159
17.1	Ingesting Data from Kafka	159
17.2	Ingesting Data from Spark	161
17.3	Business Intelligence with Tableau	164
18	Get Started	169
18.1	Provisioning	169

18.2	Connecting	171
19	Manage	173
19.1	Scaling	173
19.2	Monitoring	176
19.3	Security	185
19.4	Backup, Availability, and Replication	188
19.5	Upgrades	192
19.6	Logging	193
20	Additional Features	197
20.1	Extensions	197
20.2	Forking	198
20.3	Followers	200
20.4	Custom PostgreSQL Configuration	201
21	Support and Billing	203
21.1	Support	203
21.2	Billing	203
22	Cluster Management	205
22.1	Choosing Cluster Size	205
22.2	Initial Hardware Size	206
22.3	Scaling the cluster	206
22.4	Dealing With Node Failures	209
22.5	Tenant Isolation	209
22.6	Viewing Query Statistics	211
22.7	Resource Conservation	213
22.8	Security	213
22.9	PostgreSQL extensions	216
22.10	Creating a New Database	216
22.11	Checks For Updates and Cluster Statistics	217
23	Table Management	219
23.1	Determining Table and Relation Size	219
23.2	Vacuuming Distributed Tables	220
23.3	Analyzing Distributed Tables	220
24	Upgrading Citus	221
24.1	Upgrading Citus Versions	221
24.2	Upgrading PostgreSQL version from 10 to 11	222
25	Query Performance Tuning	227
25.1	Table Distribution and Shards	227
25.2	PostgreSQL tuning	227
25.3	Scaling Out Performance	230
25.4	Distributed Query Performance Tuning	230
25.5	Scaling Out Data Ingestion	233
26	Useful Diagnostic Queries	237
26.1	Finding which shard contains data for a specific tenant	237
26.2	Finding the distribution column for a table	237
26.3	Detecting locks	238
26.4	Querying the size of your shards	239
26.5	Querying the size of all distributed tables	240

26.6	Determining Replication Factor per Table	240
26.7	Identifying unused indices	240
26.8	Monitoring client connection count	241
26.9	Index hit rate	242
27	Common Error Messages	243
27.1	Failed to execute task <i>n</i>	243
27.2	Relation <i>foo</i> is not distributed	244
27.3	Could not receive query results	244
27.4	Canceling the transaction since it was involved in a distributed deadlock	244
27.5	Cannot establish a new connection for placement <i>n</i> , since DML has been executed on a connection that is in use	245
27.6	Could not connect to server: Cannot assign requested address	246
27.7	Could not connect to any active placements	246
27.8	Remaining connection slots are reserved for non-replication superuser connections	246
27.9	PgBouncer cannot connect to server	247
27.10	Unsupported clause type	247
27.11	Cannot open new connections after the first modification command within a transaction	247
27.12	ON CONFLICT is not supported via coordinator	248
27.13	Cannot create uniqueness constraint	248
27.14	Function create_distributed_table does not exist	249
27.15	STABLE functions used in UPDATE queries cannot be called with column references	249
28	Frequently Asked Questions	251
28.1	Can I create primary keys on distributed tables?	251
28.2	How do I add nodes to an existing Citus cluster?	251
28.3	How does Citus handle failure of a worker node?	251
28.4	How does Citus handle failover of the coordinator node?	252
28.5	How do I ingest the results of a query into a distributed table?	252
28.6	Can I join distributed and non-distributed tables together in the same query?	252
28.7	Are there any PostgreSQL features not supported by Citus?	252
28.8	How do I choose the shard count when I hash-partition my data?	253
28.9	How do I change the shard count for a hash partitioned table?	253
28.10	How does citus support count(distinct) queries?	253
28.11	In which situations are uniqueness constraints supported on distributed tables?	253
28.12	How do I create database roles, functions, extensions etc in a Citus cluster?	254
28.13	What if a worker node's address changes?	254
28.14	Which shard contains data for a particular tenant?	254
28.15	I forgot the distribution column of a table, how do I find it?	254
28.16	Can I distribute a table by multiple keys?	255
28.17	Why does pg_relation_size report zero bytes for a distributed table?	255
28.18	Why am I seeing an error about max_intermediate_result_size?	255
28.19	Can I run Citus on Heroku?	255
28.20	Can I run Citus on Amazon RDS?	255
28.21	Can I use Citus with my existing AWS account?	256
28.22	Can I shard by schema on Citus for multi-tenant applications?	256
28.23	How does cstore_fdw work with Citus?	256
28.24	What happened to pg_shard?	256
29	Related Articles	257
29.1	Introducing Citus Add-on for Heroku: Scale out your Postgres	257
29.2	Efficient Rollup Tables with HyperLogLog in Postgres	259
29.3	Distributed Distinct Count with HyperLogLog on Postgres	264
29.4	How to Scale PostgreSQL on AWS: Learnings from Citus Cloud	267

29.5	Postgres Parallel Indexing in Citus	275
29.6	Real-time Event Aggregation at Scale Using Postgres with Citus	277
29.7	How Distributed Outer Joins on PostgreSQL with Citus Work	281
29.8	Designing your SaaS Database for Scale with Postgres	286
29.9	Building a Scalable Postgres Metrics Backend using the Citus Extension	289
29.10	Sharding a Multi-Tenant App with Postgres	291
29.11	Sharding Postgres with Semi-Structured Data and Its Performance Implications	295
29.12	Scalable Real-time Product Search using PostgreSQL with Citus	297

Welcome to the documentation for Citus 8.0! Citus horizontally scales PostgreSQL across commodity servers using sharding and replication. Its query engine parallelizes incoming SQL queries across these servers to enable real-time responses on large datasets.

What is Citus?

Citus is basically [worry-free Postgres](#) that is built to scale out. It's an extension to Postgres that *distributes data* and queries in a cluster of multiple machines. As an extension (rather than a fork), Citus supports new PostgreSQL releases, allowing users to benefit from new features while maintaining compatibility with existing PostgreSQL tools.

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.

Available in Three Ways:

1. As [open source](#) to add to existing Postgres servers
2. On-premise with additional [enterprise grade](#) security and cluster management tools
3. As a fully-managed database as a service, called [Citus Cloud](#)

How Far Can Citus Scale?

Citus scales horizontally by adding worker nodes, vertically by upgrading workers/coordinator, and supports switching to *Citus MX* mode if needed. In practice our customers have achieved the following scale, with room to grow even more:

- [Algolia](#)
 - 5-10B rows ingested per day
- [Heap](#)
 - 500+ billion events
 - 900TB of data on a 40-node Citus database cluster
- [Chartbeat](#)
 - >2.6B rows of data added per month
- [Pex](#)
 - 30B rows updated/day
 - 20-node Citus database cluster on Google Cloud
 - 2.4TB memory, 1280 cores, and 60TB of data
 - ...with plans to grow to 45 nodes
- [Mixrank](#)

- 1.6PB of time series data

For more customers and statistics, see our [customer stories](#).

When to Use Citus

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.

Some advantages of Citus for multi-tenant applications:

- Fast queries for all tenants
- Sharding logic in the database, not the application
- Hold more data than possible in single-node PostgreSQL
- Scale out without giving up SQL
- Maintain performance under high concurrency
- Fast metrics analysis across customer base
- Easily scale to handle new customer signups
- Isolate resource usage of large and small customers

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. Some advantages of Citus for real-time applications:

- Maintain sub-second responses as the dataset grows
- Analyze new events and new data as it happens, in real-time
- Parallelize SQL queries
- Scale out without giving up SQL
- Maintain performance under high concurrency
- Fast responses to dashboard queries
- Use one database, not a patchwork
- Rich PostgreSQL data types and extensions

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 described here and you happen to run into an unsupported tool or query, then there's usually a good workaround.

When Citus is Inappropriate

Some workloads don't need a powerful distributed database, while others require a large flow of information between worker nodes. In the first case Citus is unnecessary, and in the second not generally performant. Here are some examples:

- When single-node Postgres can support your application and you do not expect to grow
- Offline analytics, without the need for real-time ingest nor real-time queries
- Analytics apps that do not need to support a large number of concurrent users
- Queries that return data-heavy ETL results rather than summaries

Quick Tutorials

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 \(Mac or Linux\)](#).
-

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:.
```

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 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);
```

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 Citrus 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
```

Running queries

Now that we have loaded data into the tables, let's go ahead and run some queries. Citrus 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 [Multi-Tenant Apps](#) section to see how you can model your own data for multi-tenancy.

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 \(Mac or Linux\)](#).
-

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:..
```

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);
```

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
```

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 = 'nolyouknow' WHERE user_id = 24305673;
```

With this, we come to the end of our tutorial. As a next step, you can look at the [Real-Time Apps](#) section to see how you can model your own data and power real-time analytical applications.

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.

Docker (Mac or Linux)

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

Note: The Docker image is intended for development/testing purposes only, and has not been prepared for production use. The images use default connection settings, which are very permissive, and not suitable for any kind of production setup. These should be updated before using the image for production use. The PostgreSQL manual [explains how](#) to make them more restrictive.

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/1.16.1/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
COMPOSE_PROJECT_NAME=citus docker-compose 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 choose a different port for coordinator service with the `MASTER_EXTERNAL_PORT` environment variable. For example:

```
MASTER_EXTERNAL_PORT=5433 COMPOSE_PROJECT_NAME=citus docker-compose up -d
```

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:

```
COMPOSE_PROJECT_NAME=citus docker-compose down -v
```

Note: Please note that Citus reports anonymous information about your cluster to the Citus Data company servers. To learn more about what information is collected and how to opt out of it, see [Checks For Updates and Cluster Statistics](#).

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 11 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-11-citus-8.0
```

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/11/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.

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.

Note: Please note that Citus reports anonymous information about your cluster to the Citus Data company servers. To learn more about what information is collected and how to opt out of it, see [Checks For Updates and Cluster Statistics](#).

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 11 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 citus80_11
```

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-11/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 Citrus 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 Citrus (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.

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

Note: Please note that Citrus reports anonymous information about your cluster to the Citrus Data company servers. To learn more about what information is collected and how to opt out of it, see [Checks For Updates and Cluster Statistics](#).

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.

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.

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-11-citus-8.0

# preload citus extension
sudo pg_conftool 11 main set shared_preload_libraries citus
```

This installs centralized configuration in `/etc/postgresql/11/main`, and creates a database in `/var/lib/postgresql/11/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 11 main set listen_addresses '*'
```

```
sudo vi /etc/postgresql/11/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 [Increasing 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;"
```

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

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

Note: Please note that Citus reports anonymous information about your cluster to the Citus Data company servers. To learn more about what information is collected and how to opt out of it, see [Checks For Updates and Cluster Statistics](#).

Fedora, CentOS, or Red Hat

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

Steps to be executed on all nodes

1. Add repository

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

2. Install PostgreSQL + Citrus and initialize a database

```
# install PostgreSQL with Citrus extension
sudo yum install -y citus80_11
# initialize system database (using RHEL 6 vs 7 method as necessary)
sudo service postgresql-11 initdb || sudo /usr/pgsql-11/bin/postgresql-11-setup initdb
# preload citus extension
echo "shared_preload_libraries = 'citus'" | sudo tee -a /var/lib/pgsql/11/data/
↪postgresql.conf
```

PostgreSQL adds version-specific binaries in `/usr/pgsql-11/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/11/data/postgresql.conf
```

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

```
sudo vi /var/lib/pgsql/11/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 [Increasing Worker Security](#). The PostgreSQL manual explains [how](#) to make them more restrictive.

4. Start database servers, create Citrus extension

```
# start the db server
sudo service postgresql-11 restart
```

```
# and make it start automatically when computer does
sudo chkconfig postgresql-11 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;"
```

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. The new Citus database is accessible in `psql` through the `postgres` user:

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

Note: Please note that Citus reports anonymous information about your cluster to the Citus Data company servers. To learn more about what information is collected and how to opt out of it, see [Checks For Updates and Cluster Statistics](#).

AWS CloudFormation

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

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

Introduction

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

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

1. Start a Citus cluster

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

2. Select Citus template

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

3. Fill the form with details about your cluster


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


Specify Details


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

Stack name

Parameters

AvailabilityZone1 
Select first availability zone to use

AvailabilityZone2 
Select second availability zone to use

KeyName 
The EC2 Key Pair to allow SSH access to the instances

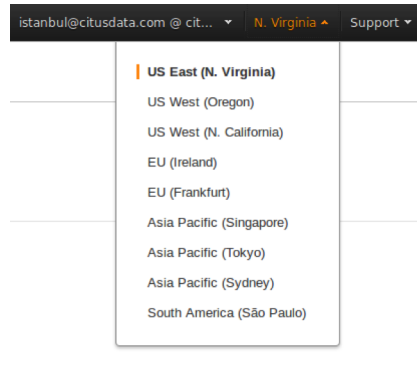
MasterInstanceType  EC2 instance type of the master node

NumWorkers The number of worker instances

WorkerInstanceType  EC2 instance type of the worker nodes

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

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



4. Acknowledge IAM capabilities

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

Finally, you need to acknowledge IAM capabilities, which give the 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.



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

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

☒ I acknowledge that this template might cause AWS CloudFormation to create IAM resources.

Cancel

Previous

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.

Stack Name	Created Time	Status	Description
<input checked="" type="checkbox"/> citusdb-test-cluster	2015-02-18 12:56:26 UTC+0100	CREATE_COMPLETE	Set up a CitusDB cluster

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

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



Troubleshooting:

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

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

6. Login to the cluster

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

Deploy on Citus Cloud

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.

Multi-tenant Applications

Contents

- *Multi-tenant Applications*
 - *Let's Make an App – Ad Analytics*
 - *Scaling the Relational Data Model*
 - *Preparing Tables and Ingesting Data*
 - * *Try it Yourself*
 - *Integrating Applications*
 - *Sharing Data Between Tenants*
 - *Online Changes to the Schema*
 - *When Data Differs Across Tenants*
 - *Scaling Hardware Resources*
 - *Dealing with Big Tenants*
 - *Where to Go From Here*

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 from and invisible to 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 Citrus provide all the tools needed to handle these challenges, so let's get building.

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 Citrus. To see how, we must become familiar with how Citrus distributes data and executes queries.

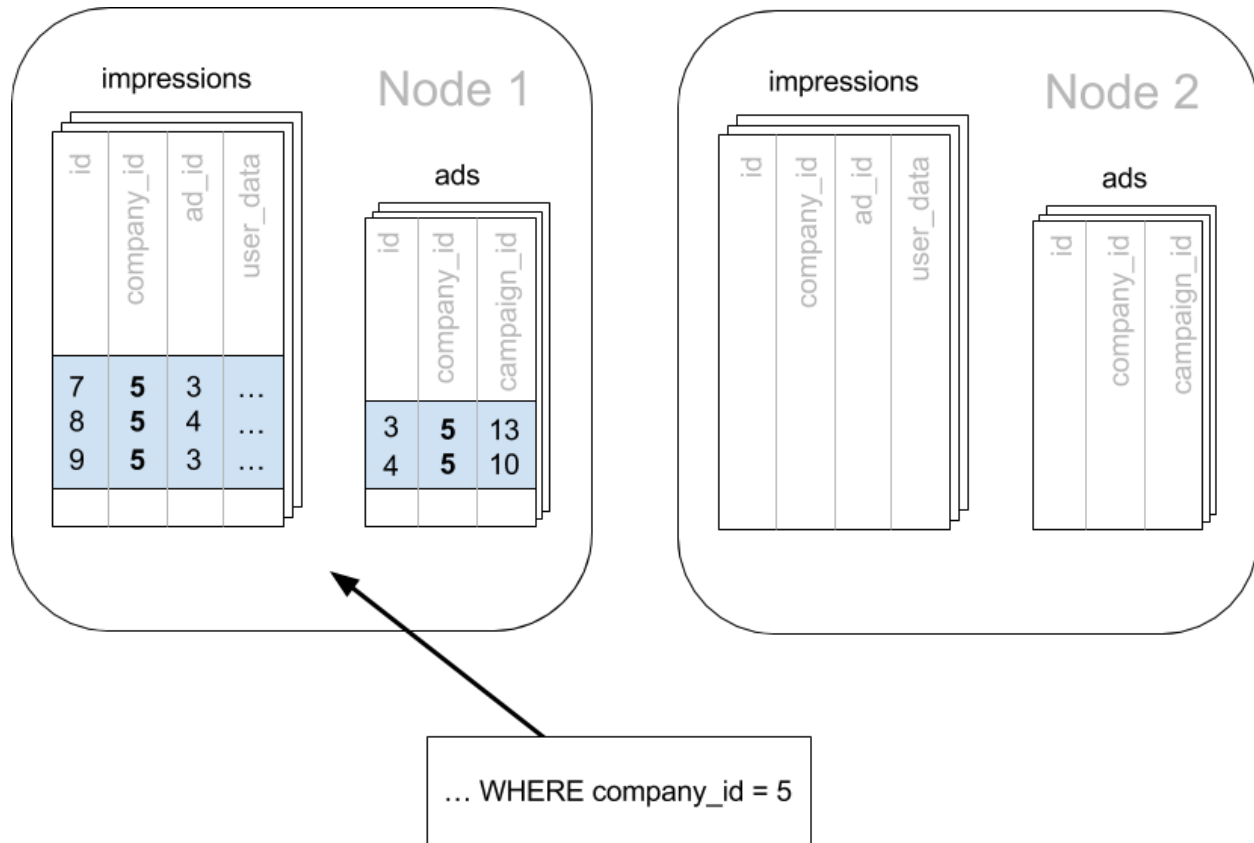
Scaling the Relational Data Model

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 Citrus you can keep your data model *and* make it scale. Citrus 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 Citrus 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 Citrus 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 Citrus to use this column to read and write rows to the same node when the rows are marked for the same company. In Citrus' terminology `company_id` will be the *distribution column*, which you can learn more about in [Distributed Data Modeling](#).

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. Citrus 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](#).

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
- [Sign up](#) for Citus Cloud and provision a cluster.

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
```

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 Citrus 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 Citrus 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, or [django-multitenant](#) for Django which will automatically add these filters to all your queries, even the complicated ones. Check out our migration guides for `rails_migration` and `django_migration`.

This guide is framework-agnostic, so we'll point out some Citrus 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 Citrus:

```
-- 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 Citrus 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
JOIN impressions as i
  ON i.company_id = a.company_id
  AND i.ad_id = a.id
WHERE a.company_id = 5
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.

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 Citrus 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 Citrus 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 addr >> c.user_ip
AND c.company_id = 5
AND c.ad_id = 290;
```

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 Citrus, you can simply use standard PostgreSQL DDL commands to change the schema of your tables, and Citrus 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 Citrus 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](#).

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;
```

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.

Formation Configuration

Total Nodes

2



RAM / vCPU per node

15 GB RAM
2 vCPUs30.5 GB RAM
4 vCPUs61 GB RAM
8 vCPUs122 GB RAM
16 vCPUs244 GB RAM
32 vCPUs

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.

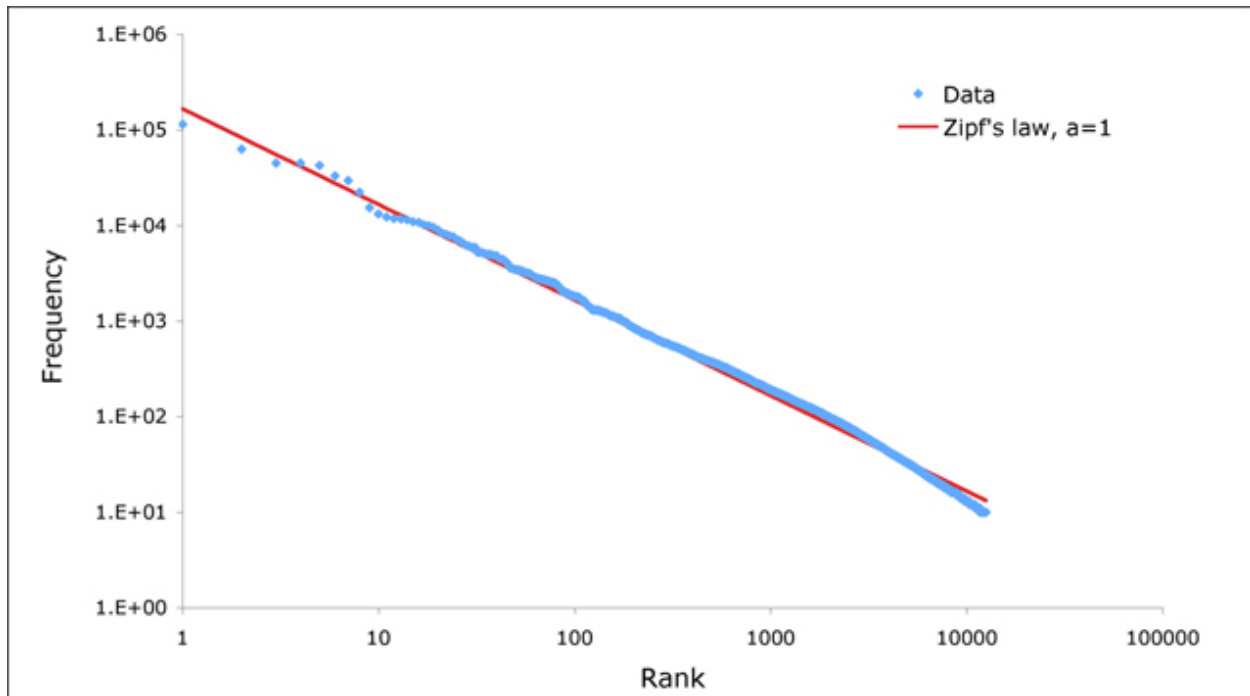
You can learn more about how shard rebalancing works here: [Scaling Out \(adding new nodes\)](#).

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. Citrus 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_placement AS placement,
     pg_dist_node AS node
WHERE placement.groupid = node.groupid
      AND node.noderole = 'primary'
      AND 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_placement` again.

Where to Go From Here

With this, you now know how to use Citrus to power your multi-tenant application for scalability. If you have an existing schema and want to migrate it for Citrus, see [Multi-Tenant Transitioning](#).

To adjust a front-end application, specifically Ruby on Rails or Django, read `rails_migration`. Finally, try [Citrus Cloud](#), the easiest way to manage a Citrus cluster, available with discounted developer plan pricing.

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.

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 Citrus 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: Citrus 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 `citrus.shard_replication_factor` to 2 or higher for fault tolerance.

With this, the system is ready to accept data and serve queries! Keep the following loop running in a `psql` console in the background while you continue with the other commands in this article. It generates fake data every second or two.

```
DO $$
BEGIN LOOP
  INSERT INTO http_request (
    site_id, ingest_time, url, request_country,
    ip_address, status_code, response_time_msec
  ) VALUES (
    trunc(random()*32), clock_timestamp(),
    concat('http://example.com/', md5(random()::text)),
    ('{China,India,USA,Indonesia}'::text[])[ceil(random()*4)],
    concat(
      trunc(random()*250 + 2), '.',
      trunc(random()*250 + 2), '.',
      trunc(random()*250 + 2), '.',
      trunc(random()*250 + 2)
    )::inet,
    ('{200,404}'::int[])[ceil(random()*2)],
    5+trunc(random()*150)
  );
  PERFORM pg_sleep(random() * 0.25);
END LOOP;
END $$;
```

Once you're ingesting data, you can run dashboard queries such as:

```
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 http_request
WHERE date_trunc('minute', ingest_time) > now() - '5 minutes'::interval
GROUP BY site_id, minute
ORDER BY minute ASC;
```

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).

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.

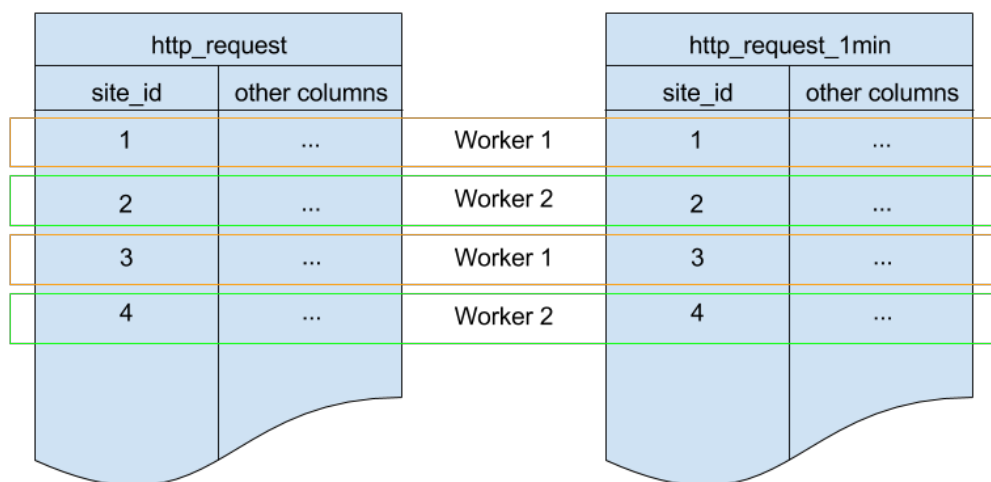
```
CREATE TABLE http_request_1min (
  site_id INT,
  ingest_time TIMESTAMPTZ, -- which minute this row represents

  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');

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 Citrus 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 an `INSERT INTO SELECT`. This is possible because the tables are co-located. The following function wraps the rollup query up for convenience.

```
-- single-row table to store when we rolled up last
CREATE TABLE latest_rollup (
    minute timestampz PRIMARY KEY,

    -- "minute" should be no more precise than a minute
    CHECK (minute = date_trunc('minute', minute))
);

-- initialize to a time long ago
INSERT INTO latest_rollup VALUES ('10-10-1901');

-- function to do the rollup
CREATE OR REPLACE FUNCTION rollup_http_request() RETURNS void AS $$
DECLARE
    curr_rollup_time timestampz := date_trunc('minute', now());
    last_rollup_time timestampz := minute from latest_rollup;
BEGIN
    INSERT INTO http_request_1min (
        site_id, ingest_time, request_count,
        success_count, error_count, average_response_time_msec
    ) SELECT
        site_id,
        date_trunc('minute', ingest_time),
        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
    -- roll up only data new since last_rollup_time
    WHERE date_trunc('minute', ingest_time) <@
        tstzrange(last_rollup_time, curr_rollup_time, '[')
    GROUP BY 1, 2;

    -- update the value in latest_rollup so that next time we run the
    -- rollup it will operate on data newer than curr_rollup_time
    UPDATE latest_rollup SET minute = curr_rollup_time;
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 rollup_http_request();'
```

Alternately, an extension such as `pg_cron` allows you to schedule recurring queries directly from the database.

The dashboard query from earlier is now a lot nicer:

```
SELECT site_id, ingest_time as minute, request_count,
       success_count, error_count, average_response_time_msec
FROM http_request_1min
WHERE ingest_time > date_trunc('minute', now()) - '5 minutes'::interval;
```

Expiring Old Data

The rollups make queries faster, but we still need to expire old data to avoid unbounded storage costs. Simply decide how long you'd like to keep data for each granularity, and use standard queries to delete expired data. In the following example, we decided to keep raw data for one day, and per-minute aggregations for one month:

```
DELETE FROM http_request WHERE ingest_time < now() - interval '1 day';
DELETE FROM http_request_1min WHERE ingest_time < now() - interval '1 month';
```

In production you could wrap these queries in a function and call it every minute in a cron job.

Data expiration can go even faster by using table range partitioning on top of Citrus hash distribution. See the [Timeseries Data](#) section for a detailed example.

Those are the basics! 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.

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. However an approximate answer is much more manageable.

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. 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:

```
-----
-- Run on all nodes -----
CREATE EXTENSION hll;
```

Note: This is not necessary on Citrus Cloud, which has HLL already installed, along with other useful [Extensions](#).

Now we're ready to track IP addresses in our rollup with HLL. First add a column to the rollup table.

```
ALTER TABLE http_request_1min ADD COLUMN distinct_ip_addresses hll;
```

Next use our custom aggregation to populate the column. Just add it to the query in our rollup function:

```
@@ -1,10 +1,12 @@
INSERT INTO http_request_1min (
    site_id, ingest_time, request_count,
    success_count, error_count, average_response_time_msec,
+   distinct_ip_addresses
```

```

) SELECT
  site_id,
  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,
+   hll_add_agg(hll_hash_text(ip_address)) AS distinct_ip_addresses
FROM http_request

```

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 site_id, ingest_time as minute, 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 ingest_time > date_trunc('minute', now()) - interval '5 minutes';

```

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 can compute distinct IP counts over a time period with the following query:

```

SELECT hll_cardinality(hll_union_agg(distinct_ip_addresses))
FROM http_request_1min
WHERE ingest_time > date_trunc('minute', now()) - '5 minutes'::interval;

```

You can find more information on HLLs [in the project's GitHub repository](#).

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 modifying the rollup function:

```

@@ -1,14 +1,19 @@
INSERT INTO http_request_1min (
  site_id, ingest_time, request_count,
  success_count, error_count, average_response_time_msec,
+   country_counters
) SELECT
  site_id,
  minute,
  COUNT(1) as request_count,
  SUM(CASE WHEN (status_code between 200 and 299) THEN 1 ELSE 0 END) as success_c

```



```

SUM(CASE WHEN (status_code between 200 and 299) THEN 0 ELSE 1 END) as error_cou
SUM(response_time_msec) / COUNT(1) AS average_response_time_msec,
- FROM http_request
+   jsonb_object_agg(request_country, country_count) AS country_counters
+ FROM (
+   SELECT *,
+     count(1) OVER (
+       PARTITION BY site_id, date_trunc('minute', ingest_time), request_country
+     ) AS country_count
+   FROM http_request
+ ) h

```

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

```

SELECT
  request_count, success_count, error_count, average_response_time_msec,
  COALESCE(country_counters->>'USA', '0')::int AS american_visitors
FROM http_request_1min
WHERE ingest_time > date_trunc('minute', now()) - '5 minutes'::interval;

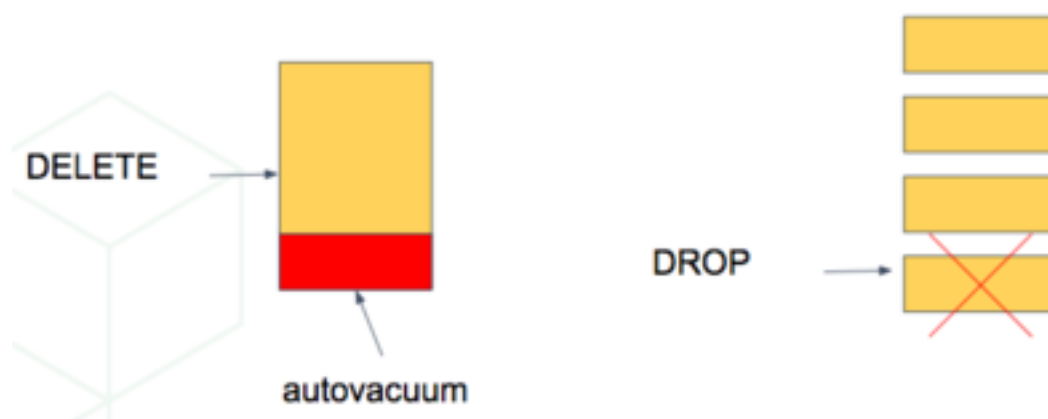
```


Timeseries Data

In a time-series workload, applications (such as some *Real-Time Apps*) query recent information, while archiving old information.

To deal with this workload, a single-node PostgreSQL database would typically use **table partitioning** to break a big table of time-ordered data into multiple inherited tables with each containing different time ranges.

Storing data in multiple physical tables speeds up data expiration. In a single big table, deleting rows incurs the cost of scanning to find which to delete, and then **vacuuming** the emptied space. On the other hand, dropping a partition is a fast operation independent of data size. It's the equivalent of simply removing files on disk that contain the data.



Partitioning a table also makes indices smaller and faster within each date range. Queries operating on recent data are likely to operate on “hot” indices that fit in memory. This speeds up reads.



Also inserts have smaller indices to update, so they go faster too.



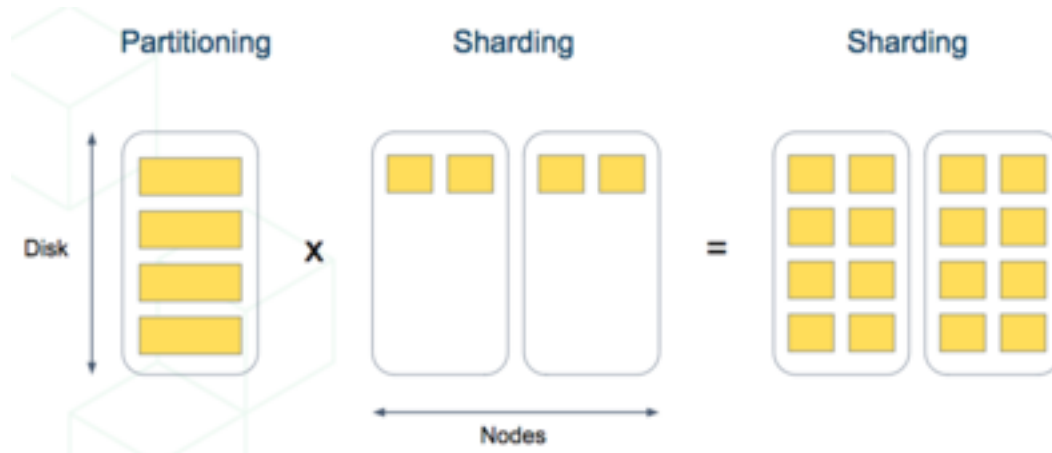
Time-based partitioning makes most sense when:

1. Most queries access a very small subset of the most recent data
2. Older data is periodically expired (deleted/dropped)

Keep in mind that, in the wrong situation, reading all these partitions can hurt overhead more than it helps. However in the right situations it is quite helpful. For example, when keeping a year of time series data and regularly querying only the most recent week.

Scaling Timeseries Data on Citus

We can mix the single-node table partitioning techniques with Citus' distributed sharding to make a scalable time-series database. It's the best of both worlds. It's especially elegant atop the declarative table partitioning in PostgreSQL 10.



For example, let's distribute *and* partition a table holding historical [GitHub events](#) data.

Each record in this GitHub data set represents an event created in GitHub, along with key information regarding the event such as event type, creation date, and the user who created the event.

The first step is to create and partition the table by time as we would in a single-node PostgreSQL database:

```
-- the separate schema will be useful later
CREATE SCHEMA github;

-- declaratively partitioned 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
) PARTITION BY RANGE (created_at);

```

Notice the `PARTITION BY RANGE (created_at)`. This tells Postgres that the table will be partitioned by the `created_at` column in ordered ranges. We have not yet created any partitions for specific ranges, though.

Before creating specific partitions, let's distribute the table in Citrus. We'll shard by `repo_id`, meaning the events will be clustered into shards per repository.

```
SELECT create_distributed_table('github.events', 'repo_id');
```

At this point Citrus has created shards for this table across worker nodes. Internally each shard is a table with the name `github.events_N` for each shard identifier `N`. Also, Citrus propagated the partitioning information, and each of these shards has `Partition key: RANGE (created_at)` declared.

A partitioned table cannot directly contain data, it is more like a view across its partitions. Thus the shards are not yet ready to hold data. We need to manually create partitions and specify their time ranges, after which we can insert data that match the ranges.

```

-- manually make a partition for 2016 events
CREATE TABLE github.events_2016 PARTITION OF github.events
FOR VALUES FROM ('2016-01-01') TO ('2016-12-31');

```

The coordinator node now has the tables `github.events` and `github.events_2016`. Citrus will propagate partition creation to all the shards, creating a partition for each shard.

Automating Partition Creation

In the previous section we manually created a partition of the `github.events` table. It's tedious to keep doing this, especially when using narrower partitions holding less than a year's range of data. It's more pleasant to let the `pg_partman` extension automatically create partitions on demand. The core functionality of `pg_partman` works out of the box with Citrus when using it with native partitioning.

First clone, build, and install the `pg_partman` extension. Then tell partman we want to make partitions that each hold one hour of data. This will create the initial empty hourly partitions:

```

CREATE SCHEMA partman;
CREATE EXTENSION pg_partman WITH SCHEMA partman;

-- Partition the table into hourly ranges of "created_at"
SELECT partman.create_parent('github.events', 'created_at', 'native', 'hourly');
UPDATE partman.part_config SET infinite_time_partitions = true;

```

Running `\d+ github.events` will now show more partitions:

```

\d+ github.events

```

Column	Type	Collation	Nullable	Default	
Storage	Stats target	Description			
event_id	bigint				plain

```

event_type | text | | | | |
↪extended | | | | | |
event_public | boolean | | | | | plain
↪ | | | | | |
repo_id | bigint | | | | | plain
↪ | | | | | |
payload | jsonb | | | | |
↪extended | | | | | |
repo | jsonb | | | | |
↪extended | | | | | |
actor | jsonb | | | | |
↪extended | | | | | |
org | jsonb | | | | |
↪extended | | | | | |
created_at | timestamp without time zone | | | | | plain
↪ | | | | | |
Partition key: RANGE (created_at)
Partitions: github.events_p2018_01_15_0700 FOR VALUES FROM ('2018-01-15 07:00:00') TO
↪ ('2018-01-15 08:00:00'),
github.events_p2018_01_15_0800 FOR VALUES FROM ('2018-01-15 08:00:00') TO
↪ ('2018-01-15 09:00:00'),
github.events_p2018_01_15_0900 FOR VALUES FROM ('2018-01-15 09:00:00') TO
↪ ('2018-01-15 10:00:00'),
github.events_p2018_01_15_1000 FOR VALUES FROM ('2018-01-15 10:00:00') TO
↪ ('2018-01-15 11:00:00'),
github.events_p2018_01_15_1100 FOR VALUES FROM ('2018-01-15 11:00:00') TO
↪ ('2018-01-15 12:00:00'),
github.events_p2018_01_15_1200 FOR VALUES FROM ('2018-01-15 12:00:00') TO
↪ ('2018-01-15 13:00:00'),
github.events_p2018_01_15_1300 FOR VALUES FROM ('2018-01-15 13:00:00') TO
↪ ('2018-01-15 14:00:00'),
github.events_p2018_01_15_1400 FOR VALUES FROM ('2018-01-15 14:00:00') TO
↪ ('2018-01-15 15:00:00'),
github.events_p2018_01_15_1500 FOR VALUES FROM ('2018-01-15 15:00:00') TO
↪ ('2018-01-15 16:00:00')

```

By default `create_parent` creates four partitions in the past, four in the future, and one for the present, all based on system time. If you need to backfill older data, you can specify a `p_start_partition` parameter in the call to `create_parent`, or `p_premake` to make partitions for the future. See the [pg_partman documentation](#) for details.

As time progresses, `pg_partman` will need to do some maintenance to create new partitions and drop old ones. Anytime you want to trigger maintenance, call:

```

-- disabling analyze is recommended for native partitioning
-- due to aggressive locks
SELECT partman.run_maintenance(p_analyze := false);

```

It's best to set up a periodic job to run the maintenance function. `Pg_partman` can be built with support for a background worker (BGW) process to do this. Or we can use another extension like `pg_cron`:

```

SELECT cron.schedule('@hourly', $$
    SELECT partman.run_maintenance(p_analyze := false);
$$);

```

Once periodic maintenance is set up, you no longer have to think about the partitions, they just work.

Finally, to configure `pg_partman` to drop old partitions, you can update the `partman.part_config` table:

```
UPDATE partman.part_config
  SET retention_keep_table = false,
      retention = '1 month'
WHERE parent_table = 'github.events';
```

Now whenever maintenance runs, partitions older than a month are automatically dropped.

Note: Be aware that native partitioning in Postgres is still quite new and has a few quirks. For example, you cannot directly create an index on a partitioned table. Instead, `pg_partman` lets you create a template table to define indexes for new partitions. Maintenance operations on partitioned tables will also acquire aggressive locks that can briefly stall queries. There is currently a lot of work going on within the postgres community to resolve these issues, so expect time partitioning in Postgres to only get better.

Concepts

Nodes

Citus is a PostgreSQL [extension](#) that allows commodity database servers (called *nodes*) to coordinate with one another in a “shared nothing” architecture. The nodes form a *cluster* that allows PostgreSQL to hold more data and use more CPU cores than would be possible on a single computer. This architecture also allows the database to scale by simply adding more nodes to the cluster.

Coordinator and Workers

Every cluster has one special node called the *coordinator* (the others are known as workers). Applications send their queries to the coordinator node which relays it to the relevant workers and accumulates the results.

For each query, the coordinator either *routes* it to a single worker node, or *parallelizes* it across several depending on whether the required data lives on a single node or multiple. The coordinator knows how to do this by consulting its metadata tables. These Citus-specific tables track the DNS names and health of worker nodes, and the distribution of data across nodes. For more information, see our [Citus Tables and Views](#).

Note: [Citus MX](#) removes the need to send all queries through the coordinator node. With MX, users can send queries directly to any worker node, which allows both reads and writes to be scaled even more.

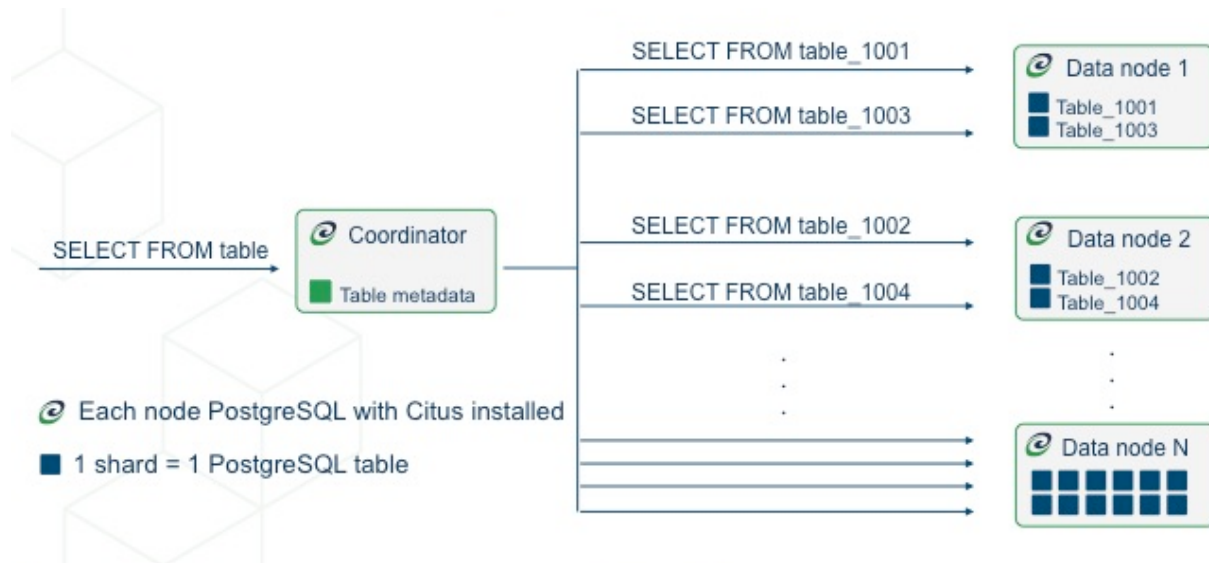
Distributed Data

Table Types

There are three types of tables in a Citus cluster, each used for different purposes.

Type 1: Distributed Tables

The first type, and most common, is *distributed* tables. These appear to be normal tables to SQL statements, but are horizontally *partitioned* across worker nodes.



Here the rows of `table` are stored in tables `table_1001`, `table_1002` etc on the workers. The component worker tables are called *shards*.

Citus runs not only SQL but DDL statements throughout a cluster, so changing the schema of a distributed table cascades to update all the table's shards across workers.

To learn how to create a distributed table, see *Creating and Modifying Distributed Tables (DDL)*.

Distribution Column

Citus uses algorithmic sharding to assign rows to shards. This means the assignment is made deterministically – in our case based on the value of a particular table column called the *distribution column*. The cluster administrator must designate this column when distributing a table. Making the right choice is important for performance and functionality, as described in the general topic of *Distributed Data Modeling*.

Type 2: Reference Tables

A reference table is a type of distributed table whose entire contents are concentrated into a single shard which is replicated on every worker. Thus queries on any worker can access the reference information locally, without the network overhead of requesting rows from another node. Reference tables have no distribution column because there is no need to distinguish separate shards per row.

Reference tables are typically small, and are used to store data that is relevant to queries running on any worker node. For example, enumerated values like order statuses, or product categories.

When interacting with a reference table we automatically perform two-phase commits (2PC) on transactions. This means that Citus makes sure your data is always in a consistent state, regardless of whether you are writing, modifying, or deleting it.

The *Reference Tables* section talks more about these tables and how to create them.

Type 3: Local Tables

When you use Citus, the coordinator node you connect to and interact with is a regular PostgreSQL database with the Citus extension installed. Thus you can create ordinary tables and choose not to shard them. This is useful for small

administrative tables that don't participate in join queries. An example would be users table for application login and authentication.

Creating standard PostgreSQL tables is easy because it's the default. It's what you get when you run CREATE TABLE. In almost every Citus deployment we see standard PostgreSQL tables co-existing with distributed and reference tables. Indeed, Citus itself uses local tables to hold cluster metadata, as mentioned earlier.

Shards

The previous section described a shard as containing a subset of the rows of a distributed table in a smaller table within a worker node. This section gets more into the technical details.

The `pg_dist_shard` metadata table on the coordinator contains a row for each shard of each distributed table in the system. The row matches a shardid with a range of integers in a hash space (shardminvalue, shardmaxvalue):

```
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)
```

If the coordinator node wants to determine which shard holds a row of `github_events`, it hashes the value of the distribution column in the row, and checks which shard's range contains the hashed value. (The ranges are defined so that the image of the hash function is their disjoint union.)

Shard Placements

Suppose that shard 102027 is associated with the row in question. This means the row should be read or written to a table called `github_events_102027` in one of the workers. Which worker? That is determined entirely by the metadata tables, and the mapping of shard to worker is known as the *shard placement*.

Joining some *metadata tables* gives us the answer. These are the types of lookups that the coordinator does to route queries. It rewrites queries into fragments that refer to the specific tables like `github_events_102027`, and runs those fragments on the appropriate workers.

```
SELECT
    shardid,
    node.nodename,
    node.nodeport
FROM pg_dist_placement placement
JOIN pg_dist_node node
    ON placement.groupid = node.groupid
    AND node.noderole = 'primary'::noderole
WHERE shardid = 102027;
```

```
-----
| shardid | nodename | nodeport |
-----
| 102027 | localhost | 5433 |
-----
```

In our example of `github_events` there were four shards. The number of shards is configurable per table at the time of its distribution across the cluster. The best choice of shard count depends on your use case, see [Shard Count](#).

Finally note that Citus allows shards to be replicated for protection against data loss. There are two replication “modes:” Citus replication and streaming replication. The former creates extra backup shard placements and runs queries against all of them that update any of them. The latter is more efficient and utilizes PostgreSQL’s streaming replication to back up the entire database of each node to a follower database. This is transparent and does not require the involvement of Citus metadata tables.

Co-Location

Since shards and their replicas can be placed on nodes as desired, it makes sense to place shards containing related rows of related tables together on the same nodes. That way join queries between them can avoid sending as much information over the network, and can be performed inside a single Citus node.

One example is a database with stores, products, and purchases. If all three tables contain – and are distributed by – a `store_id` column, then all queries restricted to a single store can run efficiently on a single worker node. This is true even when the queries involve any combination of these tables.

For a full explanation and examples of this concept, see [Table Co-Location](#).

Parallelism

Spreading queries across multiple machines allows more queries to run at once, and allows processing speed to scale by adding new machines to the cluster. Additionally splitting a single query into fragments as described in the previous section boosts the processing power devoted to it. The latter situation achieves the greatest *parallelism*, meaning utilization of CPU cores.

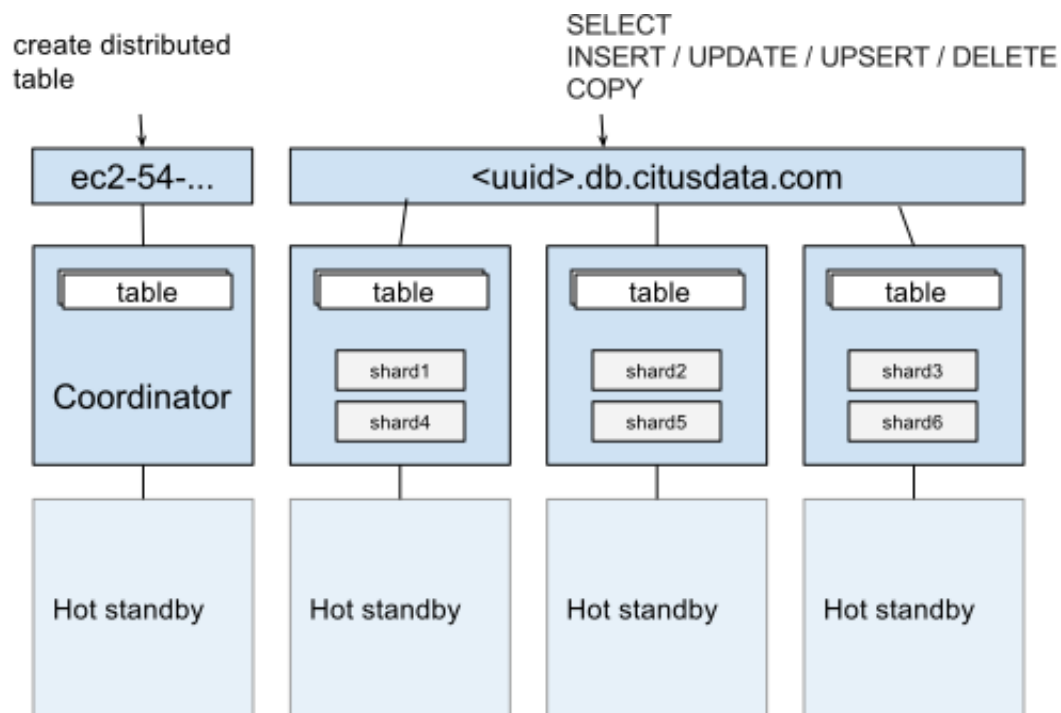
Queries reading or affecting shards spread evenly across many nodes are able to run at “real-time” speed. Note that the results of the query still need to pass back through the coordinator node, so the speedup is most apparent when the final results are compact, such as aggregate functions like counting and descriptive statistics.

[Query Processing](#) explains more about how queries are broken into fragments and how their execution is managed.

Citux MX

Citux MX is a new version of Citux that adds the ability to use hash-distributed tables from any node in a Citux 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. Citux MX is currently available in Citux Enterprise Edition and on [Citux Cloud](#).

In the Citux MX architecture, all nodes are PostgreSQL servers running the Citux 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.



Citux MX Architecture

Citux 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.

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.

The screenshot shows the Citus Cloud console interface. The top navigation bar includes the Citus logo and links for Formations, Tunes, Org, and a user profile. The main content area is titled 'Formation: MX demo' and has tabs for Overview, Nodes, Roles, Metrics, Logs, Debug Info, and Settings. The 'Overview' tab is selected, showing 'Formation Details' in a green header. Below this, a table lists the following information:

ID	a6017707-8f9c-47d4-9da1-86d53c676e9d
Coordinator URL	postgres://citus:8KIDGg26DNLvuGQyOylpg@ec2-52-45-116-32.compute-1.amazonaws.com:5432/citus?sslmode=require
Data URL	postgres://citus:8KIDGg26DNLvuGQyOylpg@a6017707-8f9c-47d4-9da1-86d53c676e9d.db.citusdata.com:5432/citus?sslmode=require
Region	us-east-1
Created	2016-09-16T15:29:18.807+00:00
Primary Spec	512 GB Storage · 15 GB RAM · HA Enabled
Worker Node Spec	4 x 512 GB Storage · 30.5 GB RAM · HA Enabled
Connections	1 / 300
Citus Version	5.2-1
Postgres Version	9.5.4

Below the 'Formation Details' section is a 'Distributed Tables' section with a dark blue header. It contains the text 'No distributed tables yet.' and a link 'Learn how to create one.'

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. 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.

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 Citrus MX.

Since Citrus is an PostgreSQL extension, you can use all the latest PostgreSQL 10 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 Citrus 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 Citrus, 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.

MX Limitations

Although MX allows direct reading and writing from worker nodes, it doesn't support all commands on workers. The coordinator node is the authoritative source of Citus metadata, so queries that change metadata must happen via the coordinator.

Supported only via coordinator

- *DDL* commands.
- *Citus Utility Functions* that change Citus metadata.
- Queries accessing *append distributed* tables.

Other query limitations

- Foreign data wrappers, including `cstore_fdw`, are not supported with Citus MX.
- Serial columns must have type "bigserial." Globally in the cluster the sequence values will not be monotonically increasing because the sixteen most significant bits hold the worker node id.

Determining Application Type

Running efficient queries on a Citus cluster requires that data be properly distributed across machines. This varies by the type of application and its query patterns.

There are broadly two kinds of applications that work very well on Citus. The first step in data modeling is to identify which of them more closely resembles your application.

At a Glance

Multi-Tenant Applications	Real-Time Applications
Sometimes dozens or hundreds of tables in schema	Small number of tables
Queries relating to one tenant (company/store) at a time	Relatively simple analytics queries with aggregations
OLTP workloads for serving web clients	High ingest volume of mostly immutable data
OLAP workloads that serve per-tenant analytical queries	Often centering around big table of events

Examples and Characteristics

Multi-Tenant Application

These are typically SaaS applications that serve other companies, accounts, or organizations. Most SaaS applications are inherently relational. They have a natural dimension on which to distribute data across nodes: just shard by `tenant_id`.

Citus enables you to scale out your database to millions of tenants without having to re-architect your application. You can keep the relational semantics you need, like joins, foreign key constraints, transactions, ACID, and consistency.

- **Examples:** Websites which host store-fronts for other businesses, such as a digital marketing solution, or a sales automation tool.
- **Characteristics:** Queries relating 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.

Scaling a multi-tenant app with Citus also requires minimal changes to application code. We have support for popular frameworks like Ruby on Rails and Django.

Real-Time Analytics

Applications needing massive parallelism, coordinating hundreds of cores for fast results to numerical, statistical, or counting queries. By sharding and parallelizing SQL queries across multiple nodes, Citus makes it possible to perform real-time queries across billions of records in under a second.

- **Examples:** Customer-facing analytics dashboards requiring sub-second response times.
- **Characteristics:** Few tables, often centering around a big table of device-, site- or user-events and requiring high ingest volume of mostly immutable data. Relatively simple (but computationally intensive) analytics queries involving several aggregations and GROUP BYs.

If your situation resembles either case above then the next step is to decide how to shard your data in the Citus cluster. As explained in the [Concepts](#) section, 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.

Choosing Distribution Column

Citus uses the distribution column in distributed tables to assign table rows to shards. Choosing the distribution column for each table is **one of the most important** modeling decisions because it determines how data is spread across nodes.

If the distribution columns are chosen correctly, then related data will group together on the same physical nodes, making queries fast and adding support for all SQL features. If the columns are chosen incorrectly, the system will run needlessly slowly, and won't be able to support all SQL features across nodes.

This section gives distribution column tips for the two most common Citus scenarios. It concludes by going in depth on “co-location,” the desirable grouping of data on nodes.

Multi-Tenant Apps

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.

Best Practices

- **Partition distributed tables by a common `tenant_id` column.** For instance, in a SaaS application where tenants are companies, the `tenant_id` will likely be `company_id`.
- **Convert small cross-tenant tables to reference tables.** When multiple tenants share a small table of information, distribute it as a *reference table*.
- **Restrict filter all application queries by `tenant_id`.** Each query should request information for one tenant at a time.

Node A

Accounts table (shard 1)

account_id	name	created_at
1	CNN	2016-07-12
5	Comcast	2016-07-19
...
1252	Walmart	2016-08-02

Campaigns table (shard 3)

campaign_id	name	account_id
1202	tv series	1
1204	superbowl	1
...
352042	chocolate	1252

Node B

Accounts table (shard 2)

account_id	name	created_at
2	AT&T	2016-07-13
3	Exxon	2016-07-14
...
1253	UPS	2016-08-03

Campaigns table (shard 4)

campaign_id	name	account_id
2742	gas state	3
2743	my phone	2
...
352423	new phone	2

Read the [Multi-tenant Applications](#) guide for a detailed example of building this kind of application.

Real-Time Apps

While the multi-tenant architecture introduces a hierarchical structure and uses data co-location to route queries per tenant, 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 entities 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 single node must do a disproportionate amount of work.

Best Practices

- **Choose a column with high cardinality as the distribution column.** For comparison, a “status” field on an order table with values “new,” “paid,” and “shipped” is a poor choice of distribution column because it assumes only those few values. The number of distinct values limits the number of shards that can hold the data, and the number of nodes that can process it. Among columns with high cardinality, it is good additionally to choose those that are frequently used in group-by clauses or as join keys.
- **Choose a column with even distribution.** If you distribute a table on a column skewed to certain common values, then data in the table will tend to accumulate in certain shards. The nodes holding those shards will end up doing more work than other nodes.
- **Distribute fact and dimension tables on their common columns.** Your fact table can have only one distribution key. Tables that join on another key will not be co-located with the fact table. Choose one dimension to

co-locate based on how frequently it is joined and the size of the joining rows.

- **Change some dimension tables into reference tables.** If a dimension table cannot be co-located with the fact table, you can improve query performance by distributing copies of the dimension table to all of the nodes in the form of a *reference table*.

Read the [Real-Time Dashboards](#) guide for a detailed example of building this kind of application.

Timeseries Data

In a time-series workload, applications query recent information while archiving old information.

The most common mistake in modeling timeseries information in Citrus is using the timestamp itself as a distribution column. A hash distribution based on time will distribute times seemingly at random into different shards rather than keeping ranges of time together in shards. However queries involving time generally reference ranges of time (for example the most recent data), so such a hash distribution would lead to network overhead.

Best Practices

- **Do not choose a timestamp as the distribution column.** Choose a different distribution column. In a multi-tenant app, use the tenant id, or in a real-time app use the entity id.
- **Use PostgreSQL table partitioning for time instead.** Use table partitioning to break a big table of time-ordered data into multiple inherited tables with each containing different time ranges. Distributing a Postgres-partitioned table in Citrus creates shards for the inherited tables.

Read the [Timeseries Data](#) guide for a detailed example of building this kind of application.

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.

Data co-location in Citrus for hash-distributed tables

The Citrus extension for PostgreSQL is unique in being able to form a distributed database of databases. Every node in a Citrus cluster is a fully functional PostgreSQL database and Citrus 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 Citrus 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.

	events shards		page shards	
	shard	hash range	shard	hash range
NODE 1	1	$-2147483648 \leq x \leq -1073741825$	5	$-2147483648 \leq x \leq -1073741825$
	2	$-1073741824 \leq x \leq -1$	6	$-1073741824 \leq x \leq -1$
NODE 2	3	$0 \leq x \leq 1073741823$	7	$0 \leq x \leq 1073741823$
	4	$1073741824 \leq x \leq 2147483647$	8	$1073741824 \leq x \leq 2147483647$

$x = \text{hash}(\text{distribution_column})$

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.

A practical example of co-location

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

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

CREATE TABLE page (
  tenant_id int,
  page_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."

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')::timestampz >= now() - interval '1 week'
) recent
USING (tenant_id, page_id)
WHERE tenant_id = 6 AND path LIKE '/blog%'
GROUP BY page_id;

```

As long as 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.

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 Citrus. 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):

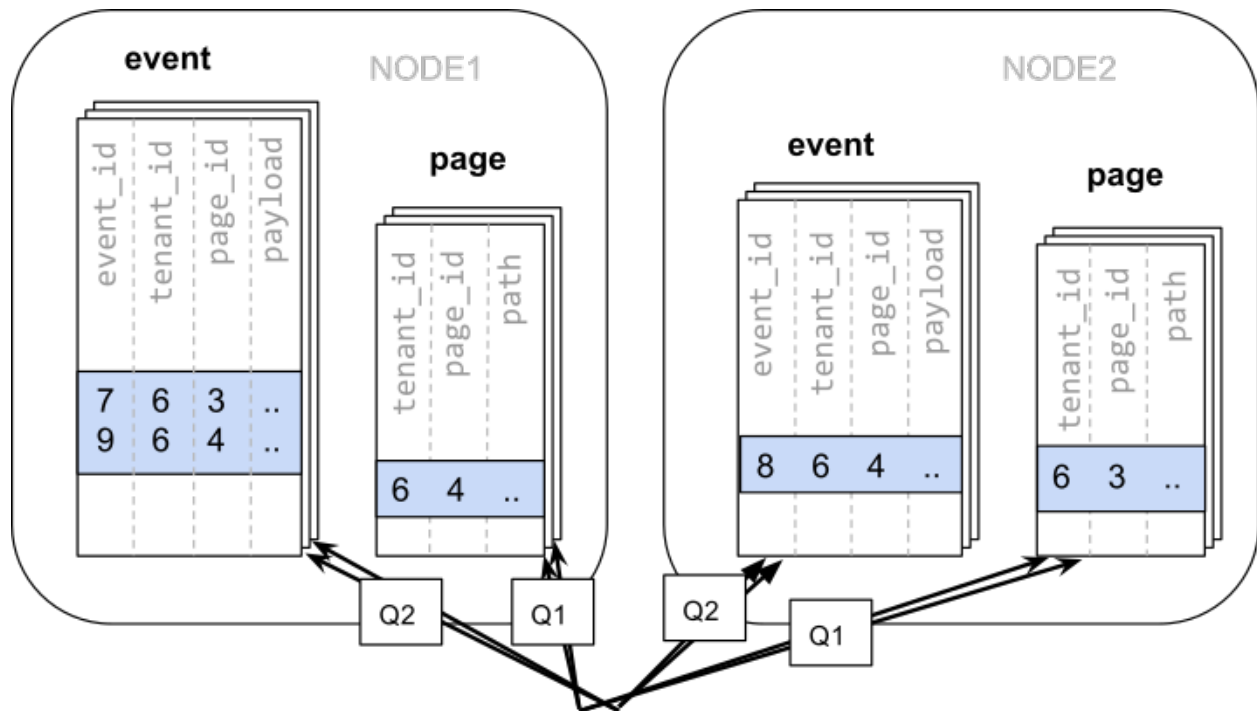
```

SELECT page_id, count(*) AS count
FROM event
WHERE page_id IN (/*...page IDs from first query...*/)
  AND tenant_id = 6
  AND (payload->>'time')::date >= 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 Citrus 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.

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 Citrus, 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');
```

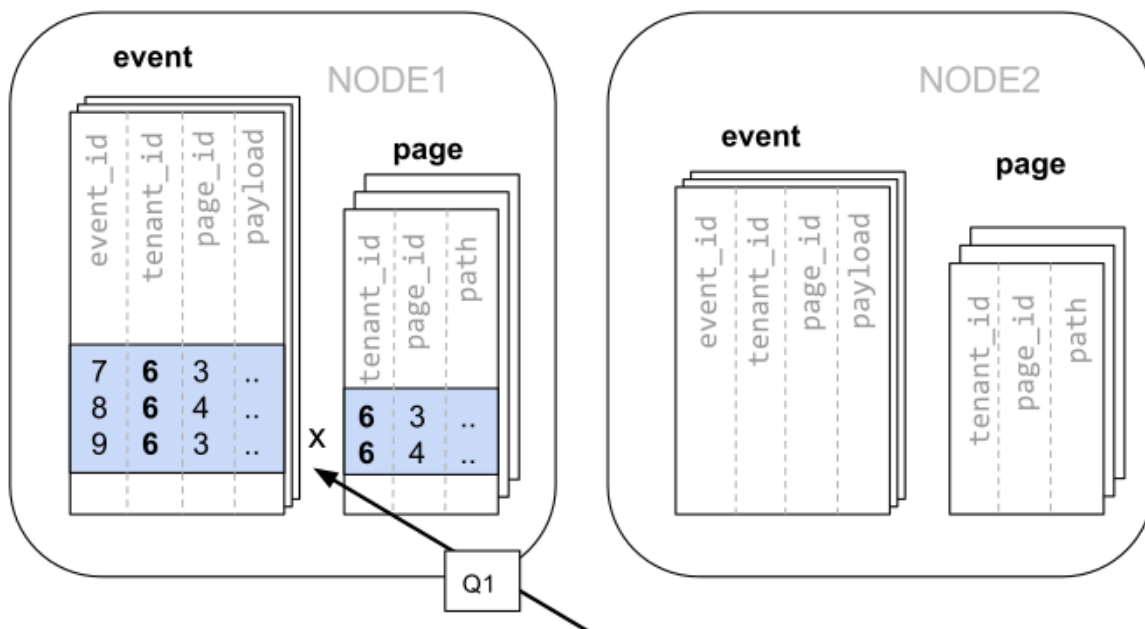
In this case, Citrus can answer the same query that you would run on a single PostgreSQL node without modification (Q1):


```

SELECT page_id, count(event_id)
FROM
  page
LEFT JOIN (
  SELECT * FROM event
  WHERE (payload->>'time')::timestampz >= 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.

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 support to 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](#).

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.

Migrating an Existing App

Migrating an existing application 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.

The first steps are to optimize the existing database schema so that it can work efficiently across multiple computers.

Identify Distribution Strategy

Pick distribution key

The first step in migrating to Citus is identifying suitable distribution keys and planning table distribution accordingly. In multi-tenant applications this will typically be an internal identifier for tenants. We typically refer to it as the “tenant ID.” The use-cases may vary, so we advise being thorough on this step.

For guidance, read these sections:

1. *Determining Application Type*
2. *Choosing Distribution Column*

We are happy to help review your environment to be sure that the ideal distribution key is chosen. To do so, we typically examine schema layouts, larger tables, long-running and/or problematic queries, standard use cases, and more.

Identify types of tables

Once a distribution key is identified, review the schema to identify how each table will be handled and whether any modifications to table layouts will be required. We typically advise tracking this with a spreadsheet, and have created a [template](#) you can use.

Tables will generally fall into one of the following categories:

1. **Ready for distribution.** These tables already contain the distribution key, and are ready for distribution.
2. **Needs backfill.** These tables can be logically distributed by the chosen key, but do not contain a column directly referencing it. The tables will be modified later to add the column.
3. **Reference table.** These tables are typically small, do not contain the distribution key, are commonly joined by distributed tables, and/or are shared across tenants. A copy of each of these tables will be maintained on all nodes. Common examples include country code lookups, product categories, and the like.

4. **Local table.** These are typically not joined to other tables, and do not contain the distribution key. They are maintained exclusively on the coordinator node. Common examples include admin user lookups and other utility tables.

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 this example stores are a natural tenant. The tenant id is in this case the `store_id`. After distributing tables in the cluster, we want rows relating to the same store to reside together on the same nodes.

Prepare Tables for Migration

Once the scope of needed database changes is identified, the next major step is to modify the data structure. First, existing tables requiring backfill are modified to add a column for the distribution key.

Add distribution keys

In our storefront example the stores and products tables have a `store_id` and are ready for distribution. Being normalized, the `line_items` table lacks a store id. If we want to distribute by `store_id`, the table needs this column.

```
-- denormalize line_items by including store_id
ALTER TABLE line_items ADD COLUMN store_id uuid;
```

Be sure to check that the distribution column has the same type in all tables, e.g. don't mix `int` and `bigint`. The column types must match to ensure proper data colocation.

Include distribution column in keys

Citrus *cannot enforce* uniqueness constraints unless a unique index or primary key contains the distribution column. Thus we must modify primary and foreign keys in our example to include `store_id`.

Here are SQL commands to turn the simple keys composite:

```
BEGIN;

-- drop simple primary keys (cascades to foreign keys)
ALTER TABLE products DROP CONSTRAINT products_pkey CASCADE;
ALTER TABLE orders DROP CONSTRAINT orders_pkey CASCADE;
ALTER TABLE line_items DROP CONSTRAINT line_items_pkey CASCADE;

-- recreate primary keys to include would-be distribution column
ALTER TABLE products ADD PRIMARY KEY (store_id, product_id);
ALTER TABLE orders ADD PRIMARY KEY (store_id, order_id);
ALTER TABLE line_items ADD PRIMARY KEY (store_id, line_item_id);

-- recreate foreign keys to include would-be distribution column
ALTER TABLE line_items ADD CONSTRAINT line_items_store_fkey
FOREIGN KEY (store_id) REFERENCES stores (store_id);
ALTER TABLE line_items ADD CONSTRAINT line_items_product_fkey
FOREIGN KEY (store_id, product_id) REFERENCES products (store_id, product_id);
```

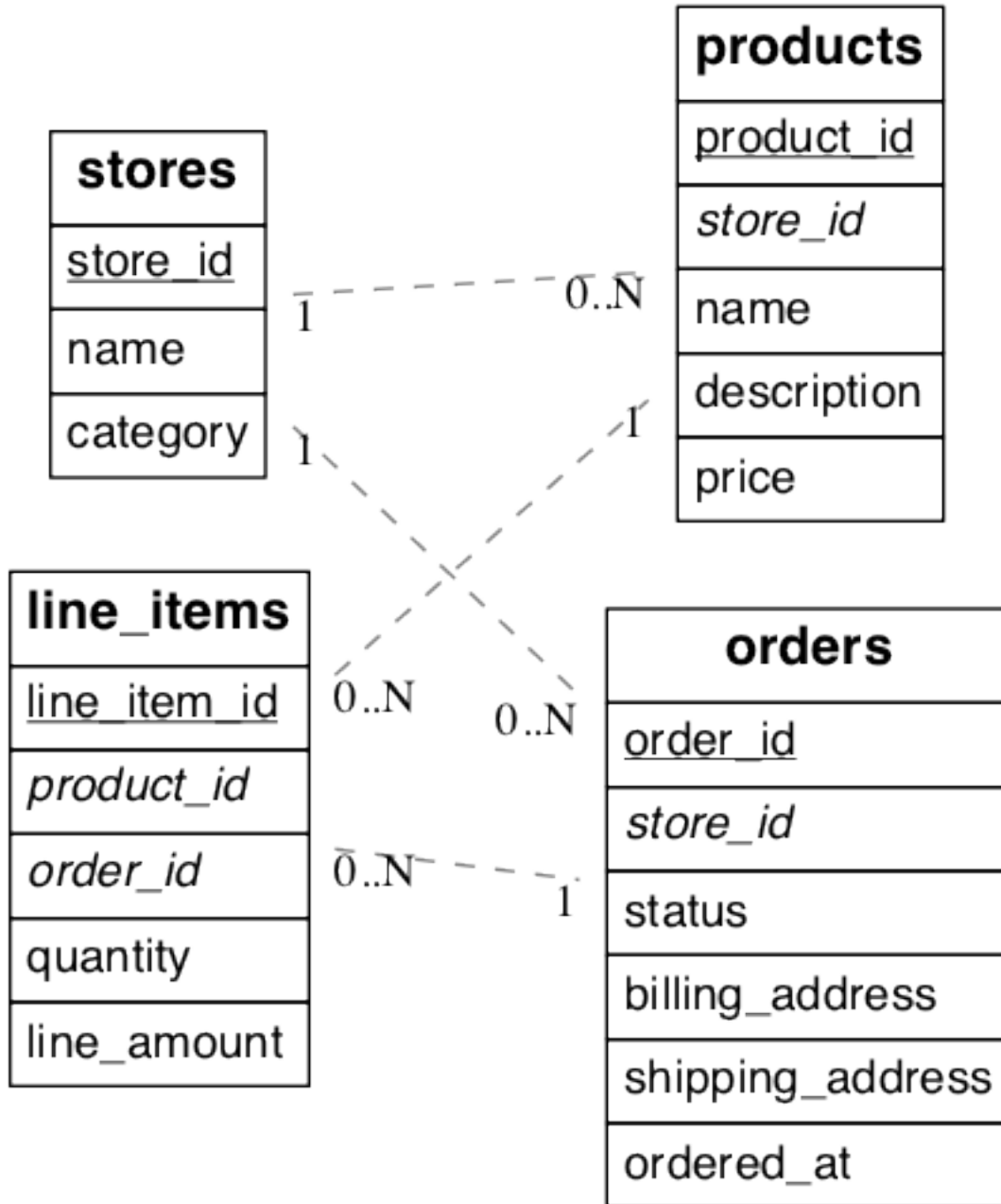


Fig. 14.1: (Underlined items are primary keys, italicized items are foreign keys.)

```
ALTER TABLE line_items ADD CONSTRAINT line_items_order_fkey
  FOREIGN KEY (store_id, order_id) REFERENCES orders (store_id, order_id);

COMMIT;
```

Thus completed, our schema will look like this:

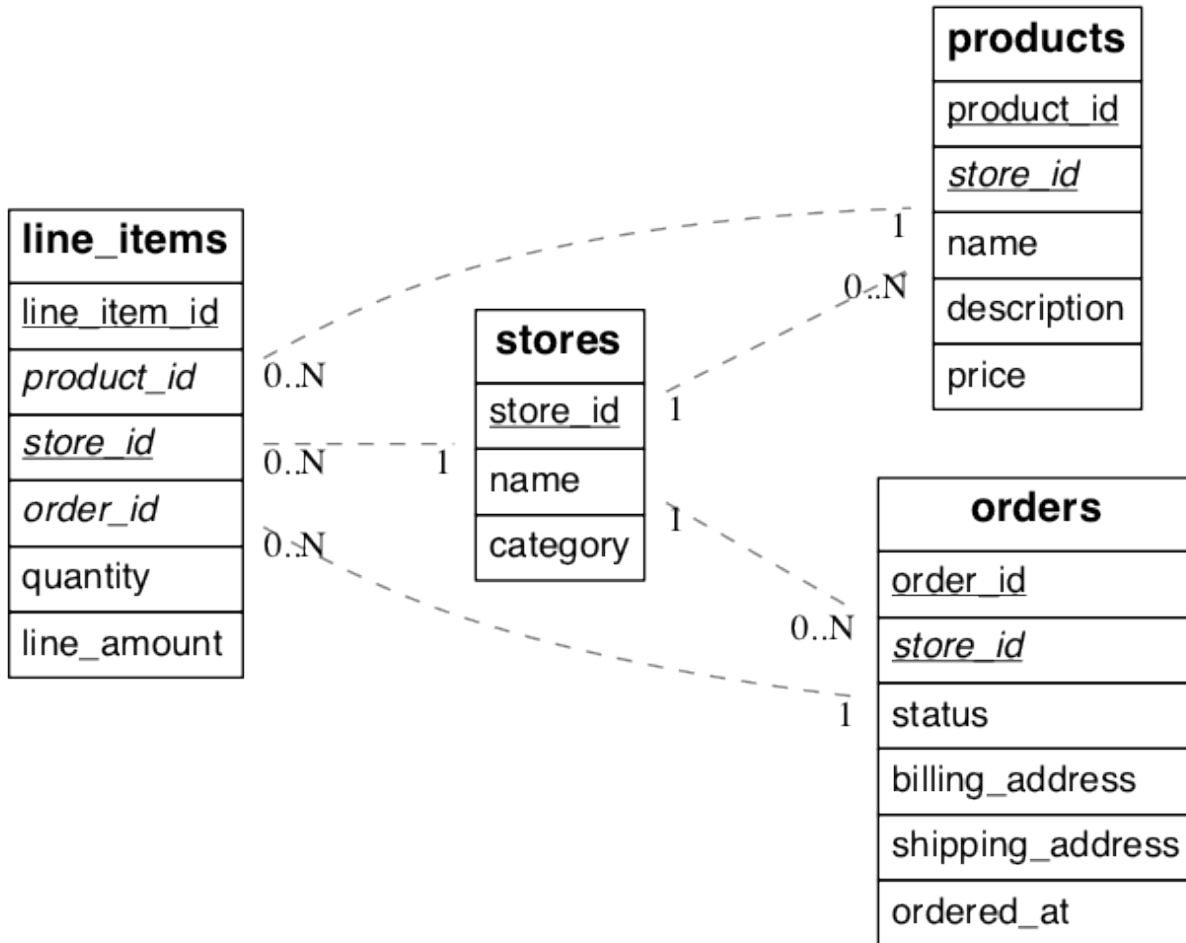


Fig. 14.2: (Underlined items are primary keys, italicized items are foreign keys.)

Be sure to modify data flows to add keys to incoming data.

Backfill newly created columns

Once the schema is updated, backfill missing values for the `tenant_id` column in tables where the column was added. In our example `line_items` requires values for `store_id`.

We backfill the table by obtaining the missing values from a join query with `orders`:

```
UPDATE line_items
  SET store_id = orders.store_id
FROM line_items
```

```
INNER JOIN orders
WHERE line_items.order_id = orders.order_id;
```

The application and other data ingestion processes should be updated to include the new column for future writes. More on that in the next section.

Next, update application code and queries to deal with the schema changes.

Prepare Application for Migration

Once the distribution key is present on all appropriate tables, the application needs to include it in queries.

Add distribution key to queries

To execute queries efficiently for a specific tenant, Citrus 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.

- Application code and any other ingestion processes that write to the tables should be updated to include the new columns.
- Running the application test suite against the modified schema on Citrus is a good way to determine which areas of the code need to be modified.
- It's a good idea to enable database logging. The logs can help uncover stray cross-shard queries in a multi-tenant app that should be converted to per-tenant queries.

Specialized Libraries

There are helper libraries for a number of popular application frameworks that make it easy to include a tenant id in queries.

- Ruby on Rails migration - uses the activerecord-multi-tenant Ruby gem
- Django migration - uses the django-multitenant Python library
- ASP.NET migration - uses the 3rd party SAASkit
- [Java Hibernate](#) - a blog post about scoping queries to tenants

It's possible to use the libraries for database writes first (including data ingestion), and later for read queries. The activerecord-multi-tenant gem for instance has a write-only mode that will modify only the write queries.

General Principles

If you're using a different ORM than those above, or doing multi-tenant queries more directly in SQL, follow these general principles. We'll use our earlier example of the ecommerce application.

Suppose we want to get the details for an order. Distributed queries that filter on the tenant id run most efficiently in multi-tenant apps, so the change below makes the query faster (while both queries return the same results):

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

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

The tenant id column is not just beneficial – but critical – for insert statements. Inserts must include a value for the tenant id column or else Citus will be unable to route the data to the correct shard and will raise an error.

Finally, when joining tables make sure to filter by tenant id too. 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'
```

Check for cross-node traffic

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 encountering 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:

```
-- adjust for your own database's name of course

ALTER DATABASE citus 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 log, instead of error out:

```
ALTER DATABASE citus SET citus.multi_task_query_log_level = 'log';
```

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

After testing the changes in a development environment, the last step is to migrate production data to a Citrus cluster and switch over the production app. We have techniques to minimize downtime for this step.

Migrate Data

At this time, having updated the database schema and application queries to work with Citrus, you're ready for the final step. It's time to migrate data to the Citrus cluster and cut over the application to its new database.

The data migration path is dependent on downtime requirements and data size, but generally falls into one of the following two categories.

Databases under 200GB

For smaller environments that can tolerate a little downtime, use a simple `pg_dump/pg_restore` process. Here are the steps.

1. Save the database structure:

```
pg_dump \
  --format=plain \
  --no-owner \
  --schema-only \
  --file=schema.sql \
  --schema=target_schema \
  postgres://user:pass@host:5432/db
```

2. Connect to the Citrus cluster using `psql` and create the schema:

```
\i schema.sql
```

3. Run your *create_distributed_table* and *create_reference_table* statements. If you get an error about foreign keys, it's generally due to the order of operations. Drop foreign keys before distributing tables and then re-add them.
4. Put the application into maintenance mode, and disable any other writes to the old database.
5. Save the data from the old database to disk with `pg_dump`:

```
pg_dump \
  --format=custom \
  --no-owner \
  --data-only \
  --file=data.dump \
  --schema=target_schema \
  postgres://user:pass@host:5432/db
```

6. Import into Citrus using `pg_restore`:

```
# remember to use connection details for Citus,  
# not the source database  
pg_restore \  
  --host=host \  
  --dbname=dbname \  
  --username=username \  
  data.dump  
  
# it'll prompt you for the connection password
```

7. Test application.

8. Launch!

Databases over 200GB

Larger environments can use Citus Warp for online replication. Citus Warp allows you to stream changes from a PostgreSQL source database into a *Citus Cloud* cluster as they happen. It's as if the application automatically writes to two databases rather than one, except with perfect transactional logic. Citus Warp works with Postgres versions 9.4 and above which have the *logical_decoding* plugin enabled (this is supported on Amazon RDS as long as you're at version 9.4 or higher).

For this process we strongly recommend contacting us by opening a ticket, contacting one of our solutions engineers on Slack, or whatever method works for you. To do a warp, we connect the coordinator node of a Citus cluster to an existing database through VPC peering or IP white-listing, and begin replication.

Here are the steps you need to perform before starting the Citus Warp process:

1. Duplicate the structure of the schema on a destination Citus cluster
2. Enable logical replication in the source database
3. Allow a network connection from Citus coordinator node to source
4. Contact us to begin the replication

Duplicate schema

The first step in migrating data to Citus is making sure that the schemas match exactly, at least for the tables you choose to migrate. One way to do this is by running `pg_dump --schema-only` against the source database. Replay the output on the coordinator Citus node. Another way to is to run application migration scripts against the destination database.

All tables that you wish to migrate must have primary keys. The corresponding destination tables must have primary keys as well, the only difference being that those keys are allowed to be composite to contain the distribution column as well, as described in *Identify Distribution Strategy*.

Also be sure to *distribute tables* across the cluster prior to starting replication so that the data doesn't have to fit on the coordinator node alone.

Enable logical replication

Some hosted databases such as Amazon RDS require enabling replication by changing a server configuration parameter. On RDS you will need to create a new parameter group, set `rds.logical_replication = 1` in it, then make the parameter group the active one. Applying the change requires a database server reboot, which can be scheduled for the next maintenance window.

If you're administering your own PostgreSQL installation, add these settings to `postgresql.conf`:

```
wal_level = logical
max_replication_slots = 5 # has to be > 0
max_wal_senders = 5      # has to be > 0
```

A database restart is required for the changes to take effect.

Open access for network connection

In the Cloud console, identify the hostname (it ends in `db.citusdata.com`). Dig the hostname to find its IP address:

```
dig +short <hostname> A
```

If you're using RDS, edit the inbound database security group to add a custom TCP rule:

Protocol TCP

Port Range 5432

Source <citus ip>/32

This white-lists the IP address of the Citus coordinator node to make an inbound connection. An alternate way to connect the two is to establish peering between their VPCs. We can help set that up if desired.

Begin Replication

Contact us by opening a support ticket in the Citus Cloud console. A Cloud engineer will connect to your database with Citus Warp to create a basebackup, open a replication slot, and begin the replication. We can include/exclude your choice of tables in the migration.

During the first stage, creating a basebackup, the Postgres write-ahead log (WAL) may grow substantially if the database is under write load. Make sure you have sufficient disk space on the source database before starting this process. We recommend 100GB free or 20% of total disk space, whichever is greater. Once the backup is complete and replication begins then the database will be able to archive unused WAL files again.

Some database schema changes are incompatible with an ongoing replication. Changing the structure of tables under replication can cause the process to stop. Cloud engineers would then need to manually restart the replication from the beginning. That costs time, so we recommend freezing the schema during replication.

Switch over to Citus and stop all connections to old database

When the replication has caught up with the current state of the source database, there is one more thing to do. Due to the nature of the replication process, sequence values don't get updated correctly on the destination databases. In order to have the correct sequence value for e.g. an id column, you need to manually adjust the sequence values before turning on writes to the destination database.

Once this is all complete, the application is ready to connect to the new database. We do not recommend writing to both the source and destination database at the same time.

When the application has cut over to the new database and no further changes are happening on the source database, contact us again to remove the replication slot. The migration is complete.

SQL Reference

Creating and Modifying Distributed Tables (DDL)

Note: Citus (including *Citus MX*) requires that DDL commands be run from the coordinator node only.

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 Functions* of our documentation.

Reference Tables

The above method distributes tables into multiple horizontal shards, but another possibility is distributing tables into a single shard and replicating the shard to every worker node. Tables distributed this way are called *reference tables*. They are used to store data that needs to be frequently accessed by multiple nodes in a cluster.

Common candidates for reference tables include:

- Smaller tables which need to join with larger distributed tables.
- Tables in multi-tenant apps which lack a tenant id column or which aren't associated with a tenant. (In some cases, to reduce migration effort, users might even choose to make reference tables out of tables associated with a tenant but which currently lack a tenant id.)
- Tables which need unique constraints across multiple columns and are small enough.

For instance suppose a multi-tenant eCommerce site needs to calculate sales tax for transactions in any of its stores. Tax information isn't specific to any tenant. It makes sense to consolidate it in a shared table. A US-centric reference table might look like this:

```
-- 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');
```

Now queries such as one calculating tax for a shopping cart can join on the `states` table with no network overhead, and can add a foreign key to the state code for better validation.

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');
```

For another example of using reference tables in a multi-tenant application, see *Sharing Data Between Tenants*.

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 it 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. However foreign keys *are* supported between distributed tables and reference tables.

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

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');
```

When a new table is not related to others in its would-be implicit co-location group, specify `colocate_with => 'none'`.

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

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

Splitting unrelated tables into their own co-location groups will improve *shard rebalancing* performance, because shards in the same group have to be moved together.

When tables are indeed related (for instance when they will be joined), it can make sense to explicitly co-locate them. The gains of appropriate co-location are more important than any rebalancing overhead.

To explicitly co-locate multiple 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.

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 using *mark_tables_colocated*:

```
-- 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.

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;
```

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.

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*.

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 like renaming it or changing its data type are fine too. However the data type of the *distribution column* cannot be altered. This column determines how table data distributes through the Citrus cluster, and modifying its data type would require moving the data.

Attempting to do so causes an error:

```
-- assuming store_id is the distribution column
-- for products, and that it has type integer

ALTER TABLE products
ALTER COLUMN store_id TYPE text;

/*
ERROR:  XX000: cannot execute ALTER TABLE command involving partition column
LOCATION:  ErrorIfUnsupportedAlterTableStmt, multi_utility.c:2150
*/
```

Adding/Removing Constraints

Using Citrus 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, Citrus will not cross-reference uniqueness constraints or referential integrity between worker nodes.

Foreign keys must always be declared between either

- Two local (non-distributed) tables,
- Two *colocated* distributed tables, or
- A distributed table and a *reference table*

To set up a foreign key between colocated distributed tables, always include the distribution column in the key. This may involve making the key compound.

Note: Primary keys and uniqueness constraints must include the distribution column. Adding them to a non-distribution column will generate an error (see *Cannot create uniqueness constraint*).

This example shows how to create primary and foreign keys on distributed tables:

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

-- We'll distribute these tables on the account_id. The ads and clicks
-- tables must use compound keys that include account_id.

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

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_ad_fk
    FOREIGN KEY (account_id, ad_id) REFERENCES ads (account_id, id);
```

Similarly, include the distribution column in uniqueness constraints:

```
-- 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 be applied to any column (distribution or not) because they require no lookups between workers.

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

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);
```

Manual Modification

Currently other DDL commands are not auto-propagated, however you can propagate the changes manually. See *Manual Query Propagation*.

Ingesting, Modifying Data (DML)

Inserting Data

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.

```
/*
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
);
*/

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, Citrus 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.

Sometimes it's convenient to put multiple insert statements together into a single insert of multiple rows. It can also be more efficient than making repeated database queries. For instance, the example from the previous section can be loaded all at once like this:

```
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'
), (
```

```
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'
);
```

“From Select” Clause (Distributed Rollups)

Citus also supports `INSERT ... SELECT` statements – which insert rows based on the results of a select query. This is a convenient way to fill tables and also allows “upserts” with the `ON CONFLICT` clause.

In Citus there are two ways that inserting from a select statement can happen. The first is if the source tables and destination table are *colocated*, and the select/insert statements both include the distribution column. In this case Citus can push the `INSERT ... SELECT` statement down for parallel execution on all nodes. Pushing the statement down supports the `ON CONFLICT` clause, the easiest way to do *distributed rollups*.

The second way of executing an `INSERT ... SELECT` statement is selecting the results from worker nodes, pulling the data up to the coordinator node, and then issuing an `INSERT` statement from the coordinator with the data. Citus is forced to use this approach when the source and destination tables are not colocated. This method does not support `ON CONFLICT`.

When in doubt about which method Citus is using, use the `EXPLAIN` command, as described in *PostgreSQL tuning*.

COPY Command (Bulk load)

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.

Caching Aggregations with Rollups

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. Citrus 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';
```

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;
```

It’s worth noting that for `INSERT INTO ... SELECT` to work on distributed tables with `ON CONFLICT`, Citrus requires the source and destination table to be co-located.

Updates and Deletion

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

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

UPDATE github_events
SET event_public = TRUE
WHERE (org->>'id')::int = 5430905;
```

When updates/deletes affect multiple shards as in the above example, Citrus defaults to using a one-phase commit protocol. For greater safety you can enable two-phase commits by setting

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

If an update or delete affects only a single shard then it runs within a single worker node. In this case enabling 2PC is unnecessary. This often happens when updates or deletes filter by a table’s distribution column:

```
-- since github_events is distributed by repo_id,
-- this will execute in a single worker node

DELETE FROM github_events
WHERE repo_id = 206084;
```

Furthermore, when dealing with a single shard, Citrus supports `SELECT ... FOR UPDATE`. This is a technique sometimes used by object-relational mappers (ORMs) to safely:

1. load rows

2. make a calculation in application code
3. update the rows based on calculation

Selecting the rows for update puts a write lock on them to prevent other processes from causing a “lost update” anomaly.

```
BEGIN;

-- select events for a repo, but
-- lock them for writing
SELECT *
FROM github_events
WHERE repo_id = 206084
FOR UPDATE;

-- calculate a desired value event_public using
-- application logic that uses those rows...

-- now make the update
UPDATE github_events
SET event_public = :our_new_value
WHERE repo_id = 206084;

COMMIT;
```

This feature is supported for hash distributed and reference tables only, and only those that have a *replication_factor* of 1.

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.

Querying Distributed Tables (SQL)

As discussed in the previous sections, Citrus is an extension which extends the latest PostgreSQL for distributed execution. This means that you can use standard PostgreSQL **SELECT** queries on the Citrus coordinator for querying. Citrus will then parallelize the SELECT queries involving complex selections, groupings and orderings, and JOINS to speed up the query performance. At a high level, Citrus 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 Citrus.

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.

Count (Distinct) Aggregates

Citus supports count(distinct) aggregates in several ways. If the count(distinct) aggregate is on the distribution column, Citrus can directly push down the query to the workers. If not, Citrus runs select distinct statements on each worker, and returns the list to the coordinator where it obtains the final count.

Note that transferring this data becomes slower when workers have a greater number of distinct items. This is especially true for queries containing multiple count(distinct) aggregates, e.g.:

```
-- multiple distinct counts in one query tend to be slow
SELECT count(distinct a), count(distinct b), count(distinct c)
FROM table_abc;
```

For these kind of queries, the resulting select distinct statements on the workers essentially produce a cross-product of rows to be transferred to the coordinator.

For increased performance you can choose to make an approximate count instead. 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, count(distinct) aggregates automatically switch to using HLL, with no changes necessary to your queries. You should be able to run approximate count distinct queries on any column of the table.

HyperLogLog Column

Certain users already store their data as HLL columns. In such cases, they can dynamically roll up those data by calling hll_union_agg(hll_column).

Estimating Top N Items

Calculating the first n elements in a set by applying count, sort, and limit is simple. However as data sizes increase, this method becomes slow and resource intensive. It's more efficient to use an approximation.

The open source [TopN extension](#) for Postgres enables fast approximate results to “top-n” queries. The extension materializes the top values into a JSON data type. TopN can incrementally update these top values, or merge them on-demand across different time intervals.

Basic Operations

Before seeing a realistic example of TopN, let's see how some of its primitive operations work. First topn_add updates a JSON object with counts of how many times a key has been seen:

```
-- starting from nothing, record that we saw an "a"
select topn_add('{}', 'a');
-- => {"a": 1}
```



```
-- record the sighting of another "a"
select topn_add(topn_add('{}', 'a'), 'a');
-- => {"a": 2}
```

The extension also provides aggregations to scan multiple values:

```
-- for normal_rand
create extension tablefunc;

-- count values from a normal distribution
SELECT topn_add_agg(floor(abs(i))::text)
  FROM normal_rand(1000, 5, 0.7) i;
-- => {"2": 1, "3": 74, "4": 420, "5": 425, "6": 77, "7": 3}
```

If the number of distinct values crosses a threshold, the aggregation drops information for those seen least frequently. This keeps space usage under control. The threshold can be controlled by the `topn.number_of_counters` GUC. Its default value is 1000.

Realistic Example

Now onto a more realistic example of how TopN works in practice. Let's ingest Amazon product reviews from the year 2000 and use TopN to query it quickly. First download the dataset:

```
curl -L https://examples.citusdata.com/customer_reviews_2000.csv.gz | \
gunzip > reviews.csv
```

Next, ingest it into a distributed table:

```
CREATE TABLE customer_reviews
(
  customer_id TEXT,
  review_date DATE,
  review_rating INTEGER,
  review_votes INTEGER,
  review_helpful_votes INTEGER,
  product_id CHAR(10),
  product_title TEXT,
  product_sales_rank BIGINT,
  product_group TEXT,
  product_category TEXT,
  product_subcategory TEXT,
  similar_product_ids CHAR(10)[]
);

SELECT create_distributed_table('customer_reviews', 'product_id');

\COPY customer_reviews FROM 'reviews.csv' WITH CSV
```

Next we'll add the extension, create a destination table to store the json data generated by TopN, and apply the `topn_add_agg` function we saw previously.

```
-- note: Citus Cloud has extension already
CREATE EXTENSION topn;
SELECT run_command_on_workers(' create extension topn; ');

-- a table to materialize the daily aggregate
CREATE TABLE reviews_by_day
```

```
(
  review_date date unique,
  agg_data jsonb
);

SELECT create_reference_table('reviews_by_day');

-- materialize how many reviews each product got per day per customer
INSERT INTO reviews_by_day
  SELECT review_date, topn_add_agg(product_id)
  FROM customer_reviews
  GROUP BY review_date;
```

Now, rather than writing a complex window function on `customer_reviews`, we can simply apply `TopN` to `reviews_by_day`. For instance, the following query finds the most frequently reviewed product for each of the first five days:

```
SELECT review_date, (topn(agg_data, 1)).*
FROM reviews_by_day
ORDER BY review_date
LIMIT 5;
```

```
-----
| review_date | item      | frequency |
-----
| 2000-01-01 | 0939173344 | 12 |
| 2000-01-02 | B000050XY8 | 11 |
| 2000-01-03 | 0375404368 | 12 |
| 2000-01-04 | 0375408738 | 14 |
| 2000-01-05 | B00000J7J4 | 17 |
-----
```

The json fields created by `TopN` can be merged with `topn_union` and `topn_union_agg`. We can use the latter to merge the data for the entire first month and list the five most reviewed products during that period.

```
SELECT (topn(topn_union_agg(agg_data), 5)).*
FROM reviews_by_day
WHERE review_date >= '2000-01-01' AND review_date < '2000-02-01'
ORDER BY 2 DESC;
```

```
-----
| item      | frequency |
-----
| 0375404368 | 217 |
| 0345417623 | 217 |
| 0375404376 | 217 |
| 0375408738 | 217 |
| 043936213X | 204 |
-----
```

For more details and examples see the [TopN readme](#).

Limit Pushdown

Citrus 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, Citrus 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, Citrus 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;
```

Views on Distributed Tables

Citus supports all views on distributed tables. For an overview of views' syntax and features, see the PostgreSQL documentation for [CREATE VIEW](#).

Note that some views cause a less efficient query plan than others. For more about detecting and improving poor view performance, see [Subquery/CTE Network Overhead](#). (Views are treated internally as subqueries.)

Citus supports materialized views as well, and stores them as local tables on the coordinator node. Using them in distributed queries after materialization requires wrapping them in a subquery, a technique described in [JOIN a local and a distributed table](#).

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 how tables are distributed. It evaluates several possible join orders and creates a join plan which requires minimum data to be transferred across network.

Co-located joins

When two tables are *co-located* then they can be joined efficiently on their common distribution columns. A co-located join is the most efficient way to join two large distributed tables.

Internally, the Citrus coordinator knows which shards of the co-located tables might match with shards of the other table by looking at the distribution column metadata. This allows Citrus 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: Be sure that the tables are distributed into the same number of shards and that the distribution columns of each table have exactly matching types. Attempting to join on columns of slightly different types such as `int` and `bigint` can cause problems.

Reference table joins

Reference Tables can be used as “dimension” tables to join efficiently with large “fact” tables. Because reference tables are replicated in full across all worker nodes, a reference join can be decomposed into local joins on each worker and

performed in parallel. A reference join is like a more flexible version of a co-located join because reference tables aren't distributed on any particular column and are free to join on any of their columns.

Repartition joins

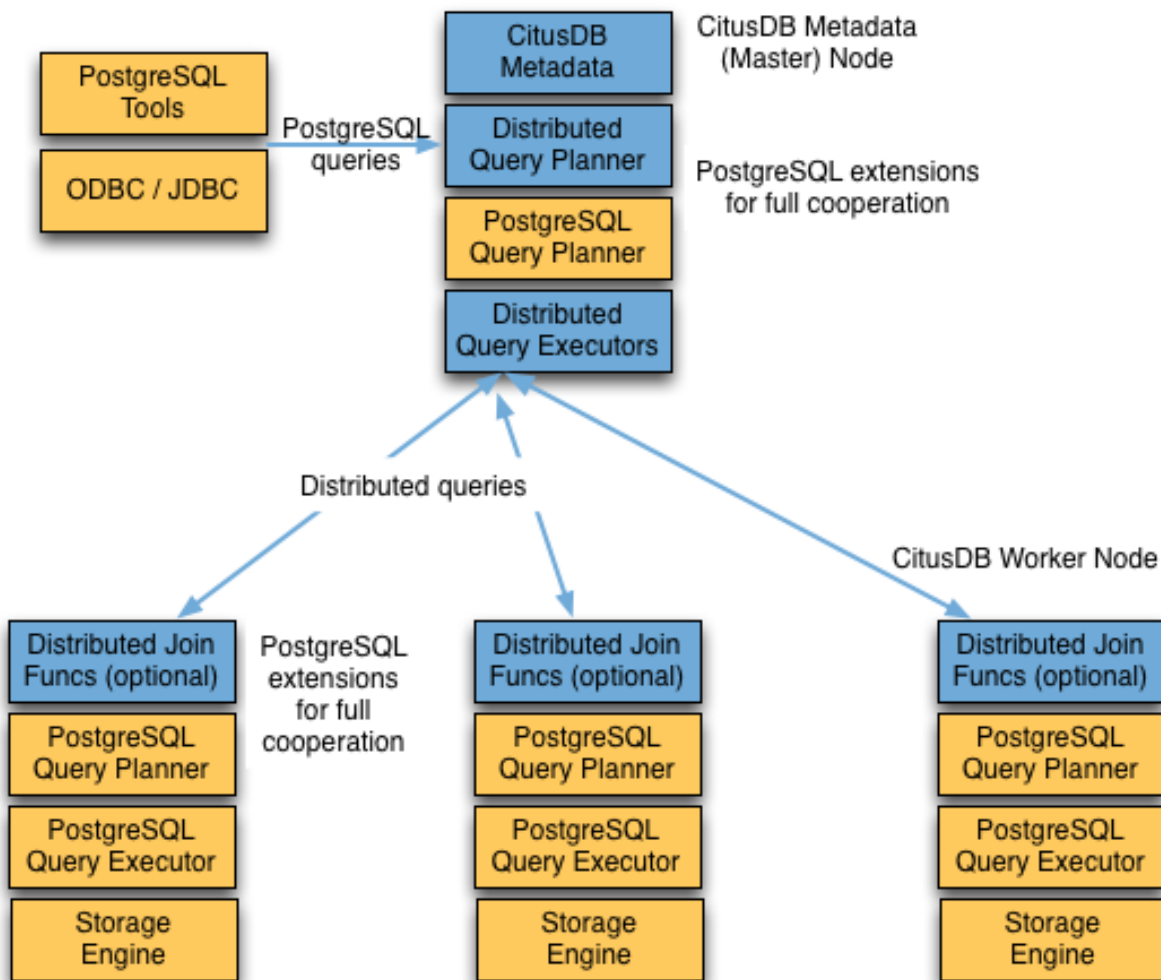
In some cases, you may need to join two tables on columns other than the distribution column. For such cases, Citus also allows joining on non-distribution key columns by dynamically repartitioning the tables for the query.

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.

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.

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.

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 three basic executor types: real time, router, and task tracker. It chooses which to use dynamically, depending on the structure of each query, and can use more than one at once for a single query, assigning different executors to different subqueries/CTEs as needed to support the SQL functionality. This process is recursive: if Citus cannot determine how to run a subquery then it examines sub-subqueries.

At a high level, the real-time executor is useful for handling simple key-value lookups and INSERT, UPDATE, and DELETE queries. The task tracker is better suited for larger SELECT queries, and the router executor for access data that is co-located in a single worker node.

The choice of executor for each query can be displayed by running PostgreSQL's `EXPLAIN` command. This can be useful for debugging performance issues.

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.

Router Executor

When all data required for a query is stored on a single node, Citus can route the entire query to the node and run it there. The result set is then relayed through the coordinator node back to the client. The router executor takes care of this type of execution.

Although Citus supports a large percentage of SQL functionality even for cross-node queries, the advantage of router execution is 100% SQL coverage. Queries executing inside a node are run in a full-featured PostgreSQL worker instance. The disadvantage of router execution is the reduced parallelism of executing a query using only one computer.

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 `citrus.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.

Subquery/CTE Push-Pull Execution

If necessary Citrus can gather results from subqueries and CTEs into the coordinator node and then push them back across workers for use by an outer query. This allows Citrus to support a greater variety of SQL constructs, and even mix executor types between a query and its subqueries.

For example, having subqueries in a `WHERE` clause sometimes cannot execute inline at the same time as the main query, but must be done separately. Suppose a web analytics application maintains a `visits` table partitioned by `page_id`. To query the number of visitor sessions on the top twenty most visited pages, we can use a subquery to find the list of pages, then an outer query to count the sessions.

```
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 20
)
GROUP BY page_id;
```

The real-time executor would like to run a fragment of this query against each shard by `page_id`, counting distinct `session_ids`, and combining the results on the coordinator. However the `LIMIT` in the subquery means the subquery cannot be executed as part of the fragment. By recursively planning the query Citrus can run the subquery separately, push the results to all workers, run the main fragment query, and pull the results back to the coordinator. The “push-pull” design supports a subqueries like the one above.

Let’s see this in action by reviewing the `EXPLAIN` output for this query. It’s fairly involved:

```
GroupAggregate (cost=0.00..0.00 rows=0 width=0)
  Group Key: remote_scan.page_id
  -> Sort (cost=0.00..0.00 rows=0 width=0)
    Sort Key: remote_scan.page_id
    -> Custom Scan (Citrus Real-Time) (cost=0.00..0.00 rows=0 width=0)
      -> Distributed Subplan 6_1
        -> Limit (cost=0.00..0.00 rows=0 width=0)
          -> Sort (cost=0.00..0.00 rows=0 width=0)
            Sort Key: COALESCE((pg_catalog.sum((COALESCE((pg_catalog.sum(remote_scan.
↪worker_column_2))::bigint, '0')::bigint)))::bigint, '0')::bigint) DESC
          -> HashAggregate (cost=0.00..0.00 rows=0 width=0)
            Group Key: remote_scan.page_id
            -> Custom Scan (Citrus 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
-> Limit (cost=1883.00..1883.05 rows=20 width=12)
-> Sort (cost=1883.00..1965.54 rows=33017 width=12)
Sort Key: (count(*)) DESC
-> HashAggregate (cost=674.25..1004.42 rows=33017 width=12)
Group Key: page_id
-> Seq Scan on visits_102264 visits (cost=0.00..509.17
↪rows=33017 width=4)
Task Count: 32
Tasks Shown: One of 32
-> Task
Node: host=localhost port=5433 dbname=postgres
-> HashAggregate (cost=734.53..899.61 rows=16508 width=8)
Group Key: visits.page_id, visits.session_id
-> Hash Join (cost=17.00..651.99 rows=16508 width=8)
Hash Cond: (visits.page_id = intermediate_result.page_id)
-> Seq Scan on visits_102264 visits (cost=0.00..509.17 rows=33017
↪width=8)
-> Hash (cost=14.50..14.50 rows=200 width=4)
-> HashAggregate (cost=12.50..14.50 rows=200 width=4)
Group Key: intermediate_result.page_id
-> Function Scan on read_intermediate_result intermediate_result
↪(cost=0.00..10.00 rows=1000 width=4)

```

Let's break it apart and examine each piece.

```

GroupAggregate (cost=0.00..0.00 rows=0 width=0)
Group Key: remote_scan.page_id
-> Sort (cost=0.00..0.00 rows=0 width=0)
Sort Key: remote_scan.page_id

```

The root of the tree is what the coordinator node does with the results from the workers. In this case it is grouping them, and GroupAggregate requires they be sorted first.

```

-> Custom Scan (Citrus Real-Time) (cost=0.00..0.00 rows=0 width=0)
-> Distributed Subplan 6_1
.

```

The custom scan has two large sub-trees, starting with a “distributed subplan.”

```

-> Limit (cost=0.00..0.00 rows=0 width=0)
-> Sort (cost=0.00..0.00 rows=0 width=0)
Sort Key: COALESCE((pg_catalog.sum((COALESCE((pg_catalog.sum(remote_scan.
↪worker_column_2))::bigint, '0'::bigint))))::bigint, '0'::bigint) DESC
-> HashAggregate (cost=0.00..0.00 rows=0 width=0)
Group Key: remote_scan.page_id
-> Custom Scan (Citrus 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
-> Limit (cost=1883.00..1883.05 rows=20 width=12)
-> Sort (cost=1883.00..1965.54 rows=33017 width=12)
Sort Key: (count(*)) DESC
-> HashAggregate (cost=674.25..1004.42 rows=33017 width=12)
Group Key: page_id
-> Seq Scan on visits_102264 visits (cost=0.00..509.17
↪rows=33017 width=4)

```


Worker nodes run the above for each of the thirty-two shards (Citrus is choosing one representative for display). We can recognize all the pieces of the `IN (...)` subquery: the sorting, grouping and limiting. When all workers have completed this query, they send their output back to the coordinator which puts it together as “intermediate results.”

```
Task Count: 32
Tasks Shown: One of 32
-> Task
Node: host=localhost port=5433 dbname=postgres
-> HashAggregate (cost=734.53..899.61 rows=16508 width=8)
Group Key: visits.page_id, visits.session_id
-> Hash Join (cost=17.00..651.99 rows=16508 width=8)
Hash Cond: (visits.page_id = intermediate_result.page_id)
```

Citus starts another real-time job in this second subtree. It’s going to count distinct sessions in visits. It uses a JOIN to connect with the intermediate results. The intermediate results will help it restrict to the top twenty pages.

```
-> Seq Scan on visits_102264 visits (cost=0.00..509.17 rows=33017
->width=8)
-> Hash (cost=14.50..14.50 rows=200 width=4)
-> HashAggregate (cost=12.50..14.50 rows=200 width=4)
Group Key: intermediate_result.page_id
-> Function Scan on read_intermediate_result intermediate_result
->(cost=0.00..10.00 rows=1000 width=4)
```

The worker internally retrieves intermediate results using a `read_intermediate_result` function which loads data from a file that was copied in from the coordinator node.

This example showed how Citrus executed the query in multiple steps with a distributed subplan, and how you can use EXPLAIN to learn about distributed query execution.

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.

Manual Query Propagation

When the user issues a query, the Citrus coordinator partitions it into smaller query fragments where each query fragment can be run independently on a worker shard. This allows Citrus to distribute each query across the cluster.

However the way queries are partitioned into fragments (and which queries are propagated at all) varies by the type of query. In some advanced situations it is useful to manually control this behavior. Citrus provides utility functions to propagate SQL to workers, shards, or placements.

Manual query propagation bypasses coordinator logic, locking, and any other consistency checks. These functions are available as a last resort to allow statements which Citrus otherwise does not run natively. Use them carefully to avoid data inconsistency and deadlocks.

Running on all Workers

The least granular level of execution is broadcasting a statement for execution on all workers. This 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; $cmd$);
```

Note: The `run_command_on_workers` function and other manual propagation commands in this section can run only queries which return a single column and single row.

Running on all Shards

The next level of granularity is running a command across all shards of a particular distributed table. It can be useful, for instance, in reading the properties of a table directly on workers. Queries run locally on a worker node have full access to metadata such as table statistics.

The `run_command_on_shards` function applies a SQL command to each shard, where the shard name is provided for interpolation in the command. Here is an example of estimating the row count for a distributed table by using the `pg_class` table on each worker to estimate the number of rows for each shard. Notice the `%s` which will be replaced with each shard's name.

```
-- Get the estimated row count for a distributed table by summing the
-- estimated counts of rows for each shard.
SELECT sum(result::bigint) AS estimated_count
FROM run_command_on_shards(
    'my_distributed_table',
    $cmd$
    SELECT reltuples
    FROM pg_class c
    JOIN pg_catalog.pg_namespace n on n.oid=c.relnamespace
    WHERE n.nspname||'.'||relname = '%s';
    $cmd$
);
```

Running on all Placements

The most granular level of execution is running a command across all shards and their replicas (aka *placements*). It can be useful for running data modification commands, which must apply to every replica to ensure consistency.

For example, suppose a distributed table has an `updated_at` field, and we want to “touch” all rows so that they are marked as updated at a certain time. An ordinary `UPDATE` statement on the coordinator requires a filter by the distribution column, but we can manually propagate the update across all shards and replicas:

```
-- note we're using a hard-coded date rather than
-- a function such as "now()" because the query will
-- run at slightly different times on each replica

SELECT run_command_on_placements(
    'my_distributed_table',
```

```
$cmd$
    UPDATE %s SET updated_at = '2017-01-01';
$cmd$
);
```

A useful companion to `run_command_on_placements` is `run_command_on_collocated_placements`. It interpolates the names of *two* placements of *co-located* distributed tables into a query. The placement pairs are always chosen to be local to the same worker where full SQL coverage is available. Thus we can use advanced SQL features like triggers to relate the tables:

```
-- Suppose we have two distributed tables
CREATE TABLE little_vals (key int, val int);
CREATE TABLE big_vals    (key int, val int);
SELECT create_distributed_table('little_vals', 'key');
SELECT create_distributed_table('big_vals',    'key');

-- We want to synchronise them so that every time little_vals
-- are created, big_vals appear with double the value
--
-- First we make a trigger function on each worker, which will
-- take the destination table placement as an argument
SELECT run_command_on_workers($cmd$
    CREATE OR REPLACE FUNCTION embiggen() RETURNS TRIGGER AS $$
    BEGIN
        IF (TG_OP = 'INSERT') THEN
            EXECUTE format(
                'INSERT INTO %s (key, val) SELECT ($1).key, ($1).val*2;',
                TG_ARGV[0]
            ) USING NEW;
        END IF;
        RETURN NULL;
    END;
    $$ LANGUAGE plpgsql;
$cmd$);

-- Next we relate the co-located tables by the trigger function
-- on each co-located placement
SELECT run_command_on_collocated_placements(
    'little_vals',
    'big_vals',
    $cmd$
        CREATE TRIGGER after_insert AFTER INSERT ON %s
        FOR EACH ROW EXECUTE PROCEDURE embiggen(%s)
    $cmd$
);
```

Limitations

- There are no safe-guards against deadlock for multi-statement transactions.
- There are no safe-guards against mid-query failures and resulting inconsistencies.
- Query results are cached in memory; these functions can't deal with very big result sets.
- The functions error out early if they cannot connect to a node.
- You can do very bad things!

SQL Support and Workarounds

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.

Citus has 100% SQL coverage for any queries it is able to execute on a single worker node. These kind of queries are called *router executable* and are common in *Multi-tenant Applications* when accessing information about a single tenant.

Even cross-node queries (used for parallel computations) support most SQL features. However some SQL features are not supported for queries which combine information from multiple nodes.

Limitations for Cross-Node SQL Queries:

- [Window functions](#) are supported only when they include the distribution column in `PARTITION BY`.
- `SELECT ... FOR UPDATE` work in *Router Executor* queries only
- `TABLESAMPLE` work in *Router Executor* queries only
- Correlated subqueries are supported only when the correlation is on the *Distribution Column* and the subqueries conform to subquery pushdown rules (e.g., grouping by the distribution column, with no `LIMIT` or `LIMIT OFFSET` clause).
- Recursive CTEs work in *Router Executor* queries only
- Grouping sets work in *Router Executor* queries only

To learn more about PostgreSQL and its features, you can visit the [PostgreSQL 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](#).

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. Even in the real-time analytics use-cases, with queries that span across nodes, Citus supports the majority of statements. The few types of unsupported queries are listed in *Are there any PostgreSQL features not supported by Citus?* Many of the unsupported features have workarounds; below are a number of the most useful.

JOIN a local and a distributed table

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

```
SELECT * FROM local JOIN dist USING (id);

/*
ERROR:  relation local is not distributed
STATEMENT:  SELECT * FROM local JOIN dist USING (id);
ERROR:  XX000: relation local is not distributed
LOCATION:  DistributedTableCacheEntry, metadata_cache.c:711
*/
```

Although you can't join such tables directly, by wrapping the local table in a subquery or CTE you can make Citus' recursive query planner copy the local table data to worker nodes. By colocating the data this allows the query to proceed.

```
-- either

SELECT *
  FROM (SELECT * FROM local) AS x
 JOIN dist USING (id);

-- or

WITH x AS (SELECT * FROM local)
SELECT * FROM x
JOIN dist USING (id);
```

Remember that the coordinator will send the results in the subquery or CTE to all workers which require it for processing. Thus it's best to either add the most specific filters and limits to the inner query as possible, or else aggregate the table. That reduces the network overhead which such a query can cause. More about this in [Subquery/CTE Network Overhead](#).

INSERT...SELECT upserts lacking distribution column

Citus supports INSERT...SELECT...ON CONFLICT statements between co-located tables when the distribution column is among those columns selected and inserted. Also aggregates in the statement must include the distribution column in the GROUP BY clause. Failing to meet these conditions will raise an error:

```
ERROR: ON CONFLICT is not supported in INSERT ... SELECT via coordinator
```

If the upsert is an important operation in your application, the ideal solution is to model the data so that the source and destination tables are co-located, and so that the distribution column can be part of the GROUP BY clause in the upsert statement (if aggregating). However if this is not feasible then the workaround is to materialize the select query in a temporary distributed table, and upsert from there.

```
-- workaround for
-- INSERT INTO dest_table <query> ON CONFLICT <upsert clause>

BEGIN;
CREATE UNLOGGED TABLE temp_table (LIKE dest_table);
SELECT create_distributed_table('temp_table', 'tenant_id');
INSERT INTO temp_table <query>;
INSERT INTO dest_table SELECT * FROM temp_table <upsert clause>;
DROP TABLE temp_table;
END;
```

Temp Tables: the Workaround of Last Resort

There are still a few queries that are *unsupported* even with the use of push-pull execution via subqueries. One of them is running window functions that partition by a non-distribution column.

Suppose we have a table called `github_events`, distributed by the column `user_id`. Then the following window function will not work:

```
-- this won't work
```

```
SELECT repo_id, org->'id' as org_id, count(*)
  OVER (PARTITION BY repo_id) -- repo_id is not distribution column
FROM github_events
WHERE repo_id IN (8514, 15435, 19438, 21692);
```

There is another trick though. We can pull the relevant information to the coordinator as a temporary table:

```
-- grab the data, minus the aggregate, into a local table

CREATE TEMP TABLE results AS (
  SELECT repo_id, org->'id' as org_id
    FROM github_events
   WHERE repo_id IN (8514, 15435, 19438, 21692)
);

-- now run the aggregate locally

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

Creating a temporary table on the coordinator is a last resort. It is limited by the disk size and CPU of the node.

Citus Utility Functions

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.

Table and Shard DDL

`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_type: (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](#).)

Keep in mind that the default value of `colocate_with` does implicit co-location. As [Table Co-Location](#) explains, this can be a great thing when tables are related or will be joined. However when two tables are unrelated but happen to use the same datatype for their distribution columns, accidentally co-locating them can decrease performance during [shard rebalancing](#). The table shards will be moved together unnecessarily in a "cascade."

If a new distributed table is not related to other tables, it's best to specify `colocate_with => 'none'`.

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)*.

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.

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');
```

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');
```

mark_tables_colocated

The `mark_tables_colocated()` function takes a distributed table (the source), and a list of others (the targets), and puts the targets into the same co-location group as the source. If the source is not yet in a group, this function creates one, and assigns the source and targets to it.

Usually colocating tables ought to be done at table distribution time via the `colocate_with` parameter of *create_distributed_table*. But `mark_tables_colocated` can take care of it if necessary.

Arguments

source_table_name: Name of the distributed table whose co-location group the targets will be assigned to match.

target_table_names: Array of names of the distributed target tables, must be non-empty. These distributed tables must match the source table in:

- distribution method
- distribution column type
- replication type
- shard count

Failing this, Citrus will raise an error. For instance, attempting to colocate tables `apples` and `oranges` whose distribution column types differ results in:

```
ERROR:  XX000: cannot colocate tables apples and oranges
DETAIL:  Distribution column types don't match for apples and oranges.
```

Return Value

N/A

Example

This example puts `products` and `line_items` in the same co-location group as `stores`. The example assumes that these tables are all distributed on a column with matching type, most likely a “store id.”

```
SELECT mark_tables_colocated('stores', ARRAY['products', 'line_items']);
```

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');
```

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);
```

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)
```

Table and Shard DML

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)
```

master_apply_delete_command

The `master_apply_delete_command()` function is used to delete shards which match the criteria specified by the delete command on an *append* distributed table. 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)
```

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 `citrus.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');
```

Metadata / Configuration Information

master_add_node

The `master_add_node()` function registers a new node addition in the cluster in the Citrus 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.

group_id: A group of one primary server and zero or more secondary servers, relevant only for streaming replication. Default 0

node_role: Whether it is 'primary' or 'secondary'. Default 'primary'

node_cluster: The cluster name. Default 'default'

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 | groupid
--|-----|-----|-----|-----|-----|-----|-----
 7      | 7       | new-node | 12345    | default  | f           | t       | 0
primary | default
(1 row)
```

master_update_node

The `master_update_node()` function changes the hostname and port for a node registered in the Citrus metadata table *pg_dist_node*.

Arguments

node_id: id from the *pg_dist_node* table.

node_name: updated DNS name or IP address for the node.

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

Return Value

N/A

Example

```
select * from master_update_node(123, 'new-address', 5432);
```

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.

group_id: A group of one primary server and zero or more secondary servers, relevant only for streaming replication. Default 0

node_role: Whether it is 'primary' or 'secondary'. Default 'primary'

node_cluster: The cluster name. Default 'default'

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 | groupid
↪-----+-----+-----+-----+-----+-----+-----+-----
↪ 7 | 7 | new-node | 12345 | default | f | f | 0
↪ primary | default
(1 row)
```

master_activate_node

The `master_activate_node` function marks a node as active in the Citrus 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 | noderole
-----+-----+-----+-----+-----+-----+-----+-----
↪ | nodecluster
↪ +-----+
       7 |       7 | new-node |   12345 | default | f           | t           | primary
↪ | default
(1 row)
```

master_disable_node

The `master_disable_node` function is the opposite of `master_activate_node`. It marks a node as inactive in the Citrus 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);
```

master_add_secondary_node

The `master_add_secondary_node()` function registers a new secondary node in the cluster for an existing primary node. It updates the Citrus metadata table `pg_dist_node`.

Arguments

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

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

primary_name: DNS name or IP address of the primary node for this secondary.

primary_port: The port on which PostgreSQL is listening on the primary node.

node_cluster: The cluster name. Default 'default'

Return Value

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

Example

```
select * from master_add_secondary_node('new-node', 12345, 'primary-node', 12345);
 nodeid | groupid | nodename | nodeport | noderack | hasmetadata | isactive |
↪-----+-----+-----+-----+-----+-----+-----+
↪-----+-----+-----+-----+-----+-----+-----+
       7 |        7 | new-node |    12345 | default | f           | t           |
↪secondary | default
(1 row)
```

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. 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)
```

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)
```

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)
```

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 Citrus 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)
```

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    |
-----
```

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
```

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
```

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
```

citus_stat_statements_reset

Removes all rows from *citus_stat_statements*. Note that this works independently from `pg_stat_statements_reset()`. To reset all stats, call both functions.

Arguments

N/A

Return Value

None

Cluster Management And Repair Functions

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);
```

master_move_shard_placement

Note: The `master_move_shard_placement` function is a part of Citus Enterprise. Please [contact us](#) to obtain this functionality.

This function moves a given shard (and shards co-located with it) from one node to another. It is typically used indirectly during shard rebalancing rather than being called directly by a database administrator.

There are two ways to move the data: blocking or nonblocking. The blocking approach means that during the move all modifications to the shard are paused. The second way, which avoids blocking shard writes, relies on Postgres 10 logical replication.

After a successful move operation, shards in the source node get deleted. If the move fails at any point, this function throws an error and leaves the source and target nodes unchanged.

Arguments

shard_id: Id of the shard to be moved.

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.

shard_transfer_mode: (Optional) Specify the method of replication, whether to use PostgreSQL logical replication or a cross-worker COPY command. The possible values are:

- `auto`: Require replica identity if logical replication is possible, otherwise use legacy behaviour (e.g. for shard repair, PostgreSQL 9.6). This is the default value.
- `force_logical`: Use logical replication even if the table doesn’t have a replica identity. Any concurrent update/delete statements to the table will fail during replication.
- `block_writes`: Use COPY (blocking writes) for tables lacking primary key or replica identity.

Return Value

N/A

Example

```
SELECT master_move_shard_placement(12345, 'from_host', 5432, 'to_host', 5432);
```

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 $(1 - \text{threshold}) * \text{average_utilization} \dots (1 + \text{threshold}) * \text{average_utilization}$ 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.

shard_transfer_mode: (Optional) Specify the method of replication, whether to use PostgreSQL logical replication or a cross-worker COPY command. The possible values are:

- `auto`: Require replica identity if logical replication is possible, otherwise use legacy behaviour (e.g. for shard repair, PostgreSQL 9.6). This is the default value.
- `force_logical`: Use logical replication even if the table doesn't have a replica identity. Any concurrent update/delete statements to the table will fail during replication.
- `block_writes`: Use COPY (blocking writes) for tables lacking primary key or replica identity.

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}');
```

get_rebalance_progress

Note: The `get_rebalance_progress()` function is a part of Citrus Enterprise. Please [contact us](#) to obtain this functionality.

Once a shard rebalance begins, the `get_rebalance_progress()` function lists the progress of every shard involved. It monitors the moves planned and executed by `rebalance_table_shards()`.

Arguments

N/A

Return Value

Tuples containing these columns:

- **sessionid**: Postgres PID of the rebalance monitor
- **table_name**: The table whose shards are moving
- **shardid**: The shard in question
- **shard_size**: Size in bytes
- **sourcename**: Hostname of the source node
- **sourceport**: Port of the source node
- **targetname**: Hostname of the destination node
- **targetport**: Port of the destination node
- **progress**: 0 = waiting to be moved; 1 = moving; 2 = complete

Example

```
SELECT * FROM get_rebalance_progress();
```

```
-----
| sessionid | table_name | shardid | shard_size | sourcename | sourceport | ↵
↪targetname | targetport | progress |
-----
|      7083 | foo       | 102008 | 1204224 | n1.foobar.com | 5432 | n4.
↪foobar.com | 5432 | 0 |
|      7083 | foo       | 102009 | 1802240 | n1.foobar.com | 5432 | n4.
↪foobar.com | 5432 | 0 |
|      7083 | foo       | 102018 | 614400 | n2.foobar.com | 5432 | n4.
↪foobar.com | 5432 | 1 |
|      7083 | foo       | 102019 | 8192 | n3.foobar.com | 5432 | n4.
↪foobar.com | 5432 | 2 |
-----
```

replicate_table_shards

Note: The `replicate_table_shards` function is a part of Citrus 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);
```

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 |
-----

```

citus_create_restore_point

Temporarily blocks writes to the cluster, and creates a named restore point on all nodes. This function is similar to `pg_create_restore_point`, but applies to all nodes and makes sure the restore point is consistent across them. This function is well suited to doing point-in-time recovery, and cluster forking.

Arguments

name: The name of the restore point to create.

Return Value

coordinator_lsn: Log sequence number of the restore point in the coordinator node WAL.

Examples

```
select citus_create_restore_point('foo');
```

```

-----
| citus_create_restore_point |
-----
| 0/1EA2808                  |
-----

```

Citus Tables and Views

Coordinator Metadata

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.

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' reference table: 'n'
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 :- citrus 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)
```

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
<code>logicalrelid</code>	<code>regclass</code>	Distributed table to which this shard belongs. This value references the <code>relfilenode</code> column in the <code>pg_class</code> system catalog table.
<code>shardid</code>	<code>bigint</code>	Globally unique identifier assigned to this shard.
<code>shardstorage</code>	<code>char</code>	Type of storage used for this shard. Different storage types are discussed in the table below.
<code>shardminvalue</code>	<code>text</code>	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).
<code>shardmaxvalue</code>	<code>text</code>	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)
```

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 <code>cstore_fdw</code> tables)
FOREIGN	'f'	Indicates that shard stores foreign data. (Used by distributed <code>file_fdw</code> tables)

Shard placement table

The `pg_dist_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 value 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.
placementid	bigint	Unique auto-generated identifier for each individual placement.
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.

```
SELECT * from pg_dist_placement;
 shardid | shardstate | shardlength | placementid | groupid
-----+-----+-----+-----+-----
  102008 |         1 |          0 |          1 |        1
  102008 |         1 |          0 |          2 |        2
  102009 |         1 |          0 |          3 |        2
  102009 |         1 |          0 |          4 |        3
  102010 |         1 |          0 |          5 |        3
  102010 |         1 |          0 |          6 |        4
  102011 |         1 |          0 |          7 |        4
```

Note: As of Citrus 7.0 the analogous table pg_dist_shard_placement has been deprecated. It included the node name and port for each 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

That information is now available by joining `pg_dist_placement` with `pg_dist_node` on the `groupid`. For compatibility Citus still provides `pg_dist_shard_placement` as a view. However we recommend using the new, more normalized, tables when possible.

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_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 <code>master_apply_delete_command</code> call and fails, the placement is moved to this state. Users can then delete these shards as a subsequent background activity.

Worker node table

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

Name	Type	Description
<code>nodeid</code>	<code>int</code>	Auto-generated identifier for an individual node.
<code>groupid</code>	<code>int</code>	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 <code>nodeid</code> .
<code>nodename</code>	<code>text</code>	Host Name or IP Address of the PostgreSQL worker node.
<code>nodeport</code>	<code>int</code>	Port number on which the PostgreSQL worker node is listening.
<code>noderack</code>	<code>text</code>	(Optional) Rack placement information for the worker node.
<code>hasmetadata</code>	<code>boolean</code>	Reserved for internal use.
<code>isactive</code>	<code>boolean</code>	Whether the node is active accepting shard placements.
<code>noderole</code>	<code>text</code>	Whether the node is a primary or secondary
<code>nodecluster</code>	<code>text</code>	The name of the cluster containing this node

```
SELECT * from pg_dist_node;
nodeid | groupid | nodename | nodeport | noderack | hasmetadata | isactive |
↪noderole | nodecluster
```

```

-----+-----+-----+-----+-----+-----+-----+-----+
↪--+-----+
      1 |      1 | localhost | 12345 | default | f          | t          | ↪
↪primary | default
      2 |      2 | localhost | 12346 | default | f          | t          | ↪
↪primary | default
      3 |      3 | localhost | 12347 | default | f          | t          | ↪
↪primary | default
(3 rows)

```

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.
distributioncolumnstype	oid	The type of the distribution column for all tables in this co-location group.

```

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

```

Query statistics table

Note: The `citus_stat_statements` view is a part of Citus Enterprise. Please [contact us](#) to obtain this functionality.

Citus provides `citus_stat_statements` for stats about how queries are being executed, and for whom. It's analogous to (and can be joined with) the `pg_stat_statements` view in PostgreSQL which tracks statistics about query

speed.

This view can trace queries to originating tenants in a multi-tenant application, which helps for deciding when to do *Tenant Isolation*.

Name	Type	Description
queryid	bigint	identifier (good for pg_stat_statements joins)
userid	oid	user who ran the query
dbid	oid	database instance of coordinator
query	text	anonymized query string
executor	text	Citus <i>executor</i> used: real-time, task-tracker, router, or insert-select
partition_key	text	value of distribution column in router-executed queries, else NULL
calls	bigint	number of times the query was run

```
-- create and populate distributed table
create table foo ( id int );
select create_distributed_table('foo', 'id');
insert into foo select generate_series(1,100);

-- enable stats
-- pg_stat_statements must be in shared_preload libraries
create extension pg_stat_statements;

select count(*) from foo;
select * from foo where id = 42;

select * from citus_stat_statements;
```

Results:

queryid	userid	dbid	query	
↪executor	partition_key	calls		
1496051219	16384	16385	select count(*) from foo;	real-
↪time	NULL	1		
2530480378	16384	16385	select * from foo where id = \$1	
↪router	42	1		
3233520930	16384	16385	insert into foo select generate_series(\$1,\$2)	
↪insert-select	NULL	1		

Caveats:

- The stats data is not replicated, and won't survive database crashes or failover
- It's a coordinator node feature, with no *Citus MX* support
- Tracks a limited number of queries, set by the `pg_stat_statements.max` GUC (default 5000)
- To truncate the table, use the `citus_stat_statements_reset()` function

Distributed Query Activity

With *Citus MX* users can execute distributed queries from any node. Examining the standard Postgres `pg_stat_activity` view on the coordinator won't include those worker-initiated queries. Also queries might get blocked on row-level locks on one of the shards on a worker node. If that happens then those queries would not show up in `pg_locks` on the Citrus coordinator node.

Citus provides special views to watch queries and locks throughout the cluster, including shard-specific queries used internally to build results for distributed queries.

- **citus_dist_stat_activity**: shows the distributed queries that are executing on all nodes. A superset of `pg_stat_activity`, usable wherever the latter is.
- **citus_worker_stat_activity**: shows queries on workers, including fragment queries against individual shards.
- **citus_lock_waits**: Blocked queries throughout the cluster.

The first two views include all columns of `pg_stat_activity` plus the host/port of the worker that initiated the query and the host/port of the coordinator node of the cluster.

For example, consider counting the rows in a distributed table:

```
-- run from worker on localhost:9701

SELECT count(*) FROM users_table;
```

We can see the query appear in `citus_dist_stat_activity`:

```
SELECT * FROM citus_dist_stat_activity;

-[ RECORD 1 ]-----+-----
query_hostname      | localhost
query_hostport      | 9701
master_query_host_name | localhost
master_query_host_port | 9701
transaction_number  | 1
transaction_stamp    | 2018-10-05 13:27:20.691907+03
datid                | 12630
datname              | postgres
pid                  | 23723
usesysid             | 10
username             | citus
application_name     | psql
client_addr          |
client_hostname      |
client_port          | -1
backend_start        | 2018-10-05 13:27:14.419905+03
xact_start           | 2018-10-05 13:27:16.362887+03
query_start          | 2018-10-05 13:27:20.682452+03
state_change         | 2018-10-05 13:27:20.896546+03
wait_event_type      | Client
wait_event           | ClientRead
state                | idle in transaction
backend_xid           |
backend_xmin         |
query                | SELECT count(*) FROM users_table;
backend_type         | client backend
```

This query requires information from all shards. Some of the information is in shard `users_table_102038` which happens to be stored in `localhost:9700`. We can see a query accessing the shard by looking at the `citus_worker_stat_activity` view:

```
SELECT * FROM citus_worker_stat_activity;

-[ RECORD 1 ]-----+-----
query_hostname      | localhost
```

```

query_hostport      | 9700
master_query_host_name | localhost
master_query_host_port | 9701
transaction_number   | 1
transaction_stamp     | 2018-10-05 13:27:20.691907+03
datid                | 12630
datname              | postgres
pid                  | 23781
usesysid             | 10
username             | citus
application_name      | citus
client_addr          | ::1
client_hostname       |
client_port          | 51773
backend_start         | 2018-10-05 13:27:20.75839+03
xact_start            | 2018-10-05 13:27:20.84112+03
query_start           | 2018-10-05 13:27:20.867446+03
state_change          | 2018-10-05 13:27:20.869889+03
wait_event_type       | Client
wait_event            | ClientRead
state                 | idle in transaction
backend_xid           |
backend_xmin          |
query                 | COPY (SELECT count(*) AS count FROM users_table_102038 users_
↪table WHERE true) TO STDOUT
backend_type           | client backend

```

The query field shows data being copied out of the shard to be counted.

Note: If a router query (e.g. single-tenant in a multi-tenant application, `SELECT * FROM table WHERE tenant_id = X`) is executed without a transaction block, then `master_query_host_name` and `master_query_host_port` columns will be NULL in `citus_worker_stat_activity`.

To see how `citus_lock_waits` works, we can generate a locking situation manually. First we'll set up a test table from the coordinator:

```

CREATE TABLE numbers AS
  SELECT i, 0 AS j FROM generate_series(1,10) AS i;
SELECT create_distributed_table('numbers', 'i');

```

Then, using two sessions on the coordinator, we can run this sequence of statements:

```

-- session 1                                -- session 2
-----
BEGIN;
UPDATE numbers SET j = 2 WHERE i = 1;
                                           BEGIN;
                                           UPDATE numbers SET j = 3 WHERE i = 1;
                                           -- (this blocks)

```

The `citus_lock_waits` view shows the situation.

```

SELECT * FROM citus_lock_waits;

-[ RECORD 1 ]-----+-----
waiting_pid        | 88624

```

```

blocking_pid          | 88615
blocked_statement     | UPDATE numbers SET j = 3 WHERE i = 1;
current_statement_in_blocking_process | UPDATE numbers SET j = 2 WHERE i = 1;
waiting_node_id       | 0
blocking_node_id      | 0
waiting_node_name     | coordinator_host
blocking_node_name     | coordinator_host
waiting_node_port      | 5432
blocking_node_port     | 5432

```

In this example the queries originated on the coordinator, but the view can also list locks between queries originating on workers (executed with Citus MX for instance).

Tables on all Nodes

Citus has other informational tables and views which are accessible on all nodes, not just the coordinator.

Connection Credentials Table

Note: This table is a part of Citus Enterprise Edition. Please [contact us](#) to obtain this functionality.

The `pg_dist_authinfo` table holds authentication parameters used by Citus nodes to connect to one another.

Name	Type	Description
<code>nodeid</code>	integer	Node id from <i>Worker node table</i> , or 0, or -1
<code>rolename</code>	name	Postgres role
<code>authinfo</code>	text	Space-separated libpq connection parameters

Upon beginning a connection, a node consults the table to see whether a row with the destination `nodeid` and desired `rolename` exists. If so, the node includes the corresponding `authinfo` string in its libpq connection. A common example is to store a password, like `'password=abc123'`, but you can review the [full list](#) of possibilities.

The parameters in `authinfo` are space-separated, in the form `key=val`. To write an empty value, or a value containing spaces, surround it with single quotes, e.g., `keyword='a value'`. Single quotes and backslashes within the value must be escaped with a backslash, i.e., `\'` and `\\`.

The `nodeid` column can also take the special values 0 and -1, which mean *all nodes* or *loopback connections*, respectively. If, for a given node, both specific and all-node rules exist, the specific rule has precedence.

```

SELECT * FROM pg_dist_authinfo;

 nodeid | rolename | authinfo
-----+-----+-----
    123 | jdoe    | password=abc123
(1 row)

```

Connection Pooling Credentials

Note: This table is a part of Citus Enterprise Edition. Please [contact us](#) to obtain this functionality.

If you want to use a connection pooler to connect to a node, you can specify the pooler options using `pg_dist_poolinfo`. This metadata table holds the host, port and database name for Citus to use when connecting to a node through a pooler.

If pool information is present, Citus will try to use these values instead of setting up a direct connection. The `pg_dist_poolinfo` information in this case supersedes `pg_dist_node`.

Name	Type	Description
nodeid	integer	Node id from <i>Worker node table</i>
poolinfo	text	Space-separated parameters: host, port, or dbname

Note: In some situations Citus ignores the settings in `pg_dist_poolinfo`. For instance *Shard rebalancing* is not compatible with connection poolers such as pgbouncer. In these scenarios Citus will use a direct connection.

```
-- how to connect to node 1 (as identified in pg_dist_node)

INSERT INTO pg_dist_poolinfo (nodeid, poolinfo)
VALUES (1, 'host=127.0.0.1 port=5433');
```

Shards and Indices on MX Workers

Note: The `citus_shards_on_worker` and `citus_shard_indexes_on_worker` views are relevant in Citus MX only. In the non-MX scenario they contain no rows.

Worker nodes store shards as tables that are ordinarily hidden in Citus MX (see *[citus.override_table_visibility \(boolean\)](#)*). The easiest way to obtain information about the shards on each worker is to consult that worker's `citus_shards_on_worker` view. For instance, here are some shards on a worker for the distributed table `test_table`:

```
SELECT * FROM citus_shards_on_worker ORDER BY 2;
 Schema |      Name      | Type | Owner
-----+-----+-----+-----
 public | test_table_1130000 | table | citus
 public | test_table_1130002 | table | citus
```

Indices for shards are also hidden, but discoverable through another view, `citus_shard_indexes_on_worker`:

```
SELECT * FROM citus_shard_indexes_on_worker ORDER BY 2;
 Schema |      Name      | Type | Owner |      Table
-----+-----+-----+-----+-----
 public | test_index_1130000 | index | citus | test_table_1130000
 public | test_index_1130002 | index | citus | test_table_1130002
```

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](#)*.

As an example you can update a setting with:

```
ALTER DATABASE citus SET citus.multi_task_query_log_level = 'log';
```

General configuration

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.

citus.use_secondary_nodes (enum)

Sets the policy to use when choosing nodes for SELECT queries. If this is set to 'always', then the planner will query only nodes which are marked as 'secondary' noderole in *pg_dist_node*.

The supported values for this enum are:

- **never:** (default) All reads happen on primary nodes.
- **always:** Reads run against secondary nodes instead, and insert/update statements are disabled.

citus.cluster_name (text)

Informs the coordinator node planner which cluster it coordinates. Once *cluster_name* is set, the planner will query worker nodes in that cluster alone.

citus.enable_version_checks (boolean)

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.

citus.log_distributed_deadlock_detection (boolean)

Whether to log distributed deadlock detection related processing in the server log. It defaults to false.

citus.distributed_deadlock_detection_factor (floating point)

Sets the time to wait before checking for distributed deadlocks. In particular the time to wait will be this value multiplied by PostgreSQL's *deadlock_timeout* setting. The default value is 2. A value of -1 disables distributed deadlock detection.

citus.node_conninfo (text)

The `citus.node_conninfo` GUC sets non-sensitive `libpq` connection parameters used for all inter-node connections.

```
-- key=value pairs separated by spaces.
-- For example, ssl options:

ALTER DATABASE foo
SET citus.node_conninfo =
    'sslrootcert=/path/to/citus.crt sslmode=verify-full';
```

Citus honors only a whitelisted subset of the options, namely:

- `application_name`
- `connect_timeout`
- `gsslib`[†]
- `keepalives`
- `keepalives_count`
- `keepalives_idle`
- `keepalives_interval`
- `krbsrvname`[†]
- `sslcompression`
- `sslcr1`
- `sslmode`
- `sslrootcert`

([†] = subject to the runtime presence of optional PostgreSQL features)

citus.override_table_visibility (boolean)

Note: This GUC has an effect on Citus MX only.

Shards are stored on the worker nodes as regular tables with an identifier appended to their names. Unlike regular Citus, by default Citus MX does not show shards in the list of tables, when for instance a user runs `\d` in `psql`. However the shards can be made visible in MX by updating the GUC:

```
SET citus.override_table_visibility TO FALSE;
```

```
\d
```

Schema	Name	Type	Owner
public	test_table	table	citus
public	test_table_102041	table	citus
public	test_table_102043	table	citus
public	test_table_102045	table	citus

```
| public      | test_table_102047 | table | citus      |
+-----+-----+-----+-----+
```

Now the `test_table` shards (`test_table_<n>`) appear in the list.

Another way to see the shards is by querying the `citus_shards_on_worker` view.

Data Loading

`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:

- **2pc:** (default) 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`.
- **1pc:** The transactions in 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).

`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.

`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.

`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.

Planner Configuration

`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.

`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.

`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.

Intermediate Data Transfer

`citus.binary_worker_copy_format` (boolean)

Use the binary copy format to transfer intermediate data between workers. During large table joins, Citrus 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.

`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.

`citus.max_intermediate_result_size` (integer)

The maximum size in KB of intermediate results for CTEs and complex subqueries. The default is 1GB, and a value of -1 means no limit. Queries exceeding the limit will be canceled and produce an error message.

DDL

`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 cluster 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*.

Executor Configuration

`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.

`citus.max_task_string_size` (integer)

Sets the maximum size (in bytes) of a worker task call string. Changing this value requires a server restart, it cannot be changed at runtime.

Active worker tasks are tracked in a shared hash table on the master node. This configuration value limits the maximum size of an individual worker task, and affects the size of pre-allocated shared memory.

Minimum: 8192, Maximum 65536, Default 12288

`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.

`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.

`citrus.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. The log line will include the SQL query that was run.
- **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` during development testing, and a lower log-level like `log` during actual production deployment. Choosing `log` will cause multi-task queries to appear in the database logs with the query itself shown after "STATEMENT."

```
LOG: multi-task query about to be executed
HINT: Queries are split to multiple tasks if they have to be split into several_
      ↪queries on the workers.
STATEMENT: select * from foo;
```

Real-time executor configuration

The Citrus 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.

citus.enable_repartition_joins (boolean)

Ordinarily, attempting to perform *Repartition joins* with the real-time executor will fail with an error message. However setting `citus.enable_repartition_joins` to `true` allows Citus to temporarily switch into the task-tracker executor to perform the join. The default value is `false`.

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 daemon 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. Citrus 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.

Explain output

`citus.explain_all_tasks` (boolean)

By default, Citrus 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. Occasionally, 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.

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 Citrus' 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 manipulating 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 Citrus 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 Citrus query planner can easily determine which shards overlap that range and send the query 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, Citrus will maintain minimum and maximum hash ranges for all the created shards. Whenever a row is inserted, updated or deleted, Citrus 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.

Citrus 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.

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.

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` \geq '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 [Citrus Utility Functions](#) 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.

Deleting Data

The most flexible way to modify or delete rows throughout a Citrus cluster with regular SQL statements:

```
DELETE FROM github_events
WHERE created_at >= '2015-01-01 00:03:00';
```

Unlike `master_apply_delete_command`, standard SQL 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.

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;
```

Data Loading

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

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.

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 Functions* section of the documentation.

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.

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.

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 `citushard_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_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 Functions](#) section of the documentation.

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_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.

Pre-processing Data in Citrus

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 Citrus](#).

External Integrations

Ingesting Data from Kafka

Citus can leverage existing Postgres data ingestion tools. For instance, we can use a tool called `kafka-sink-pg-json` to copy JSON messages from a Kafka topic into a database table. As a demonstration, we'll create a `kafka_test` table and ingest data from the `test` topic with a custom mapping of JSON keys to table columns.

The easiest way to experiment with Kafka is using the [Confluent platform](#), which includes Kafka, Zookeeper, and associated tools whose versions are verified to work together.

```
# we're using Confluent 2.0 for kafka-sink-pg-json support
curl -L http://packages.confluent.io/archive/2.0/confluent-2.0.0-2.11.7.tar.gz \
  | tar zx

# Now get the jar and conf files for kafka-sink-pg-json
mkdir sink
curl -L https://github.com/justonedb/kafka-sink-pg-json/releases/download/v1.0.2/
↪justone-jafka-sink-pg-json-1.0.zip -o sink.zip
unzip -d sink $_ && rm $_
```

The download of `kafka-sink-pg-json` contains some configuration files. We want to connect to the coordinator Citus node, so we must edit the configuration file `sink/justone-kafka-sink-pg-json-connector.properties`:

```
# add to sink/justone-kafka-sink-pg-json-connector.properties

# the kafka topic we will use
topics=test

# db connection info
# use your own settings here
db.host=localhost:5432
db.database=postgres
db.username=postgres
db.password=bar

# the schema and table we will use
db.schema=public
db.table=kafka_test

# the JSON keys, and columns to store them
```

```
db.json.parse=@a,/b
db.columns=a,b
```

Notice `db.columns` and `db.json.parse`. The elements of these lists match up, with the items in `db.json.parse` specifying where to find values inside incoming JSON objects.

Note: The paths in `db.json.parse` are written in a language that allows some flexibility in getting values out of JSON. Given the following JSON,

```
{
  "identity":71293145,
  "location": {
    "latitude":51.5009449,
    "longitude":-2.4773414
  },
  "acceleration":[0.01,0.0,0.0]
}
```

here are some example paths and what they match:

- `/@identity` - the path to element 71293145.
- `/@location/@longitude` - the path to element -2.4773414.
- `/@acceleration/#0` - the path to element 0.01
- `/@location` - the path to element `{"latitude":51.5009449,"longitude":-2.4773414}`

Our own scenario is simple. Our events will be objects like `{"a":1,"b":2}`. The parser will pull those values into eponymous columns.

Now that the configuration file is set up, it's time to prepare the database. Connect to the coordinator node with `psql` and run this:

```
-- create metadata tables for kafka-sink-pg-json
\i sink/install-justone-kafka-sink-pg-1.0.sql

-- create and distribute target ingestion table
create table kafka_test ( a int, b int );
select create_distributed_table('kafka_test', 'a');
```

Start the Kafka machinery:

```
# save some typing
export C=confluent-2.0.0

# start zookeeper
$C/bin/zookeeper-server-start \
  $C/etc/kafka/zookeeper.properties

# start kafka server
$C/bin/kafka-server-start \
  $C/etc/kafka/server.properties

# create the topic we'll be reading/writing
$C/bin/kafka-topics --create --zookeeper localhost:2181 \
  --replication-factor 1 --partitions 1 \
  --topic test
```


Run the ingestion program:

```
# the jar files for this are in "sink"
export CLASSPATH=$PWD/sink/*

# Watch for new events in topic and insert them
$C/bin/connect-standalone \
  sink/justone-kafka-sink-pg-json-standalone.properties \
  sink/justone-kafka-sink-pg-json-connector.properties
```

At this point Kafka-Connect is watching the test topic, and will parse events there and insert them into kafka_test. Let's send an event from the command line.

```
echo '{"a":42,"b":12}' | \
  $C/bin/kafka-console-producer --broker-list localhost:9092 --topic test
```

After a small delay the new row will show up in the database.

```
select * from kafka_test;

-----
| a  | b  |
-----
| 42 | 12 |
-----
```

Caveats

- At the time of this writing, kafka-sink-pg-json requires Kafka version 0.9 or earlier.
- The kafka-sink-pg-json connector config file does not provide a way to connect with SSL support, so this tool will not work with Citrus Cloud which requires secure connections.
- A malformed JSON string in the Kafka topic will cause the tool to become stuck. Manual intervention in the topic is required to process more events.

Ingesting Data from Spark

People sometimes use Spark to transform Kafka data, such as by adding computed values. In this section we'll see how to ingest Spark dataframes into a distributed Citrus table.

First let's start a local Spark cluster. It has several moving parts, so the easiest way is to run the pieces with docker-compose.

```
wget https://raw.githubusercontent.com/gettyimages/docker-spark/master/docker-compose.
→ yml

# this may require "sudo" depending on the docker daemon configuration
docker-compose up
```

To do the ingestion into PostgreSQL, we'll be writing custom Scala code. We'll use the Scala Build Tool (SBT) to load dependencies and run our code, so [download SBT](#) and install it on your machine.

Next create a new directory for our project.

```
mkdir sparkcitrus
```

Create a file called `sparkcitrus/build.sbt` to tell SBT our project configuration, and add this:

```
// add this to build.sbt

name := "sparkcitrus"
version := "1.0"

scalaVersion := "2.10.4"

resolvers += Seq(
  "Maven Central" at "http://central.maven.org/maven2/"
)

libraryDependencies += Seq(
  "org.apache.spark" %% "spark-core" % "2.2.1",
  "org.apache.spark" %% "spark-sql" % "2.2.1",
  "org.postgresql" % "postgresql" % "42.2.2"
)
```

Next create a helper Scala class for doing ingestion through JDBC. Add the following to `sparkcitrus/copy.scala`:

```
import java.io.InputStream
import java.sql.DriverManager
import java.util.Properties

import org.apache.spark.sql.{DataFrame, Row}
import org.postgresql.copy.CopyManager
import org.postgresql.core.BaseConnection

object CopyHelper {

  def rowsToInputStream(rows: Iterator[Row]): InputStream = {
    val bytes: Iterator[Byte] = rows.map { row =>
      (row.toSeq
        .map { v =>
          if (v == null) {
            """\N"""
          } else {
            "\"" + v.toString.replaceAll("\"", "\\\"") + "\""
          }
        }
        .mkString("\t") + "\n").getBytes
    }.flatten

    new InputStream {
      override def read(): Int = {
        if (bytes.hasNext) {
          bytes.next & 0xff // make the signed byte an unsigned int
        } else {
          -1
        }
      }
    }
  }

  def copyIn(url: String, df: DataFrame, table: String): Unit = {
```

```

var cols = df.columns.mkString(",")

df.foreachPartition { rows =>
  val conn = DriverManager.getConnection(url)
  try {
    val cm = new CopyManager(conn.asInstanceOf[BaseConnection])
    cm.copyIn(
      s"COPY $table ($cols) " + "\"FROM STDIN WITH (NULL '\n', FORMAT CSV, \
→DELIMITER E'\t')\"",
      rowsToInputStream(rows))
    ()
  } finally {
    conn.close()
  }
}
}

```

Continuing the setup, save some sample data into `people.json`. Note the intentional lack of surrounding square brackets. Later we'll create a Spark dataframe from the data.

```

{"name":"Tanya Rosenau" , "age": 24},
{"name":"Rocky Slay" , "age": 85},
{"name":"Tama Erdmann" , "age": 48},
{"name":"Jared Olivero" , "age": 42},
{"name":"Gudrun Shannon" , "age": 53},
{"name":"Quentin Yoon" , "age": 32},
{"name":"Yanira Huckstep" , "age": 53},
{"name":"Brendon Wesley" , "age": 19},
{"name":"Minda Nordeen" , "age": 79},
{"name":"Katina Woodell" , "age": 83},
{"name":"Nevada Mckinnon" , "age": 65},
{"name":"Georgine Mcbee" , "age": 56},
{"name":"Mittie Vanetten" , "age": 17},
{"name":"Lecia Boyett" , "age": 37},
{"name":"Tobias Mickel" , "age": 69},
{"name":"Jina Mccook" , "age": 82},
{"name":"Cassidy Turrell" , "age": 37},
{"name":"Cherly Skalski" , "age": 29},
{"name":"Reita Bey" , "age": 69},
{"name":"Keely Symes" , "age": 34}

```

Finally, create and distribute a table in Citrus:

```

create table spark_test ( name text, age integer );
select create_distributed_table('spark_test', 'name');

```

Now we're ready to hook everything together. Start up `sbt`:

```

# run this in the sparkcitrus directory

sbt

```

Once inside `sbt`, compile the project and then go into the “console” which is a Scala repl that loads our code and dependencies:

```

sbt:sparkcitrus> compile
[success] Total time: 3 s

```

```
sbt:sparkcitus> console
[info] Starting scala interpreter...

scala>
```

Type these Scala commands into the console:

```
// inside the sbt scala interpreter

import org.apache.spark.sql.SparkSession

// open a session to the Spark cluster
val spark = SparkSession.builder().appName("sparkcitus").config("spark.master", "local
↪").getOrCreate()

// load our sample data into Spark
val df = spark.read.json("people.json")

// this is a simple connection url (it assumes Citus
// is running on localhost:5432), but more complicated
// JDBC urls differ subtly from Postgres urls, see:
// https://jdbc.postgresql.org/documentation/head/connect.html
val url = "jdbc:postgresql://localhost/postgres"

// ingest the data frame using our CopyHelper class
CopyHelper.copyIn(url, df, "spark_test")
```

This uses the CopyHelper to ingest the information. At this point the data will appear in the distributed table.

Note: Our method of ingesting the dataframe is straightforward but doesn't protect against Spark errors. Spark guarantees “at least once” semantics, i.e. a read error can cause a subsequent read to encounter previously seen data.

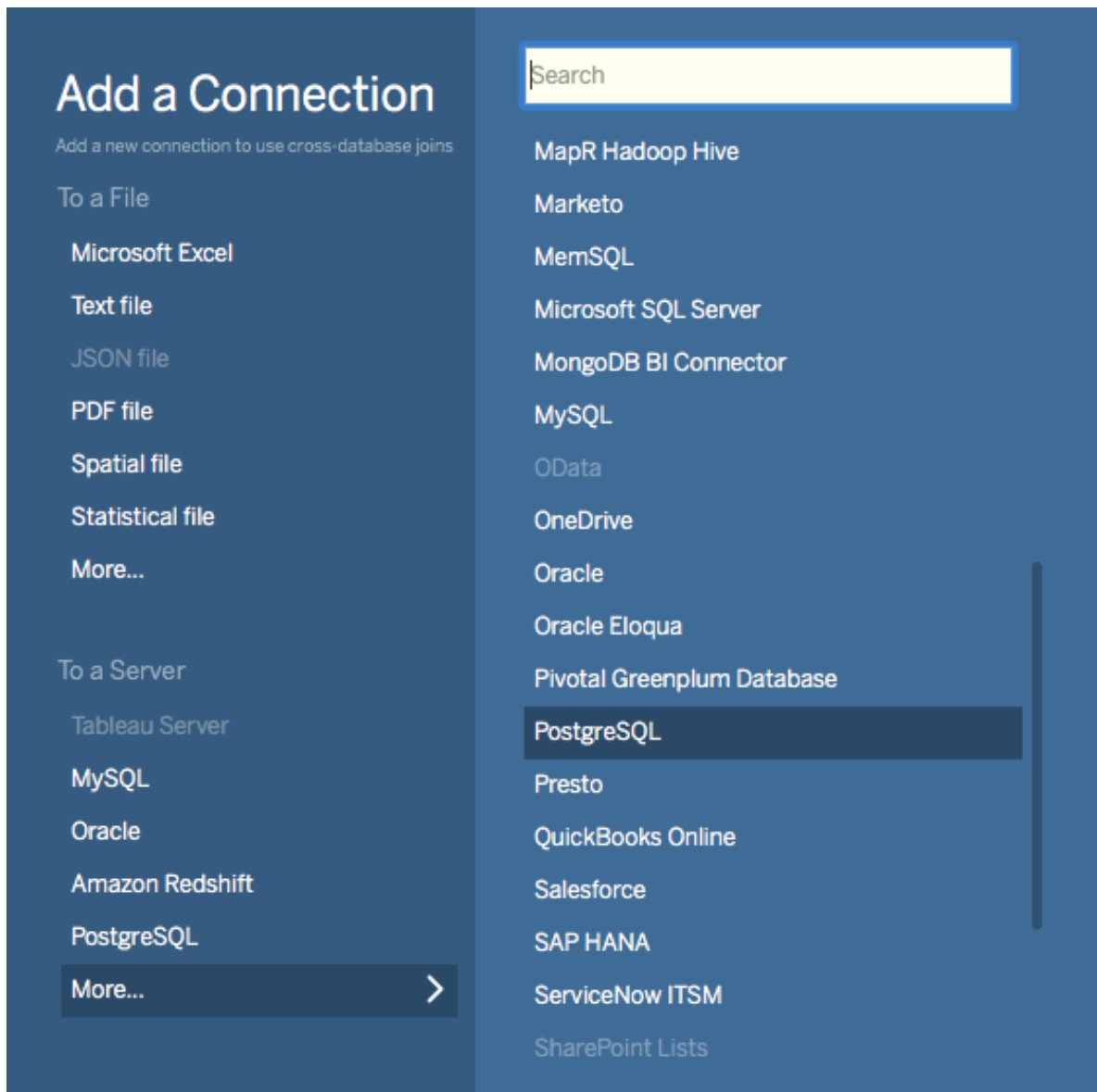
A more complicated, but robust, approach is to use the custom Spark partitioner `spark-citus` so that partitions match up exactly with Citus shards. This allows running transactions directly on worker nodes which can rollback on read failure. See the presentation linked in that repository for more information.

Business Intelligence with Tableau

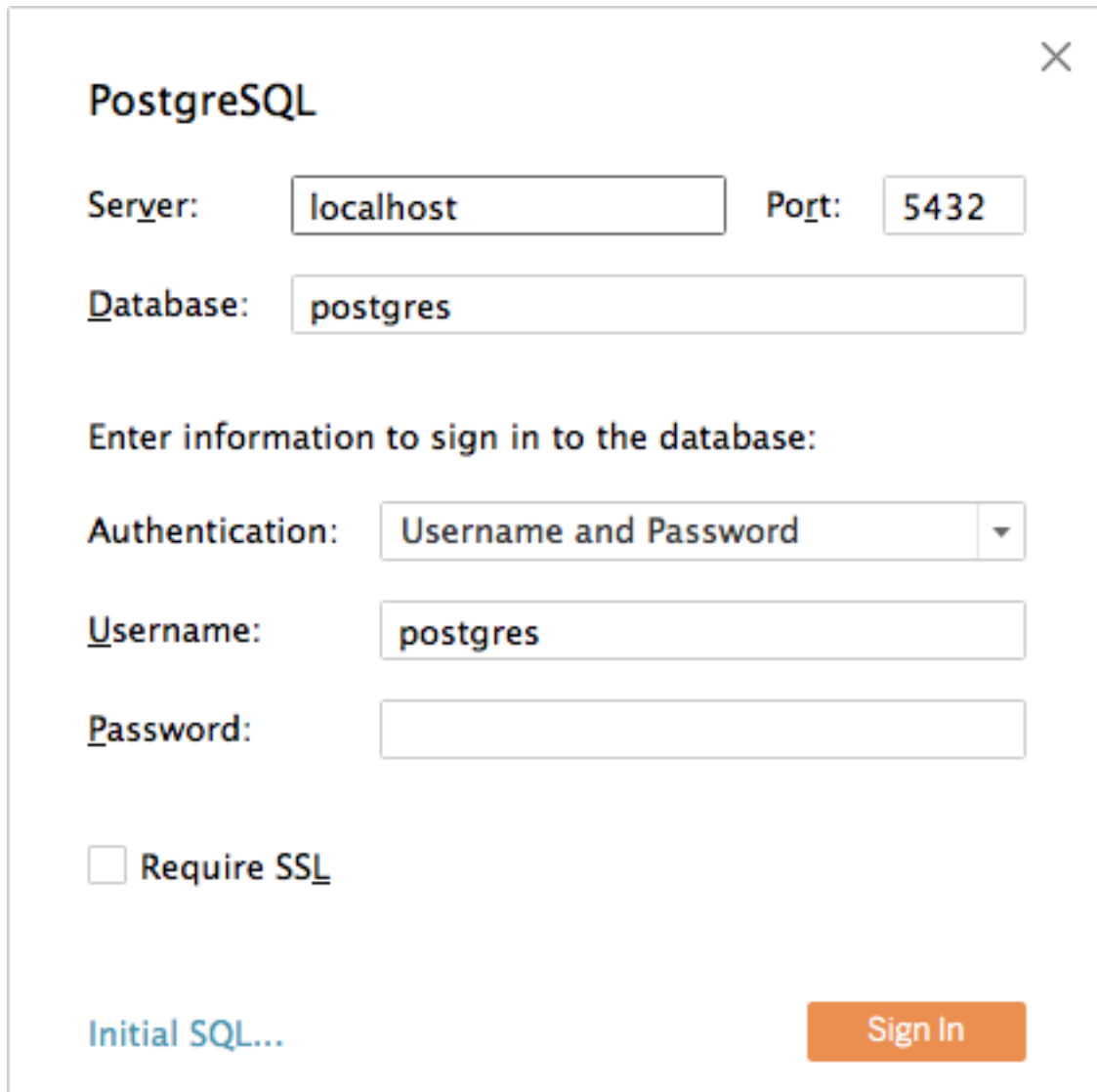
Tableau is a popular business intelligence and analytics tool for databases. Citus and Tableau provide a seamless experience for performing ad-hoc reporting or analysis.

You can now interact with Tableau using the following steps.

- Choose PostgreSQL from the “Add a Connection” menu.



- Enter the connection details for the coordinator node of your Citrus cluster. (Note if you're connecting to Citrus Cloud you must select "Require SSL.")

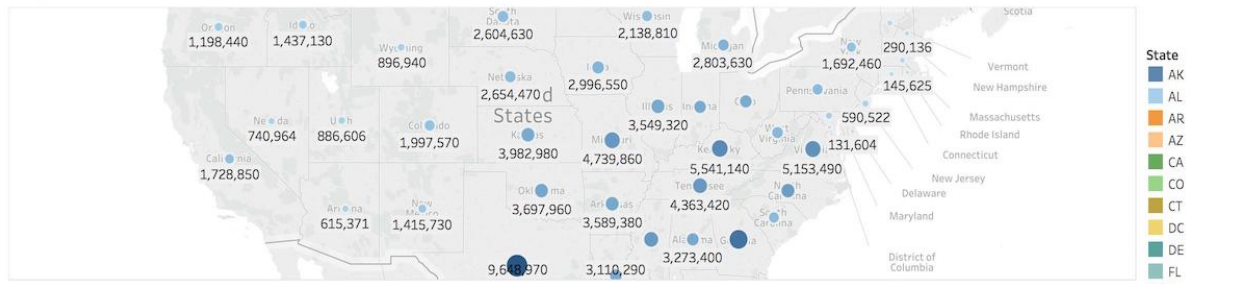


The screenshot shows a 'PostgreSQL' connection dialog box. It has a title bar with a close button (X). The fields are as follows:

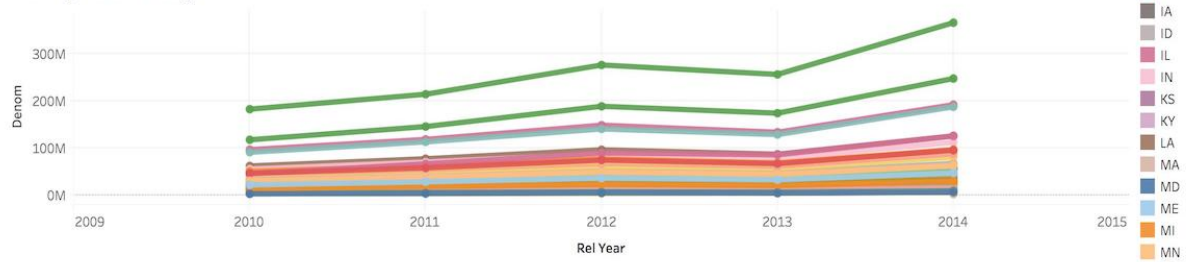
- Server:** localhost
- Port:** 5432
- Database:** postgres
- Enter information to sign in to the database:**
- Authentication:** Username and Password (dropdown menu)
- Username:** postgres
- Password:** (empty text box)
- ☐ **Require SSL**
- Initial SQL...** (link)
- Sign In** (button)

- Once you connect to Tableau, you will see the tables in your database. You can define your data source by dragging and dropping tables from the “Table” pane. Or, you can run a custom query through “New Custom SQL”.
- You can create your own sheets by dragging and dropping dimensions, measures, and filters. You can also create an interactive user interface with Tableau. To do this, Tableau automatically chooses a date range over the data. Citrus can compute aggregations over this range in human real-time.

Upper bound by state



Year by Year Change



Get Started

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.

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.

Total Nodes

2

RAM / vCPU per node

15 GB RAM 2 vCPUs	30.5 GB RAM 4 vCPUs	61 GB RAM 8 vCPUs	122 GB RAM 16 vCPUs	244 GB RAM 32 vCPUs
----------------------	------------------------	----------------------	------------------------	------------------------

Storage per node

512 GB	1 TB	1.5 TB	2 TB
--------	------	--------	------

Coordinator Size ?

7.5 GB RAM / 4 vCPUs / 512 GB STORAGE [Change coordinator size](#)

☐ High Availability

Need help sizing the Citus cluster that is right for you? We're here to help. [Contact Us](#).

\$890/mo

ESTIMATED PRICE & SUMMARY

TOTAL NODES

2

ACTIVE

DISTRIBUTED RAM / vCPU

30 GB RAM

4 vCPUs

ACTIVE

DISTRIBUTED STORAGE

1 TB

ACTIVE

COORDINATOR SIZE

7.5 GB RAM

4 vCPUs

512 GB

ACTIVE

Create New Formation

[Copy link to this custom configuration](#)

i We bill on a pro-rated, hourly basis—easily spin up a formation for a few hours of testing.

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 1 TB 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

Supported Regions

Citus Cloud runs on top of Amazon Web Services. During provisioning you're able to select your database region. We currently support:

- US East (N. Virginia) [us-east-1]
- US East (Ohio) [us-east-2]
- Asia Pacific (Tokyo) [ap-northeast-1]
- Asia Pacific (Seoul) [ap-northeast-2]
- Asia Pacific (Singapore) [ap-southeast-1]
- Asia Pacific (Sydney) [ap-southeast-2]
- Asia Pacific (Mumbai) [ap-south-1]
- EU (Frankfurt) [eu-central-1]
- EU (Ireland) [eu-west-1]
- EU (London) [eu-west-2]
- EU (Paris) [eu-west-3]
- South America (São Paulo) [sa-east-1]
- US West (N. California) [us-west-1]
- US West (Oregon) [us-west-2]
- Canada (Central) [ca-central-1]

If there is an AWS region you do not see listed but would like for us to add support for please [contact us](#) and we'd be happy to look into it.

Other infrastructure providers

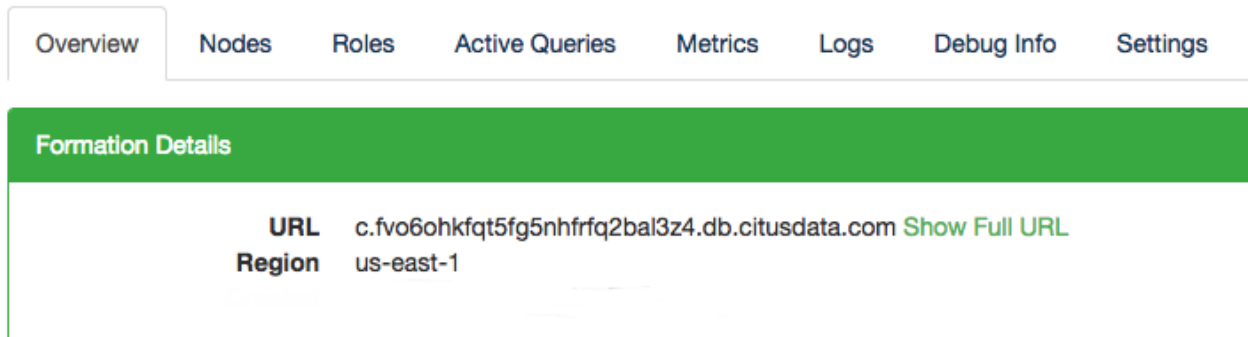
At this time we only support Citus Cloud on top of Amazon Web Services. We are continually exploring other infrastructure providers to make Citus Cloud available on. If you have immediate needs you could consider running Citus Community Edition or Citus Enterprise Edition. Or if you have questions about our timeline for other infrastructure providers please feel free to [reach out](#).

Connecting

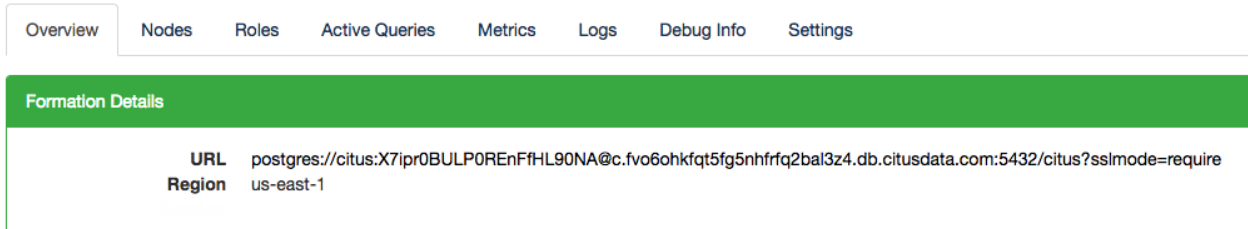
Applications connect to Citrus the same way they would PostgreSQL, using a [connection URI](#). This is a string which includes network and authentication information, and has the form:

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

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

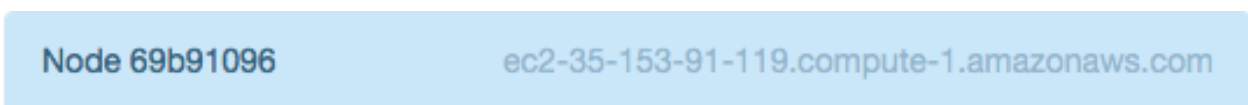


Notice how the end of the connection string contains `?sslmode=require`. Citrus Cloud accepts only SSL connections for security, so this url parameter is required. When connecting from an application framework such as Rails, Django, or Spring you may need to explicitly specify the `sslmode` as well.

Connecting Directly to a Worker

The previous section shows how to get a connection string for the coordinator node. To connect a database client such as `psql` to an individual worker node instead, we’ll need to create new a connection string by replacing the hostname in the coordinator connection string.

Under the “Nodes” tab in Cloud console each node lists its `amazonaws.com` hostname, like:



Replace the host in the coordinator connection string with this value, consulting the previous section for the connection string format. (As shown in the previous section, the hostname to be replaced will be immediately following an `@` sign.) Leave all other parameters unchanged. This will be the new connection string for the worker.

Manage


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:

Formation Configuration

Total Nodes
2


RAM / vCPU per node

15 GB RAM 2 vCPUs	30.5 GB RAM 4 vCPUs	61 GB RAM 8 vCPUs	122 GB RAM 16 vCPUs	244 GB RAM 32 vCPUs
------------------------------------	--------------------------------------	------------------------------------	--------------------------------------	--------------------------------------

The slider, **Total Nodes**, scales out the cluster by adding new nodes. The **RAM** buttons scale it up by changing the instance size (RAM and CPU cores) of existing nodes.

For example, just drag the slider for node count:

Formation Configuration

Total Nodes

4

RAM / vCPU per node

15 GB RAM 2 vCPUs	30.5 GB RAM 4 vCPUs	61 GB RAM 8 vCPUs	122 GB RAM 16 vCPUs	244 GB RAM 32 vCPUs
----------------------	------------------------	----------------------	------------------------	------------------------

After you adjust the slider and/or buttons and accept the changes, Citrus 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.

Maintenance Window

Maintenance Window

Any Time (default)

Save

Citrus 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:

Change In Progress: Your formation is currently scaling to 3 data nodes.

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

Change In Progress: Your data node type is scheduled to change to r4.xlarge.

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, Citrus Cloud does

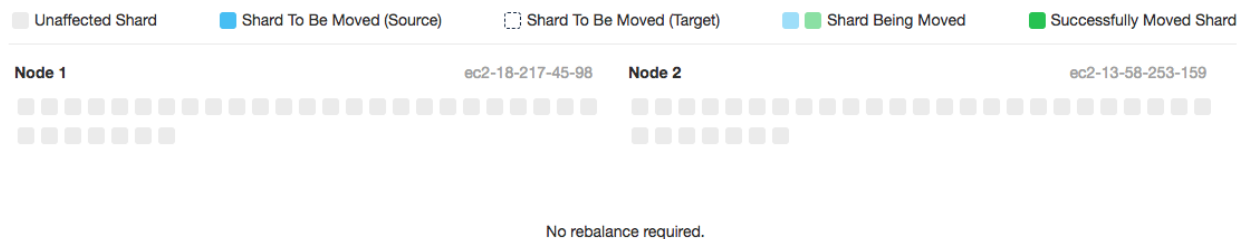
a quick switchover from the existing primary nodes to their followers which takes about two minutes. The creation and switchover process uses the same well-tested replication mechanism that powers Cloud’s *Backup, Availability, and Replication* feature. During the switchover period clients may experience errors and have to retry queries, especially cross-tenant queries hitting multiple nodes.

Scaling Out (adding new nodes)

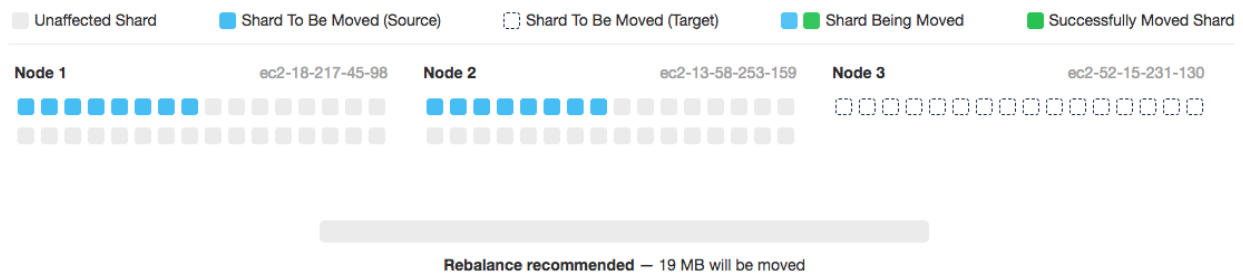
Node addition completes in five to ten minutes, which is faster than node resizing because the new nodes are created without data. To take advantage of the new nodes you still must adjust manually rebalance the shards, meaning move some shards from existing nodes to the new ones.

Rebalancing

You can go to the “Rebalancer” tab in the Cloud console to see the shard balance across nodes. Ordinarily this page will show, “No rebalance required.”



However if the shards could be placed more evenly, such as after a new node has been added to the cluster, the page will show a “Rebalance recommended.”



For maximum control, the choice of when to run the shard rebalancer is left to the database administrator. Citrus does not automatically rebalance on node creation. To start the shard rebalancer, connect to the cluster coordinator node with `psql` and run:

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

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.

Learn more about this function in *Rebalance Shards without Downtime*.

Citrus will output progress in both `psql` (saying which shards are moving) and graphically in the Cloud console:

The rebalance progress is also queryable in SQL with the `get_rebalance_progress()` function.

Scaling Connections (pgBouncer)

Each client connection to PostgreSQL consumes a noticeable amount of resources. To protect resource usage Citus Cloud enforces a hard limit of 300 concurrent connections to the coordinator node.

For further scaling we provide PgBouncer out of the box on Cloud. If your application requires more than 300 connections, change the port in the Cloud 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.

When connecting to PgBouncer you have:

- 1800 idle connections available
- 300 active connections to Citus available

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

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

Monitoring

Cloud Platform Status

Any events affecting the Citus Cloud platform as a whole are recorded on status.citusdata.com. This page shows a chart of recent uptime, as well as listing any past incidents.

Resources Usage

Citus Cloud metrics enable you to get information about your cluster's health and performance. The "Metrics" tab of the Cloud Console provides graphs for a number of measurements, all viewable per node. Below are the metrics, broken into categories, with details about the less obvious ones.

Amazon EBS Volume Metrics

(A full description of the metrics reported by Citus Cloud for underlying EBS resources can be found in the AWS documentation about [I/O Characteristics and Monitoring](#).)

IOPS are a unit of measure representing input/output operations per second. The operations are measured in KiB, and the underlying drive technology determines the maximum amount of data that a volume type counts as a single I/O.

- Read IOPS
- Write IOPS

Volume queue length is the number of pending I/O requests for a device. Latency is the true end-to-end client time of an I/O operation, in other words, the time elapsed between sending an I/O and receiving an acknowledgement that the I/O read or write is complete.

- Average Queue Length (Count)

- Average Read Latency (Seconds)
- Average Write Latency (Seconds)
- Bytes Read / Second
- Bytes Written / Second

CPU and Network

- CPU Utilization (Percent)
- Network - Bytes In / Second
- Network - Bytes Out / Second

PostgreSQL Write-Ahead Log

- WAL Bytes Written / Second

Formation Events Feed

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.



Formations

[Formation Events](#)
[+ New Formation](#)

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

Disk Almost Full

The “disk-almost-full” event happens if the disk utilization in the Postgres data directory reaches or exceeds 90%.

- Formation name
- Formation url
- Server
- “used_bytes”
- “total_bytes”
- “threshold_pct” (always 90)

The alert will appear once every twenty-four hours until the disk usage is resolved.

High CPU Utilization

The “cpu-utilization-high” event happens when there is CPU utilization reaches or exceeds 90%.

- Formation name
- Formation url
- Server
- “util_pct” (0-100% utilization value)
- “threshold_pct” (always 90)

The alert will appear once per hour until CPU utilization goes down to a normal level.

StatsD external reporting

Citus Cloud can send events to an external [StatsD](#) server for detailed monitoring. Citus Cloud sends the following statsd metrics:

Metric	Notes
<code>citus.disk.data.total</code>	
<code>citus.disk.data.used</code>	
<code>citus.load.1</code>	Load in past 1 minute
<code>citus.load.5</code>	Load in past 5 minutes
<code>citus.load.15</code>	Load in past 15 minutes
<code>citus.locks.granted.<mode>.<locktype>.count</code>	See below
<code>citus.mem.available</code>	
<code>citus.mem.buffered</code>	
<code>citus.mem.cached</code>	
<code>citus.mem.commit_limit</code>	Memory currently available to be allocated on the system
<code>citus.mem.committed_as</code>	Total amount of memory estimated to complete the workload
<code>citus.mem.dirty</code>	Amount of memory waiting to be written back to the disk
<code>citus.mem.free</code>	Amount of physical RAM left unused
<code>citus.mem.total</code>	Total amount of physical RAM
<code>citus.pgouncer_outbound.cl_active</code>	Active client connections
<code>citus.pgouncer_outbound.cl_waiting</code>	Waiting client connections
<code>citus.pgouncer_outbound.sv_active</code>	Active server connections
<code>citus.pgouncer_outbound.sv_idle</code>	Idle server connections
<code>citus.pgouncer_outbound.sv_used</code>	Server connections idle more than <code>server_check_delay</code>
<code>citus.postgres_connections.active</code>	
<code>citus.postgres_connections.idle</code>	
<code>citus.postgres_connections.unknown</code>	
<code>citus.postgres_connections.used</code>	

Notes:

- The `citus.mem.*` metrics are reported in kilobytes, and are also recorded in megabytes as `system.mem.*`. Memory metrics come from `/proc/meminfo`, and the [proc\(5\)](#) man page contains a description of each.
- The `citus.load.*` metrics are duplicated as `system.load.*`.
- `citus.locks.granted.*` and `citus.locks.not_granted.*` use `mode` and `locktype` as present in Postgres' [pg_locks](#) table.
- See the [pgBouncer docs](#) for more details about the `pgouncer_outbound` metrics.

To send these metrics to a statsd server, use the “Create New Metrics Destination” button in the “Metrics” tab of Cloud Console.

Overview
Nodes
Roles
Active Queries
Metrics
Logs
Rebalancer
Fork / PITR / Follower
Configuration
Debug Info
Settings

Metrics Destinations

Hostname	Port	Created At
<div>Create New Metrics Destination</div>		

Then fill in the host details in the resulting dialog box.

The statsd protocol is not encrypted, so we advise setting up *VPC peering* between the server and your Citrus Cloud cluster.

Example: Datadog with statsd

Datadog is a product which receives application metrics in the statsd protocol and makes them available in a web interface with sophisticated queries and reports. Here are the steps to connect it to Citrus Cloud.

1. Sign up for a Datadog account and take note of your personal API key. It is available at <https://app.datadoghq.com/account/settings#api>
2. Launch a Linux server, for instance on EC2.
3. In that server, install the Datadog Agent. This is a program which listens for statsd input and translates it into Datadog API requests. In the server command line, run:

```
# substitute your own API key
DD_API_KEY=1234567890 bash -c \
  "$ (curl -L https://raw.githubusercontent.com/DataDog/datadog-agent/master/cmd/
  ↪agent/install_script.sh) "
```

4. Configure the agent. (If needed, see Datadog [per-platform guides](#))


```
cat - | sudo tee -a /etc/datadog-agent/datadog.yaml << CONF
non_local_traffic: yes
use_dogstatsd: yes
dogstatsd_port: 8125
dogstatsd_non_local_traffic: yes
log_level: info
log_file: /var/log/datadog/agent.log
CONF

# this is how to do it on ubuntu
sudo systemctl restart datadog-agent
```

5. Fill in the agent server information as a new metrics destination in the Cloud Console. See the previous section for details.
6. The agent should now appear in the [Infrastructure](#) section in Datadog.

1 Host up / **1** total

Filter by name, tag, or app Group by tags: Showing the only entry.

Hostname	Status	CPU ↓	IOWait	Load 15	Apps
i-0cbfc38b44bcd5449	 UP	0.1%	0%	0	ntp system

Clicking the hostname link goes into a full dashboard of all the metrics, with the ability to write queries and set alerts.

VividCortex External Monitoring

Like the systems above, VividCortex provides a metrics dashboard. While the other systems mostly focus on computer resources, VividCortex focuses on the performance of queries. It tracks their throughput, error rate, 99th percentile latency, and concurrency.

To integrate VividCortex with Citrus Cloud we'll be using the [Off-Host Configuration](#). In this mode we create a database role with permissions to read the PostgreSQL statistics tables, and give the role's login information to the VividCortex agent. VividCortex then connects and periodically collects information.

Here's a step-by-step guide to get started.

1. Create a special VividCortex schema and relations on the Citrus coordinator node.

```
# Use their SQL script to create schema and
# helper functions to monitor the cluster

curl -L https://docs.vividcortex.com/create-stat-functions-v96.sql | \
psql [connection_uri]
```

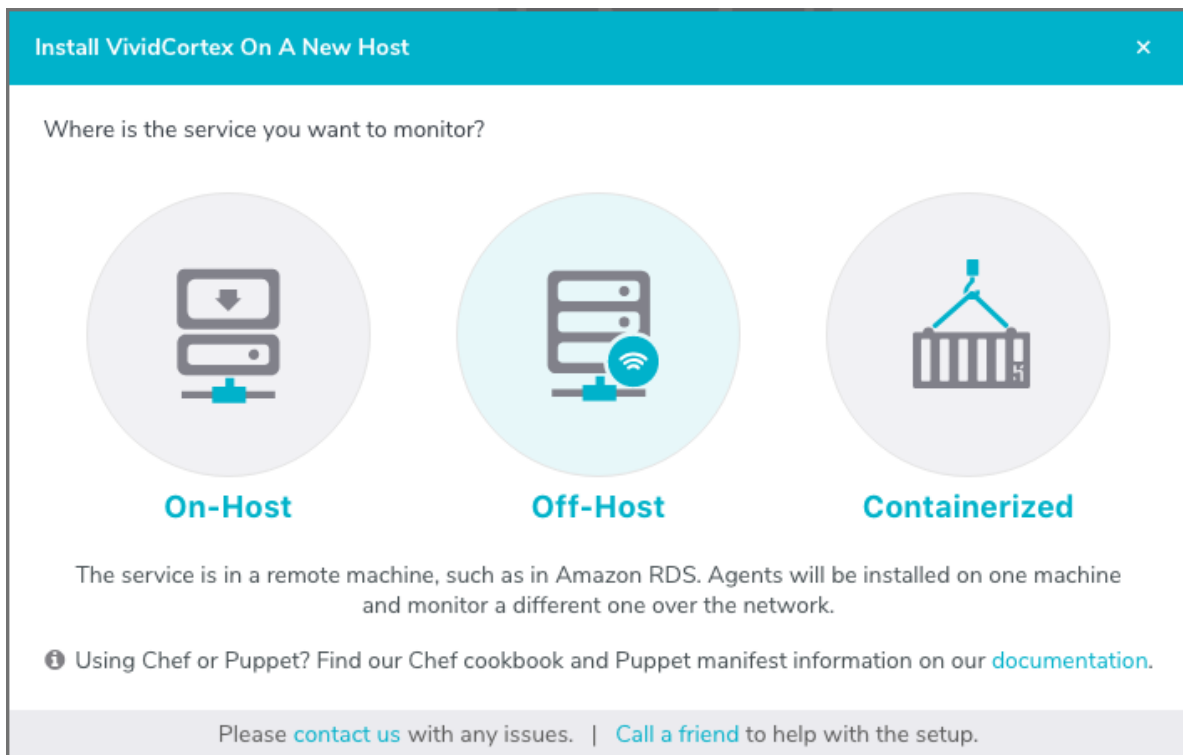
2. Create a VividCortex account.
2. On the **inventory** page, click "Setup your first host." This will open a wizard.



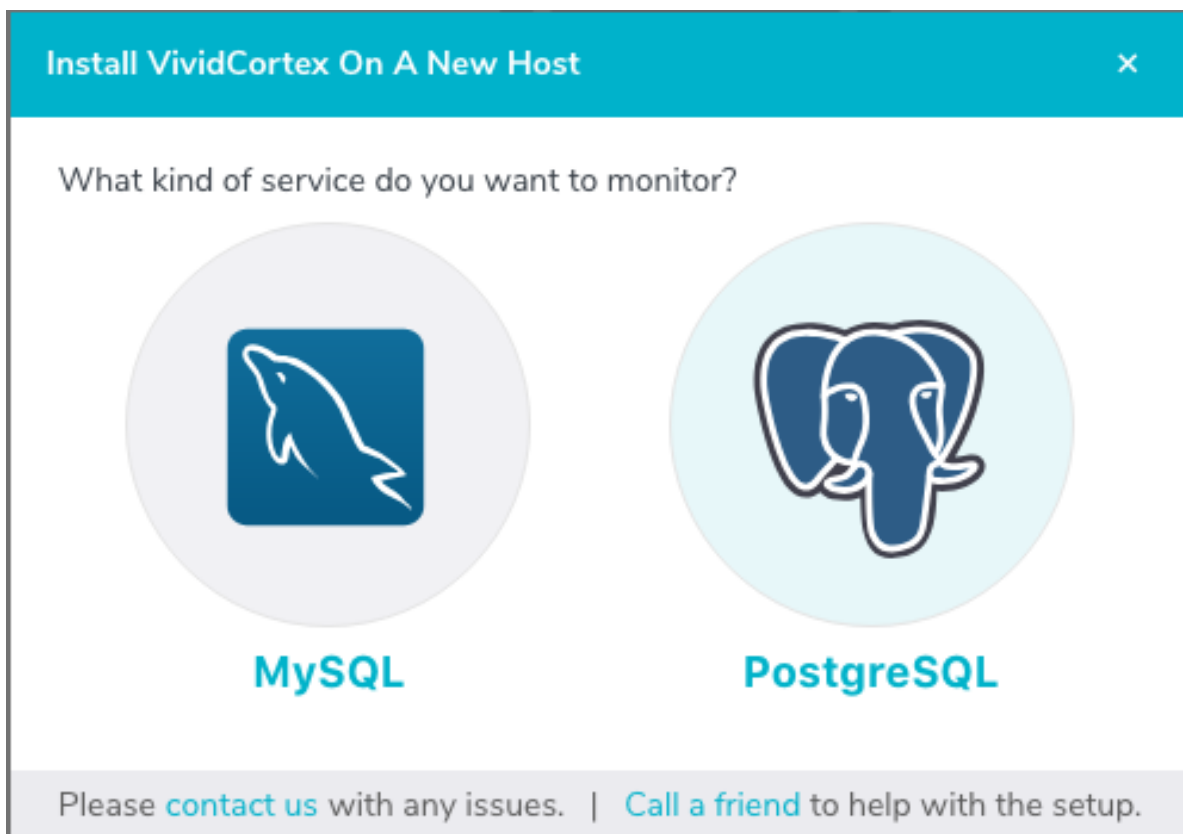
There are no active hosts. [Learn more.](#)

[Setup your first host](#)

3. Choose the off-host installation method.



4. Select the PostgreSQL database.



5. In Citrus Cloud, *create a new role* called `vividcortex`. Then grant it access to the VividCortex schema like

so:

```
# Grant our new role access to vividcortex schema

psql [connection_uri] -c \
    "GRANT USAGE ON SCHEMA vividcortex TO vividcortex;"
```

Finally note the generated password for the new account. Click “Show full URL” to see it.

Roles			
Name	Connection URL	Created At	Actions
citus	Show Full URL	Apr 17, 2018	Reset Password
vividcortex	Show Full URL	Apr 30, 2018	Reset Password Delete Role

- Input the connection information into the credentials screen in the VividCortex wizard. Make sure SSL Enabled is on, and that you’re using SSL Mode “Verify Full.” Specify `/etc/ssl/certs/citus.crt` for the SSL Authority.

Install VividCortex On A New Host

Set Credentials | Create User | Check configuration | Select the OS Host | Check Agents | Share with your team

Choose PostgreSQL Credentials
Enter the credentials the agent should use to connect to PostgreSQL.

Address:
Examples: localhost:5432, 10.0.10.1:5432, [::1]:5432

Database:

User:

Password:

SSL Enabled: ☒ ON

SSL Mode:
Always SSL (verify that the certification presented by the server was signed by a trusted CA)

SSL Authority:

SSL Certificate:

SSL Key:

Next step: [Create User >](#)

Please [contact us](#) with any issues. | [Call a friend](#) to help with the setup.

- Provision a server to act as the VividCortex agent. For instance a small EC2 instance will do. On this new host install the Citrus Cloud SSL certificate.

```
sudo curl -L https://console.citusdata.com/citus.crt \
    -o /etc/ssl/certs/citus.crt
```

- Advance to the next screen in the wizard. It will contain commands to run on the agent server, customized with a token for your account.

Install VividCortex On A New Host

Set Credentials Create User Check configuration **Select the OS Host** Check Agents Share with your team

Install The Agents On Your Server

Run the commands below as root on the server where you want the agent to run. This is the host you will monitor your database from, not the host of the database itself. The installation script will download the supervisor and install the VividCortex init script. If you need to configure outbound firewall rules or a proxy, see the [docs](#).

Options

Configure capture of query samples: [Learn more](#)

```
curl https://download.vividcortex.com/install > install
sh install --token s1lWtKhQXGo01b82jMTieJI2e9rV9Tm9 --autostart --start-now --proxy auto --off-host
```

Select Host

Select your new host below. It may take a moment to appear after you run the installer.

	ip-172-31-41-2 Server ip-172-31-41-2 version 4.4.0-1052-aws	a few seconds ago Apr 30, 2018 6:35:01 PM CDT
--	--	--

[Back](#) Next step: [Check Agents >](#)

Please [contact us](#) with any issues. | [Call a friend](#) to help with the setup.

After running the commands on your server, the server will appear under “Select host.” Click it and then continue.

After these steps, VividCortex should show all systems as activated. You can then proceed to the dashboard to monitor queries on your Citrus cluster.

Install VividCortex On A New Host

Set Credentials Create User Check configuration Select the OS Host **Check Agents** Share with your team

After completing the previous steps, wait for the agents to check in with our app and send the all-clear signal. This might take a few moments.

- ✓ OS Metrics
- ✓ PostgreSQL Metrics
- ✓ Supervisor
- ✓ Aggregator

[Back](#) [Next >](#)

Please [contact us](#) with any issues. | [Call a friend](#) to help with the setup.

Security

Connecting with SSL

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.

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.

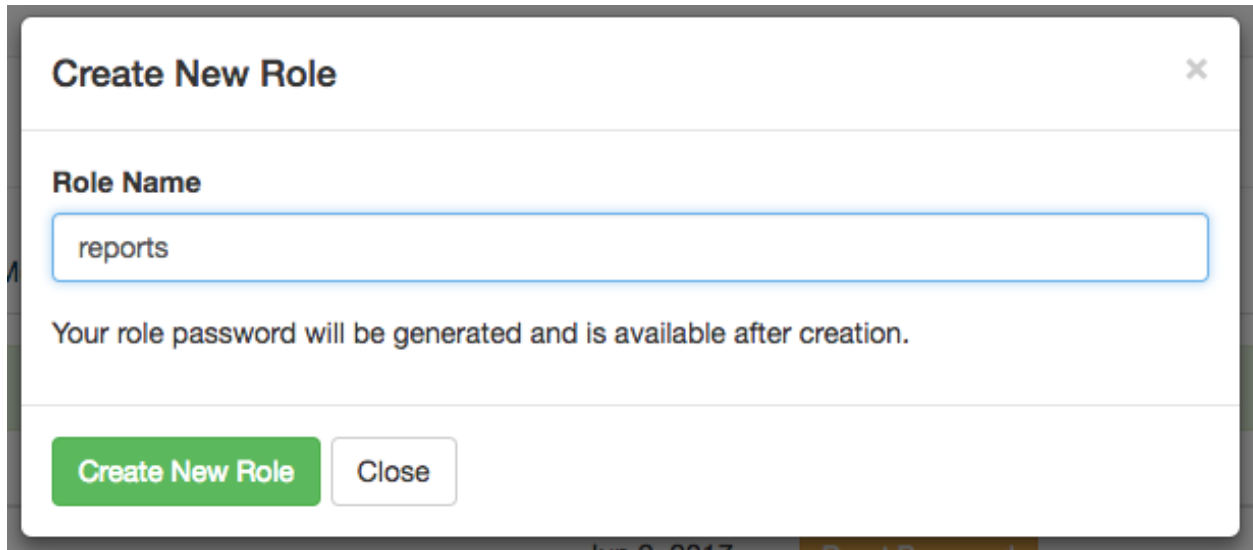
Users and Permissions

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."

Roles			
Name	Connection URL	Created At	Actions
citus	Show Full URL	Jun 9, 2017	Reset Password
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](#).

Citus Cloud also supports *Row-Level Security* for greater control in a multi-tenant environment.

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;'
);
```

Encryption at Rest

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.

Network Perimeter Controls

All Citus Cloud clusters run in their own isolated Amazon Virtual Private Cloud (VPC). In addition to the options discussed earlier, Citus Cloud provides the ability to further secure your cluster by limiting network access in two ways:

VPC Peering

VPC peering forms a network connection between two VPCs which routes traffic securely between them using private IPv4 addresses or IPv6 addresses. Instances in either VPC can communicate with each other as if they are within the same network. To set up a VPC peering connecting between an existing Citus Cloud formation and an Amazon VPC, open a support ticket and we will initiate the peering request.

IP Whitelisting

IP whitelisting restricts access to servers within a Citus Cloud cluster so that only designated IP addresses are able to connect to them, typically the addresses of application servers.

To enable IP whitelisting on your Citus Cloud formation, go to the VPCs tab in the Cloud Console:

[Support](#)[Formations](#)[VPCs](#)[Tunes](#)[Org](#) ▾[User](#) ▾

Then find the VPC containing your formation and click View Details:

VPC: vpc-363a185e

VPC ID	vpc-363a185e
Region	us-east-2
Formations	whitelist-demo

[View Details](#)

Finally, in the “IP Whitelist / Ingress Rules” tab, enter the allowable IP addresses or CIDR blocks, one per line.

0.0.0.0/0

[Save](#)

The special address 0.0.0.0/0 means unrestricted access. Note that permitted connections still require a valid username and password to actually access your database.

Backup, Availability, and Replication

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.

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.

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.

High-Availability (HA) Replication

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.

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.

Disaster Recovery (DR)

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.

Comparison of HA and DR

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.

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, leaving uncertainty on whether 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.

Trade-offs between latency and reliability for backups

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.

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.

Point-in-Time Recovery

While *Forking* means making a copy of the database at the current moment, point-in-time recovery (PITR) means making a copy from a specific moment in the past. PITR acts as a time machine, allowing simple restoration from even the most severe data loss.

Suppose a customer deletes a record stored in your Citus cluster and you want to recover it, or an engineer drops a table by accident. Just navigate to the "Fork / PITR" tab in the Citus Cloud console and select the "Recover to a point in time" option:

Fork This Formation / Point In Time Recovery (PITR)

Forking a formation creates a copy of all data as of right now. This process uses the continuous backups for your formation and has no impact on the source database.

Fork or Point In Time Recovery?

- ☐ Fork to now
☒ Recover to a point in time

Earliest Restore Time: Sunday, October 1, 2017 1:03 AM UTC

Current Time: Tuesday, October 10, 2017 8:08 PM UTC

Selected Restore Time: **Tuesday, October 10, 2017 8:07 PM UTC**

October 2017						
Su	Mo	Tu	We	Th	Fr	Sa
24	25	26	27	28	29	30
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31	1	2	3	4

8:07 PM

Name

Region

Which node sizes?

- ☒ Keep same node size
☐ Change node size

Price per Month: \$99

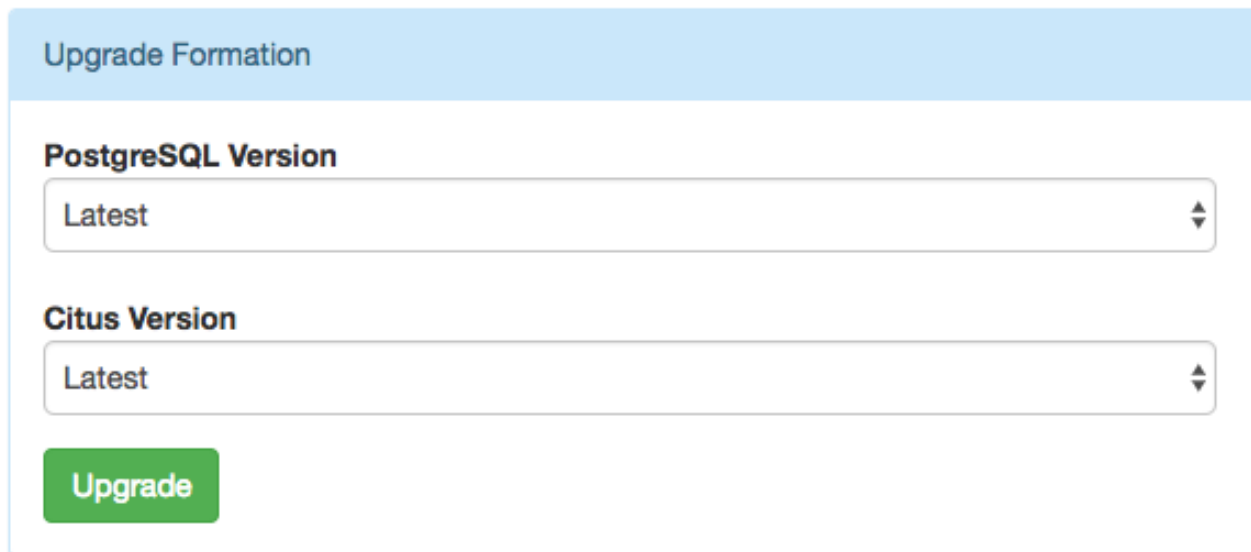
Similar to forking, PITR creates a copy of your formation and leaves the original unaffected. It uses the same operations internally as forking does: base backups and WAL shipping. This means that performing PITR causes no extra load on the original database.

Citus Cloud stores base backups and WAL records for up to ten days, which limits the recovery period to that amount of time. The user interface will prevent you from going back further, and will show the earliest possible recovery time.

Note that Citus worker nodes have different base backup timings and sizes, so it can happen that one node restores faster than another. In general the recovery process begins with restoring base backups at roughly 30MB/second. When that is complete the Cloud Console will provide a progress meter as it replays transactions from the WAL. The length of time for this stage varies by the number of transactions that must be replayed.

Upgrades

Regularly upgrading PostgreSQL and the Citus extension keeps you on top of all the latest features and bug fixes, and on Citus Cloud an upgrade is as easy as a few clicks of a button. Just visit the Settings tab in the cloud console, and scroll down to the upgrade section:



The screenshot shows a light blue header with the text "Upgrade Formation". Below this, there are two sections. The first section is titled "PostgreSQL Version" and contains a dropdown menu with "Latest" selected. The second section is titled "Citus Version" and also contains a dropdown menu with "Latest" selected. At the bottom of these sections is a green button with the text "Upgrade".

Note: We strongly recommend *forking* a formation and trying the upgrade on the fork before upgrading the production formation. It takes more time, but provides a dry run for the real upgrade.

You may want to set a maintenance window prior to starting an upgrade. Part of the upgrade process requires a few minutes of downtime, and setting a window will ensure that the downtime happens during a specific time of day. This option is available in the Settings tab of the cloud console.

Maintenance Window

Maintenance Window

Any Time (default)

Save

When you do start the upgrade, Citus Cloud creates new servers for the coordinator and worker nodes with the requested software versions, and replays the write-ahead log from the original nodes to transfer data. This can take a fair amount of time depending on the amount of data in existing nodes. During the transfer, the formation overview tab will contain a notice:

Change In Progress: Upgrade operation is continuing.

When the new nodes have caught up (and during a maintenance window if one has been set), Citus Cloud switches over to the new nodes. During the period of redirection there may be a few minutes of downtime. After everything is done, the upgrade section in Settings will show:

Upgrade Formation

Your formation is running the latest PostgreSQL and Citus version.

If *high-availability replication* is enabled, Citus Cloud will automatically create new standby servers. However *follower replicas* will not receive the updates and must be manually destroyed and recreated.

Logging

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

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.

[Overview](#)
[Nodes](#)
[Roles](#)
[Metrics](#)
[Logs](#)
[Debug Info](#)
[Settings](#)

Recent Logs

Primary ede35931
Primary 49e757f8
Node ffb04c7
Node 031c3caf
Node 632b8a17
Node 5d645591

```

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

```

Last Updated: 2016-09-02T23:59:16+00:00 [Refresh](#)

Log Destinations

Hostname	Port	TLS	Description	Created At
Create New Log Destination				

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:

Create New Log Destination

Hostname
logs4.papertrailapp.com

Port
19493

☒ TLS

Message Template

Optional, can be used to include tokens or change the structure of the log message.
[Details](#)

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

[Create New Log Destination](#) [Close](#)

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.

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

LogDNA

Hostname	syslog-a.logdna.com
Port	514
TLS	No
Protocol	BSD Syslog over TCP
Message Template	<\${PRI}>1 \${ISODATE} \${HOST} \${PROGRAM} \${PID} \${MSGID} [logdna@48950 key=\\ "<token>\\ "] \$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.

Additional Features

Extensions

To keep a standard Cloud installation for all customers and improve our ability to troubleshoot and provide support, we do not provide superuser access to Cloud clusters. Thus customers are not able to install PostgreSQL extensions themselves.

Generally there is no need to install extensions, however, because every Cloud cluster comes pre-loaded with many useful ones:

Name	Version	Schema	Description
btree_gin	1.2	public	support for indexing common datatypes in GIN
btree_gist	1.5	public	support for indexing common datatypes in GiST
citext	1.4	public	data type for case-insensitive character strings
citus	8.0-8	pg_catalog	Citus distributed database
cube	1.2	public	data type for multidimensional cubes
dblink	1.2	public	connect to other PostgreSQL databases from within a database
earthdistance	1.1	public	calculate great-circle distances on the surface of the Earth
fuzzystrmatch	1.1	public	determine similarities and distance between strings
hll	2.12	public	type for storing hyperloglog data
hstore	1.4	public	data type for storing sets of (key, value) pairs
intarray	1.2	public	functions, operators, and index support for 1-D arrays of integers
ltree	1.1	public	data type for hierarchical tree-like structures
pg_buffercache	1.3	public	examine the shared buffer cache
pg_cron	1.0	public	Job scheduler for PostgreSQL
pg_freespacemap	1.2	public	examine the free space map (FSM)
pg_partman	4.0.0	partman	Extension to manage partitioned tables by time or ID
pg_prewarm	1.1	public	prewarm relation data
pg_stat_statements	1.5	public	track execution statistics of all SQL statements executed
pg_trgm	1.3	public	text similarity measurement and index searching based on trigrams
pgcrypto	1.3	public	cryptographic functions
pgrowlocks	1.2	public	show row-level locking information
pgstattuple	1.5	public	show tuple-level statistics
plpgsql	1.0	pg_catalog	PL/pgSQL procedural language
session_analytics	1.0	public	
shard_rebalancer	8.0	public	
sslnfo	1.2	public	information about SSL certificates
tablefunc	1.0	public	functions that manipulate whole tables, including crosstab

Continued on next page

Table 20.1 – continued from previous page

Name	Version	Schema	Description
topn	2.1.0	public	type for top-n JSONB
unaccent	1.1	public	text search dictionary that removes accents
uuid-osp	1.1	public	generate universally unique identifiers (UUIDs)
xml2	1.1	public	XPath querying and XSLT

Forking

Forking a Citrus Cloud formation makes a copy of the cluster’s data at the current point 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.

How to Fork a Formation

Citrus 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.

Fork This Formation / Point In Time Recovery (PITR)

Forking a formation creates a copy of all data as of right now. This process uses the continuous backups for your formation and has no impact on the source database.

Fork or Point In Time Recovery?

- ☒ Fork to now
- ☐ Recover to a point in time

Name

Region

Which node sizes?

- ☒ Keep same node size
- ☐ Change node size

Price per Month: \$99

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*.

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 colleagues 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 Citrus 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.

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.

Followers

Citus Cloud allows you to create a read-only replica of a formation, called a “follower.” Any changes that happen to the original formation get promptly reflected in its follower, and queries against the follower cause no extra load on the original. The replica is a safe place for business analysts to run big report queries. In general followers are a useful tool to improve performance for read-only workloads.

Contrast followers with *Forking*. In a fork the copied formation does not receive post-copy changes from the original, and can diverge with its own updates. A follower, on the other hand, remains faithful to changes in the original.

To create a follower, head to the “Fork / PITR / Follower” tab in the Cloud console. Select the “Create follower formation” radio button, and fill in a name.

Forking a formation or creating a follower formation creates a copy of all data as of right now. This process uses the continuous backups for your formation and has no impact on the source database.

Fork, Point In Time Recovery or Create Follower Formation?

- ☐ Fork to now
- ☐ Recover to a point in time
- ☒ Create follower formation

Name

Region

VPC

Which node sizes?

- ☒ Keep same node size
- ☐ Change node size

Price per Month: \$99

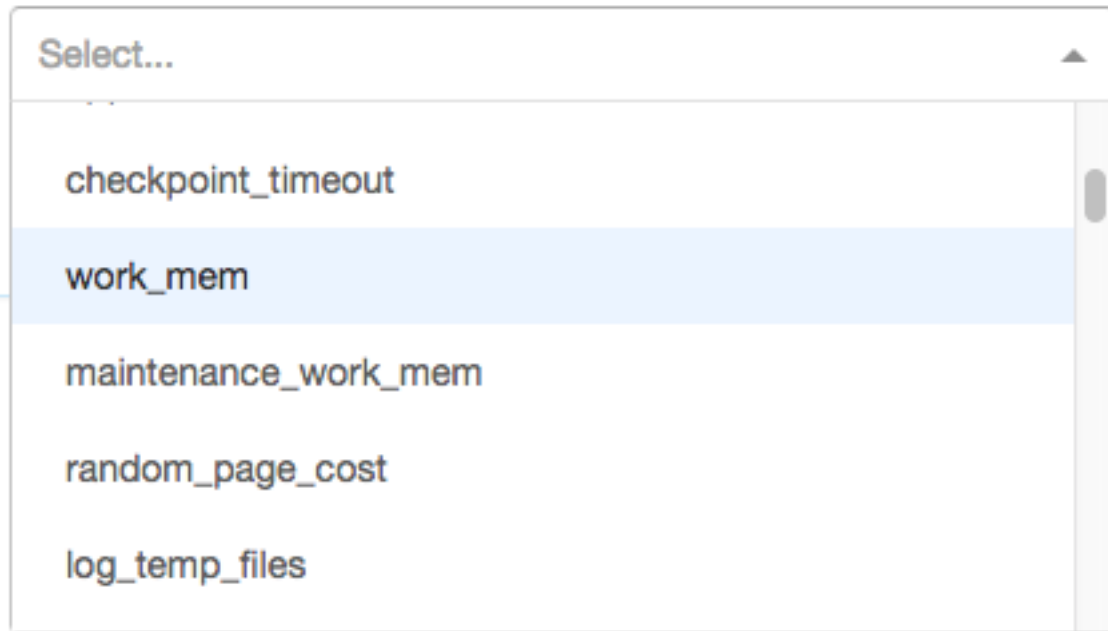
Click “Create Follower Formation” and wait. On completion the process will redirect you to a console for the new formation. The follower formation is distinct from the original and has its own database connection string.

Custom PostgreSQL Configuration

Citus Cloud supports changing of a number of PostgreSQL and Citus database server configuration parameters. Adjusting the parameters can help tune the server for particular workloads.

Within the configuration tab you can see any configuration parameters that have been customized as well as add new ones. Any configurations that require a database failover or restart must be configured by the Citus Cloud team – in order to customize one of those configurations please open a support ticket. Parameters that do not require a server restart are available directly in the customize tab. These settings will be immediately propagated to your cluster when set.

To access these settings, go to the Configuration tab in your Cloud formation and click “Change a Parameter.” It will present a dropdown list of config parameters:



The options are grouped by which server and system they control:

- Coordinator PostgreSQL
- Coordinator Inbound PgBouncer
- Worker PostgreSQL

Selecting an option opens an input box that accepts the appropriate values, whether numerical, textual, or a pre-set list. The selected option also shows a link to learn more about the configuration parameter.

Numerical parameters do not yet allow specifying units in this interface, and are interpreted as their default unit. The default unit appears in the description under the selected parameter. For instance in the picture above it says, “in kilobytes.” In this example one could specify a `work_mem` of 1GB using the value 1048576 (= 1024*1024).

Support and Billing

All Citus Cloud plans come with 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.

Support

Web based support is available on all Citus Cloud plans. You can open a [support inquiry](#) within the Citus Cloud console. Support response times for ticket classification of Citus Cloud are:

- Urgent (production database offline) - 1 hour response time
- High (production database impacted) - 4 hour response time
- Normal (general support) - 1 business day response time
- Low (general question) - 3 business days response time

Billing

Pricing

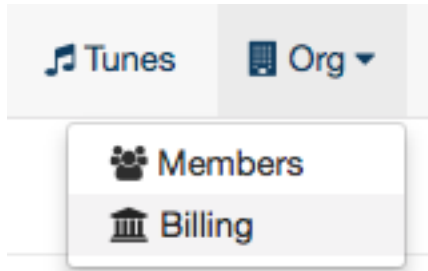
Citus Cloud bills on a per minute basis. We bill for a minimum of one hour of usage across all plans. Pricing varies based on the size and configuration of the cluster. A few factors that determine your price are:

- 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](#).

Payments

We accept payments through credit card or directly through bank ACH. To set up payment details, go to “Org” -> “Billing” in the main menu of the Cloud Console.



Then choose either “Credit Card” or “Bank Account” and fill out the form for the desired payment option:

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.

Choosing Cluster Size

This section explores configuration settings for running a cluster in production.

Shard Count

The number of nodes in a cluster is easy to change (see *Scaling the cluster*), but the number of shards to distribute among those nodes is more difficult to change after cluster creation. Choosing the shard count for each distributed table is a balance between the flexibility of having more shards, and the overhead for query planning and execution across them.

Multi-Tenant SaaS Use-Case

The optimal choice varies depending on your access patterns for the data. For instance, in the *Multi-Tenant Database* use-case we recommend choosing between **32 - 128 shards**. For smaller workloads say <100GB, you could start with 32 shards and for larger workloads you could choose 64 or 128. This means that you have the leeway to scale from 32 to 128 worker machines.

Real-Time Analytics Use-Case

In the *Real-Time Analytics* use-case, shard count should be related to the total number of cores on the workers. 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.

However keep in mind that for each query Citrus opens one database connection per shard, and these connections are limited. Be careful to keep the shard count small enough that distributed queries won't often have to wait for a connection. Put another way, the connections needed, `(max concurrent queries * shard count)`, should generally not exceed the total connections possible in the system, `(number of workers * max_connections per worker)`.

Initial Hardware Size

The size of a cluster, in terms of number of nodes and their hardware capacity, is easy to change. (*Scaling* on Citrus Cloud is especially easy.) However you still need to choose an initial size for a new cluster. Here are some tips for a reasonable initial cluster size.

Multi-Tenant SaaS Use-Case

For those migrating to Citrus from an existing single-node database instance, we recommend choosing a cluster where the number of worker cores and RAM in total equals that of the original instance. In such scenarios we have seen 2-3x performance improvements because sharding improves resource utilization, allowing smaller indices etc.

The coordinator node needs less memory than workers, so you can choose a compute-optimized machine for running the coordinator. The number of cores required depends on your existing workload (write/read throughput). By default in Citrus Cloud the workers use Amazon EC2 instance type R4S, and the coordinator uses C4S.

Real-Time Analytics Use-Case

Total cores: when working data fits in RAM, you can expect a linear performance improvement on Citrus proportional to the number of worker cores. To determine the right number of cores for your needs, consider the current latency for queries in your single-node database and the required latency in Citrus. Divide current latency by desired latency, and round the result.

Worker RAM: the best case would be providing enough memory that the majority of the working set fits in memory. The type of queries your application uses affect memory requirements. You can run `EXPLAIN ANALYZE` on a query to determine how much memory it requires.

Scaling the cluster

Citrus'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 Citrus cluster in order to improve query performance / scalability.

Add a worker

Citrus 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);
```

The new node is available for shards of new distributed tables. Existing shards will stay where they are unless redistributed, so adding a new worker may not help performance without further steps.

Rebalance Shards without Downtime

If you want to move existing shards to a newly added worker, Citrus Enterprise and Citrus Cloud provide a *rebalance_table_shards* function to make it easier. This function will move the shards of a given table to distribute them evenly among the workers.

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

Many products, like multi-tenant SaaS applications, cannot tolerate downtime, and Citrus rebalancing is able to honor this requirement on PostgreSQL 10 or above. This means reads and writes from the application can continue with minimal interruption while data is being moved.

How it Works

Citus' shard rebalancing uses PostgreSQL logical replication to move data from the old shard (called the “publisher” in replication terms) to the new (the “subscriber.”) Logical replication allows application reads and writes to continue uninterrupted while copying shard data. Citrus puts a brief write-lock on a shard only during the time it takes to update metadata to promote the subscriber shard as active.

As the PostgreSQL docs [explain](#), the source needs a *replica identity* configured:

A published table must have a “replica identity” configured in order to be able to replicate UPDATE and DELETE operations, so that appropriate rows to update or delete can be identified on the subscriber side. By default, this is the primary key, if there is one. Another unique index (with certain additional requirements) can also be set to be the replica identity.

In other words, if your distributed table has a primary key defined then it's ready for shard rebalancing with no extra work. However if it doesn't have a primary key or an explicitly defined replica identity, then attempting to rebalance it will cause an error. For instance:

```
-- creating the following table without REPLICA IDENTITY or PRIMARY KEY
CREATE TABLE test_table (key int not null, value text not null);
SELECT create_distributed_table('test_table', 'key');

-- running shard rebalancer with default behavior
SELECT rebalance_table_shards('test_table');

/*
NOTICE:  Moving shard 102040 from localhost:9701 to localhost:9700 ...
ERROR:  cannot use logical replication to transfer shards of the
        relation test_table since it doesn't have a REPLICA IDENTITY or
        PRIMARY KEY
DETAIL:  UPDATE and DELETE commands on the shard will error out during
        logical replication unless there is a REPLICA IDENTITY or PRIMARY KEY.
HINT:   If you wish to continue without a replica identity set the
        shard_transfer_mode to 'force_logical' or 'block_writes'.
*/
```

Here's how to fix this error.

First, does the table have a unique index?

If the table to be replicated already has a unique index which includes the distribution column, then choose that index as a replica identity:

```
-- supposing my_table has unique index my_table_idx
-- which includes distribution column
```

```
ALTER TABLE my_table REPLICA IDENTITY
USING INDEX my_table_idx;
```

Note: While `REPLICA IDENTITY USING INDEX` is fine, we recommend **against** adding `REPLICA IDENTITY FULL` to a table. This setting would result in each update/delete doing a full-table-scan on the subscriber side to find the tuple with those rows. In our testing we've found this to result in worse performance than even solution four below.

Otherwise, can you add a primary key?

Add a primary key to the table. If the desired key happens to be the distribution column, then it's quite easy, just add the constraint. Otherwise, a primary key with a non-distribution column must be composite and contain the distribution column too.

Unwilling to add primary key or unique index?

If the distributed table doesn't have a primary key or replica identity, and adding one is unclear or undesirable, you can still force the use of logical replication on PostgreSQL 10 or above. It's OK to do this on a table which receives only reads and inserts (no deletes or updates). Include the optional `shard_transfer_mode` argument of `rebalance_table_shards`:

```
SELECT rebalance_table_shards(
    'test_table',
    shard_transfer_mode => 'force_logical'
);
```

In this situation if an application does attempt an update or delete during replication, then the request will merely return an error. Deletes and writes will become possible again after replication is complete.

What about PostgreSQL 9.x?

On PostgreSQL 9.x and lower, logical replication is not supported. In this case we must fall back to a less efficient solution: locking a shard for writes as we copy it to its new location. Unlike logical replication, this approach introduces downtime for write statements (although read queries continue unaffected).

To choose this replication mode, use the `shard_transfer_mode` parameter again. Here is how to block writes and use the `COPY` command for replication:

```
SELECT rebalance_table_shards(
    'test_table',
    shard_transfer_mode => 'block_writes'
);
```

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](#).

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.

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 coordinator 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.

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.

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_placement AS placement,
     pg_dist_node AS node
WHERE placement.groupid = node.groupid
      AND node.noderole = 'primary'
      AND 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.

Viewing Query Statistics

Note: The `citus_stat_statements` view is a feature of **Citus Enterprise Edition** and *Citus Cloud* only.

When administering a Citus cluster it's useful to know what queries users are running, which nodes are involved, and which execution method Citus is using for each query. Citus records query statistics in a metadata view called *citus_stat_statements*, named analogously to Postgres' `pg_stat_statements`. Whereas `pg_stat_statements` stores info about query duration and I/O, `citus_stat_statements` stores info about Citus execution methods and shard partition keys (when applicable).

Citus requires the `pg_stat_statements` extension to be installed in order to track query statistics. On Citus Cloud this extension will be pre-activated, but on a self-hosted Postgres instance you must load the extension in `postgresql.conf` via `shared_preload_libraries`, then create the extension in SQL:

```
CREATE EXTENSION pg_stat_statements;
```

Let's see how this works. Assume we have a table called `foo` that is hash-distributed by its `id` column.

```
-- create and populate distributed table
create table foo ( id int );
select create_distributed_table('foo', 'id');

insert into foo select generate_series(1,100);
```

We'll run two more queries, and `citus_stat_statements` will show how Citus chooses to execute them.

```
-- counting all rows executes on all nodes, and sums
-- the results on the coordinator
SELECT count(*) FROM foo;

-- specifying a row by the distribution column routes
-- execution to an individual node
SELECT * FROM foo WHERE id = 42;
```

To find how these queries were executed, ask the stats table:

```
SELECT * FROM citus_stat_statements;
```

Results:

```
-----
| queryid | userid | dbid | query |
-- executor | partition_key | calls |
```

1496051219	16384	16385	select count(*) from foo;	real-
↪time	NULL		1	
2530480378	16384	16385	select * from foo where id = \$1	_
↪router	42		1	
3233520930	16384	16385	insert into foo select generate_series(\$1,\$2)	_
↪insert-select	NULL		1	

We can see that Citus used the real-time executor to run the count query. This executor fragments the query into constituent queries to run on each node and combines the results on the coordinator node. In the case of the second query (filtering by the distribution column `id = $1`), Citus determined that it needed the data from just one node. Thus Citus chose the router executor to send the whole query to that node for execution. Lastly, we can see that the `insert into foo select...` statement ran with the insert-select executor which provides flexibility to run these kind of queries. The insert-select executor must sometimes pull results to the coordinator and push them back down to workers.

So far the information in this view doesn't give us anything we couldn't already learn by running the `EXPLAIN` command for a given query. However in addition to getting information about individual queries, the `citus_stat_statements` view allows us to answer questions such as “what percentage of queries in the cluster are run by which executors?”

```
SELECT executor, sum(calls)
FROM citus_stat_statements
GROUP BY executor;
```

executor	sum	

insert-select	1	
real-time	1	
router	1	

In a multi-tenant database, for instance, we would expect to see “router” for the vast majority of queries, because the queries should route to individual tenants. Seeing too many real-time queries may indicate that queries do not have the proper filters to match a tenant, and are using unnecessary resources.

We can also find which partition_ids are the most frequent targets of router execution. In a multi-tenant application these would be the busiest tenants.

```
SELECT partition_key, sum(calls) as total_queries
FROM citus_stat_statements
WHERE coalesce(partition_key, '') <> ''
GROUP BY partition_key
ORDER BY total_queries desc
LIMIT 10;
```

```
-----
| partition_key | total_queries |
-----
| 42            |              1 |
-----
```

Finally, by joining `citus_stat_statements` with `pg_stat_statements` we can gain a better view into not only how many queries use different executor types, but how much time each executor spends running queries.

```
SELECT executor, sum(css.calls), sum(pss.total_time)
FROM citus_stat_statements css
JOIN pg_stat_statements pss USING (queryid)
GROUP BY executor;
```

```
-----
| executor      | sum |      sum      |
-----
| insert-select |    1 | 18.393312     |
| real-time     |    1 | 3.537155      |
| router        |    1 | 0.392590      |
-----
```

Resource Conservation

Limiting Long-Running Queries

Long running queries can hold locks, queue up WAL, or just consume a lot of system resources, so in a production environment it's good to prevent them from running too long. You can set the `statement_timeout` parameter on the coordinator and workers to cancel queries that run too long.

```
-- limit queries to five minutes
ALTER DATABASE citus
SET statement_timeout TO 300000;
SELECT run_command_on_workers($cmd$
ALTER DATABASE citus
SET statement_timeout TO 300000;
$cmd$);
```

The timeout is specified in milliseconds.

Note: It's not possible to make an exception to the timeout for individual queries. Citrus does not yet propagate SET LOCAL to the workers, so the `statement_timeout` GUC cannot be adjusted within a session.

Security

Connection Management

When Citrus nodes communicate with one another they consult a GUC for connection parameters and, in the Enterprise Edition of Citrus, a table with connection credentials. This gives the database administrator flexibility to adjust parameters for security and efficiency.

To set non-sensitive libpq connection parameters to be used for all node connections, update the `citus.node_conninfo` GUC:

```
-- key=value pairs separated by spaces.
-- For example, ssl options:

ALTER DATABASE foo
```

```
SET citus.node_conninfo =  
    'sslrootcert=/path/to/citus.crt sslmode=verify-full';
```

There is a whitelist of parameters that the GUC accepts. See the [node_conninfo](#) reference for details.

Citus Enterprise Edition includes an extra table used to set sensitive connection credentials. This is fully configurable per host/user. It's easier than managing `.pgpass` files through the cluster and additionally supports certificate authentication.

```
-- only superusers can access this table  
  
INSERT INTO pg_dist_authinfo  
    (nodeid, rolename, authinfo)  
VALUES  
    (123, 'jdoe', 'password=abc123');
```

After this INSERT, any query needing to connect to node 123 as the user jdoe will use the supplied password. The documentation for [pg_dist_authinfo](#) has more info.

Increasing 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. In Citus Enterprise the `pg_dist_authinfo` table can provide that information, as discussed earlier. However in Citus Community Edition the authentication information has to be maintained in a `.pgpass` file. Edit `.pgpass` in 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.

Row-Level Security

Note: Row-level security support is a part of Citus Enterprise. Please [contact us](#) to obtain this functionality.

PostgreSQL [row-level security](#) policies restrict, on a per-user basis, which rows can be returned by normal queries or inserted, updated, or deleted by data modification commands. This can be especially useful in a multi-tenant

Citus cluster because it allows individual tenants to have full SQL access to the database while hiding each tenant's information from other tenants.

We can implement the separation of tenant data by using a naming convention for database roles that ties into table row-level security policies. We'll assign each tenant a database role in a numbered sequence: `tenant_1`, `tenant_2`, etc. Tenants will connect to Citus using these separate roles. Row-level security policies can compare the role name to values in the `tenant_id` distribution column to decide whether to allow access.

Here is how to apply the approach on a simplified events table distributed by `tenant_id`. First create the roles `tenant_1` and `tenant_2` (it's easy on Citus Cloud, see [Users and Permissions](#)). Then run the following as an administrator:

```
CREATE TABLE events(
  tenant_id int,
  id int,
  type text
);

SELECT create_distributed_table('events','tenant_id');

INSERT INTO events VALUES (1,1,'foo'), (2,2,'bar');

-- assumes that roles tenant_1 and tenant_2 exist
GRANT select, update, insert, delete
  ON events TO tenant_1, tenant_2;
```

As it stands, anyone with select permissions for this table can see both rows. Users from either tenant can see and update the row of the other tenant. We can solve this with row-level table security policies.

Each policy consists of two clauses: `USING` and `WITH CHECK`. When a user tries to read or write rows, the database evaluates each row against these clauses. Existing table rows are checked against the expression specified in `USING`, while new rows that would be created via `INSERT` or `UPDATE` are checked against the expression specified in `WITH CHECK`.

```
-- first a policy for the system admin "citus" user
CREATE POLICY admin_all ON events
  TO citus          -- apply to this role
  USING (true)      -- read any existing row
  WITH CHECK (true); -- insert or update any row

-- next a policy which allows role "tenant_<n>" to
-- access rows where tenant_id = <n>
CREATE POLICY user_mod ON events
  USING (current_user = 'tenant_' || tenant_id::text);
  -- lack of CHECK means same condition as USING

-- enforce the policies
ALTER TABLE events ENABLE ROW LEVEL SECURITY;
```

Now roles `tenant_1` and `tenant_2` get different results for their queries:

Connected as tenant_1:

```
SELECT * FROM events;
```

```
-----
| tenant_id | id | type |
-----
```

```
|          1 | 1 | foo |
-----
```

Connected as tenant_2:

```
SELECT * FROM events;
```

```
-----
| tenant_id | id | type |
-----
|          2 | 2 | bar  |
-----
```

```
INSERT INTO events VALUES (3,3,'surprise');
/*
ERROR:  42501: new row violates row-level security policy for table "events_102055"
*/
```

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](#) 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.

In addition to our core Citus extension, we also maintain several others:

- [cstore_fdw](#) - Columnar store for analytics. The columnar nature delivers performance by reading only relevant data from disk, and it may compress data 6x-10x to reduce space requirements for data archival.
- [pg_cron](#) - Run periodic jobs directly from the database.
- [postgresql-topn](#) - Returns the top values in a database according to some criteria. Uses an approximation algorithm to provide fast results with modest compute and memory resources.
- [postgresql-hll](#) - HyperLogLog data structure as a native data type. It's a fixed-size, set-like structure used for distinct value counting with tunable precision.

Creating a New Database

Each PostgreSQL server can hold [multiple databases](#). However new databases do not inherit the extensions of any others; all desired extensions must be added afresh. To run Citus on a new database, you'll need to create the database on

the coordinator and workers, create the Citrus extension within that database, and register the workers in the coordinator database.

On an existing database on the coordinator run:

```
-- create the new db on coordinator and workers
CREATE DATABASE newbie;
SELECT run_command_on_workers('CREATE DATABASE newbie;');

-- review the worker nodes registered in current db
SELECT * FROM master_get_active_worker_nodes();

-- switch to new db on coordinator
\c newbie

-- create citus extension in new db
CREATE EXTENSION citus;

-- register workers in new db
SELECT * from master_add_node('node-name', 5432);
SELECT * from master_add_node('node-name2', 5432);
-- ... for each of them
```

In the new db on every worker, manually run:

```
CREATE EXTENSION citus;
```

Now the new database will be operating as another Citrus cluster.

Checks For Updates and Cluster Statistics

Unless you opt out, Citrus checks if there is a newer version of itself during installation and every twenty-four hours thereafter. If a new version is available, Citrus emits a notice to the database logs:

```
a new minor release of Citrus (X.Y.Z) is available
```

During the check for updates, Citrus also sends general anonymized information about the running cluster to Citrus Data company servers. This helps us understand how Citrus is commonly used and thereby improve the product. As explained below, the reporting is opt-out and does **not** contain personally identifying information about schemas, tables, queries, or data.

What we Collect

1. Citrus checks if there is a newer version of itself, and if so emits a notice to the database logs.
2. Citrus collects and sends these statistics about your cluster:
 - Randomly generated cluster identifier
 - Number of workers
 - OS version and hardware type (output of `uname -psr` command)
 - Number of tables, rounded to a power of two
 - Total size of shards, rounded to a power of two
 - Whether Citrus is running in Docker or natively

Because Citus is an open-source PostgreSQL extension, the statistics reporting code is available for you to audit. See [statistics_collection.c](#).

How to Opt Out

If you wish to disable our anonymized cluster statistics gathering, set the following GUC in `postgresql.conf` on your coordinator node:

```
citus.enable_statistics_collection = off
```

This disables all reporting and in fact all communication with Citus Data servers, including checks for whether a newer version of Citus is available.

If you have super-user SQL access you can also achieve this without needing to find and edit the configuration file. Just execute the following statement in `psql`:

```
ALTER SYSTEM SET citus.enable_statistics_collection = 'off';
```

Since Docker users won't have the chance to edit this PostgreSQL variable before running the image, we added a Docker flag to disable reports.

```
# Docker flag prevents reports  
  
docker run -e DISABLE_STATS_COLLECTION=true citusdata/citus:latest
```

Table Management

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
<code>citus_relation_size(relation_name)</code>	<ul style="list-style-type: none"> • Size of actual data in table (the “main fork”). • A relation can be the name of a table or an index.
<code>citus_table_size(relation_name)</code>	<ul style="list-style-type: none"> • <code>citus_relation_size</code> plus: <ul style="list-style-type: none"> – size of free space map – size of visibility map
<code>citus_total_relation_size(relation_name)</code>	<ul style="list-style-type: none"> • <code>citus_table_size</code> plus: <ul style="list-style-type: none"> – 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 |
-----
```

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 Citrus 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.

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;
```

Citrus propagates the ANALYZE command to all worker node placements.

Upgrading Citus

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.

Patch Version Upgrade

To upgrade a Citus version to its latest patch, issue a standard upgrade command for your package manager. Assuming version 8.0 is currently installed on Postgres 11:

Ubuntu or Debian

```
sudo apt-get update
sudo apt-get install --only-upgrade postgresql-11-citus-8.0
sudo service postgresql restart
```

Fedora, CentOS, or Red Hat

```
sudo yum update citus80_11
sudo service postgresql-11.0 restart
```

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 7.5 to 8.0 for instance:

Step 1. Update Citus Package

Ubuntu or Debian

```
sudo apt-get update
sudo apt-get install postgresql-10-citus-8.0
sudo service postgresql restart
```

Fedora, CentOS, or Red Hat

```
# Fedora, CentOS, or Red Hat
sudo yum swap citus75_10 citus80_10
sudo service postgresql-10 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.

Upgrading PostgreSQL version from 10 to 11

Note: Do not attempt to upgrade *both* Citus and Postgres versions at once. If both upgrades are desired, upgrade Citus first.

Also **Citus 7.x is not compatible with Postgres 11**. Before upgrading Postgres 10 to 11, be sure to follow the above steps to upgrade from Citus 7.x to 8.0.

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

Existing data directory (e.g. /opt/pgsql/10/data) `export OLD_PG_DATA=/opt/pgsql/10/data`

Existing PostgreSQL installation path (e.g. /usr/pgsql-10) `export OLD_PG_PATH=/usr/pgsql-10`

New data directory after upgrade `export NEW_PG_DATA=/opt/pgsql/11/data`

New PostgreSQL installation path `export NEW_PG_PATH=/usr/pgsql-11`

On Every Node (Coordinator and workers)

1. 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_placement AS
SELECT * FROM pg_catalog.pg_dist_placement;
CREATE TABLE      public.pg_dist_node_metadata AS
SELECT * FROM pg_catalog.pg_dist_node_metadata;
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;

```

2. Configure the new database instance to use Citrus.

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

```
shared_preload_libraries = 'citrus'
```

- **DO NOT CREATE** Citrus extension yet
- **DO NOT** start the new server

3. Stop the old server.

4. 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 Citrus extension **IS NOT** created

5. 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

```

6. Start the new server.

- **DO NOT** run any query before running the queries given in the next step

7. 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_placement
SELECT * FROM public.pg_dist_placement;
INSERT INTO pg_catalog.pg_dist_node_metadata
SELECT * FROM public.pg_dist_node_metadata;
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;

```

8. Drop temporary metadata tables.

```

DROP TABLE public.pg_dist_partition;
DROP TABLE public.pg_dist_shard;
DROP TABLE public.pg_dist_placement;
DROP TABLE public.pg_dist_node_metadata;
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;

```

9. 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_placement_placementid_seq', (SELECT
↪MAX(placementid)+1 AS max_placement_id FROM pg_dist_placement), 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_colocationid_seq', (SELECT
↪MAX(colocationid)+1 AS max_colocation_id FROM pg_dist_colocation), false);

```

10. 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);
```

11. 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 = 'citrus') as refobjid,
    0 as refobjsubid ,
    'n' as deptype
FROM
    pg_dist_partition p;
```

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.

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.

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 (Citrus 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 Citrus real-time executor to execute this query:

```
-> Custom Scan (Citrus 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 a lot of disk activity on your worker node in spite 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 `citrus.explain_all_tasks` is enabled, EXPLAIN plans are retrieved sequentially, which may take a long time for EXPLAIN ANALYZE.

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 Citrus 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 Citrus. 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. Citrus 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 Citrus 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.

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.

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 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](#).

Subquery/CTE Network Overhead

In the best case Citrus can execute queries containing subqueries and CTEs in a single step. This is usually because both the main query and subquery filter by tables' distribution column in the same way, and can be pushed down to worker nodes together. However Citrus is sometimes forced to execute subqueries *before* executing the main query, copying the intermediate subquery results to other worker nodes for use by the main query. This technique is called *Subquery/CTE Push-Pull Execution*.

It's important to be aware when subqueries are executed in a separate step, and avoid sending too much data between worker nodes. The network overhead will hurt performance. The EXPLAIN command allows you to discover how queries will be executed, including whether multiple steps are required. For a detailed example see [Subquery/CTE Push-Pull Execution](#).

Also you can defensively set a safeguard against large intermediate results. Adjust the `max_intermediate_result_size` limit in a new connection to the coordinator node. By default the max intermediate result size is 1GB, which is large enough to allow some inefficient queries. Try turning it down and running your queries:

```
-- set a restrictive limit for intermediate results
SET citus.max_intermediate_result_size = '512kB';

-- attempt to run queries
-- SELECT ...
```

If the query has subqueries or CTEs that exceed this limit, the query will be canceled and you will see an error message:

```
ERROR:  the intermediate result size exceeds citus.max_intermediate_result_size_
↳ (currently 512 kB)
DETAIL:  Citus restricts the size of intermediate results of complex subqueries and_
↳ CTEs to avoid accidentally pulling large result sets into once place.
HINT:  To run the current query, set citus.max_intermediate_result_size to a higher_
↳ value or -1 to disable.
```

If this happens, consider whether you can move limits or filters inside CTEs/subqueries. For instance

```
-- It's slow to retrieve all rows and limit afterward

WITH cte_slow AS (SELECT * FROM users_table)
SELECT * FROM cte_slow LIMIT 10;

-- Limiting inside makes the intermediate results small

WITH cte_fast AS (SELECT * FROM users_table LIMIT 10)
SELECT * FROM cte_fast;
```

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 Citrus 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. Citrus 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 `citrus.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 `citrus.max_running_tasks_per_node`. This configuration value sets the maximum number of tasks to execute concurrently on one worker 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 `citrus.max_running_tasks_per_node` without much concern.

With this, we conclude our discussion about performance tuning in Citrus. To learn more about the specific configuration parameters discussed in this section, please visit the [Configuration Reference](#) section of our documentation.

Scaling Out Data Ingestion

Citrus 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.

Real-time Insert and Updates

On the Citrus 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), Citrus first finds the right shard placements based on the value in the distribution column. Citrus 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 Citrus coordinator node however can process concurrent INSERTs to reach high throughputs.

Insert Throughput

To measure data ingest rates with Citrus, 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 `INSERT`s 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 `INSERT`s 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.

Update Throughput

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 `INSERT`s. 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.

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 `INSERT`s per second. This translates to approximately 1 to 4 billion `INSERT`s 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.

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 Citrus 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 response 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_TIMESTAMP); -- Time: 3.132 ms
```

Staging Data Temporarily

When loading data for temporary staging, consider using an **unlogged table**. These are tables which are not backed by the Postgres write-ahead log. This makes them faster for inserting rows, but not suitable for long term data storage. You can use an unlogged table as a place to load incoming data, prior to manipulating the data and moving it to permanent tables.

```
-- example unlogged table
CREATE UNLOGGED TABLE unlogged_table (
    key text,
    value text
);

-- its shards will be unlogged as well when
-- the table is distributed
SELECT create_distributed_table('unlogged_table', 'key');

-- ready to load data
```

Bulk Copy (250K - 2M/s)

Distributed tables support **COPY** from the Citrus 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: Make sure your benchmarking setup is well configured so you can observe optimal COPY performance. Follow these tips:

- We recommend a large batch size (~ 50000-100000). You can benchmark with multiple files (1, 10, 1000, 10000 etc), each of that batch size.
 - Use parallel ingestion. Increase the number of threads/ingestors to 2, 4, 8, 16 and run benchmarks.
 - Use a compute-optimized coordinator. For the workers choose memory-optimized boxes with a decent number of vcpus.
 - Go with a relatively small shard count, 32 should suffice but you could benchmark with 64, too.
 - Ingest data for a suitable amount of time (say 2, 4, 8, 24 hrs). Longer tests are more representative of a production setup.
-

Citus MX (50k/s-500k/s)

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 available in Citus Enterprise Edition and on Citus Cloud. For more information see [Citus MX](#).

Useful Diagnostic Queries

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 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 shardid, shardstate, shardlength, nodename, nodeport, placementid
FROM pg_dist_placement AS placement,
     pg_dist_node AS node
WHERE placement.groupid = node.groupid
      AND node.noderole = 'primary'
      AND 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 |
-----
```

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;
```

Example output:

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

Detecting locks

This query will run across all worker nodes and identify locks, how long they've been open, and the offending queries:

```
SELECT run_command_on_workers($cmd$
SELECT array_agg(
    blocked_statement || ' $ ' || cur_stmt_blocking_proc
    || ' $ ' || cnt::text || ' $ ' || age
)
FROM (
    SELECT blocked_activity.query      AS blocked_statement,
           blocking_activity.query    AS cur_stmt_blocking_proc,
           count(*)                   AS cnt,
           age(now(), min(blocked_activity.query_start)) AS "age"
    FROM pg_catalog.pg_locks          blocked_locks
    JOIN pg_catalog.pg_stat_activity blocked_activity
      ON blocked_activity.pid = blocked_locks.pid
    JOIN pg_catalog.pg_locks          blocking_locks
      ON blocking_locks.locktype = blocked_locks.locktype
    AND blocking_locks.DATABASE IS NOT DISTINCT FROM blocked_locks.DATABASE
    AND blocking_locks.relation IS NOT DISTINCT FROM blocked_locks.relation
    AND blocking_locks.page IS NOT DISTINCT FROM blocked_locks.page
    AND blocking_locks.tuple IS NOT DISTINCT FROM blocked_locks.tuple
    AND blocking_locks.virtualxid IS NOT DISTINCT FROM blocked_locks.virtualxid
    AND blocking_locks.transactionid IS NOT DISTINCT FROM blocked_locks.transactionid
    AND blocking_locks.classid IS NOT DISTINCT FROM blocked_locks.classid
```

```

        AND blocking_locks.objid IS NOT DISTINCT FROM blocked_locks.objid
        AND blocking_locks.objsubid IS NOT DISTINCT FROM blocked_locks.objsubid
        AND blocking_locks.pid != blocked_locks.pid
    JOIN pg_catalog.pg_stat_activity blocking_activity ON blocking_activity.pid =
↪ blocking_locks.pid
    WHERE NOT blocked_locks.GRANTED
        AND blocking_locks.GRANTED
    GROUP BY blocked_activity.query,
             blocking_activity.query
    ORDER BY 4
) a
$cmd$);

```

Example output:

```

-----
|                               run_command_on_workers                               |
-----
| (localhost,5433,t,"")                                                         |
| (localhost,5434,t,"""update ads_102277 set name = 'new name' where id = 1; $↪
↪ sel...|
| ...ect * from ads_102277 where id = 1 for update; $ 1 $ 00:00:03.729519""")      |
↪ |
-----

```

Querying the size of your shards

This query will provide you with the size of every shard of a given distributed table, designated here with the placeholder `my_distributed_table`:

```

SELECT *
FROM run_command_on_shards('my_distributed_table', $cmd$
    SELECT json_build_object(
        'shard_name', '%1$s',
        'size',      pg_size_pretty(pg_table_size('%1$s'))
    );
$cmd$);

```

Example output:

```

-----
| shardid | success | result                                                                 |
↪ |
-----
| 102008 | t       | {"shard_name" : "my_distributed_table_102008", "size" : "2416 kB
↪ "} |
| 102009 | t       | {"shard_name" : "my_distributed_table_102009", "size" : "3960 kB
↪ "} |
| 102010 | t       | {"shard_name" : "my_distributed_table_102010", "size" : "1624 kB
↪ "} |
| 102011 | t       | {"shard_name" : "my_distributed_table_102011", "size" : "4792 kB
↪ "} |
-----

```

Querying the size of all distributed tables

This query gets a list of the sizes for each distributed table plus the size of their indices.

```
SELECT
    tablename,
    pg_size_pretty(
        citus_total_relation_size(tablename::text)
    ) AS total_size
FROM pg_tables pt
JOIN pg_dist_partition pp
    ON pt.tablename = pp.logicalrelid::text
WHERE schemaname = 'public';
```

Example output:

```
-----
|  tablename  | total_size |
-----
| github_users | 39 MB      |
| github_events | 98 MB      |
-----
```

Note that this query works only when `citus.shard_replication_factor = 1`. Also there are other Citrus functions for querying distributed table size, see [Determining Table and Relation Size](#).

Determining Replication Factor per Table

When using Citrus replication rather than PostgreSQL streaming replication, each table can have a customized “replication factor.” This controls the number of redundant copies Citrus keeps of each of the table’s shards. (See [Worker Node Failures](#).)

To see an overview of this setting for all tables, run:

```
SELECT logicalrelid AS tablename,
       count(*)/count(DISTINCT ps.shardid) AS replication_factor
FROM pg_dist_shard_placement ps
JOIN pg_dist_shard p ON ps.shardid=p.shardid
GROUP BY logicalrelid;
```

Example output:

```
-----
|  tablename  | replication_factor |
-----
| github_events | 1 |
| github_users | 1 |
-----
```

Identifying unused indices

This query will run across all worker nodes and identify any unused indexes for a given distributed table, designated here with the placeholder `my_distributed_table`:


```

SELECT *
FROM run_command_on_shards('my_distributed_table', $cmd$
  SELECT array_agg(a) as infos
  FROM (
    SELECT (
      schemaname || '.' || relname || '##' || indexrelname || '##'
      || pg_size_pretty(pg_relation_size(i.indexrelid))::text
      || '##' || idx_scan::text
    ) AS a
    FROM pg_stat_user_indexes ui
    JOIN pg_index i
    ON ui.indexrelid = i.indexrelid
    WHERE NOT indisunique
    AND idx_scan < 50
    AND pg_relation_size(relid) > 5 * 8192
    AND schemaname || '.' || relname = '%s'
    ORDER BY
      pg_relation_size(i.indexrelid) / NULLIF(idx_scan, 0) DESC nulls first,
      pg_relation_size(i.indexrelid) DESC
    ) sub
  $cmd$);

```

Example output:

```

-----
| shardid | success | result
-----
| 102008 | t       |
| 102009 | t       | {"public.my_distributed_table_102009##stupid_index_102009##28 MB
| 102010 | t       |
| 102011 | t       |
-----

```

Monitoring client connection count

This query will give you the connection count by each type that are open on the coordinator:

```

SELECT state, count(*)
FROM pg_stat_activity
GROUP BY state;

```

Exxample output:

```

-----
| state | count |
-----
| active | 3 |
|  | 1 |
-----

```

Index hit rate

This query will provide you with your index hit rate across all nodes. Index hit rate is useful in determining how often indices are used when querying:

```
SELECT nodename, result as index_hit_rate
FROM run_command_on_workers($cmd$
  SELECT CASE sum(idx_blks_hit)
    WHEN 0 THEN 'NaN'::numeric
    ELSE to_char((sum(idx_blks_hit) - sum(idx_blks_read)) / sum(idx_blks_hit + idx_
    ↪blks_read), '99.99')::numeric
  END AS ratio
  FROM pg_statio_user_indexes
$cmd$);
```

Example output:

	index_hit_rate

ec2-13-59-96-221.us-east-2.compute.amazonaws.com	0.88
ec2-52-14-226-167.us-east-2.compute.amazonaws.com	0.89

Common Error Messages

Failed to execute task *n*

This happens when a task fails due to an issue on a particular worker node. For instance, consider a query that generally succeeds but raises an error on one of the workers:

```
CREATE TABLE pageviews (
  page_id int,
  good_views int,
  total_views int
);

INSERT INTO pageviews
VALUES (1, 50, 100), (2, 0, 0);

SELECT create_distributed_table('pageviews', 'page_id');

SELECT page_id,
       good_views / total_views AS goodness
FROM pageviews;
```

The SELECT query fails:

```
ERROR: failed to execute task 50
STATEMENT: SELECT page_id, good_views/total_views AS goodness FROM pageviews;
ERROR: XX000: failed to execute task 50
LOCATION: MultiRealTimeExecute, multi_real_time_executor.c:255
```

To find out what's really going on, we have to examine the database logs inside worker nodes. In our case `page_id=1` is stored on one worker and `page_id=2` on another. The logs for the latter reveal:

```
ERROR: division by zero
STATEMENT: COPY (SELECT page_id, (good_views / total_views) AS goodness FROM_
↪pageviews_102480 pageviews WHERE true) TO STDOUT
WARNING: division by zero
CONTEXT: while executing command on localhost:5433
WARNING: 22012: division by zero
LOCATION: ReportResultError, remote_commands.c:293
```

That's because `total_views` is zero in a row in shard `pageviews_102480`.

Resolution

Check the database logs on worker nodes to identify which query is failing. Common real-life causes for query failure on workers include invalid concatenation of jsonb objects, and typecasting errors. If PgBouncer is between the coordinator and workers, check that it is working properly as well.

Relation *foo* is not distributed

This is caused by attempting to join local and distributed tables in the same query.

Resolution

For an example, with workarounds, see *JOIN a local and a distributed table*.

Could not receive query results

Caused when the *Router Executor* on the coordinator node is unable to connect to a worker. (The *Real-time Executor*, on the other hand, issues *Failed to execute task n* in this situation.)

```
SELECT 1 FROM companies WHERE id = 2928;
```

```
WARNING: connection error: ec2-52-21-20-100.compute-1.amazonaws.com:5432
DETAIL: no connection to the server
ERROR: could not receive query results
```

Resolution

To fix, check that the worker is accepting connections, and that DNS is correctly resolving.

Canceling the transaction since it was involved in a distributed deadlock

Deadlocks can happen not only in a single-node database, but in a distributed database, caused by queries executing across multiple nodes. Citrus has the intelligence to recognize distributed deadlocks and defuse them by aborting one of the queries involved.

We can see this in action by distributing rows across worker nodes, and then running two concurrent transactions with conflicting updates:

```
CREATE TABLE lockme (id int, x int);
SELECT create_distributed_table('lockme', 'id');
```

```
-- id=1 goes to one worker, and id=2 another
INSERT INTO lockme VALUES (1,1), (2,2);
```

```
----- TX 1 ----- TX 2 -----
BEGIN;                                BEGIN;
```

```
UPDATE lockme SET x = 3 WHERE id = 1;
UPDATE lockme SET x = 4 WHERE id = 2;
UPDATE lockme SET x = 3 WHERE id = 2;
UPDATE lockme SET x = 4 WHERE id = 1;
```

```
ERROR:  40P01: canceling the transaction since it was involved in a distributed_
↪deadlock
LOCATION:  ProcessInterrupts, postgres.c:2988
```

Resolution

Detecting deadlocks and stopping them is part of normal distributed transaction handling. It allows an application to retry queries or take another course of action.

Cannot establish a new connection for placement n , since DML has been executed on a connection that is in use

```
BEGIN;
INSERT INTO http_request (site_id) VALUES (1337);
INSERT INTO http_request (site_id) VALUES (1338);
SELECT count(*) FROM http_request;
```

```
ERROR:  25001: cannot establish a new connection for placement 314, since DML has_
↪been executed on a connection that is in use
LOCATION:  FindPlacementListConnection, placement_connection.c:612
```

This is a current limitation. In a single transaction Citrus does not support running insert/update statements with the *Router Executor* that reference multiple shards, followed by a read query that consults both of the shards.

Note: A similar error also occurs (misleadingly) when the *create_distributed_table* function is executed on a table by a role other than the table's owner. See this [github discussion](#) for details. To resolve this particular problem, identify the table's owner, switch roles, and try again.

```
-- find the role
SELECT tablename, tableowner FROM pg_tables;
-- switch into it
SET ROLE table_owner_name;
```

Also note that `table_owner_name` must have LOGIN permissions on the worker nodes.

Resolution

Consider moving the read query into a separate transaction.

Could not connect to server: Cannot assign requested address

```
WARNING: connection error: localhost:9703
DETAIL: could not connect to server: Cannot assign requested address
```

This occurs when there are no more sockets available by which the coordinator can respond to worker requests.

Resolution

Configure the operating system to re-use TCP sockets. Execute this on the shell in the coordinator node:

```
sysctl -w net.ipv4.tcp_tw_reuse=1
```

This allows reusing sockets in TIME_WAIT state for new connections when it is safe from a protocol viewpoint. Default value is 0 (disabled).

Could not connect to any active placements

When all available worker connection slots are in use, further connections will fail.

```
WARNING: 08006: connection error: hostname:5432
ERROR: XX000: could not connect to any active placements
DETAIL: FATAL: sorry, too many clients already
LOCATION: OpenCopyConnections, multi_copy.c:884
```

Resolution

This error happens most often when copying data into Citrus in parallel. The COPY command opens up one connection per shard. If you run M concurrent copies into a destination with N shards, that will result in M*N connections. To solve the error, reduce the shard count of target distributed tables, or run fewer \copy commands in parallel.

Remaining connection slots are reserved for non-replication superuser connections

This occurs when PostgreSQL runs out of available connections to serve concurrent client requests.

Resolution

The `max_connections` GUC adjusts the limit, with a typical default of 100 connections. Note that each connection consumes resources, so adjust sensibly. When increasing `max_connections` it's usually a good idea to increase `memory limits` too.

Using [PgBouncer](#) can also help by queueing connection requests which exceed the connection limit. Citrus Cloud has a built-in PgBouncer instance, see [Scaling Connections \(pgBouncer\)](#) to learn how to connect through it.

PgBouncer cannot connect to server

In a self-hosted Citrus cluster, this error indicates that the coordinator node is not responding to PgBouncer.

Resolution

Try connecting directly to the server with psql to ensure it is running and accepting connections.

Unsupported clause type

This error no longer occurs in the current version of citrus. It used to happen when executing a join with an inequality condition:

```
SELECT *
FROM identified_event ie
JOIN field_calculator_watermark w ON ie.org_id = w.org_id
WHERE w.org_id = 42
AND ie.version > w.version
LIMIT 10;
```

```
ERROR:  unsupported clause type
```

Resolution

Upgrade to Citrus 7.2 or higher.

Cannot open new connections after the first modification command within a transaction

This error no longer occurs in the current version of citrus except in certain unusual shard repair scenarios. It used to happen when updating rows in a transaction, and then running another command which would open new coordinator-to-worker connections.

```
BEGIN;
-- run modification command that uses one connection via
-- the router executor
DELETE FROM http_request
WHERE site_id = 8
AND ingest_time < now() - '1 week'::interval;

-- now run a query that opens connections to more workers
SELECT count(*) FROM http_request;
```

```
ERROR:  cannot open new connections after the first modification command within a
↳ transaction
```

Resolution

Upgrade to Citrus 7.2 or higher.

ON CONFLICT is not supported via coordinator

Running an INSERT...SELECT statement with an ON CONFLICT clause will fail unless the source and destination tables are co-located, and unless the distribution column is among the columns selected from the source and inserted in the destination. Also if there is a GROUP BY clause it must include the distribution column. Failing to meet these conditions will raise an error:

```
ERROR: ON CONFLICT is not supported in INSERT ... SELECT via coordinator
```

Resolution

Add the table distribution column to both the select and insert statements, as well as the statement GROUP BY if applicable. For more info as well as a workaround, see *INSERT...SELECT upserts lacking distribution column*.

Cannot create uniqueness constraint

As a distributed system, Citrus can guarantee uniqueness only if a unique index or primary key constraint includes a table's distribution column. That is because the shards are split so that each shard contains non-overlapping partition column values. The index on each worker node can locally enforce its part of the constraint.

Trying to make a unique index on a non-distribution column will generate an error:

```
ERROR: 0A000: cannot create constraint on "foo"  
DETAIL: Distributed relations cannot have UNIQUE, EXCLUDE, or PRIMARY KEY  
→constraints that do not include the partition column (with an equality operator if  
→EXCLUDE).  
LOCATION: ErrorIfUnsupportedConstraint, multi_utility.c:2505
```

Enforcing uniqueness on a non-distribution column would require Citrus to check every shard on every INSERT to validate, which defeats the goal of scalability.

Resolution

There are two ways to enforce uniqueness on a non-distribution column:

1. Create a composite unique index or primary key that includes the desired column (*C*), but also includes the distribution column (*D*). This is not quite as strong a condition as uniqueness on *C* alone, but will ensure that the values of *C* are unique for each value of *D*. For instance if distributing by `company_id` in a multi-tenant system, this approach would make *C* unique within each company.
2. Use a *reference table* rather than a hash distributed table. This is only suitable for small tables, since the contents of the reference table will be duplicated on all nodes.

Function `create_distributed_table` does not exist

```
SELECT create_distributed_table('foo', 'id');
/*
ERROR:  42883: function create_distributed_table(unknown, unknown) does not exist
LINE 1: SELECT create_distributed_table('foo', 'id');
HINT:  No function matches the given name and argument types. You might need to add
      ↪explicit type casts.
*/
```

Resolution

When basic *Citus Utility Functions* are not available, check whether the Citrus extension is properly installed. Running `\dx` in `psql` will list installed extensions.

One way to end up without extensions is by creating a new database in a Postgres server, which requires extensions to be re-installed. See *Creating a New Database* to learn how to do it right.

STABLE functions used in UPDATE queries cannot be called with column references

Each PostgreSQL function is marked with a *volatility*, which indicates whether the function can update the database, and whether the function's return value can vary over time given the same inputs. A *STABLE* function is guaranteed to return the same results given the same arguments for all rows within a single statement, while an *IMMUTABLE* function is guaranteed to return the same results given the same arguments forever.

Non-immutable functions can be inconvenient in distributed systems because they can introduce subtle changes when run at slightly different times across shard replicas. Differences in database configuration across nodes can also interact harmfully with non-immutable functions.

One of the most common ways this can happen is using the `timestamp` type in Postgres, which unlike `timestampz` does not keep a record of time zone. Interpreting a timestamp column makes reference to the database timezone, which can be changed between queries, hence functions operating on timestamps are not immutable.

Citus forbids running distributed queries that filter results using stable functions on columns. For instance:

```
-- foo_timestamp is timestamp, not timestampz
UPDATE foo SET ... WHERE foo_timestamp < now();
```

```
ERROR:  0A000: STABLE functions used in UPDATE queries cannot be called with column
      ↪references
```

In this case the comparison operator `<` between `timestamp` and `timestampz` is not immutable.

Resolution

Avoid stable functions on columns in a distributed `UPDATE` statement. In particular, whenever working with times use `timestampz` rather than `timestamp`. Having a time zone in `timestampz` makes calculations immutable.

Frequently Asked Questions

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.

How do I add nodes to an existing Citus cluster?

On Citus Cloud it's as easy as dragging a slider in the user interface. The *Scaling Out (adding new nodes)* section has instructions. In Citus Community edition you can add nodes manually by calling the `master_add_node` UDF with the hostname (or IP address) and port number of the new node.

Either way, 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, Citus Cloud and Enterprise edition provide a shard rebalancer utility. You can find more information in the *Rebalance Shards without Downtime* section.

How does Citus handle failure of a worker node?

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 coordinator 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.

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*.

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.

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 wrap the local table in a subquery. Citus' subquery execution logic will allow the join to work. See *JOIN a local and a distributed table*.

Are there any PostgreSQL features not supported by Citus?

Since Citus provides distributed functionality by extending PostgreSQL, it uses the standard PostgreSQL SQL constructs. The vast majority of queries are supported, even when they combine data across the network from multiple database nodes. This includes transactional semantics across nodes. Currently all SQL is supported except:

- Correlated subqueries
- Recursive CTEs
- Table sample
- `SELECT ... FOR UPDATE`
- Grouping sets
- Window functions that do not include the distribution column in `PARTITION BY`

What's more, Citus has 100% 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*).

Remember that – even with this extensive SQL coverage – data modeling can have a significant impact on query performance. See the section on *Query Processing* for details on how Citus executes queries.

How do I choose the shard count when I hash-partition my data?

One of the choices when first distributing a table is its shard count. This setting can be set differently for each table, and the optimal value depends on use-case. It is possible, but difficult, to change the count after cluster creation, so use these guidelines to choose the right size.

In the *Multi-Tenant Database* use-case we recommend choosing between 32 - 128 shards. For smaller workloads say <100GB, you could start with 32 shards and for larger workloads you could choose 64 or 128. This means that you have the leeway to scale from 32 to 128 worker machines.

In the *Real-Time Analytics* use-case, shard count should be related to the total number of cores on the workers. 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.

To choose a shard count for a table you wish to distribute, update the `citus.shard_count` variable. This affects subsequent calls to `create_distributed_table`. For example

```
SET citus.shard_count = 64;
-- any tables distributed at this point will have
-- sixty-four shards
```

For more guidance on this topic, see *Choosing Cluster Size*.

How do I change the shard count for a hash partitioned table?

Note that it is not straightforward to change the shard count of an already distributed table. If you need to do so, please [Contact Us](#). It's good to think about shard count carefully at distribution time, see *How do I choose the shard count when I hash-partition my data?*.

How does citus support count(distinct) queries?

Citus can evaluate count(distinct) aggregates both in and across worker nodes. When aggregating on a table's distribution column, Citus can push the counting down inside worker nodes and total the results. Otherwise it can pull distinct rows to the coordinator and calculate there. If transferring data to the coordinator is too expensive, fast approximate counts are also available. More details in *Count (Distinct) Aggregates*.

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.

How do I create database roles, functions, extensions etc in a Citrus cluster?

Certain commands, when run on the coordinator node, do not get propagated to the workers:

- CREATE ROLE/USER
- CREATE FUNCTION
- CREATE TYPE
- CREATE EXTENSION
- CREATE DATABASE
- ALTER ... SET SCHEMA
- ALTER TABLE ALL IN TABLESPACE

It is still possible to use these commands by explicitly running them on all nodes. Citrus provides a function to execute queries across all workers:

```
SELECT run_command_on_workers($cmd$
/* the command to run */
CREATE FUNCTION ...
$cmd$);
```

Learn more in *Manual Query Propagation*. Also note that even after manually propagating CREATE DATABASE, Citrus must still be installed there. See *Creating a New Database*.

What if a worker node's address changes?

If the hostname or IP address of a worker changes, you need to let the coordinator know using *master_update_node*:

```
-- update worker node metadata on the coordinator
-- (remember to replace 'old-address' and 'new-address'
-- with the actual values for your situation)

select master_update_node(nodeid, 'new-address', nodeport)
from pg_dist_node
where nodename = 'old-address';
```

Until you execute this update, the coordinator will not be able to communicate with that worker for queries.

Which shard contains data for a particular tenant?

Citrus 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.

I forgot the distribution column of a table, how do I find it?

The Citrus coordinator node metadata tables contain this information. See *Finding the distribution column for a table*.

Can I distribute a table by multiple keys?

No, you must choose a single column per table as the distribution column. A common scenario where people want to distribute by two columns is for timeseries data. However for this case we recommend using a hash distribution on a non-time column, and combining this with PostgreSQL partitioning on the time column, as described in [Timeseries Data](#).

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. Citrus provides helper functions to query this information. See [Determining Table and Relation Size](#) to learn more.

Why am I seeing an error about `max_intermediate_result_size`?

Citus has to use more than one step to run some queries having subqueries or CTEs. Using [Subquery/CTE Push-Pull Execution](#), it pushes subquery results to all worker nodes for use by the main query. If these results are too large, this might cause unacceptable network overhead, or even insufficient storage space on the coordinator node which accumulates and distributes the results.

Citus has a configurable setting, `citus.max_intermediate_result_size` to specify a subquery result size threshold at which the query will be canceled. If you run into the error, it looks like:

```
ERROR:  the intermediate result size exceeds citus.max_intermediate_result_size,
↪(currently 1 GB)
DETAIL:  Citus restricts the size of intermediate results of complex subqueries and
↪CTEs to avoid accidentally pulling large result sets into once place.
HINT:  To run the current query, set citus.max_intermediate_result_size to a higher
↪value or -1 to disable.
```

As the error message suggests, you can (cautiously) increase this limit by altering the variable:

```
SET citus.max_intermediate_result_size = '3GB';
```

Can I run Citrus on Heroku?

Yes, the [Citrus Heroku add-on](#) provisions a Citrus cluster and makes it available to a Heroku app.

Can I run Citrus on Amazon RDS?

At this time Amazon does not support running Citrus directly on top of Amazon RDS.

If you are looking for something similar, [Citrus 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 Citrus' horizontal scaling.

Can I use Citus with my existing AWS account?

Yes, [Citus Cloud on the AWS Marketplace](#) allows you to provision a Citus Cloud cluster directly through Amazon Web Services.

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](#).

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`.

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.

Related Articles

Introducing Citus Add-on for Heroku: Scale out your Postgres

(Copy of [original publication](#))

Just as Heroku has made it simple for you to deploy applications, at [Citus Data](#) we aim to make it simple for you to scale out your Postgres database.

Once upon a time at Heroku, it all started with `git push heroku master`. Later, the team at [Heroku](#) made it easy to add any service you could want within your app via `heroku addons:create foo`. The simplicity of dragging a slider to scale up your dynos is the type of awesome customer experience we strive to create at Citus. With Citus Cloud (our fully-managed [database as a service](#)), you can simply drag and save—then voila, you’ve scaled the resources to your database.

Today we are excited to announce our [Citus add-on for Heroku](#). Now with `heroku addons:create citus` you can get a database that scales out just as easily as your app does.

Fig. 29.1: Citus console provisioning slider

When running Postgres on a single node, the need to scale out can occur anywhere from 100 GB of data up to 1 TB. If you’re running Postgres already, then Citus is a natural fit. With Citus you’ll get much of the experience you expect on Heroku, coupled with the ability to keep scaling out your database. Let’s take a deeper look at a few of the features available for the Citus add-on.

Continuous protection and monitoring with Citus Cloud

Citus leverages the same underlying tools as Heroku Postgres for continuous protection. In fact, the team that runs Citus Cloud is the founding engineering team from Heroku Postgres.

With continuous protection, Citus takes incremental backups of your database every minute. From there our [control plane](#) then monitors your database every 30 seconds to make sure your database is healthy and active. If anything goes wrong, our [Citus state machine](#) begins a series of automated processes to restore availability. If our systems determine it’s needed, we may perform an [HA failover or disaster recovery](#), leveraging the continuous protection and frequent incremental backups to ensure your data is still safe.

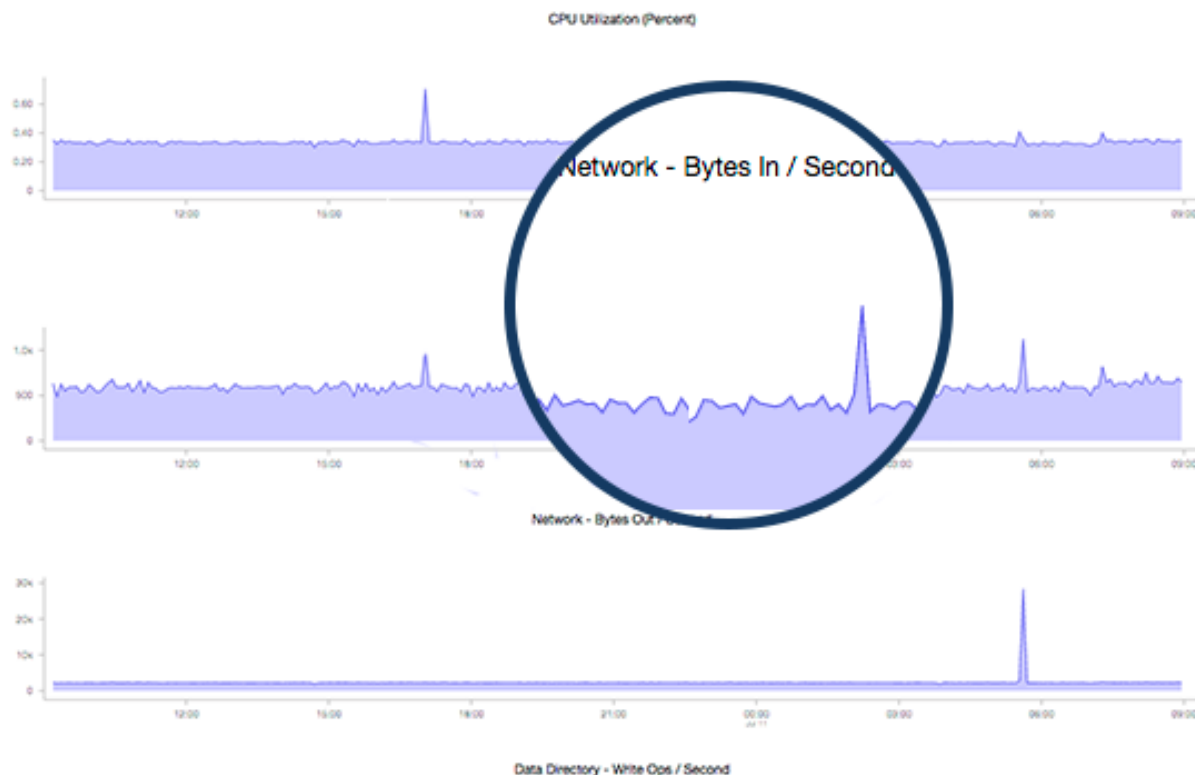
I like to say that we wear the pager for you, to keep you up and running. So you can focus on your app, not your database.

The insights you need

When managing an app, there can be a lot to juggle: there are endless options for metrics and what to pay attention to. Many of the standard best practices for Postgres performance apply—most importantly, I always recommend you pay attention to your [cache hit rate](#) and index hit rate, both of which we make available for you in the Citus Cloud dashboard.

We also make it easy for you to centralize your logs into any one of the [Heroku add-on providers](#) of your choosing. To do this you can create a new secure TLS drain which will send the logs from all Citus database nodes to your logging system.

Finally, with Citus, you get insights into system level metrics such as disk i/o, network i/o, and perhaps most key, the CPU load:



Granular access control in Citus Cloud

We know that Heroku makes it easy to add new collaborators to your app. And when it comes to your database you may want to grant access to users that are a bit less familiar with a CLI—or perhaps you don’t want some of your users to have full access to your Heroku app. With Citus Cloud, you can add new users and restrict their permissions however you see fit. Want to allow inserts but not deletes? Done. Want them to only have read-only permissions? Done. You have the ultimately flexibility to control roles how you see fit.

Scale out Postgres on Heroku today

At Citus we get that flexibility matters. So we’re making it simple for you by allowing custom Citus add-on plans to be created. If you don’t see the Heroku add-on plan that makes sense for you, just let us or let the Heroku team know

and we'll make sure to get it created and available for you.

We want scaling your Postgres database on Heroku to be as easy as scaling your Heroku app. Which is why we're excited to roll out the Citrus add-on to help you grow. If you're using Heroku Postgres and are looking for scale, [give the Citrus add-on for Heroku](#) a try today.

Efficient Rollup Tables with HyperLogLog in Postgres

(Copy of [original publication](#))

Rollup tables are commonly used in Postgres when you don't need to perform detailed analysis, but you still need to answer basic aggregation queries on older data.

With rollup tables, you can pre-aggregate your older data for the queries you still need to answer. Then you no longer need to store all of the older data, rather, you can delete the older data or roll it off to slower storage—saving space and computing power.

Let's walk through a rollup table example in Postgres without using HLL.

Rollup tables without HLL—using GitHub events data as an example

For this example we will create a rollup table that aggregates historical data: [a GitHub events data set](#).

Each record in this GitHub data set represents an event created in GitHub, along with key information regarding the event such as event type, creation date, and the user who created the event. (Craig Kerstiens has written about this same data set in the past, in his [getting started with GitHub event data on Citus](#) post.)

If you want to create a chart to show the number of GitHub event creations in each minute, a rollup table would be super useful. With a rollup table, you won't need to store all user events in order to create the chart. Rather, you can aggregate the number of event creations for each minute and just store the aggregated data. You can then throw away the rest of the events data, if you are trying to conserve space.

To illustrate the example above, let's create `github_events` table and load data to the table:

```
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
);

\COPY github_events FROM events.csv CSV
```

In this example, I'm assuming you probably won't perform detailed analysis on your older data on a regular basis. So there is no need to allocate resources for the older data, instead you can use rollup tables and just keep the necessary information in memory. You can create a rollup table for this purpose:

```
CREATE TABLE github_events_rollup_minute
(
    created_at timestamp,
```

```
    event_count bigint
);
```

And populate with INSERT/SELECT:

```
INSERT INTO github_events_rollup_minute(
    created_at,
    event_count
)
SELECT
    date_trunc('minute', created_at) AS created_at,
    COUNT(*) AS event_count
FROM github_events
GROUP BY 1;
```

Now you can store the older (and bigger) data in a less expensive resource like disk so that you can access it in the future—and keep the `github_events_rollup_minute` table in memory so you can create your analytics dashboard.

By aggregating the data by minute in the example above, you can answer queries like hourly and daily total event creations, but unfortunately it is not possible to know the more granular event creation count for each second.

Further, since you did not keep event creations for each user separately (at least not in this example), you cannot have a separate analysis for each user with this rollup table. All of these are trade-offs.

Without HLL, rollup tables have a few limitations

For queries involving distinct count, rollup tables are less useful. For example, if you pre-aggregate over minutes, you cannot answer queries asking for distinct counts over an hour. You cannot add each minute's result to have hourly event creations by unique users. Why? Because you are likely to have overlapping records in different minutes.

And if you want to calculate distinct counts constrained by combinations of columns, you would need multiple rollup tables.

Sometimes you want to get event creation count by unique users filtered by date and sometimes you want to get unique event creation counts filtered by event type (and sometimes a combination of both.) With HLL, one rollup table can answer all of these queries—but without HLL, you would need a separate rollup table for each of these different types of queries.

HLL to the rescue

If you do rollups with the HLL data type (instead of rolling up the final unique user count), you can easily overcome the overlapping records problem. HLL encodes the data in a way that allows summing up individual unique counts without re-counting overlapping records.

HLL is also useful if you want to calculate distinct counts constrained by combinations of columns. For example, if you want to get unique event creation counts per date and/or per event type, with HLL, you can use just one rollup table for all combinations.

Whereas without HLL, if you want to calculate distinct counts constrained by combinations of columns, you would need to create:

- 7 different rollup tables to cover all combinations of 3 columns
- 15 rollup tables to cover all combinations of 4 columns
- $2^n - 1$ rollup tables to cover all combinations in n columns

HLL and rollup tables in action, together

Let's see how HLL can help us to answer some typical distinct count queries on GitHub events data. If you did not create a `github_events` table in the previous example, create and populate it now with the [GitHub events data set](#):

```
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
);

\COPY github_events FROM events.csv CSV
```

After creating your table, let's also create a rollup table. We want to get distinct counts both per `user` and per `event_type` basis. Therefore you should use a slightly different rollup table:

```
DROP TABLE IF EXISTS github_events_rollup_minute;

CREATE TABLE github_events_rollup_minute(
    created_at timestamp,
    event_type text,
    distinct_user_id_count hll
);
```

Finally, you can use `INSERT/SELECT` to populate your rollup table and you can use `hll_hash_bigint` function to hash each `user_id`. (For an explanation of why you need to hash elements, be sure to read our [Citius blog post on distributed counts with HyperLogLog on Postgres](#)):

```
INSERT INTO github_events_rollup_minute(
    created_at,
    event_type,
    distinct_user_id_count
)
SELECT
    date_trunc('minute', created_at) AS created_at,
    event_type,
    sum(hll_hash_bigint(user_id))
FROM github_events
GROUP BY 1, 2;

INSERT 0 2484
```

What kinds of queries can HLL answer?

Let's start with a simple case to see how to materialize HLL values to actual distinct counts. To demonstrate that, we will answer the question:

How many distinct users created an event for each event type at each minute at 2016-12-01 05:35:00?

We will just need to use the `hll_cardinality` function to materialize the HLL data structures to actual distinct count.

```
SELECT
    created_at,
    event_type,
    hll_cardinality(distinct_user_id_count) AS distinct_count
FROM
    github_events_rollup_minute
WHERE
    created_at = '2016-12-01 05:35:00'::timestamp
ORDER BY 2;
```

created_at	event_type	distinct_count
2016-12-01 05:35:00	CommitCommentEvent	1
2016-12-01 05:35:00	CreateEvent	59
2016-12-01 05:35:00	DeleteEvent	6
2016-12-01 05:35:00	ForkEvent	20
2016-12-01 05:35:00	GollumEvent	2
2016-12-01 05:35:00	IssueCommentEvent	42
2016-12-01 05:35:00	IssuesEvent	13
2016-12-01 05:35:00	MemberEvent	4
2016-12-01 05:35:00	PullRequestEvent	24
2016-12-01 05:35:00	PullRequestReviewCommentEvent	4
2016-12-01 05:35:00	PushEvent	254.135297564883
2016-12-01 05:35:00	ReleaseEvent	4
2016-12-01 05:35:00	WatchEvent	57

(13 rows)

Then let's continue with a query which we could not answer without HLL:

How many distinct users created an event during this one-hour period?

With HLLs, this is easy to answer.

```
SELECT
    hll_cardinality(SUM(distinct_user_id_count)) AS distinct_count
FROM
    github_events_rollup_minute
WHERE
    created_at BETWEEN '2016-12-01 05:00:00'::timestamp AND '2016-12-01 06:00:00'::
    ↳timestamp;
```

distinct_count
10978.2523520687

(1 row)

Another question where we can use HLL's additivity property to answer would be:

How many unique users created an event during each hour at 2016-12-01?

```
SELECT
    EXTRACT(HOUR FROM created_at) AS hour,
    hll_cardinality(SUM(distinct_user_id_count)) AS distinct_count
FROM
    github_events_rollup_minute
WHERE
```

```

    created_at BETWEEN '2016-12-01 00:00:00'::timestamp AND '2016-12-01 23:59:59'::
↪timestamp
GROUP BY 1
ORDER BY 1;

 hour | distinct_count
-----+-----
    5 | 10598.637184899
    6 | 17343.2846931687
    7 | 18182.5699816622
    8 | 12663.9497604266
(4 rows)

```

Since our data is limited, the query only returned 4 rows, but that is not the point of course. Finally, let's answer a final question:

How many distinct users created a PushEvent during each hour?

```

SELECT
    EXTRACT(HOUR FROM created_at) AS hour,
    hll_cardinality(SUM(distinct_user_id_count)) AS distinct_push_count
FROM
    github_events_rollup_minute
WHERE
    created_at BETWEEN '2016-12-01 00:00:00'::timestamp AND '2016-12-01 23:59:59'::
↪timestamp
    AND event_type = 'PushEvent'::text
GROUP BY 1
ORDER BY 1;

 hour | distinct_push_count
-----+-----
    5 | 6206.61586498546
    6 | 9517.80542100396
    7 | 10370.4087640166
    8 | 7067.26073810357
(4 rows)

```

A rollup table with HLL is worth a thousand rollup tables without HLL

Yes, I believe a rollup table with HLL is worth a thousand rollup tables without HLL.

Well, maybe not a thousand, but it is true that one rollup table with HLL can answer lots of queries where otherwise you would need a different rollup table for each query. Above, we demonstrated that with HLL, 4 example queries all can be answered with a single rollup table— and without HLL, we would have needed 3 separate rollup tables to answer all these queries.

In the real world, if you do not take advantage of HLL you are likely to need even more rollup tables to support your analytics queries. Basically for all combinations of n constraints, you would need $2^n - 1$ rollup tables whereas with HLL just one rollup table can do the job.

One rollup table (with HLL) is obviously much easier to maintain than multiple rollup tables. And that one rollup table uses significantly less memory too. In some cases, without HLL, the overhead of using rollup tables can become too expensive and exceeds the benefit of using rollup tables, so people decide not to use rollup tables at all.

Want to learn more about HLL in Postgres?

HLL is not only useful to create rollup tables, HLL is useful in distributed systems, too. Just as with rollup tables, in a distributed system, such as Citrus, we often place different parts of our data in different nodes, hence we are likely to have overlapping records at different nodes. Thus, the clever techniques HLL uses to encode data to merge separate unique counts (and address the overlapping record problem) can also help in distributed systems.

If you want to learn more about HLL, read [how HLL can be used in distributed systems](#), where we explained the internals of HLL and how HLL merges separate unique counts without counting overlapping records.

Distributed Distinct Count with HyperLogLog on Postgres

(Copy of [original publication](#))

Running `SELECT COUNT(DISTINCT)` on your database is all too common. In applications it's typical to have some analytics dashboard highlighting the number of unique items such as unique users, unique products, unique visits. While traditional `SELECT COUNT(DISTINCT)` queries works well in single machine setups, it is a difficult problem to solve in distributed systems. When you have this type of query, you can't just push query to the workers and add up results, because most likely there will be overlapping records in different workers. Instead you can do:

- Pull all distinct data to one machine and count there. (Doesn't scale)
- Do a map/reduce. (Scales but it's very slow)

This is where approximation algorithms or sketches come in. Sketches are probabilistic algorithms which can generate approximate results efficiently within mathematically provable error bounds. There are many of them out there, but today we're just going to focus on one, HyperLogLog or HLL. HLL is very successful for estimating unique number of elements in a list. First we'll look some at the internals of the HLL to help us understand why HLL algorithm is useful to solve distinct count problem in a scalable way, then how it can be applied in a distributed fashion. Then we will see some examples of HLL usage.

What does HLL do behind the curtains?

Hash all elements

HLL and almost all other probabilistic counting algorithms depend on uniform distribution of the data. Since in the real world, our data is generally not distributed uniformly, HLL firsts hashes each element to make the data distribution more uniform. Here, by uniform distribution, we mean that each bit of the element has 0.5 probability of being 0 or 1. We will see why this is useful in couple of minutes. Apart from uniformity, hashing allows HLL to treat all data types same. As long as you have a hash function for your data type, you can use HLL for cardinality estimation.

Observe the data for rare patterns

After hashing all the elements, HLL looks for the binary representation of each hashed element. It mainly looks if there are bit patterns which are less likely to occur. Existence of such rare patterns means that we are dealing with large dataset.

For this purpose, HLL looks number of leading zero bits in the hash value of each element and finds maximum number of leading zero bits. Basically, to be able to observe k leading zeros, we need $2k+1$ trials (i.e. hashed numbers). Therefore, if maximum number of leading zeros is k in a data set, HLL concludes that there are approximately $2k+1$ distinct elements.

This is pretty straightforward and simple estimation method. However; it has some important properties, which are especially shine in distributed environment;

- HLL has very low memory footprint. For maximum number n , we need to store just $\log \log n$ bits. For example; if we hash our elements into 64 bit integers, we just need to store 6 bits to make an estimation. This saves a lot of memory especially compared with naive approach where we need to remember all the values.
- We only need to do one pass on the data to find maximum number of leading zeros.
- We can work with streaming data. After calculating maximum number of leading zeros, if some new data arrives we can include them into calculation without going over whole data set. We only need to find number of leading zeros of each new element, compare them with maximum number of leading zeros of whole dataset and update maximum number of leading zeros if necessary.
- We can merge estimations of two separate datasets efficiently. We only need to pick bigger number of leading zeros as maximum number of leading zeros of combined dataset. This allow us to separate the data into shards, estimate their cardinality and merge the results. This is called additivity and it allow us to use HLL in distributed systems.

Stochastic Averaging

If you think above is not that good estimation, you are right. First of all, our prediction is always in the form of $2k$. Secondly we may end up with pretty far estimates if the data distribution is not uniform enough.

One possible fix for these problems could be just repeating the process with different hash functions and taking the average, which would work fine but hashing all the data multiple times is expensive. HLL fixes this problem something called stochastic averaging. Basically, we divide our data into buckets and use aforementioned algorithm for each bucket separately. Then we just take the average of the results. We use first few bits of the hash value to determine which bucket a particular element belongs to and use remaining bits to calculate maximum number of leading zeros.

Moreover, we can configure precision by choosing number of buckets to divide the data. We will need to store $\log \log n$ bits for each bucket. Since we can store each estimation in $\log \log n$ bits, we can create lots of buckets and still end up using insignificant amount of memory. Having such small memory footprint is especially important while operating on large scale data. To merge two estimations, we will merge each bucket then take the average. Therefore, if we plan to do merge operation, we should keep each bucket's maximum number of leading zeros.

More?

HLL does some other things too to increase accuracy of the estimation, however observing bit patterns and stochastic averaging is the key points of HLL. After these optimizations, HLL can estimate cardinality of a dataset with typical error rate 2% error rate using 1.5 kB of memory. Of course is is possible to increase accuracy by using more memory. We will not go into details of other steps but there are tons of content on the internet about HLL.

HLL in distributed systems

As we mentioned, HLL has additivity property. This means you can divide your dataset into several parts, operate on them with HLL algorithm to find unique element count of each part. Then you can merge intermediate HLL results efficiently to find unique element count of all data without looking back to original data.

If you work on large scale data and you keep parts of your data in different physical machines, you can use HLL to calculate unique count over all your data without pulling whole data to one place. In fact, Citus can do this operation for you. There is a [HLL extension](#) developed for PostgreSQL and it is fully compatible with Citus. If you have HLL extension installed and want to run COUNT(DISTINCT) query on a distributed table, Citus automatically uses HLL. You do not need to do anything extra once you configured it.

Hands on with HLL

Setup

To play with HLL we will use Citus Cloud and GitHub events data. You can see and learn more about Citus Cloud and this data set from [here](#). Assuming you created your Citus Cloud instance and connected it via psql, you can create HLL extension by;

```
CREATE EXTENSION hll;
```

You should create the extension at master and all workers. Then 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 and use more memory for computation. We recommend setting this to 0.005.

```
SET citus.count_distinct_error_rate TO 0.005;
```

Different from [previous blog post](#), we will only use github_events table and we will use [large_events.csv](#) data set;

```
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
);

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

\COPY github_events FROM large_events.csv CSV
```

Examples

After distributing the table, we can use regular COUNT(DISTINCT) query to find out how many unique users created an event;

```
SELECT
    COUNT(DISTINCT user_id)
FROM
    github_events;
```

It should return something like this;

```
count
-----
264227
(1 row)
```

It looks like this query does not have anything with HLL. However if you set *citus.count_distinct_error_rate* to something bigger than 0 and issue COUNT(DISTINCT) query; Citus automatically uses HLL. For simple use-cases like

this, you don't even need to change your queries. Exact distinct count of users who created an event is 264198, so our error rate is little bigger than 0.0001.

We can also use constraints to filter out some results. For example we can query number of unique users who created a `PushEvent`;

```
SELECT
    COUNT(DISTINCT user_id)
FROM
    github_events
WHERE
    event_type = 'PushEvent'::text;
```

It would return;

```
count
-----
157471
(1 row)
```

Similarly exact distinct count for this query is 157154 and our error rate is little bigger than 0.002.

Conclusion

If you're having trouble scaling `count (distinct)` in Postgres give HLL a look it may be useful if close enough counts are feasible for you.

How to Scale PostgreSQL on AWS: Learnings from Citus Cloud

(Copy of [original publication](#))

One challenge associated with building a distributed relational database (RDBMS) is that they require notable effort to deploy and operate. To remove these operational barriers, we had been thinking about offering Citus as a managed database for a long time.

Naturally, we were also worried that providing a native database offering on AWS could split our startup's focus and take up significant engineering resources. (Honestly, if the founding engineers of the Heroku PostgreSQL team didn't join Citus, we might have decided to wait on this.) Since making Citus Cloud publicly available, we are now more bullish on the cloud then ever.

It turns out that targeting an important use case for your customers and delivering it to them in a way that removes their pain points, matters more than anything else. In this article, we'll only focus on removing operational pain points and not on use cases: Why is cloud changing the way databases are delivered to customers? What AWS technologies Citus Cloud is using to enable that in a unique way?

The following sections highlight our learnings from talking to hundreds of prospective customers. Early on, we thought that our customers were asking for something that was more magical than attainable – Citus should handle failovers in a way that's transparent to the application, elastically scale up and scale out, take automatic backups for disaster recovery, have monitoring and alarms built in, and provide patches and upgrades with no downtime.

We then started breaking down each customer conversation into smaller pieces and found that our customers had seven operational pain points. In the following, we're going to talk about each one of these seven points as a mini-learning. Although we describe these learnings in the context of PostgreSQL, Citus, and AWS, they are also applicable to other databases and cloud vendors.

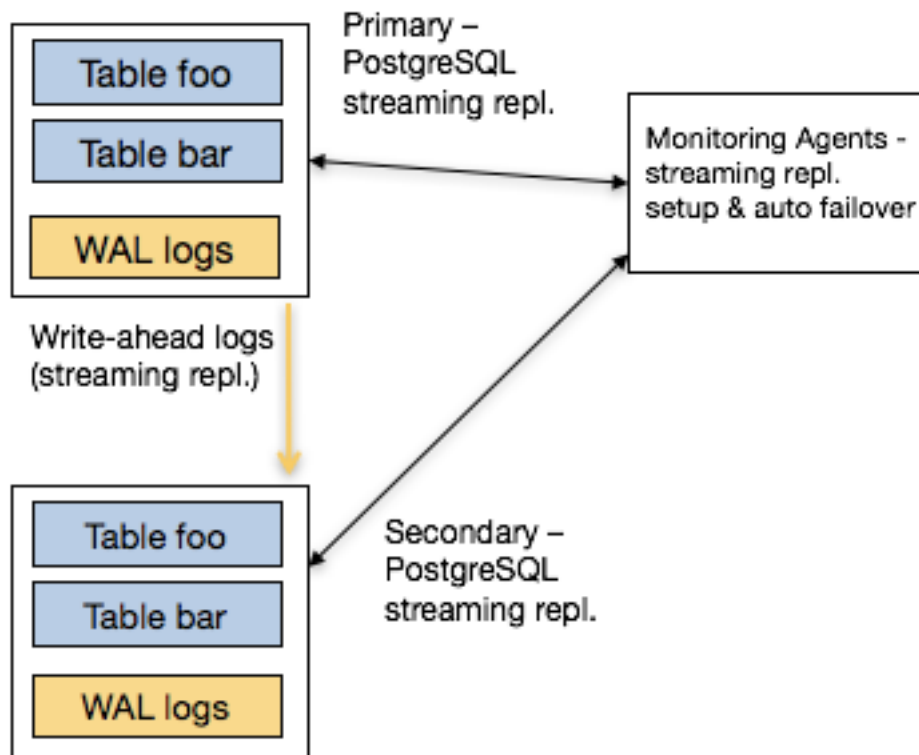
Since Citus is an extension to PostgreSQL (not a fork), it makes sense to start with a pain most PostgreSQL users have.

1. High Availability

PostgreSQL provides [streaming replication](#) as a way to provide high availability since its v9.0 release. Streaming replication works by designating one PostgreSQL instance as the primary node. The primary node takes all writes and executes all changes against the database. It then ships those changes to one or more secondary nodes for them to apply.

When you talk to users who use streaming replication in production, they express two pain points with it. The first is the complexity associated with it: [it takes twelve steps](#) to setup streaming replication, and you may also need to understand concepts such as the number of wal segments and replication lag in the process.

The second and more important one relates to auto failover. If the primary node fails, how do you detect the failure and promote a secondary to be the new primary? (In distributed systems, this is also known as the leader election problem.) To enable this, you need a monitoring component that watches the primary and secondary nodes and reliably initiates their failovers.



A common way to do that is through integrating PostgreSQL streaming replication with a distributed synchronization solution such as etcd or ZooKeeper. In fact, open source solutions such as [Governor](#) and [Patroni](#) aim to do just that. That said, this integration again comes with a complexity cost. You now need to have someone in your organization who understands these systems – so that you can resolve any issues should they come up in production.

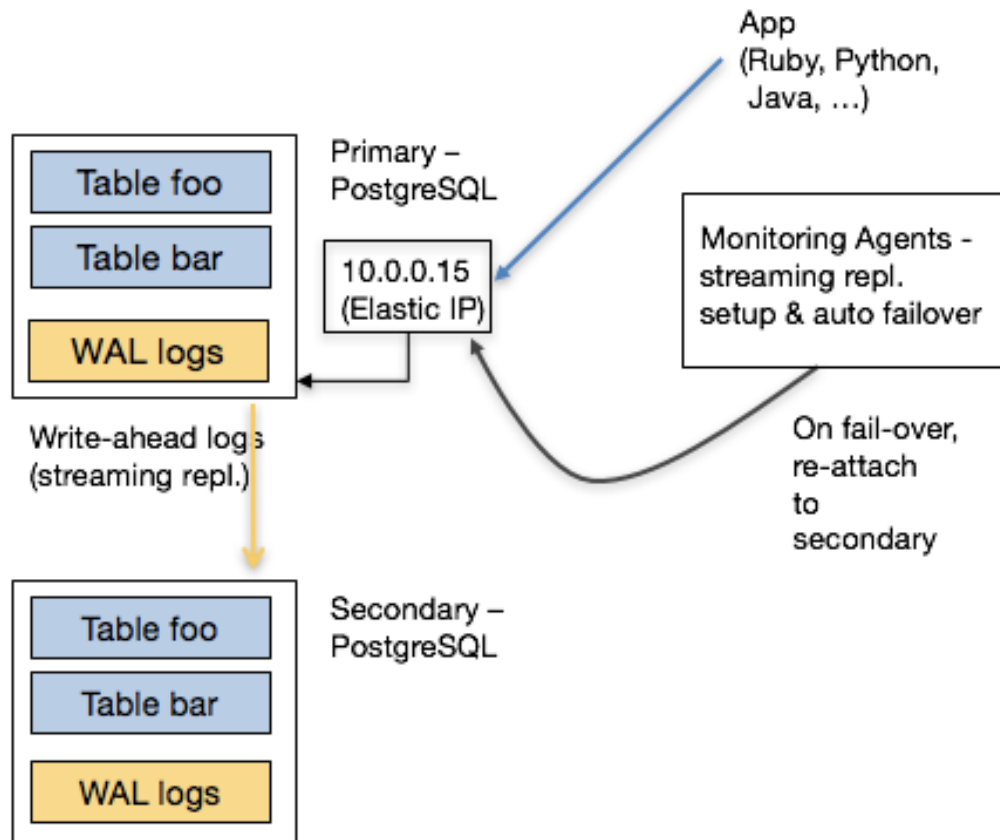
2. Effortless Failovers

Once the underlying HA solution provides auto failover capabilities, the next question becomes this: “My PostgreSQL driver / ORM tool talks to an IP address. When the primary node changes, do I need to add logic into my application

to detect this failover?”

PostgreSQL has a massive ecosystem that supports many programming languages and frameworks. And most PostgreSQL clients don’t have a mechanism to automatically retry different endpoints in case of a failure. This either means that the user would need to implement this logic into their application or the underlying database provides it for them.

Fortunately, AWS provides powerful software defined networking (SDN) primitives. You can therefore start by attaching an elastic IP to the primary node. When your monitoring agent promotes a new secondary, you can detach this elastic IP from the old primary and attach it to the newly promoted machine. This way, the application doesn’t need to change the endpoint it talks to – it happens magically.



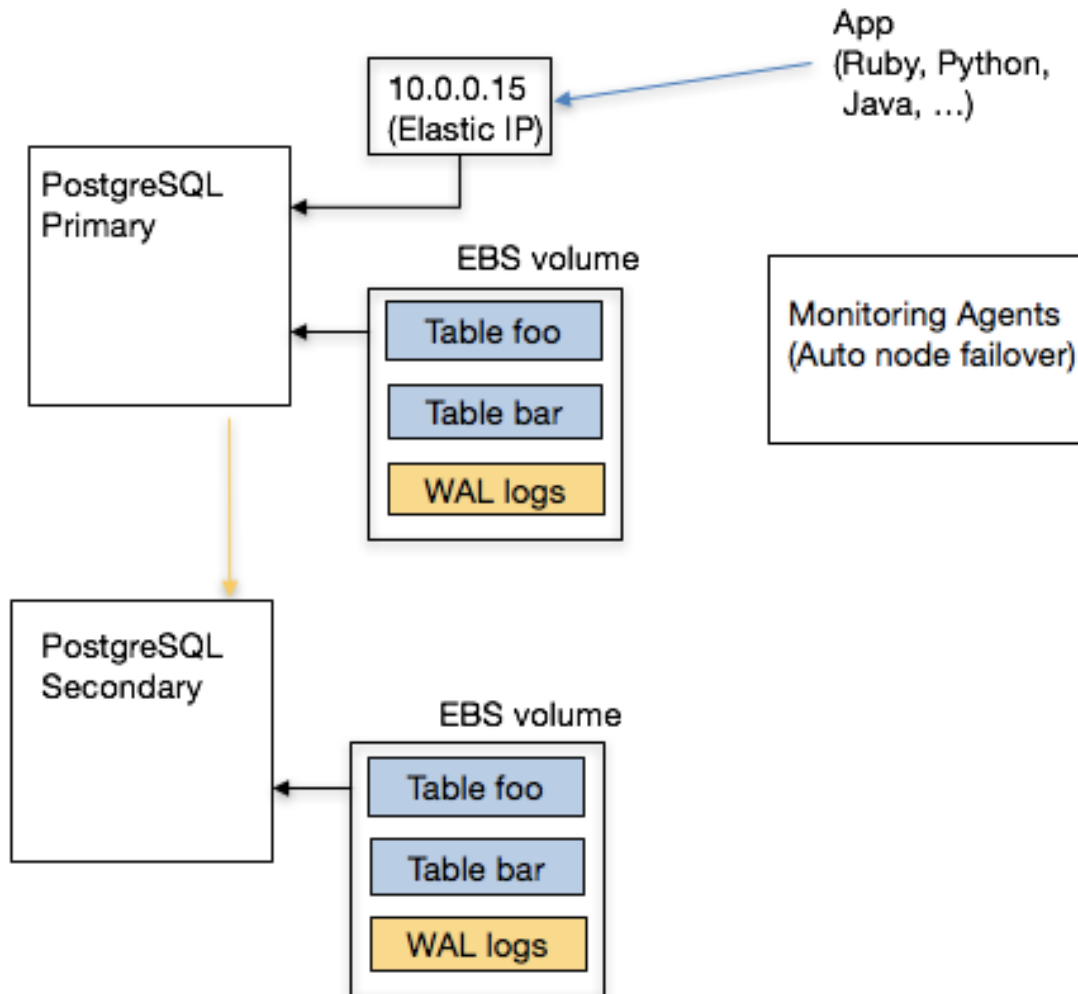
Having a powerful SDN stack also helps with consistency. When the monitoring agent decides to failover the primary node, it can ask the networking layer to no longer send traffic to the old primary. This is also known as fencing – making sure that you disabled the old primary for all clients so that your data doesn’t diverge.

3. Vertical Scalability

Another feature request that we hear from customers is vertical scalability. For example, you may want to instantly scale up CPU and memory resources on your database before you scale out. AWS’ Elastic Block Storage (EBS) enables you to quickly detach from one EC2 instance and attach to another instance that has more resources.

EBS also provides security and availability benefits. When an EC2 instance observes a temporary failure, one for example requires an instance restart, you don’t have any guarantees on whether you will get the same instance back

(or someone else will). With an EBS volume, you can encrypt the data at rest and securely wipe out an EBS volume before terminating it.



One known drawback to attaching storage over the network, instead of using your local storage, is performance. If your working set doesn't fit into memory, your database will pay the penalty of constantly fetching data over the network. Further, as your business grows, your expectations from the underlying database will continue to increase. In that case, you will need your database to scale out.

4. Horizontal Scalability

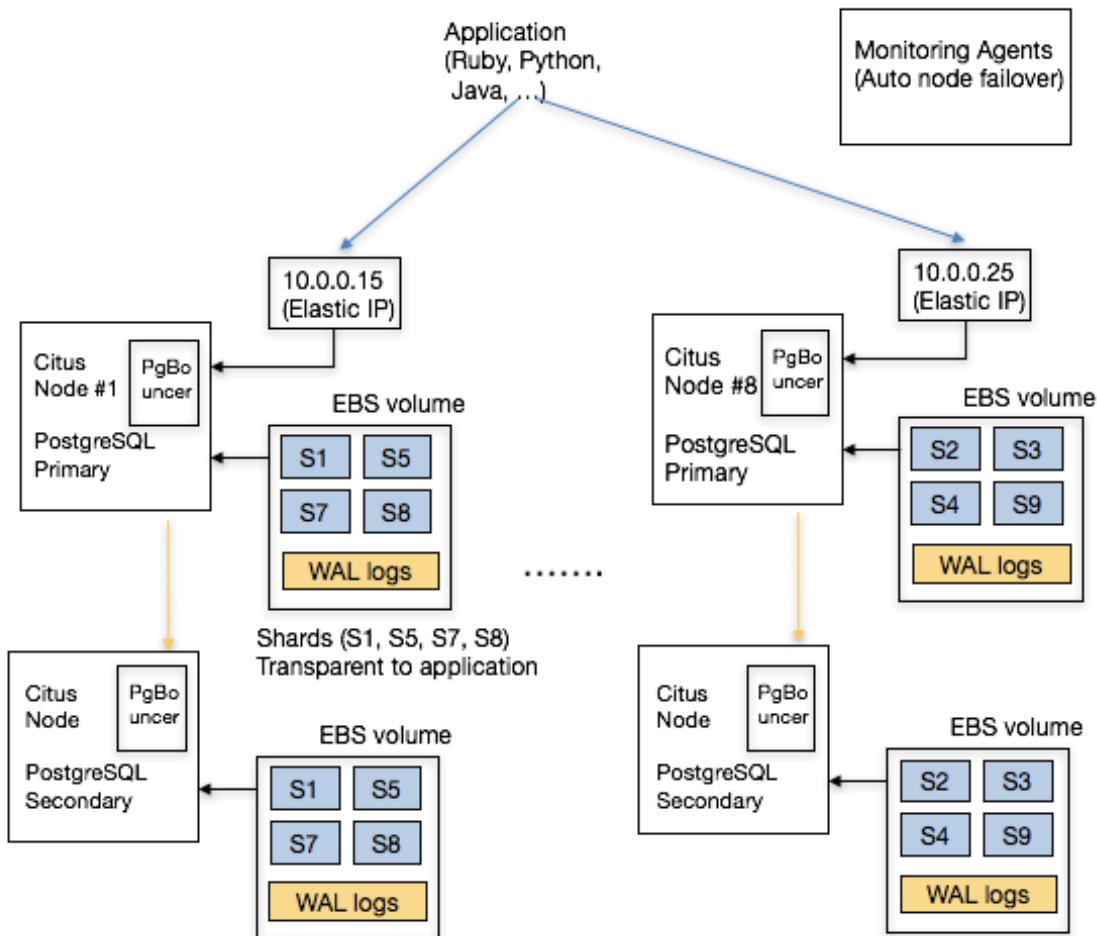
Deploying a distributed RDBMS into production requires a good understanding of both relational databases and distributed systems. This knowledge becomes important in two ways:

1. Data modeling: When you're modeling your data for a distributed environment, you need to pick the right sharding keys for your tables for example.
2. Operational: You need to set up multiple instances and the networking layer between them. More importantly, you need to have an understanding of load management and networking so that you can take the right action.

Running your database on the cloud removes significant operational burden associated with deploying a distributed RDBMS. In particular, Citus Cloud can differentiate between temporary and permanent failures by tracking historical

networking data. It can also deprovision faulty instances and start up new ones in their place, without any involvement from the user. Further, if your database needs to more power, you can elastically add more machines to your cluster.

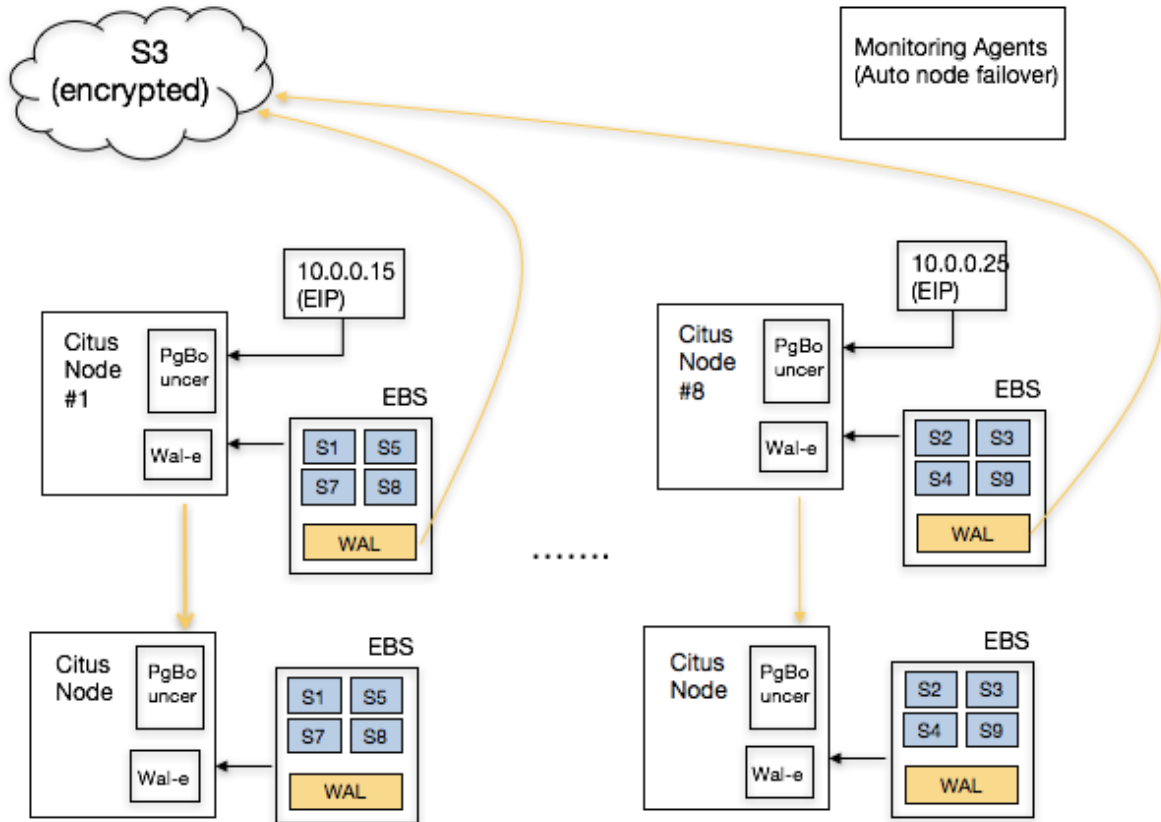
Other operational benefits show themselves with high read/write throughputs. For example, you may need your application to talk to multiple endpoints instead of just one. In this case, Citus Cloud can set up DNS in front of your database instances and configure [PgBouncer](#) to make sure that your distributed database stays within reasonable connection limits. The following shows a simplified architecture diagram.



5. Automatic Backups for Disaster Recovery

If your relational database powers your core business, and acts as your system of record, you need to take regular backups and store those backups in a highly durable location. This way, when the unexpected happens, you can resume your business and not lose it. Therefore, an integrated solution for disaster recovery is among the top feature requests prospective Citus customers ask for.

The thing about database backups is that they are hard to get right. Fortunately, PostgreSQL has a rich ecosystem and comes with open source technologies for automatic backups. For example, [wal-e](#) encrypts and continuously archives your data to a durable storage service, such as S3. In other words, wal-e make sure that your backup solution does the right thing, and cloud service providers make sure that your backups don't get lost.



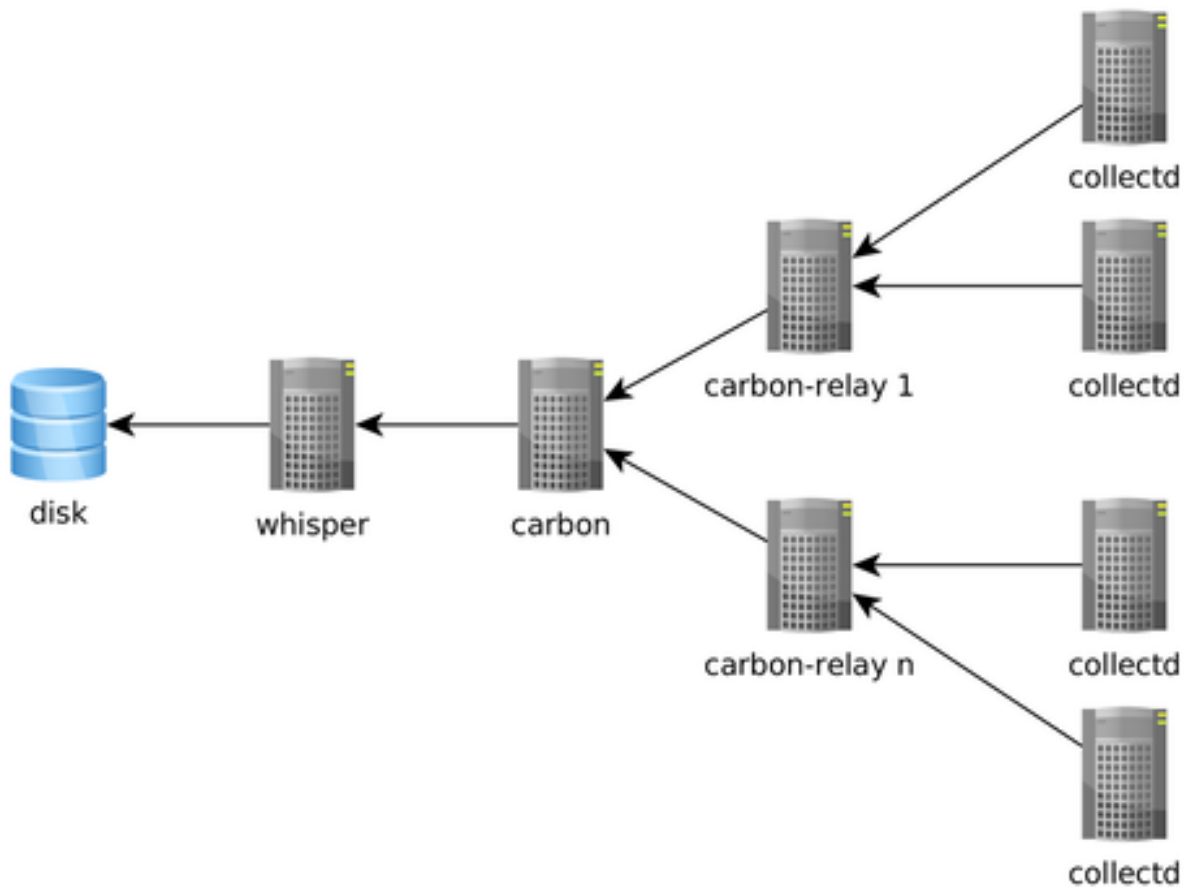
For Citus backups, wal-e helps with part of the picture. Taking distributed database backups are even harder. In this case, it helps when the author of wal-e also happens to be part of the Citus Cloud team. This way, when you deploy a Citus Cloud formation, we can automatically set up regular backups for your distributed database.

6. Monitoring, Alerts, and Logging

An important part of running a relational database in production includes monitoring your database, alerting on anomalies, and setting up a unified logging infrastructure. This seems simple at first, but setting up this infrastructure usually takes time to set up and operate.

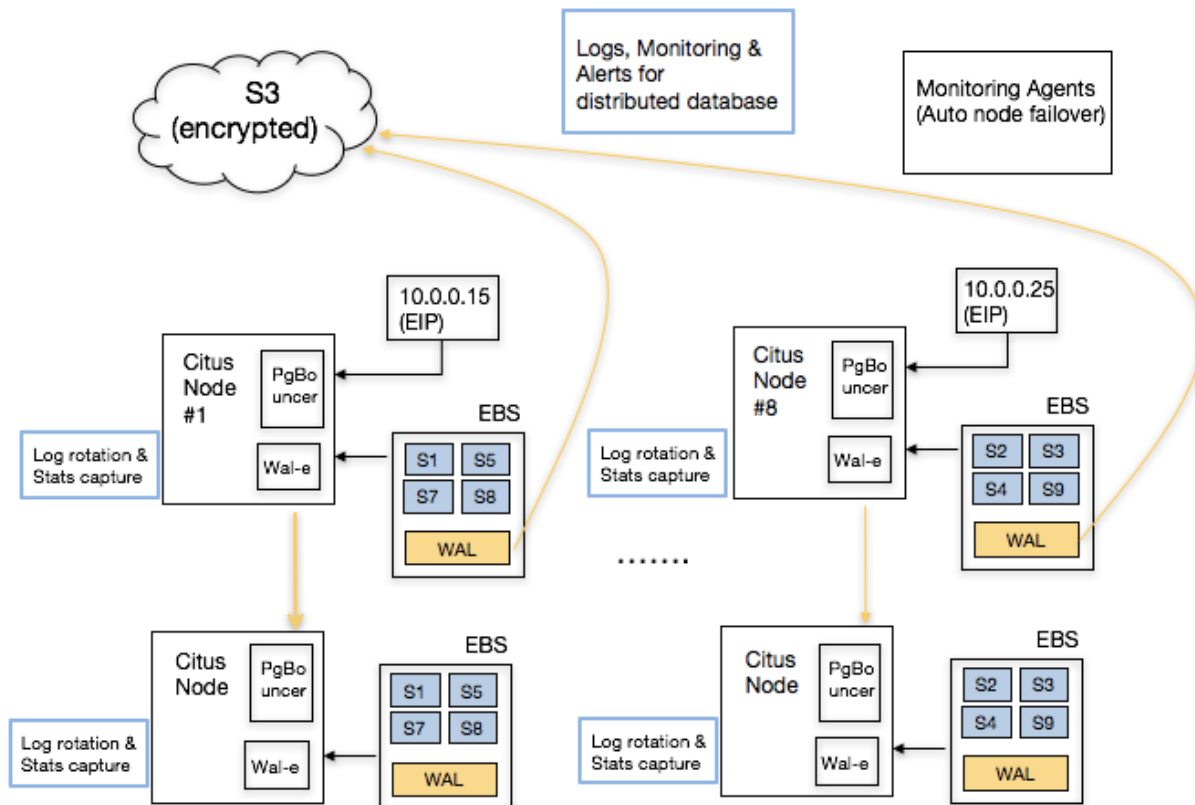
At a minimum, you need to decide on the type of metrics you'd like to track. Some common ones include hardware metrics (such as CPU, memory, network), OS level statistics (VM, RSS, page faults), PostgreSQL metrics (pg_stat views, table and index hit ratios), and active and long running queries.

Once you decide on these metrics, you need to have a daemon that periodically collects these metrics, a solution to serialize and store these metrics, and a visualization layer to display them. A common stack that accomplishes this can run collectd on database instances, Graphite to store them, and Grafana to display them.



If you're thinking of setting up a monitoring infrastructure for PostgreSQL, two good resources to read are Compose's blog post on [monitoring PostgreSQL](#) and Sebastien's presentation at PGConf.EU on [Watching Elephants](#). It's also worth looking at projects that take a more PostgreSQL centric approach to monitoring, such as [PgBadger](#) and [pganalyze](#).

Naturally, setting up a monitoring and logging infrastructure for a distributed RDBMS requires more effort than a single node one. In this case, we were lucky to have `pganalyze`'s author as part of the Citrus Cloud team. We also only needed to provide this only on the AWS platform and that helped us deliver production-grade monitoring much simpler.



7. Auto-configure, patch, and upgrade

Most PostgreSQL power users have three more questions when it comes to running their database in production: configuration, deploying security and bug fix patches, and upgrading a database to a new version – preferably with no downtime.

If you have a solid DBA, they are already turning these knobs for you and deploying new versions without you knowing about it. If you don't, [The Accidental DBA](#) tutorial provides a good introduction on these topics. That said, you will need to keep on learning more as your database's scale increases. And with bigger scale, tuning the right settings in `postgresql.conf`, deploying patches, and upgrading to newer versions will become more important and require deeper database know-how.

Providing a database on the cloud significantly helps deploying patches and new versions – since Citus Cloud has hundreds of these machines running in production, we can automate how to deploy new versions in the way they should be. Citus Cloud can also override certain `postgresql.conf` fields by examining the underlying hardware configuration and workload characteristics.

Conclusion

When we started Citus, we used to think that databases were about databases and we just needed to scale out the RDBMS. After talking to hundreds of customers, we realized that databases were also about native integration with applications, use cases, and operations.

At Citus, as we provided better integration with PostgreSQL drivers and tools, and focused on use cases, we started hearing more questions on the operational components. The seven questions above became part of everyday conversation.

And answering these questions without offering costly services and training work was hard. When a prospective customer asked us about how to handle failover without changing IP addresses, take automatic backups, integrate with monitoring and logging tools, upgrade their PostgreSQL version with no downtime, or elastically scale out their cluster by adding more machines, we'd tell them about the work involved. In fact, there were calls where we quoted \$300K for the services work, and never heard from that user again.

That's the really exciting part about Citrus Cloud. These days, when we hear the same questions, we smile and ask our users to simply click a button. Behind the covers, Citrus Cloud deploys a production grade distributed database, one that natively integrates with PostgreSQL. What was once only accessible to large enterprises with solutions such as Oracle RAC, is now becoming accessible to everyone with open source technologies like PostgreSQL and Citrus, and the cloud.

Postgres Parallel Indexing in Citrus

(Copy of [original publication](#))

Indexes are an essential tool for optimizing database performance and are becoming ever more important with big data. However, as the volume of data increases, index maintenance often becomes a write bottleneck, especially for [advanced index types](#) which use a lot of CPU time for every row that gets written. Index creation may also become prohibitively expensive as it may take hours or even days to build a new index on terabytes of data in PostgreSQL. Citrus makes creating and maintaining indexes that much faster through parallelization.

Citrus can be used to distribute PostgreSQL tables across many machines. One of the many advantages of Citrus is that you can keep adding more machines with more CPUs such that you can keep increasing your write capacity even if indexes are becoming the bottleneck. Citrus allows `CREATE INDEX` to be performed in a massively parallel fashion, allowing fast index creation on large tables. Moreover, the [COPY command](#) can write multiple rows in parallel when used on a distributed table, which greatly improves performance for use-cases which can use bulk ingestion (e.g. sensor data, click streams, telemetry).

To show the benefits of parallel indexing, we'll walk through a small example of indexing ~200k rows containing large JSON objects from the [GitHub archive](#). To run the examples, we set up a formation using [Citrus Cloud](#) consisting of four worker nodes with four cores each, running PostgreSQL 9.6.

You can download the sample data by running the following commands:

```
wget http://examples.citusdata.com/github_archive/github_events-2015-01-01-{0..24}.
→csv.gz
gzip -d github_events-*.gz
```

Next let's create the table for the GitHub events once as a regular PostgreSQL table and then distribute it across the four nodes:

```
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
);

-- (distributed table only) Shard the table by repo_id
SELECT create_distributed_table('github_events', 'repo_id');
```

```
-- Initial data load: 218934 events from 2015-01-01
\COPY github_events FROM PROGRAM 'cat github_events-*.csv' WITH (FORMAT CSV)
```

Each event in the GitHub data set has a detailed payload object in JSON format. Building a GIN index on the payload gives us the ability to quickly perform fine-grained searches on events, such as finding commits from a specific author. However, building such an index can be very expensive. Fortunately, parallel indexing makes this a lot faster by using all cores at the same time and building many smaller indexes:

```
CREATE INDEX github_events_payload_idx ON github_events USING GIN (payload);
```

	Regular table	Distributed table	Speedup
CREATE INDEX on 219k rows	33.2s	2.6s	13x

To test how well this scales we took the opportunity to run our test multiple times. Interestingly, parallel CREATE INDEX exhibits super-linear speedups giving >16x speedup despite having only 16 cores. This is likely due to the fact that inserting into one big index is less efficient than inserting into a small, per-shard index (following $O(\log N)$ for N rows), which gives an additional performance benefit to sharding.

	Regular table	Distributed table	Speedup
CREATE INDEX on 438k rows	55.9s	3.2s	17x
CREATE INDEX on 876k rows	110.9s	5.0s	22x
CREATE INDEX on 1.8M rows	218.2s	8.9s	25x

Once the index is created, the COPY command also takes advantage of parallel indexing. Internally, COPY sends a large number of rows over multiple connections to different workers asynchronously which then store and index the rows in parallel. This allows for much faster load times than a single PostgreSQL process could achieve. How much speedup depends on the data distribution. If all data goes to a single shard, performance will be very similar to PostgreSQL.

```
\COPY github_events FROM PROGRAM 'cat github_events-*.csv' WITH (FORMAT CSV)
```

	Regular table	Distributed table	Speedup
COPY 219k rows no index	18.9s	12.4s	1.5x
COPY 219k rows with GIN	49.3s	12.9s	3.8x

Finally, it's worth measuring the effect that the index has on query time. We try two different queries, one across all repos and one with a specific repo_id filter. This distinction is relevant to Citrus because the github_events table is sharded by repo_id. A query with a specific repo_id filter goes to a single shard, whereas the other query is parallelised across all shards.

```
-- Get all commits by test@gmail.com from all repos
SELECT repo_id, jsonb_array_elements(payload->'commits')
FROM github_events
WHERE event_type = 'PushEvent' AND
      payload @> '{"commits":[{"author":{"email":"test@gmail.com"}}]}';

-- Get all commits by test@gmail.com from a single repo
SELECT repo_id, jsonb_array_elements(payload->'commits')
FROM github_events
WHERE event_type = 'PushEvent' AND
      payload @> '{"commits":[{"author":{"email":"test@gmail.com"}}]}' AND
      repo_id = 17330407;
```

On 219k rows, this gives us the query times below. Times marked with * are of queries that are executed in parallel by Citus. Parallelisation creates some fixed overhead, but also allows for more heavy lifting, which is why it can either be much faster or a bit slower than queries on a regular table.

	Regular table	Distributed table
SELECT no indexes, all repos	900ms	68ms*
SELECT with GIN on payload, all repos	2ms	11ms*
SELECT no indexes, single repo	900ms	28ms
SELECT with indexes, single repo	2ms	2ms

Indexes in PostgreSQL can dramatically reduce query times, but at the same time dramatically slow down writes. Citus gives you the possibility of scaling out your cluster to get good performance on both sides of the pipeline. A particular sweet spot for Citus is parallel ingestion and single-shard queries, which gives querying performance that is better than regular PostgreSQL, but with much higher and more scalable write throughput.

Real-time Event Aggregation at Scale Using Postgres with Citus

(Copy of [original publication](#))

Citus is commonly used to scale out event data pipelines on top of PostgreSQL. Its ability to transparently shard data and parallelise queries over many machines makes it possible to have real-time responsiveness even with terabytes of data. Users with very high data volumes often store pre-aggregated data to avoid the cost of processing raw data at run-time. For large datasets, querying pre-computed aggregation tables can be orders of magnitude faster than querying the facts table on demand.

To create aggregations for distributed tables, the latest version of Citus supports the `INSERT .. SELECT` syntax for tables that use the same distribution column. Citus automatically ‘co-locates’ the shards of distributed tables such that the same distribution column value is always placed on the same worker node, which allows us to transfer data between tables as long as the distribution column value is preserved. A common way of taking advantage of co-location is to follow the [multi-tenant data model](#) and shard all tables by `tenant_id` or `customer_id`. Even without that model, as long as your tables share the same distribution column, you can leverage the `INSERT .. SELECT` syntax.

`INSERT .. SELECT` queries that can be pushed down to the workers are supported, which excludes some SQL functionality such as limits, and unions. Since the result will be inserted into a co-located shard in the destination table, we need to make sure that the distribution column (e.g. `tenant_id`) is preserved in the aggregation and is included in joins. `INSERT .. SELECT` commands on distributed tables will usually look like:

```
INSERT INTO aggregation_table (tenant_id, ...)
SELECT tenant_id, ... FROM facts_table ...
```

Now let’s walk through the steps of creating aggregations for a typical example of high-volume data: page views. We set up a [Citus Cloud](#) formation consisting of 4 workers with 4 cores each, and create a distributed `facts` table with several indexes:

```
CREATE TABLE page_views (
    tenant_id int,
    page_id int,
    host_ip inet,
    view_time timestamp default now()
);
CREATE INDEX view_tenant_idx ON page_views (tenant_id);
CREATE INDEX view_time_idx ON page_views USING BRIN (view_time);
SELECT create_distributed_table('page_views', 'tenant_id');
```

Next, we generate 100 million rows of fake data (takes a few minutes) and load it into the database:

```
\COPY (SELECT s % 307, (random()*5000)::int, '203.0.113.' || (s % 251), now() +
↪random() * interval '60 seconds' FROM generate_series(1,100000000) s) TO '/tmp/
↪views.csv' WITH CSV

\COPY page_views FROM '/tmp/views.csv' WITH CSV
```

We can now perform aggregations at run-time by performing a SQL query against the facts table:

```
-- Most views in the past week
SELECT page_id, count(*) AS view_count
FROM page_views
WHERE tenant_id = 5 AND view_time >= date '2016-11-23'
GROUP BY tenant_id, page_id
ORDER BY view_count DESC LIMIT 3;
page_id | view_count
-----+-----
    2375 |          99
    4538 |          95
    1417 |          93
(3 rows)

Time: 269.125 ms
```

However, we can do *much* better by creating a pre-computed aggregation, which we also distribute by tenant_id. Citrus automatically co-locates the table with the page_views table:

```
CREATE TABLE daily_page_views (
    tenant_id int,
    day date,
    page_id int,
    view_count bigint,
    primary key (tenant_id, day, page_id)
);

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

We can now populate the aggregation using a simple INSERT..SELECT command, which is parallelised across the cores in our workers, processing around *10 million events per second* and generating 1.7 million aggregates:

```
INSERT INTO daily_page_views (tenant_id, day, page_id, view_count)
SELECT tenant_id, view_time::date AS day, page_id, count(*) AS view_count
FROM page_views
GROUP BY tenant_id, view_time::date, page_id;

INSERT 0 1690649

Time: 10649.870 ms
```

After creating the aggregation, we can get the results from the aggregation table in a fraction of the query time:

```
-- Most views in the past week
SELECT page_id, view_count
FROM daily_page_views
WHERE tenant_id = 5 AND day >= date '2016-11-23'
ORDER BY view_count DESC LIMIT 3;
page_id | view_count
```

```

-----+-----
2375 |      99
4538 |      95
1417 |      93
(3 rows)

Time: 4.528 ms

```

We typically want to keep aggregations up-to-date, even as the current day progresses. We can achieve this by expanding our original command to only consider new rows and updating existing rows to consider the new data using **ON CONFLICT**. If we insert data for a primary key (`tenant_id`, `day`, `page_id`) that already exists in the aggregation table, then the count will be added instead.

```

INSERT INTO page_views VALUES (5, 10, '203.0.113.1');

INSERT INTO daily_page_views (tenant_id, day, page_id, view_count)
  SELECT tenant_id, view_time::date AS day, page_id, count(*) AS view_count
  FROM page_views
 WHERE view_time >= '2016-11-23 23:00:00' AND view_time < '2016-11-24 00:00:00'
 GROUP BY tenant_id, view_time::date, page_id
 ON CONFLICT (tenant_id, day, page_id) DO UPDATE SET
   view_count = daily_page_views.view_count + EXCLUDED.view_count;

INSERT 0 1

Time: 2787.081 ms

```

To regularly update the aggregation, we need to keep track of which rows in the facts table have already been processed as to avoid counting them more than once. A basic approach is to aggregate up to the current time, store the timestamp in a table, and continue from that timestamp on the next run. We do need to be careful that there may be in-flight requests with a lower timestamp, which is especially true when using bulk ingestion through COPY. We therefore roll up to a timestamp that lies slightly in the past, with the assumption that all requests that started before then have finished by now. We can easily codify this logic into a PL/pgSQL function:

```

CREATE TABLE aggregations (name regclass primary key, last_update timestamp);
INSERT INTO aggregations VALUES ('daily_page_views', now());

CREATE OR REPLACE FUNCTION compute_daily_view_counts()
  RETURNS void LANGUAGE plpgsql AS $function$
  DECLARE
    start_time timestamp;
    end_time timestamp := now() - interval '1 minute'; -- exclude in-flight requests
  BEGIN
    SELECT last_update INTO start_time FROM aggregations WHERE name = 'daily_page_views'
    ↪ '::regclass;
    UPDATE aggregations SET last_update = end_time WHERE name = 'daily_page_views'::
    ↪ regclass;

    SET LOCAL citus.all_modifications_commutative TO on; -- for on-premises,
    ↪ replication factor >1 only

    EXECUTE $$
      INSERT INTO daily_page_views (tenant_id, day, page_id, view_count)
        SELECT tenant_id, view_time::date AS day, page_id, count(*) AS view_count
        FROM page_views

```

```

WHERE view_time >= $1 AND view_time < $2
GROUP BY tenant_id, view_time::date, page_id
ON CONFLICT (tenant_id, day, page_id) DO UPDATE SET
view_count = daily_page_views.view_count + EXCLUDED.view_count$$
USING start_time, end_time;
END;
$function$;

```

After creating the function, we can periodically call `SELECT compute_daily_view_counts()` to continuously update the aggregation with 1-2 minutes delay. More advanced approaches can bring down this delay to a few seconds.

A few caveats to note:

- In this example, we used a single, database-generated time column, but it's generally better to distinguish between the time at which the event happened at the source and the database-generated ingestion time used to keep track of whether an event was already processed.
- When running Citus on-premises with built-in replication, we recommend you set `citus.all_modifications_commutative` to on before any `INSERT..SELECT` command, since Citus otherwise locks the source tables to avoid inconsistencies between replicas. *On Citus Cloud this is a non-issue as we leverage Postgres streaming replication.*

You might be wondering why we used a `page_id` in the examples instead of something more meaningful like a URL. Are we trying to dodge the overhead of storing URLs for every page view to make our numbers look better? We certainly are! With Citus you can often avoid the cost of denormalization that you would pay in distributed databases that don't support joins. You can simply put the static details of a page inside another table and perform a join:

```

CREATE TABLE pages (
    tenant_id int,
    page_id int,
    url text,
    language varchar(2),
    primary key (tenant_id, page_id)
);

SELECT create_distributed_table('pages', 'tenant_id');

... insert pages ...

-- Most views in the past week
SELECT url, view_count
FROM daily_page_views JOIN pages USING (tenant_id, page_id)
WHERE tenant_id = 5 AND day >= date '2016-11-23'
ORDER BY view_count DESC LIMIT 3;
  url      | view_count
-----+-----
/home      |          99
/contact   |          95
/product   |          93
(3 rows)

Time: 7.042 ms

```

You can also perform joins in the `INSERT..SELECT` command, allowing you to create more detailed aggregations, e.g. by language.

Distributed aggregation adds another tool to Citus' broad toolchest in dealing with big data problems. With parallel `INSERT .. SELECT`, parallel indexing, parallel querying, scaling write throughput through [Citus MX](#), and many other

features, Citrus can not only horizontally scale your multi-tenant database, but can also unify many different parts of your data pipeline into one platform.

How Distributed Outer Joins on PostgreSQL with Citrus Work

(Copy of [original publication](#))

SQL is a very powerful language for analyzing and reporting against data. At the core of SQL is the idea of joins and how you combine various tables together. One such type of join: outer joins are useful when we need to retain rows, even if it has no match on the other side.

And while the most common type of join, inner join, against tables A and B would bring only the tuples that have a match for both A and B, outer joins give us the ability to bring together from say all of table A even if they don't have a corresponding match in table B. For example, let's say you keep customers in one table and purchases in another table. When you want to see all purchases of customers, you may want to see all customers in the result even if they did not do any purchases yet. Then, you need an outer join. Within this post we'll analyze a bit on what outer joins are, and then how we support them in a distributed fashion on Citrus.

Let's say we have two tables, customer and purchase:

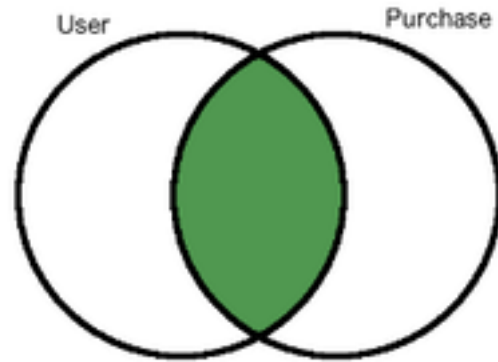
```
customer table:
customer_id |      name
-----+-----
          1 | Corra Ignacio
          3 | Warren Brooklyn
          2 | Jalda Francis

purchase table:
purchase_id | customer_id | category |      comment
-----+-----+-----+-----
        1000 |           1 | books    | Nice to Have!
        1001 |           1 | chairs   | Comfortable
        1002 |           2 | books    | Good Read, cheap price
        1003 |          -1 | hardware | Not very cheap
        1004 |          -1 | laptops  | Good laptop but expensive...
```

The following queries and results help clarifying the inner and outer join behaviors:

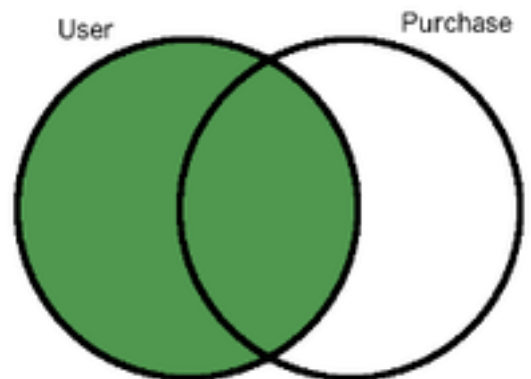
```
SELECT customer.name, purchase.comment
FROM customer JOIN purchase ON customer.customer_id = purchase.customer_id
ORDER BY purchase.comment;
```

name	comment
Corra Ignacio	Comfortable
Jalda Francis	Good Read, cheap price
Corra Ignacio	Nice to Have!



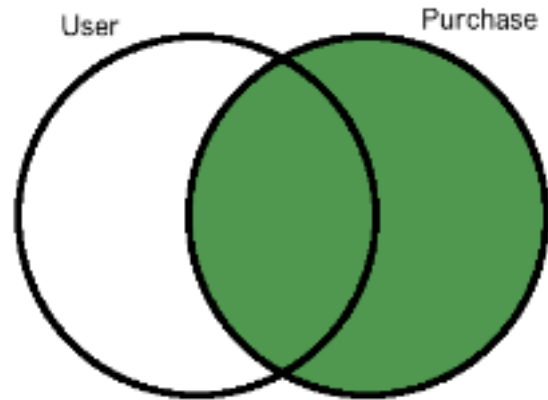
```
SELECT customer.name, purchase.comment
FROM customer INNER JOIN purchase ON customer.customer_id = purchase.customer_id
ORDER BY purchase.comment;
```

name	comment
Corra Ignacio	Comfortable
Jalda Francis	Good Read, cheap price
Corra Ignacio	Nice to Have!



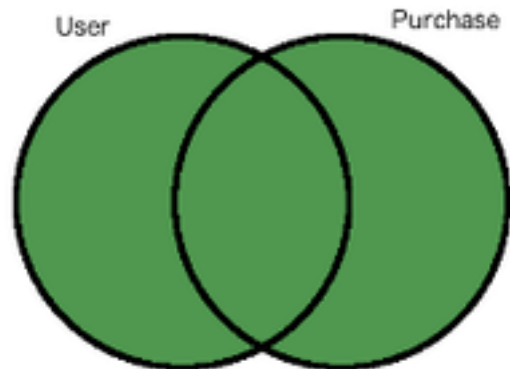
```
SELECT customer.name, purchase.comment
FROM customer LEFT JOIN purchase ON customer.customer_id = purchase.customer_id
ORDER BY purchase.comment;
```

name	comment
Corra Ignacio	Comfortable
Jalda Francis	Good Read, cheap price
Corra Ignacio	Nice to Have!
Warren Brooklyn	



```
SELECT customer.name, purchase.comment
FROM customer RIGHT JOIN purchase ON customer.customer_id = purchase.customer_id
ORDER BY purchase.comment;
```

name	comment
Corra Ignacio	Comfortable
Jalda Francis	Good Read, cheap price
	Good laptop but expensive...
Corra Ignacio	Nice to Have!
	Not very cheap



```
SELECT customer.name, purchase.comment
FROM customer FULL JOIN purchase ON customer.customer_id = purchase.customer_id
ORDER BY purchase.comment;
```

name	comment
Corra Ignacio	Comfortable
Jalda Francis	Good Read, cheap price
	Good laptop but expensive...
Corra Ignacio	Nice to Have!
	Not very cheap
Warren Brooklyn	

Distributed Outer Joins with Citrus

The Citrus extension allows PostgreSQL to distribute big tables into smaller fragments called “shards” and performing outer joins on these distributed tables becomes a bit more challenging, since the union of outer joins between individual shards does not always give the correct result. Currently, Citrus support distributed outer joins under some criteria:

- Outer joins should be between distributed(sharded) tables only, i.e. it is not possible to outer join a sharded table with a regular PostgreSQL table.
- Join criteria should be on [partition columns](#) of the distributed tables.
- The query should join the distributed tables on the equality of partition columns (table1.a = table2.a)
- Shards of the distributed table should match one to one, i.e. each shard of table A should overlap with one and only one shard from table B.

For example lets assume we 3 hash distributed tables X, Y and Z and let X and Y have 4 shards while Z has 8 shards.

```
CREATE TABLE user (user_id int, name text);
SELECT create_distributed_table('user', 'user_id');

CREATE TABLE purchase (user_id int, amount int);
SELECT create_distributed_table('purchase', 'user_id');

CREATE TABLE comment (user_id int, comment text, rating int);
SELECT create_distributed_table('comment', 'user_id');
```

The following query would work since distributed tables user and purchase have the same number of shards and the join criteria is equality of partition columns:

```
SELECT * FROM user OUTER JOIN purchase ON user.user_id = purchase.user_id;
```

The following queries are not supported out of the box:

```
-- user and comment tables doesn't have the same number of shards:
SELECT * FROM user OUTER JOIN comment ON user.user_id = comment.user_id;

-- join condition is not on the partition columns:
SELECT * FROM user OUTER JOIN purchase ON user.user_id = purchase.amount;

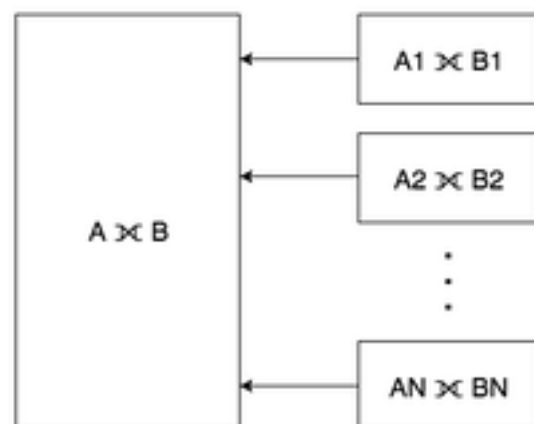
-- join condition is not equality:
SELECT * FROM user OUTER JOIN purchase ON user.user_id < purchase.user_id;
```

How Citrus Processes OUTER JOINS When one-to-one matching between shards exists, then performing an outer join on large tables is equivalent to combining outer join results of corresponding shards.

Let’s look at how Citrus handles an outer join query:

```
SELECT table1.a, table1.b AS b1, table2.
↪b AS b2, table3.b AS b3, table4.b AS b4
FROM table1
FULL JOIN table2 ON table1.a = table2.a
FULL JOIN table3 ON table1.a = table3.a
FULL JOIN table4 ON table1.a = table4.a;
```

First, the query goes through the standard PostgreSQL planner and Citrus uses this plan to generate a distributed plan where various checks about Citrus’ support of the query are



performed. Then individual queries that will go to workers for distributed table fragments are generated.

```
SELECT table1.a, table1.b AS b1, table2.
↪b AS b2, table3.b AS b3, table4.b AS b4
FROM (((table1_102359 table1
FULL JOIN table2_
↪102363 table2 ON ((table1.a = table2.a)))
FULL JOIN table3_
↪102367 table3 ON ((table1.a = table3.a)))
FULL JOIN table4_102371 table4_
↪ON ((table1.a = table4.a))) WHERE true
```

```
SELECT table1.a, table1.b AS b1, table2.
↪b AS b2, table3.b AS b3, table4.b AS b4
FROM (((table1_102360 table1
FULL JOIN table2_
↪102364 table2 ON ((table1.a = table2.a)))
FULL JOIN table3_
↪102368 table3 ON ((table1.a = table3.a)))
FULL JOIN table4_102372 table4_
↪ON ((table1.a = table4.a))) WHERE true
```

```
SELECT table1.a, table1.b AS b1, table2.
↪b AS b2, table3.b AS b3, table4.b AS b4
FROM (((table1_102361 table1
FULL JOIN table2_
↪102365 table2 ON ((table1.a = table2.a)))
FULL JOIN table3_
↪102369 table3 ON ((table1.a = table3.a)))
FULL JOIN table4_102373 table4_
↪ON ((table1.a = table4.a))) WHERE true
```

```
SELECT table1.a, table1.b AS b1, table2.
↪b AS b2, table3.b AS b3, table4.b AS b4
FROM (((table1_102362 table1
FULL JOIN table2_
↪102366 table2 ON ((table1.a = table2.a)))
FULL JOIN table3_
↪102370 table3 ON ((table1.a = table3.a)))
FULL JOIN table4_102374 table4_
↪ON ((table1.a = table4.a))) WHERE true
```

The resulting queries may seem complex at first but you can see that they are actually the same with the original query with just the table names are a bit different. This is because Citrus stores the data in standard postgres tables called shards with the name as `_`. With 1-1 matching of shards, the distributed outer join is equivalent to the union of all outer joins of individual matching shards. In many cases you don't even have to think about this as Citrus simply takes care of you. If you're sharding on some shared id, as is common in certain [use cases](#), then Citrus will do the join on the appropriate node without any inter-worker communication.

We hope you found the insight into how we perform distributed outer joins valuable. If you're curious about trying Citrus or learning how more works we encourage you to join the conversation with us on Slack.

Designing your SaaS Database for Scale with Postgres

(Copy of [original publication](#))

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

With smaller amounts of data (10s of GB), it's easy to throw more hardware at the problem and scale up your database. As these tables grow however, you need to think about ways to scale your multi-tenant database across dozens or hundreds of machines.

After our blog post on [sharding a multi-tenant app with Postgres](#), we received a number of questions on architectural patterns for multi-tenant databases and when to use which. At a high level, developers have three options:

1. Create one database per tenant
2. Create one schema per tenant
3. Have all tenants share the same table(s)

The option you pick has implications on scalability, how you handle data that varies across tenants, isolation, and ease-of-maintenance. And these implications have been discussed in detail across many [StackOverflow questions](#) and database articles. So, what is the best solution?

In practice, each of the three design options -with enough effort- can address questions around scale, data that varies across tenants, and isolation. The decision depends on the *primary* dimension you're building/optimizing for. The tldr:

- If you're building for scale: Have all tenants share the same table(s)
- If you're building for isolation: Create one database per tenant

In this article, we'll focus on the scaling dimension, as we found that more users who talked to us had questions in that area. (We also intend to describe considerations around isolation in a follow-up blog post.)

To expand on this further, if you're planning to have 5 or 50 tenants in your B2B application, and your database is running into scalability issues, then you can create and maintain a separate database for each tenant. If however you plan to have thousands of tenants, then sharding your tables on a `tenant_id/account_id` column will help you scale in a much better way.

Common benefits of having all tenants share the same database are:

Resource pooling (reduced cost): If you create a separate database for each tenant, then you need to allocate resources to that database. Further, databases usually make assumptions about resources available to them—for example, PostgreSQL has `shared_buffers`, makes good use of the operating system cache, comes with connection count settings, runs processes in the background, and writes logs and data to disk. If you're running 50 of these databases on a few physical machines, then resource pooling becomes tricky even with today's virtualization tech.

If you have a distributed database that manages all tenants, then you're using your database for what it's designed to do. You could shard your tables on `tenant_id` and easily support 1000s or tens of thousands of tenants.

[Google's F1 paper](#) is a good example that demonstrates a multi-tenant database that scales this way. The paper talks about technical challenges associated with scaling out the Google AdWords platform; and at its core describes a multi-tenant database. The F1 paper also highlights how best to model data to support many tenants/customers in a distributed database.

The data model on the left-hand side follows the relational database model and uses foreign key constraints to ensure data integrity in the database. This strict relational model introduces certain drawbacks in a distributed environment however.

Accounts table (shard 1)

account_id	name	created_at
1	CNN	2016-07-12
5	Comcast	2016-07-19
...
1252	Walmart	2016-08-02

Accounts table (shard 2)

account_id	name	created_at
2	AT&T	2016-07-13
3	Exxon	2016-07-14
...
1253	UPS	2016-08-03

Campaigns table (shard 3)

campaign_id	name	account_id
1202	tv series	1
1204	superbowl	1
...
352042	chocolate	1252

Campaigns table (shard 4)

campaign_id	name	account_id
2742	gas state	3
2743	my phone	2
...
352423	new phone	2

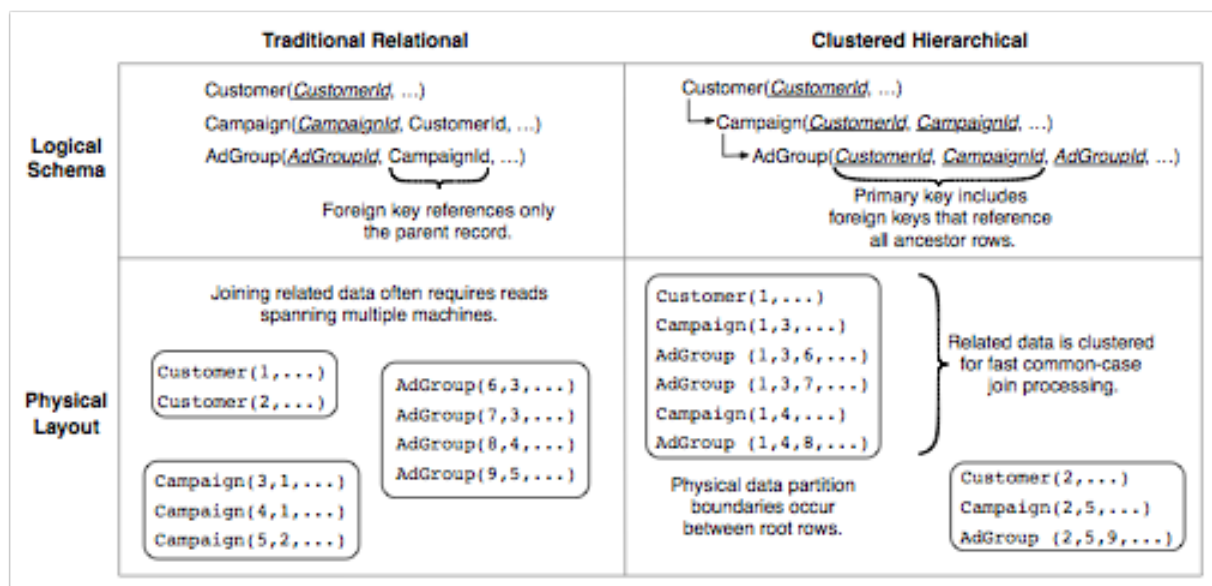


Figure 2: The logical and physical properties of data storage in a traditional normalized relational schema compared with a clustered hierarchical schema used in an F1 database.

In particular, most transactions and joins you perform on your database, and constraints you'd like to enforce across your tables, have a customer/tenant dimension to them. If you shard your tables on their primary key column (in the relational model), then most distributed transactions, joins, and constraints become expensive. Network and machine failures further add to this cost.

The diagram on the right-hand side proposes the hierarchical database model. This model is the one used by F1 and resolves the previously mentioned issues. In its simplest form, you add a `customer_id/tenant_id` column to your tables and shard them on `customer_id`. This ensures that data from the same customer gets **colocated together** – co-location dramatically reduces the cost associated with distributed transactions, joins, and **foreign key constraints**.

Ease of maintenance: Another challenge associated with supporting 100-100K tenants is schema changes (Alter Table) and index creations (Create Index). As your application grows, you will iterate on your database model and make improvements.

If you're following an architecture where each tenant lives in a separate database, then you need to implement an infrastructure that ensures that each schema change either succeeds across all tenants or gets eventually rolled back. For example, what happens when you changed the schema for 5,000 of 10K tenants and observed a failure? How do you handle that?

When you shard your tables for multi-tenancy, then you're having your database do the work for you. The database will either ensure that an Alter Table goes through across all shards, or it will roll it back.

What about data that varies across tenants? Another challenge with scaling to thousands of tenants relates to handling data that varies across tenants. Your multi-tenant application will naturally include a standard database setup with default tables, fields, queries, and relationships that are appropriate to your solution. But different tenants/organizations may have their own unique needs that a rigid, inextensible default data model won't be able to address. For example, one organization may need to track their stores in the US through their zip codes. Another customer in Europe might not care about US zip codes, but may be interested to keep tax ratios for each store.

This used to be an area where having a tenant per database offered the most flexibility, at the cost of extra maintenance work from the developer(s). You could create separate tables or columns per tenant in each database, and manage those differences across time.

If then you wanted to scale your infrastructure to thousands of tenants, you'd create a huge table with many string columns (Value0, Value1, ... Value500). Probably, the best known example of this model is [Salesforce's multi-tenant architecture](#).

campaign_id	name	account_id	V1	V2	V3
1202	tv series	1	null	"Paris"	null
1204	big bang	1	null	94210	0.08
3492	World Cup	93	null	"processed"	"2016-08-02"
352042	Chocolate	1252	8600	"paym.due"	0.08

In this database model, your tables have a preset collection of custom columns, labeled in this image as V1, V2, and V3. Dates and Numbers are stored as strings in a format such that they can be converted to their native types. When you're storing data associated with a particular tenant, you can then use these custom columns and tailor them to each tenant's special needs.

Fortunately, designing your database to account for "flexible" columns became significantly easier with the introduction of semi-structured data types. PostgreSQL has a rich set of semi-structured data types that include `hstore`, `json`, and `jsonb`. You can now represent the previous database schema by simply declaring a `jsonb` column and scale to thousands of tenants.

Of course, these aren't the only design criteria and questions to be aware of. If you shard your database tables, how do you handle isolation or integrate with ORM libraries? What happens if you have a table that you can't easily

campaign_id	name	account_id	payment_info
1202	tv series	1	"location": "Paris"
1204	big bang	1	"zip": 94210, "tax": 0.08
3492	World Cup	93	"status": "processed", "date": "2016-08-02"
352042	Chocolate	1252	"status": "paym.due", "amount": 8600

add a `tenant_id` column? In this article, we focused on building multi-tenant databases with scaling as the primary consideration in mind; and skipped over certain points. If you're looking to learn more about designing multi-tenant databases, see [Multi-tenant Applications](#).

The good news is, databases have advanced quite a bit in the past ten years in accommodating SaaS applications at scale. What was once only available to the likes of Google and Salesforce with significant engineering effort, is now becoming accessible to everyone with open-source technologies such as PostgreSQL and Citrus.

Building a Scalable Postgres Metrics Backend using the Citrus Extension

(Copy of [original publication](#))

From nearly the beginning of the Citrus Cloud service, we've had an internal formation provisioned and managed by the service itself. Dogfooding in this manner brings all the usual benefits such as gaining operational knowledge, customer empathy, and etc.

However, more interesting than yet another blog post going over the importance of dogfooding is the two different ways we're using our Citrus formation. Setting up a distributed table requires a bit more forethought than a normal Postgres table, because the choice of shard column has a big impact on the types of queries and joins you can do with the data.

We're going to look at two cases here: a time-series metrics table and an events table.

Time-Series Metrics

The first is a metrics table that we fill with data from AWS Cloud Watch for all servers and volumes that we manage. We then show graphs of this both for internal and customer-facing dashboards. The table has the following structure and is sharded on `server_id`. The primary key is several columns wide to both serve as a unique constraint preventing duplicate data points and as an index over all the columns we query on.

Table "public.cw_metrics"		
Column	Type	Modifiers
server_id	uuid	not null
aws_id	text	collate C not null
name	text	collate C not null
timestamp	timestamp with time zone	not null
sample_count	bigint	not null
average	double precision	not null
sum	double precision	not null
minimum	double precision	not null
maximum	double precision	not null
unit	text	collate C not null
Indexes:		
"cw_metrics_pkey" PRIMARY KEY, btree (server_id, timestamp, aws_id, name)		

Some sample rows:

```
citrus=> select * from cw_metrics order by timestamp desc limit 2;
-[ RECORD 1 ]+-----
server_id    | f2239a4b-7297-4b66-b9e3-851291760b70
aws_id       | i-723a927805464ac8b
name         | NetworkOut
timestamp    | 2016-07-28 14:13:00-07
sample_count | 5
average      | 127505
sum          | 637525
minimum      | 111888
maximum      | 144385
unit         | Bytes
-[ RECORD 2 ]+-----
server_id    | f2239a4b-7297-4b66-b9e3-851291760b70
aws_id       | i-723a927805464ac8b
name         | NetworkIn
timestamp    | 2016-07-28 14:13:00-07
sample_count | 5
average      | 32930.8
sum          | 164654
minimum      | 18771
maximum      | 46584
unit         | Bytes
```

There are currently only two queries on this table. The first is simply inserting data after periodically fetching data from AWS CloudWatch.

The other gets the data for the graphs that are shown both on the internal admin site and on the customer-facing console and looks like `select ... where server_id in (?,) and name in (?,) and timestamp > now() - '? hours'::interval`. Because Citrus shards are just normal postgres tables, this query is parallelized by going to only the shards necessary for the `server_id` list. Once on each shard, finding the data is very fast because the other two where conditions are covered by the primary key.

The main downside to sharding on `server_id` is expiring old data is a little cumbersome. We have to go to each shard and run a `delete ... where timestamp > '?'`. This can take a while depending on how big a window we're pruning, and it leaves a bloated table that requires vacuuming. A nice alternative is to use standard time-based table partitioning for each of the shards, and then simply drop the old time tables. We haven't done this yet because expiry hasn't been a problem so far, but it's nice the option is there.

Events

The other table is a general event table. We are using a hosted exception tracker to discover problems in production. However we were also sending that service unexceptional exceptions. That is, these were expected errors, such as the failure to ssh into a server that was still booting. Sometimes an increased rate of exceptions in a particular category can indicate a problem even though a normal baseline rate is okay.

However the exception tracker was not the right place for this. It made it harder than necessary to spot real errors, so we moved these events to a distributed Citrus table which looks like this:

Table "public.events"		
Column	Type	Modifiers
id	uuid	not null

```

name          | text                               | not null
created_at    | timestamp with time zone          | not null
data          | jsonb                             |

```

Indexes:

```

"events_pkey" PRIMARY KEY, btree (id)
"events_created_at_idx" brin (created_at)
"events_data_idx" gin (data jsonb_path_ops)
"events_name_idx" btree (name)

```

The `id` column is a randomly generated uuid and the shard key, which gives a roughly equal distribution amongst the shards as events come in. Also because Citrus is just an extension on top of Postgres, we're able to take advantage of the powerful `jsonb` data type with the corresponding `gin` index which gives us very fast lookups on arbitrary keys, and the new `brin` index type.

Here are some example rows from the events table:

```

citrus=> select * from events order by created_at desc limit 2;
-[ RECORD 1 ]-
id          | 9a3dfdbd-c395-40bb-8d25-45ee7c913662
name        | Timeout::Error
created_at  | 2016-07-28 13:18:47.289917-07
data        | {"id": "5747a999-9768-429c-b13c-c7c0947dd950", "class": "Server",
↪ "message": "execution expired"}
-[ RECORD 2 ]-
id          | ba9d6a13-0832-47fb-a849-02f1362c9019
name        | Sequel::DatabaseConnectionError
created_at  | 2016-07-28 12:58:40.506267-07
data        | {"id": "232835ec-31a1-44d0-ae5b-edafb2cf6978", "class": "Timeline",
↪ "message": "PG::ConnectionBad: could not connect to server: Connection_
↪ refused\n\tIs the server running on host \"ec2-52-207-18-20.compute-1.amazonaws.com\"
↪ \" (52.207.18.20) and accepting\n\tTCP/IP connections on port 5432?\n"}

```

This data is currently mostly used to show graphs on the admin dashboard to spot outliers. The query to gather data is for the graphs is

```

SELECT count(*), name, date_trunc('hour', created_at) as hour
FROM events
WHERE created_at > now() - '1 week'::interval
GROUP BY name, hour;

```

And the graphs look like

This clearly shows a time period of something not quite right. Sometimes we've gone into `psql` to look at the `jsonb` to get details if there is a high rate of some particular error to figure out which server is causing it. That is currently a manual process, and perhaps sample json bodies could be put into the UI, but doing the work for that hasn't been worth it yet.

A more exciting project would be to use some machine learning on past time periods to automatically detect outliers. If we ever do that, I'll be sure to put a writeup on the experience on this blog.

Sharding a Multi-Tenant App with Postgres

(Copy of [original publication](#))

Whether you're building marketing analytics, a portal for e-commerce sites, or an application to cater to schools, if you're building an application and your customer is another business then a multi-tenant approach is the norm. The

Recent Events

Name	Last Week	Last Day	Last Hour	Chart
Timeout::Error	1923	437	10	
Sequel::DatabaseConnectionError	7861	38	0	
Actor::EarlyExit	431	32	0	
Errno::ECONNREFUSED	22	0	0	
Aws::S3::Errors::OperationAborted	7	0	0	
Net::SSH::AuthenticationFailed	12	0	0	
Aws::EC2::Errors::IncorrectState	10	0	0	

FormationIndex

same code runs for all customers, but each customer sees their own private data set, *except in some cases of holistic internal reporting*.

Early in your application's life customer data has a simple structure which evolves organically. Typically all information relates to a central customer/user/tenant table. With a smaller amount of data (10's of GB) it's easy to scale the application by throwing more hardware at it, but what happens when you've had enough success and data that you have no longer fits in memory on a single box, or you need more concurrency? You scale out, often painfully.

This scale out model has worked well for the likes of [Google](#) and [Instagram](#), but also doesn't have to be as complicated as you might think. If you're able to model your multi-tenant data in the right way sharding can become much simpler and still give you the power you need from a database including joins, indexing, and more. While Citrus lets you scale out your processing power and memory, how you model your data may determine the ease and flexibility you get from the system. If you're building a multi-tenant SaaS application hopefully the following example highlights how you can plan early for scaling without having to contort too much of your application.

Tenancy

At the core of most non-consumer focused applications tenancy is already built in, whether you realize it or not. As we mentioned above you may have a users table. Let's look at a very basic SaaS schema that highlights this:

```
CREATE TABLE stores (
  id UUID,
  owner_email VARCHAR(255),
  owner_password VARCHAR(255),
  name VARCHAR(255),
  url VARCHAR(255),
  last_login_at TIMESTAMPTZ,
  created_at TIMESTAMPTZ
)

CREATE TABLE products (
  id UUID,
  name VARCHAR(255),
  description TEXT,
  price INTEGER,
  quantity INTEGER,
  store_id UUID,
  created_at TIMESTAMPTZ,
  updated_at TIMESTAMPTZ
)

CREATE TABLE purchases (
  id UUID,
  product_id UUID,
  customer_id UUID,
  store_id UUID,
  price INTEGER,
  purchased_at TIMESTAMPTZ,
)
```

The above schema highlights an *overly simplified* multi-tenant e-commerce site. Say for example someone like an Etsy. And of course there are a number of queries you would run against this:

List the products for a particular store:

```
SELECT id,
       name,
```

```
    price
FROM products
WHERE store_id = 'foo';
```

Or let's say you want to compute how many purchases exist weekly for a given store:

```
SELECT date_trunc('week', purchased_at),
       sum(price * quantity)
FROM purchases,
     stores
WHERE stores.id = products.stores_id
      AND store_id = 'foo'
```

From here you could envision how to give each store its own presence and analytics. Now if we fast-forward a bit and start to look at scaling this out then we have a choice to make on how we'll shard the data. The easiest level to do this at is the tenant level or in this case on store id. With the above data model the largest tables over time are likely to be products and purchases, we could shard on both of these. Though if we choose products or purchases the difficulty lies in the fact that we may want to do queries that focus on some high level item such as store. If we choose store id then all data for a particular store would exist on the same node, this would allow you push down all computations directly to the a single node.

Multi-tenancy and co-location, a perfect pair

Co-locating data within the same physical instance avoids sending data over the network during joins. This can result in much faster operations. With Citus there are a number of ways to move your data around so you can join and query it in a flexible manner, but for this class of multi-tenant SaaS apps it's simple if you can ensure data ends up on the shard. To do this though we need to push down our store id to all of our tables.

The key that makes this all possible is including your `store_id` on all tables. By doing this you can easily shard out all your data so it's located on the same shard. In the above data model we coincidentally had `store_id` on all of our tables, but if it weren't there you could add it. This would put you in a good position to distribute all your data so it's stored on the same nodes. So now lets try sharding our tenants, in this case stores:

```
SELECT create_distributed_table('stores', 'id');
SELECT create_distributed_table('products', 'store_id');
SELECT create_distributed_table('purchases', 'store_id');
```

Now you're all set. *Again, you'll notice that we shard everything by store_id—this allows all queries to be routed to a single Postgres instance.* The same queries as before should work just fine for you as long as you have `store_id` on your query. An example layout of your data now may look something like:

The alternative to colocation is to choose some lower level shard key such as orders or products. This has a trade-off of making joins and querying more difficult because you have to send more data over the network and make sure things work in a distributed way. This lower level key can be useful for consumer focused datasets, if your analytics are always against the entire data set as is often the case in metrics-focused use cases.

In conclusion

It's important to note that different distribution models can have different benefits and trade-offs. In some cases modeling on a lower level entity id such as products or purchases can be the right choice. You gain more parallelism for analytics and trade off simplicity in querying a single store. Either choice of picking a multi-tenant data model or adopt a more *distributed document model* can be made to scale, but each comes with its own trade-offs. If you have the need today to scale out your multi-tenant app then give [Citrus Cloud](#) a try or if you have any questions on which might work best for your situation please don't hesitate to [reach out to us](#). We can help.

Shard 1

Stores

id	name
1	my book store
5	my other store

Products

id	name	store_id
1	foo	1
2	bar	1
3	baz	1

Purchases

id	product_id	store_id	price
1	2	1	1000
2	1	1	1200
3	3	1	1199

Shard 2

Stores

id	name
2	my sock store
6	old things

Products

id	name	store_id
33	new socks	2
34	old socks	6
35	old tie	6

Purchases

id	product_id	store_id	price
102	35	6	600
43	33	2	800

Sharding Postgres with Semi-Structured Data and Its Performance Implications

(Copy of [original publication](#))

If you're looking at Citus it's likely you've outgrown a single node database. In most cases your application is no longer performing as you'd like. In cases where your data is still under 100 GB a single Postgres instance will still work well for you, and is a great choice. At levels beyond that Citus can help, but how you model your data has a major impact on how much performance you're able to get out of the system.

Some applications fit naturally in this scaled out model, but others require changes in your application. The model you choose can determine the queries you'll be able to run in a performant manner. You can approach this in two ways either from how your data may already be modeled today or more ideally by examining the queries you're looking to run and needs on performance of them to inform which data model may make the most sense.

One large table, without joins

We've found that storing semi-structured data in JSONB helps reduce the number of tables required, which improves scalability. Let's look at the example of web analytics data. They traditionally store a table of events with minimal information, and use lookup tables to refer to the events and record extra information. Some events have more associated information than others. By replacing the lookup tables by a JSONB column you can easily query and filter while still having great performance. Let's take a look at what an example schema might look like following by a few queries to show what's possible:

```
CREATE TABLE visits AS (
  id UUID,
  site_id uuid,
  visited_at TIMESTAMPTZ,
```

```
session_id UUID,  
page TEXT,  
url_params JSONB  
)
```

Note that url parameters for an event are open-ended, and no parameters are guaranteed. Even the common “utm” parameters (such as `utm_source`, `utm_medium`, `utm_campaign`) are by no means universal. Our choice of using a **JSONB** column for `url_params` is much more convenient than creating columns for each parameter. With JSONB we can get both the flexibility of schema, and combined with **GIN indexing** we can still have performant queries against all keys and values without having to index them individually.

Enter Citus

Assuming you do need to scale beyond a single node, **Citus** can help at scaling out your processing power, memory, and storage. In the early stages of utilizing Citus you’ll create your schema, then tell the system how you wish to shard your data.

In order to determine the ideal sharding key you need to examine the query load and types of operations you’re looking to perform. If you are storing aggregated data and all of your queries are per customer then a shard key such as `customer_id` or `tenant_id` can be a great choice. Even if you have minutely rollups and then need to report on a daily basis this can work well. This allows you to easily route queries to shards just for that customer. As a result of routing queries to a single shard this can allow you a higher concurrency.

In the case where you are storing raw data, there often ends up being a lot of data per customer. Here it can be more difficult to get sub-second response without further parallelizing queries per customer. *It may also be difficult to get predictable sub-second responsiveness if you have a low number of customers or if 80% of your data comes from one customer.* In the above mentioned cases, picking a shard key that’s more granular than customer or tenant id can be ideal.

The distribution of your data and query workload is what will heavily determine which key is right for you.

With the above example if all of your sites have the same amount of traffic then `site_id` might be reasonable, but if either of the above cases is true then something like `session_id` could be a more ideal distribution key.

The query workload

With a sharding key of `session_id` we could easily perform a number of queries such as:

Top page views over the last 7 days for a given site:

```
SELECT page,  
       count(*)  
FROM visits  
WHERE site_id = 'foo'  
      AND visited_at > now() - '7 days'::interval  
GROUP BY page  
ORDER BY 2 DESC;
```

Unique sessions today:

```
SELECT distinct(session_id)  
FROM visits  
WHERE site_id = 'foo'  
      AND visited_at > date_trunc('date', now())
```


And assuming you have an index on `url_params` you could easily do various rollups on it... Such as find the campaigns that have driven the most traffic to you over the past 30 days and which pages received the most benefit:

```
SELECT url_params ->> 'utm_campaign',
       page,
       count(*)
FROM visits
WHERE url_params ? 'utm_campaign'
      AND visited_at >= now() - '30 days'::interval
      AND site_id = 'foo'
GROUP BY url_params ->> 'utm_campaign',
         page
ORDER BY 3 DESC;
```

Every distribution has its thorns

Choosing a sharding key always involves trade-offs. If you're optimising to get the maximum parallelism out of your database then matching your cores to the number of shards ensures that every query takes full advantage of your resources. In contrast if you're optimising for higher read concurrency, then allowing queries to run against only a single shard will allow more queries to run at once, although each individual query will experience less parallelism.

The choice really comes down to what you're trying to accomplish in your application. If you have questions about what method to use to shard your data, or what key makes sense for your application please feel free to reach out to us or join our slack channel.

Scalable Real-time Product Search using PostgreSQL with Citrus

(Copy of [original publication](#))

Product search is a common, yet sometimes challenging use-case for online retailers and marketplaces. It typically involves a combination of full-text search and filtering by attributes which differ for every product category. More complex use-cases may have many sellers that offer the same product, but with a different price and different properties.

PostgreSQL has the functionality required to build a product search application, but performs poorly when indexing and querying large catalogs. With Citrus, PostgreSQL can distribute tables and parallelize queries across many servers, which lets you scale out memory and compute power to handle very large catalogs. While the search functionality is not as comprehensive as in dedicated search solutions, a huge benefit of keeping the data in PostgreSQL is that it can be updated in real-time and tables can be joined. This post will go through the steps of setting up an experimental products database with a parallel search function using PostgreSQL and Citrus, with the goal of showcasing several powerful features.

We start by setting up a [multi-node Citrus cluster on EC2](#) using 4 m3.2xlarge instances as workers. An even easier way to get started is to use [Citrus Cloud](#), which gives you a managed Citrus cluster with full auto-failover. The main table in our [database schema](#) is the “product” table, which contains the name and description of a product, its price, and attributes in [JSON format](#) such that different types of products can use different attributes:

```
$ psql
CREATE TABLE product (
  product_id int primary key,
  name text not null,
  description text not null,
  price decimal(12,2),
  attributes jsonb
);
```

To distribute the table using Citrus, we call the functions for *Creating and Modifying Distributed Tables (DDL)* the table into 16 shards (one per physical core). The shards are distributed and replicated across the 4 workers.

```
SELECT create_distributed_table('product', 'product_id');
```

We create a [GIN index](#) to allow fast filtering of attributes by the JSONB containment operator. For example, a search query for English books might have the following expression: `attributes @> '{"category": "books", "language": "english"}'`, which can use the GIN index.

```
CREATE INDEX attributes_idx ON product USING GIN (attributes jsonb_path_ops);
```

To filter products by their name and description, we use the [full text search functions](#) in PostgreSQL to find a match with a user-specified query. A text search operation is performed on a text search vector (tsvector) using a text search query (tsquery). It can be useful to define an intermediate function that generates the tsvector for a product. The `product_text_search` function below combines the name and description of a product into a tsvector, in which the name is assigned the highest weight (from 'A' to 'D'), such that matches with the name will show up higher when sorting by relevance.

```
CREATE FUNCTION product_text_search(name text, description text)
RETURNS tsvector LANGUAGE sql IMMUTABLE AS $function$
    SELECT setweight(to_tsvector(name), 'A') ||
           setweight(to_tsvector(description), 'B');
$function$;
```

To use the `product_text_search` function in queries and indexes, it also needs to be created on the workers. We'll use `run_command_on_workers` to do this (see [Running on all Workers](#) for more info).

```
$ psql
SELECT run_command_on_workers($cmd$
    CREATE FUNCTION product_text_search(name text, description text)
    RETURNS tsvector LANGUAGE sql IMMUTABLE AS $function$
        SELECT setweight(to_tsvector(name), 'A') ||
               setweight(to_tsvector(description), 'B');
    $function$;
$cmd$);
```

After setting up the function, we define a GIN index on it, which speeds up text searches on the product table.

```
$ psql
CREATE INDEX text_idx ON product USING GIN (product_text_search(name, description));
```

We don't have a large product dataset available, so instead we generate 10 million mock products (7GB) by appending random words to generate names, descriptions, and attributes, using a [simple generator function](#). This is probably not be the fastest way to generate mock data, but we're PostgreSQL geeks :). After adding some words to the words table, we can run:

```
\COPY (SELECT * FROM generate_products(10000000)) TO '/data/base/products.tsv'
```

The new COPY feature in Citrus can be used to load the data into the product table. COPY for hash-partitioned tables is currently available in the [latest version of Citrus](#) and in [Citrus Cloud](#). A benefit of using COPY on distributed tables is that workers can process multiple rows in parallel. Because each shard is indexed separately, the indexes are also kept small, which improves ingestion rate for GIN indexes.

```
\COPY product FROM '/data/base/products.tsv'
```

The data load takes just under 7 minutes; roughly 25,000 rows/sec on average. We also loaded data into a regular PostgreSQL table in 45 minutes (3,700 rows/sec) by creating the index after copying in the data.

Now let's search products! Assume the user is searching for "copper oven". We can convert the phrase into a tsquery using the `plainto_tsquery` function and match it to the name and description using the `@@` operator. As an additional filter, we require that the "food" attribute of the product is either "waste" or "air". We're using very random words :). To order the query by relevance, we can use the `ts_rank` function, which takes the tsvector and tsquery as input.

```
SELECT p.product_id, p.name, p.price
FROM product p
WHERE product_text_search(name, description) @@ plainto_tsquery('copper oven')
  AND (attributes @> '{"food":"waste"}' OR attributes @> '{"food":"air"}')
ORDER BY ts_rank(product_text_search(name, description),
                 plainto_tsquery('copper oven')) DESC
LIMIT 10;
```

product_id	name	price
2016884	oven copper hot	32.33
8264220	rifle copper oven	92.11
4021935	argument chin rub	79.33
5347636	oven approval circle	50.78

(4 rows)

Time: 68.832 ms (~78ms on non-distributed table)

The query above uses both GIN indexes to do a very fast look-up of a small number of rows. A much broader search can take longer because of the need to sort all the results by their rank. For example, the following query has 294,000 results that it needs to sort to get the first 10:

```
SELECT p.product_id, p.name, p.price
FROM product p
WHERE product_text_search(name, description) @@ plainto_tsquery('oven')
  AND price < 50
ORDER BY ts_rank(product_text_search(name, description),
                 plainto_tsquery('oven')) DESC
LIMIT 10;
```

product_id	name	price
6295883	end oven oven	7.80
3304889	oven punishment oven	28.27
2291463	town oven oven	7.47
...		

(10 rows)

Time: 2262.502 ms (37 seconds on non-distributed table)

This query gets the top 10 results from each of the 16 shards, which is where the majority of time is spent, and the master sorts the final 160 rows. By using more machines and more shards, the number of rows that needs to be sorted in each shard is lowered significantly, but the amount of sorting work done by the master is still trivially small. This means that we can get significantly lower query times by using a bigger cluster with more shards.

In addition to products, imagine the retailer also has a marketplace where third-party sellers can offer products at different prices. Those offers should also show up in searches if their price is under the maximum. A product can have many such offers. We create an additional distributed table, which we distribute by `product_id` and assign the same number of shards, such that we can perform joins on the *co-located* product / offer tables on `product_id`.

```
CREATE TABLE offer (
  product_id int not null,
  offer_id int not null,
  seller_id int,
  price decimal(12,2),
  new bool,
  primary key(product_id, offer_id)
);
SELECT create_distributed_table('offer', 'product_id');
```

We load 5 million random offers generated using the `generate_offers` function and `COPY`. The following query searches for popcorn oven products priced under \$70, including products with offers under \$70. Offers are included in the results as an array of JSON objects.

```
SELECT p.product_id, p.name, p.price, to_json(array_agg(to_json(o)))
FROM   product p LEFT JOIN offer o USING (product_id)
WHERE  product_text_search(p.name, p.description) @@ plainto_tsquery('popcorn oven')
      AND (p.price < 70 OR o.price < 70)
GROUP BY p.product_id, p.name, p.description, p.price
ORDER BY ts_rank(product_text_search(p.name, p.description),
               plainto_tsquery('popcorn oven')) DESC
LIMIT 10;
```

product_id	name	price	
9354998	oven popcorn bridge	41.18	[null]
1172380	gate oven popcorn	24.12	[{"product_id":1172380,"offer_id":
4853987			"seller_id":2088,"price":55.00,"new":true}]
985098	popcorn oven scent	73.32	[{"product_id":985098,"offer_id":
5890813			"seller_id":5727,"price":67.00,"new":true}]
...			

(10 rows)

Time: 337.441 ms (4 seconds on non-distributed tables)

Given the wide array of features available in PostgreSQL, we can keep making further enhancements. For example, we could convert the entire row to JSON, or add a filter to only return reasonably close matches, and we could make sure only lowest priced offers are included in the results. We can also start doing real-time inserts and updates in the product and offer tables.