Citus Documentation

Release 11.3

Citus Data

Jul 18, 2023

GET STARTED

1 What is Citus? 3

1.1 Citus Gives You Postgres At Any Scale . . . 3

2 When to Use Citus 5

2.1 Multi-Tenant SaaS 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 . . . 11

4 Single-Node Citus 15

4.1 Docker (Mac or Linux) . . . 15
4.2 Ubuntu or Debian . . . 16
4.3 Fedora, CentOS, or Red Hat . . . 17

5 Multi-Node Citus 19

5.1 Ubuntu or Debian . . . 19
5.2 Fedora, CentOS, or Red Hat . . . 21

6 Managed Service 25

7 Multi-tenant Applications 27

7.1 Let's Make an App – Ad Analytics . . . 28
7.2 Scaling the Relational Data Model . . . 29
7.3 Preparing Tables and Ingesting Data . . . 30
7.4 Integrating Applications . . . 33
7.5 Sharing Data Between Tenants . . . 34
7.6 Online Changes to the Schema . . . 35
7.7 When Data Differs Across Tenants . . . 36
7.8 Scaling Hardware Resources . . . 37
7.9 Dealing with Big Tenants . . . 37
7.10 Where to Go From Here . . . 39

8 Real-Time Dashboards 41

8.1 Data Model . . . 41
8.2 Rollups . . . 43
8.3 Expiring Old Data . . . 45
8.4 Approximate Distinct Counts . . . 45

8.5 Unstructured Data with JSONB . . . 46

9 Timeseries Data 49
9.1 Scaling Timeseries Data on Citus . . . 50
9.2 Automating Partition Creation . . . 51
9.3 Archiving with Columnar Storage . . . 52

10 Concepts 55
10.1 Nodes . . . 55
10.2 Distributed Data . . . 55
10.3 Query Execution . . . 58

11 Determining Application Type 61
11.1 At a Glance . . . 61
11.2 Examples and Characteristics . . . 61

12 Choosing Distribution Column 63
12.1 Multi-Tenant Apps . . . 63
12.2 Real-Time Apps . . . 64
12.3 Timeseries Data . . . 65
12.4 Table Co-Location . . . 65

13 Migrating an Existing App 71
13.1 Identify Distribution Strategy . . . 71
13.2 Prepare Source Tables for Migration . . . 73
13.3 Prepare Application for Citus . . . 74
13.4 Migrate Production Data . . . 91

14 SQL Reference 95
14.1 Creating and Modifying Distributed Objects (DDL) . . . 95
14.2 Ingesting, Modifying Data (DML) . . . 103
14.3 Caching Aggregations with Rollups . . . 105
14.4 Querying Distributed Tables (SQL) . . . 108
14.5 Query Processing . . . 114
14.6 Manual Query Propagation . . . 119
14.7 SQL Support and Workarounds . . . 121

15 Citus API 125
15.1 Citus Utility Functions . . . 125
15.2 Citus Tables and Views . . . 157
15.3 Configuration Reference . . . 178

16 External Integrations 191
16.1 Change Data Capture (CDC) . . . 191
16.2 Ingesting Data from Kafka . . . 194
16.3 Ingesting Data from Spark . . . 197
16.4 Business Intelligence with Tableau . . . 200

17 Cluster Management 205
17.1 Choosing Cluster Size . . . 205
17.2 Initial Hardware Size . . . 206
17.3 Scaling the cluster . . . 206
17.4 Dealing With Node Failures . . . 209
17.5 Tenant Isolation . . . 209
17.6 Viewing Query Statistics . . . 211

17.7 Resource Conservation . . . 213
17.8 Security . . . 213
17.9 PostgreSQL extensions . . . 218
17.10 Creating a New Database . . . 219

18 Table Management **221**
18.1 Determining Table and Relation Size . . . 221
18.2 Vacuuming Distributed Tables . . . 222
18.3 Analyzing Distributed Tables . . . 222
18.4 Columnar Storage . . . 223

19 Upgrading Citus **229**
19.1 Upgrading Citus Versions . . . 229
19.2 Upgrading PostgreSQL version from 14 to 15 . . . 231

20 Query Performance Tuning **233**
20.1 Table Distribution and Shards . . . 233
20.2 PostgreSQL tuning . . . 233
20.3 Scaling Out Performance . . . 236
20.4 Distributed Query Performance Tuning . . . 237
20.5 Scaling Out Data Ingestion . . . 241

21 Useful Diagnostic Queries **245**
21.1 Finding which shard contains data for a specific tenant . . . 245
21.2 Finding the distribution column for a table . . . 245
21.3 Detecting locks . . . 246
21.4 Querying the size of your shards . . . 246
21.5 Querying the size of all distributed tables . . . 247
21.6 Identifying unused indices . . . 247
21.7 Monitoring client connection count . . . 248
21.8 Viewing system queries . . . 248
21.9 Index hit rate . . . 249
21.10 Cache hit rate . . . 250

22 Common Error Messages **251**
22.1 Could not receive query results . . . 251
22.2 Canceling the transaction since it was involved in a distributed deadlock . . . 251
22.3 Could not connect to server: Cannot assign requested address . . . 252
22.4 SSL error: certificate verify failed . . . 252
22.5 Could not connect to any active placements . . . 253
22.6 Remaining connection slots are reserved for non-replication superuser connections . . . 253
22.7 PgBouncer cannot connect to server . . . 253
22.8 Relation *foo* is not distributed . . . 254
22.9 Unsupported clause type . . . 254
22.10 Cannot open new connections after the first modification command within a transaction . . . 254
22.11 Cannot create uniqueness constraint . . . 255
22.12 Function create_distributed_table does not exist . . . 255
22.13 STABLE functions used in UPDATE queries cannot be called with column references . . . 256

23 Frequently Asked Questions **257**
23.1 Can I create primary keys on distributed tables? . . . 257
23.2 How do I add nodes to an existing Citus cluster? . . . 257
23.3 How does Citus handle failure of a worker node? . . . 257
23.4 How does Citus handle failover of the coordinator node? . . . 257
23.5 Are there any PostgreSQL features not supported by Citus? . . . 258

23.6 How do I choose the shard count when I hash-partition my data? 258
23.7 How do I change the shard count for a hash partitioned table? 259
23.8 How does citus support count(distinct) queries? 259
23.9 In which situations are uniqueness constraints supported on distributed tables? 259
23.10 How do I create database roles, functions, extensions etc in a Citus cluster? 259
23.11 What if a worker node's address changes? 260
23.12 Which shard contains data for a particular tenant? 260
23.13 I forgot the distribution column of a table, how do I find it? 260
23.14 Can I distribute a table by multiple keys? 260
23.15 Why does pg_relation_size report zero bytes for a distributed table? 260
23.16 Why am I seeing an error about max_intermediate_result_size? 261
23.17 Can I run Citus on Microsoft Azure? 261
23.18 Can I shard by schema on Citus for multi-tenant applications? 261
23.19 How does cstore_fdw work with Citus? 261
23.20 What happened to pg_shard? 262

24 Related Articles **263**
24.1 Efficient Rollup Tables with HyperLogLog in Postgres 263
24.2 Distributed Distinct Count with HyperLogLog on Postgres 268
24.3 Postgres Parallel Indexing in Citus 272
24.4 Real-time Event Aggregation at Scale Using Postgres with Citus 274
24.5 How Distributed Outer Joins on PostgreSQL with Citus Work 278
24.6 Designing your SaaS Database for Scale with Postgres 283
24.7 Building a Scalable Postgres Metrics Backend using the Citus Extension 286
24.8 Sharding a Multi-Tenant App with Postgres 290
24.9 Sharding Postgres with Semi-Structured Data and Its Performance Implications 292
24.10 Scalable Real-time Product Search using PostgreSQL with Citus 295

Welcome to the documentation for Citus 11.3! Citus is an open source extension to PostgreSQL that transforms Postgres into a distributed database. To scale out Postgres horizontally, Citus employs distributed tables, reference tables, and a distributed SQL query engine. The query engine parallelizes SQL queries across multiple servers in a database cluster to deliver dramatically improved query response times, even for data-intensive applications.

CHAPTER

ONE

WHAT IS CITUS?

The Citus database is an open source extension to Postgres that gives you all the greatness of Postgres, at any scale—from a single node to a large distributed database cluster. Because Citus is an extension (not a fork) to Postgres, when you use Citus, you are also using Postgres. You can leverage the latest Postgres features, tooling, and ecosystem.

With Citus you get distributed Postgres features like sharding, distributed tables, reference tables, a distributed query engine, columnar storage—and as of Citus 11.0, the ability to query from any node. The Citus combination of parallelism, keeping more data in memory, and higher I/O bandwidth can lead to significant performance improvements for multi-tenant SaaS applications, customer-facing real-time analytics dashboards, and time series workloads.

Two Ways to Get Citus:

1. **Open source**: Citus is 100% open source. You can download Citus open source, or to see the source code and build it yourself, visit the Citus repo on GitHub.
2. **Managed service**: The Citus database is available as a managed service in the cloud with Azure Cosmos DB for PostgreSQL, formerly known as Hyperscale (Citus) in Azure Database for PostgreSQL.

1.1 Citus Gives You Postgres At Any Scale

You can start using Citus on a single node, using a distributed data model from the beginning so you are "scale out ready". When your Postgres workload needs to scale, it's easy to add worker nodes to the Citus database cluster, and/or to scale up the coordinator and worker nodes in your cluster.

Sometimes people ask "how big can Citus scale?" Here are a few examples of large-scale customers—but please keep in mind that there are many, many 2-node and 3-node Citus clusters in the wild, too.

- Algolia
  - 5-10B rows ingested per day
- Heap
  - 700+ billion events
  - 1.4PB of data on a 100-node Citus database cluster
- Pex
  - 80B rows updated/day
  - 20-node Citus database cluster
  - 2.4TB memory, 1280 cores, and 80TB of data
  - . . . with plans to grow to 45 nodes
- MixRank

  - 10 PB of time series data

For more customers and statistics, see our customer stories.

CHAPTER

TWO

WHEN TO USE CITUS

2.1 Multi-Tenant SaaS 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

2.2 Real-Time Analytics

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

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

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

2.3 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.

2.4 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 you do not expect your workload to ever grow beyond a single Postgres node
- 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

CHAPTER

THREE

QUICK TUTORIALS

3.1 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 setup Citus locally using one of the options from *Single-Node Citus*.

3.1.1 Data model and sample data

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

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

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

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

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

3.1.2 Creating tables

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

If you are using native Postgres, as installed in our *Single-Node Citus* guide, the coordinator node will be running on port 9700.

```
psql -p 9700
```

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

```
docker exec -it citus 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);
```

3.1.3 Distributing tables and loading data

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

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

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

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

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

3.1.4 Running queries

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

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

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

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

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

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

```
BEGIN;
DELETE FROM campaigns WHERE id = 46 AND company_id = 5;
DELETE FROM ads WHERE campaign_id = 46 AND company_id = 5;
COMMIT;
```

Each statement in a transactions causes roundtrips between the coordinator and workers in multi-node Citus. For multi-tenant workloads, it's more efficient to run transactions in distributed functions. The efficiency gains become more apparent for larger transactions, but we can use the small transaction above as an example.

First create a function that does the deletions:

```
CREATE OR REPLACE FUNCTION
  delete_campaign(company_id int, campaign_id int)
```

(continues on next page)

(continued from previous page)

```
RETURNS void LANGUAGE plpgsql AS $fn$
BEGIN
  DELETE FROM campaigns
   WHERE id = $2 AND campaigns.company_id = $1;
  DELETE FROM ads
   WHERE ads.campaign_id = $2 AND ads.company_id = $1;
END;
$fn$;
```

Next use *create_distributed_function* to instruct Citus to run the function directly on workers rather than on the coordinator (except on a single-node Citus installation, which runs everything on the coordinator). It will run the function on whatever worker holds the *Shards* for tables `ads` and `campaigns` corresponding to the value `company_id`.

```
SELECT create_distributed_function(
  'delete_campaign(int, int)', 'company_id',
  colocate_with := 'campaigns'
);

-- you can run the function as usual
SELECT delete_campaign(5, 46);
```

Besides transactional operations, you can also run analytics queries 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 ads.campaign_id = campaigns.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.

3.2 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 setup Citus locally using one of the options from *Single-Node Citus*.

3.2.1 Data model and sample data

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

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

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

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

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

3.2.2 Creating tables

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

If you are using native Postgres, as installed in our *Single-Node Citus* guide, the coordinator node will be running on port 9700.

```
psql -p 9700
```

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

```
docker exec -it citus 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,
```

(continues on next page)

(continued from previous page)

```
    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);
```

3.2.3 Distributing tables and loading data

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

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

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

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

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

3.2.4 Running queries

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

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

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

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

(continues on next page)

(continued from previous page)

```
GROUP BY minute
ORDER BY minute;
```

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

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

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

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

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

CHAPTER

FOUR

SINGLE-NODE CITUS

The installation instructions below will guide you to install Citus open source packages (along with Postgres packages) on a single node. Single-node Citus gives you an easy way to start small in production, while being ready for future scale and growth. Single-node Citus is also useful for early development and testing.

4.1 Docker (Mac or Linux)

Note: The Docker image is intended for development/testing purposes only, and has not been prepared for production use.

You can start Citus in Docker with one command:

```
# start the image
docker run -d --name citus -p 5432:5432 -e POSTGRES_PASSWORD=mypass \
           citusdata/citus:11.3

# verify it's running, and that Citus is installed:
psql -U postgres -h localhost -d postgres -c "SELECT * FROM citus_version();"
```

You should see the latest version of Citus.

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.

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 Citus image attempts to bind to the standard PostgreSQL port 5432. To fix this, choose a different port with the -p option. You will need to also use the new port in the psql command below as well.

4.2 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 15 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-15-citus-11.3
```

2. Initialize the Cluster

Let's create a new database on disk. 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/15/bin

cd ~
mkdir citus
initdb -D citus
```

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/postgresql.conf
```

3. Start the database server

Finally, we'll start an instance of PostgreSQL for the new directory:

```
pg_ctl -D citus -o "-p 9700" -l citus_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;"
```

4. Verify that installation has succeeded

To verify that the installation has succeeded, and Citus is installed:

```
psql -p 9700 -c "select citus_version();"
```

You should see details of the Citus extension.

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.

4.3 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 15 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 citus113_15
```

2. Initialize the Cluster

Let's create a new database on disk. 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-15/bin

cd ~
mkdir citus
initdb -D citus
```

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/postgresql.conf
```

3. Start the database server

Finally, we'll start an instance of PostgreSQL for the new directory:

```
pg_ctl -D citus -o "-p 9700" -l citus_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;"
```

4. Verify that installation has succeeded

To verify that the installation has succeeded, and Citus is installed:

```
psql -p 9700 -c "select citus_version();"
```

You should see details of the Citus extension.

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.

CHAPTER

FIVE

MULTI-NODE CITUS

The *Single-Node Citus* section has instructions for installing Citus open source packages on a single node. If you are looking to deploy the Citus database across multiple nodes (along with Postgres packages, too) you can use the guide below.

5.1 Ubuntu or Debian

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

5.1.1 Steps to be executed on all nodes

1. Add repository

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

2. Install PostgreSQL + Citus and initialize a database

```
# install the server and initialize db
sudo apt-get -y install postgresql-15-citus-11.3

# preload citus extension
sudo pg_conftool 15 main set shared_preload_libraries citus
```

This installs centralized configuration in */etc/postgresql/15/main*, and creates a database in */var/lib/postgresql/15/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 15 main set listen_addresses '*'
```

```
sudo vi /etc/postgresql/15/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
```

(continues on next page)

(continued from previous page)

```

# 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;"
```

5.1.2 Steps to be executed on the coordinator node

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

1. Add worker node information

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

```
# Register the hostname that future workers will use to connect
# to the coordinator node.
#
# You'll need to change the example, 'coord.example.com',
# to match the actual hostname

sudo -i -u postgres psql -c \
  "SELECT citus_set_coordinator_host('coord.example.com', 5432);"

# Add the worker nodes.
#
# Similarly, you'll need to change 'worker-101' and 'worker-102' to the
# actual hostnames

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

5.2 Fedora, CentOS, or Red Hat

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

5.2.1 Steps to be executed on all nodes

1. Add repository

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

2. Install PostgreSQL + Citus and initialize a database

```
# install PostgreSQL with Citus extension
sudo yum install -y citus113_15
# initialize system database (using RHEL 6 vs 7 method as necessary)
sudo service postgresql-15 initdb || sudo /usr/pgsql-15/bin/postgresql-15-setup initdb
# preload citus extension
echo "shared_preload_libraries = 'citus'" | sudo tee -a /var/lib/pgsql/15/data/
↪postgresql.conf
```

PostgreSQL adds version-specific binaries in */usr/pgsql-15/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/15/data/postgresql.conf
```

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

```
sudo vi /var/lib/pgsql/15/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 Citus extension

```
# start the db server
sudo service postgresql-15 restart
# and make it start automatically when computer does
sudo chkconfig postgresql-15 on
```

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

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

5.2.2 Steps to be executed on the coordinator node

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

1. Add worker node information

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

```
# Register the hostname that future workers will use to connect
# to the coordinator node.
#
# You'll need to change the example, 'coord.example.com',
# to match the actual hostname

sudo -i -u postgres psql -c \
  "SELECT citus_set_coordinator_host('coord.example.com', 5432);"

# Add the worker nodes.
#
# Similarly, you'll need to change 'worker-101' and 'worker-102' to the
# actual hostnames

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

CHAPTER

SIX

MANAGED SERVICE

One of the easiest ways to get started with Citus is to spin up a cluster in the cloud on Azure. The Citus managed service is available as Azure Cosmos DB for PostgreSQL, formerly known as Hyperscale (Citus) in Azure Database for PostgreSQL.

To get started with Citus on Azure, you can take advantage of our quickstart for Azure Cosmos DB for PostgreSQL in the Azure docs, or go straight to the free trial on Azure.)

CHAPTER

SEVEN

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 Citus provide all the tools needed to handle these challenges, so let's get building.

7.1 Let's Make an App – Ad Analytics

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

```sql
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,
```

(continues on next page)

(continued from previous page)

```
  cost_per_click_usd numeric(20,10),
  user_ip inet NOT NULL,
  user_data jsonb NOT NULL
);

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

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

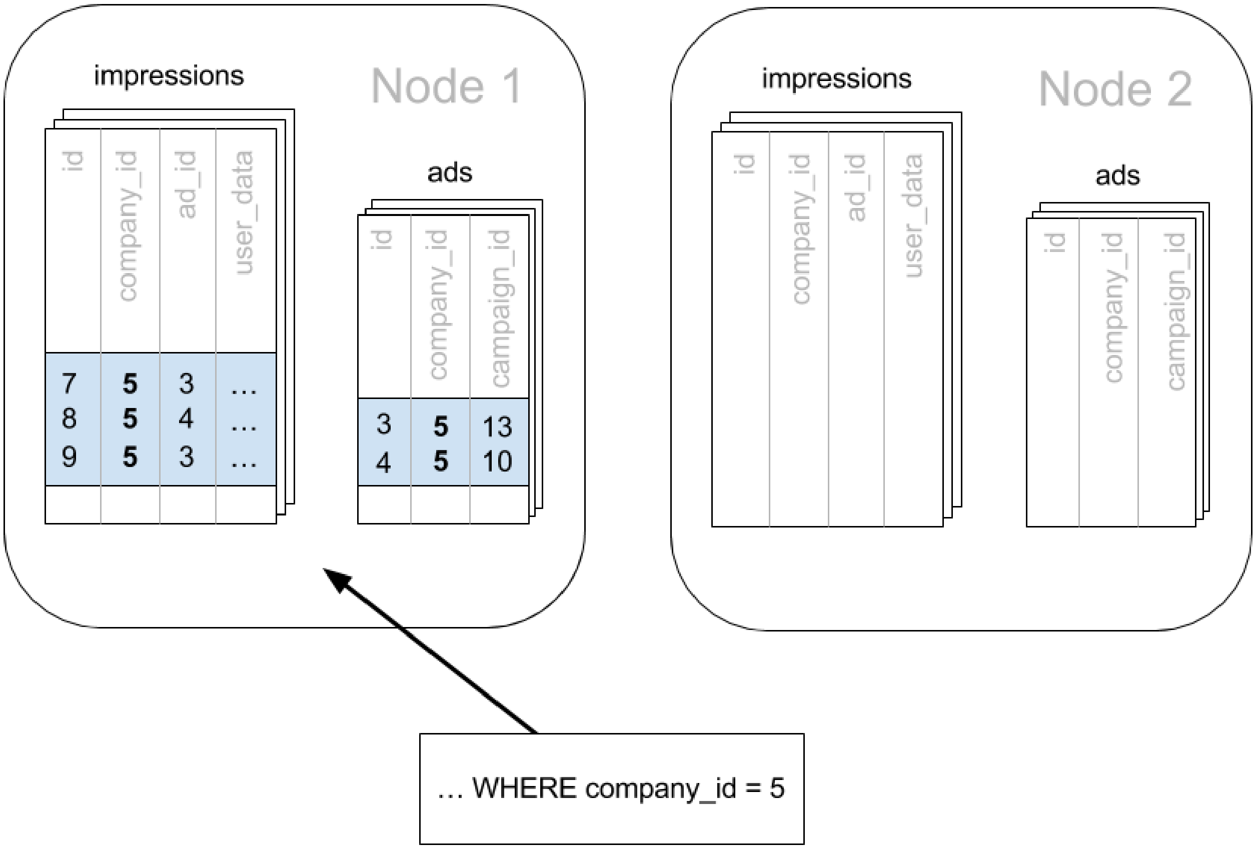
7.2 Scaling the Relational Data Model

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

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

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

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



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

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

7.3 Preparing Tables and Ingesting Data

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

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

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

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

```
CREATE TABLE companies (
  id bigserial PRIMARY KEY,
  name text NOT NULL,
```

(continues on next page)

(continued from previous page)

```
  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),
```

(continues on next page)

(continued from previous page)

```
  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*.

7.3.1 Try it Yourself

Note: This guide is designed so you can follow along in your own Citus database. This tutorial assumes that you already have Citus installed and running. If you don't have Citus running, you can setup Citus locally using one of the options from *Single-Node Citus*.

You'll run the SQL commands using psql and connect to the Coordinator node:

- **Docker**: `docker exec -it citus psql -U postgres`

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:.
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
```

7.4 Integrating Applications

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

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

ActiveRecord:

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

Django:

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

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

Also, to make it even simpler, you can use our activerecord-multi-tenant library for Rails, 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 *Ruby on Rails* and Django.

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

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

```
-- campaigns with highest budget

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

-- double the budgets!

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

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

```
-- transactionally reallocate campaign budget money

BEGIN;

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

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

COMMIT;
```

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

```
SELECT a.campaign_id,
       RANK() OVER (
         PARTITION BY a.campaign_id
         ORDER BY a.campaign_id, count(*) desc
       ), count(*) as n_impressions, a.id
  FROM ads as a
  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.

7.5 Sharing Data Between Tenants

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

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

(continues on next page)

(continued from previous page)

```
);
CREATE INDEX ON geo_ips USING gist (addrs inet_ops);
```

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

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

SELECT create_reference_table('geo_ips');
```

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

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

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

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

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

7.6 Online Changes to the Schema

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

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

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

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

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

7.7 When Data Differs Across Tenants

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

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

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

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

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

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

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

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

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

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

7.8 Scaling Hardware Resources

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 Azure Cosmos DB for PostgreSQL (formerly known as Hyperscale (Citus) in Azure Database for PostgreSQL), you can use the Azure Portal to add the required number of nodes. Alternatively, if you run your own Citus installation, you can add nodes manually with the *citus_add_node* UDF.

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

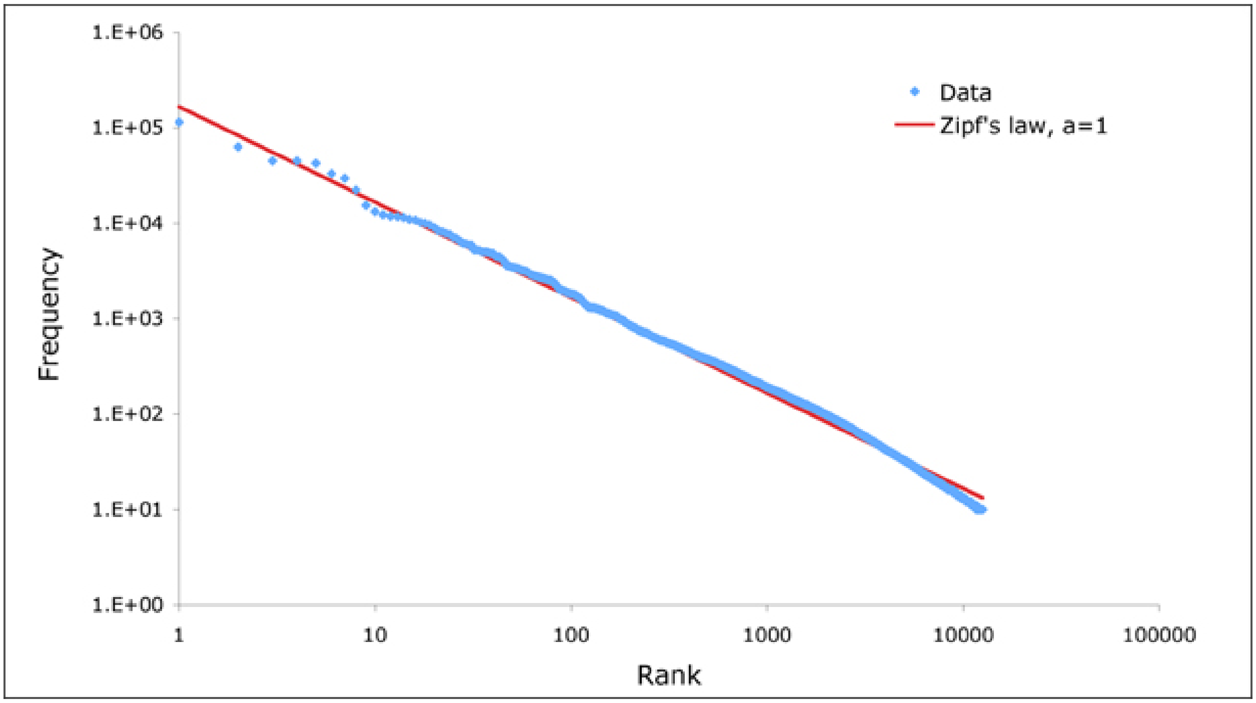
```
SELECT citus_rebalance_start();
```

With our cloud_topic, or with Community edition 11.0 and above, 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. In Citus Community edition, writes to shards are blocked during rebalancing but reads are unaffected.

7.9 Dealing with Big Tenants

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

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



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

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

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

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

First isolate the tenant's data to a dedicated shard suitable to move. The CASCADE option also applies this change to the rest of our tables distributed by `company_id`.

```sql
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)

-- note that you should set wal_level for all nodes to be >= logical
-- to use citus_move_shard_placement.
-- you also need to restart your cluster after setting wal_level in
-- postgresql.conf files.

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

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

7.10 Where to Go From Here

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

To adjust a front-end application, specifically Ruby on Rails or Django, read *Ruby on Rails* or Django migration. Finally, try Azure Cosmos DB for PostgreSQL, the easiest way to manage a Citus cluster.

CHAPTER

EIGHT

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.

8.1 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.

```sql
-- this is run on the coordinator

CREATE TABLE http_request (
  site_id INT,
  ingest_time TIMESTAMPTZ DEFAULT now(),

  url TEXT,
  request_country TEXT,
  ip_address TEXT,

  status_code INT,
  response_time_msec INT
);

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

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

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

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

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:

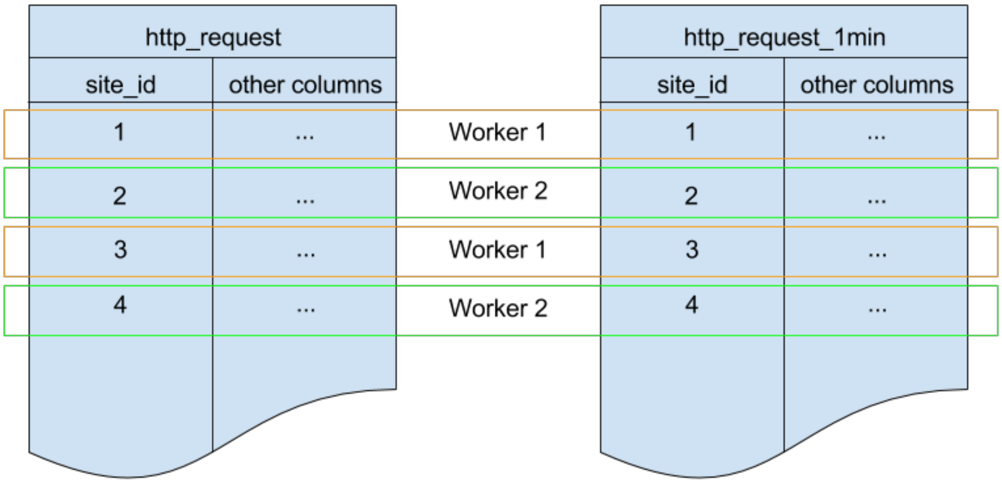
- Your HTTP analytics dashboard must go over each row every time it needs to generate a graph. For example, if your clients are interested in trends over the past year, your queries will aggregate every row for the past year from scratch.
- Your storage costs will grow proportionally with the ingest 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).

8.2 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. Because all three of those match, there's a 1-to-1 correspondence between `http_request` shards and `http_request_1min` shards, and Citus will place matching shards on the same worker. This is called *co-location*; it makes queries such as joins faster and our rollups possible.



In order to populate `http_request_1min` we're going to periodically run 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 timestamptz PRIMARY KEY,

```

(continues on next page)

(continued from previous page)

```
  -- "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 timestamptz := date_trunc('minute', now() - interval '1 minute');
  last_rollup_time timestamptz := 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();'
```

Alternatively, 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;
```

8.3 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 Citus 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.

8.4 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:

```
CREATE EXTENSION hll;
```

Note: This is not necessary when using Citus in the cloud with Azure Cosmos DB for PostgreSQL, 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
```

(continues on next page)

(continued from previous page)

```
  ) 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
+   , 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.

8.5 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,
```

(continues on next page)

(continued from previous page)

```
    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
+   , 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;
```

CHAPTER

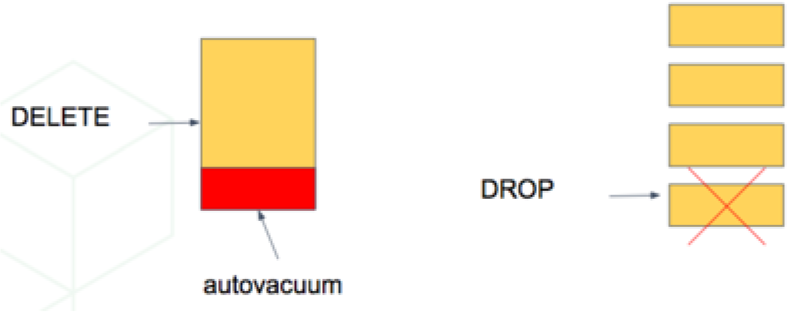
NINE

TIMESERIES DATA

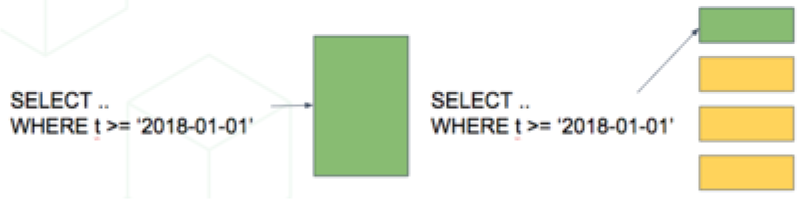
In a time-series workload, applications (such as some *Real-Time Apps*) query recent information, while archiving old information.

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.

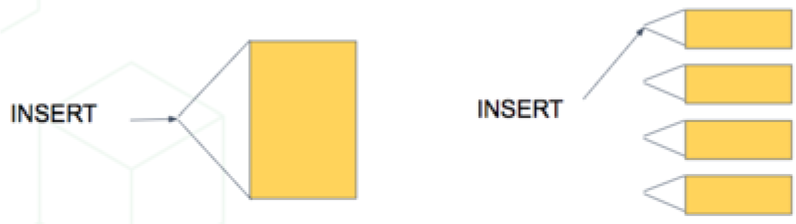
Storing data in multiple physical tables speeds up data expiration. In a single big table, deleting rows incurs the cost of scanning to find which to delete, and then vacuuming the emptied space. On the other hand, dropping a partition is a fast operation independent of data size. It's the equivalent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

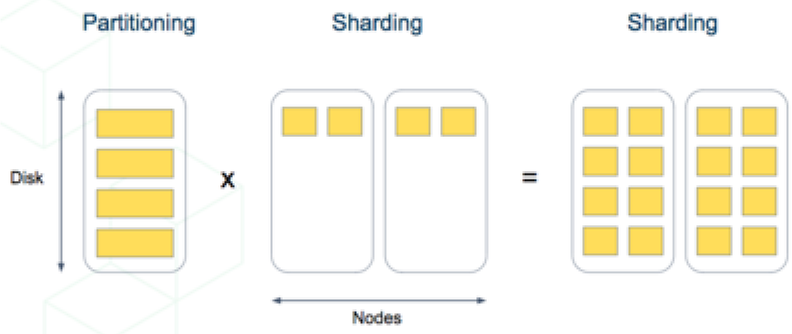
1. Most queries access a very small subset of the most recent data

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.

9.1 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 Postgres's declarative table partitioning.



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:

```
-- 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 Citus. 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 Citus 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, Citus 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 create partitions and specify their time ranges, after which we can insert data that match the ranges.

9.2 Automating Partition Creation

Citus provides helper functions for partition management. We can create a batch of monthly partitions using `create_time_partitions()`:

```
SELECT create_time_partitions(
  table_name         := 'github_events',
  partition_interval := '1 month',
  end_at             := now() + '12 months'
);
```

Citus also includes a view, *time_partitions*, for an easy way to investigate the partitions it has created.

```
SELECT partition
  FROM time_partitions
 WHERE parent_table = 'github_events'::regclass;

┌────────────────────────┐
│       partition        │
├────────────────────────┤
│ github_events_p2021_10 │
│ github_events_p2021_11 │
│ github_events_p2021_12 │
│ github_events_p2022_01 │
│ github_events_p2022_02 │
│ github_events_p2022_03 │
│ github_events_p2022_04 │
│ github_events_p2022_05 │
│ github_events_p2022_06 │
│ github_events_p2022_07 │
│ github_events_p2022_08 │
│ github_events_p2022_09 │
│ github_events_p2022_10 │
└────────────────────────┘
```

As time progresses, you will need to do some maintenance to create new partitions and drop old ones. It's best to set up a periodic job to run the maintenance functions with an extension like pg_cron:

```
-- set two monthly cron jobs:

-- 1. ensure we have partitions for the next 12 months

SELECT cron.schedule('create-partitions', '0 0 1 * *', $$
  SELECT create_time_partitions(
      table_name         := 'github_events',
      partition_interval := '1 month',
      end_at             := now() + '12 months'
  )
```

(continues on next page)

(continued from previous page)

```
$$);

-- 2. (optional) ensure we never have more than one year of data

SELECT cron.schedule('drop-partitions', '0 0 1 * *', $$
  CALL drop_old_time_partitions(
      'github_events',
      now() - interval '12 months' /* older_than */
  );
$$);
```

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

Note: Be aware that native partitioning in Postgres is still quite new and has a few quirks. Maintenance operations on partitioned tables will 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.

9.3 Archiving with Columnar Storage

Some applications have data that logically divides into a small updatable part and a larger part that's "frozen." Examples include logs, clickstreams, or sales records. In this case we can combine partitioning with *columnar table storage* (introduced in Citus 10) to compress historical partitions on disk. Citus columnar tables are currently append-only, meaning they do not support updates or deletes, but we can use them for the immutable historical partitions.

A partitioned table may be made up of any combination of row and columnar partitions. When using range partitioning on a timestamp key, we can make the newest partition a row table, and periodically roll the newest partition into another historical columnar partition.

Let's see an example, using GitHub events again. We'll create a new table called `github_columnar_events` for disambiguation from the earlier example. To focus entirely on the columnar storage aspect, we won't distribute this table.

Next, download sample data:

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

```
-- our new table, same structure as the example in
-- the previous section

CREATE TABLE github_columnar_events ( LIKE github_events )
PARTITION BY RANGE (created_at);

-- create partitions to hold two hours of data each

SELECT create_time_partitions(
  table_name         := 'github_columnar_events',
  partition_interval := '2 hours',
  start_from         := '2015-01-01 00:00:00',
  end_at             := '2015-01-01 08:00:00'
```

(continues on next page)

(continued from previous page)

```
);

-- fill with sample data
-- (note that this data requires the database to have UTF8 encoding)

\COPY github_columnar_events FROM 'github_events.csv' WITH (format CSV)

-- list the partitions, and confirm they're
-- using row-based storage (heap access method)

SELECT partition, access_method
  FROM time_partitions
 WHERE parent_table = 'github_columnar_events'::regclass;
```

```
                partition                | access_method
-----------------------------------------+---------------
 github_columnar_events_p2015_01_01_0000 | heap
 github_columnar_events_p2015_01_01_0200 | heap
 github_columnar_events_p2015_01_01_0400 | heap
 github_columnar_events_p2015_01_01_0600 | heap
```

```
-- convert older partitions to use columnar storage

CALL alter_old_partitions_set_access_method(
  'github_columnar_events',
  '2015-01-01 06:00:00' /* older_than */,
  'columnar'
);

-- the old partitions are now columnar, while the
-- latest uses row storage and can be updated

SELECT partition, access_method
  FROM time_partitions
 WHERE parent_table = 'github_columnar_events'::regclass;
```

```
                partition                | access_method
-----------------------------------------+---------------
 github_columnar_events_p2015_01_01_0000 | columnar
 github_columnar_events_p2015_01_01_0200 | columnar
 github_columnar_events_p2015_01_01_0400 | columnar
 github_columnar_events_p2015_01_01_0600 | heap
```

To see the compression ratio for a columnar table, use `VACUUM VERBOSE`. The compression ratio for our three columnar partitions is pretty good:

```
VACUUM VERBOSE github_columnar_events;
```

```
INFO:  statistics for "github_columnar_events_p2015_01_01_0000":
storage id: 10000000003
total file size: 4481024, total data size: 4444425
compression rate: 8.31x
total row count: 15129, stripe count: 1, average rows per stripe: 15129
chunk count: 18, containing data for dropped columns: 0, zstd compressed: 18

INFO:  statistics for "github_columnar_events_p2015_01_01_0200":
storage id: 10000000004
total file size: 3579904, total data size: 3548221
compression rate: 8.26x
total row count: 12714, stripe count: 1, average rows per stripe: 12714
chunk count: 18, containing data for dropped columns: 0, zstd compressed: 18

INFO:  statistics for "github_columnar_events_p2015_01_01_0400":
storage id: 10000000005
total file size: 2949120, total data size: 2917407
compression rate: 8.51x
total row count: 11756, stripe count: 1, average rows per stripe: 11756
chunk count: 18, containing data for dropped columns: 0, zstd compressed: 18
```

One power of the partitioned table `github_columnar_events` is that it can be queried in its entirety like a normal table.

```
SELECT COUNT(DISTINCT repo_id)
  FROM github_columnar_events;
```

```
┌───────┐
│ count │
├───────┤
│ 16001 │
└───────┘
```

Entries can be updated or deleted, as long as there's a WHERE clause on the partition key which filters entirely into row table partitions.

9.3.1 Archiving a Row Partition to Columnar Storage

When a row partition has filled its range, you can archive it to compressed columnar storage. We can automate this with pg_cron like so:

```
-- a monthly cron job

SELECT cron.schedule('compress-partitions', '0 0 1 * *', $$
  CALL alter_old_partitions_set_access_method(
    'github_columnar_events',
    now() - interval '6 months' /* older_than */,
    'columnar'
  );
$$);
```

For more information, see *Columnar Storage*.

CHAPTER

TEN

CONCEPTS

10.1 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.

10.1.1 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*.

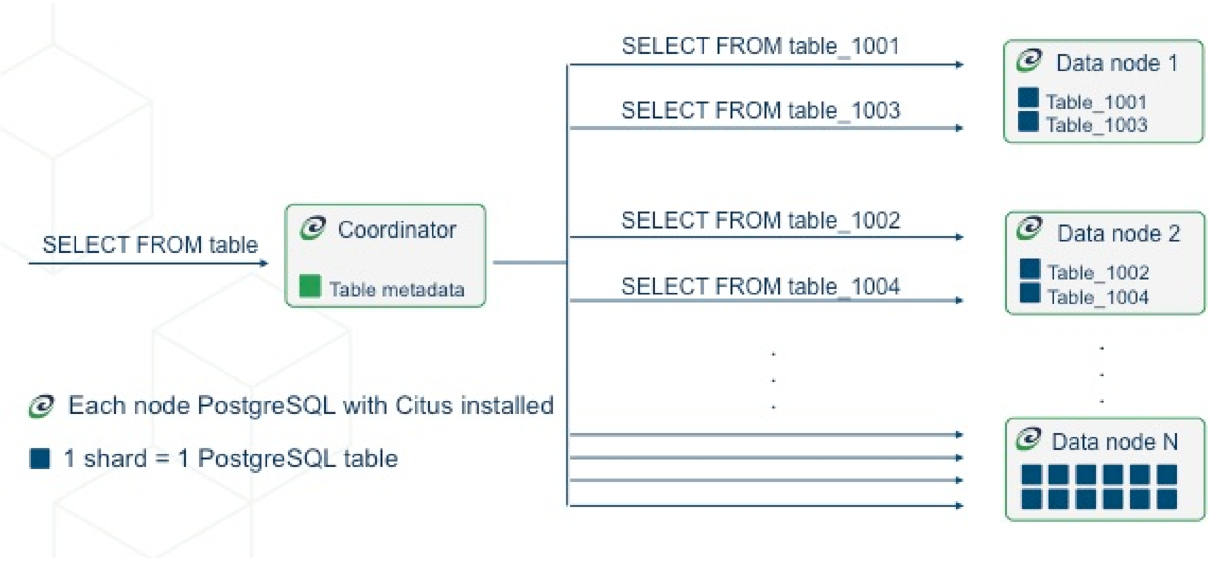
10.2 Distributed Data

10.2.1 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 Objects (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.

10.2.2 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 using PostgreSQL streaming replication. 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.

10.2.3 Co-Location

Since shards 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*.

10.2.4 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.

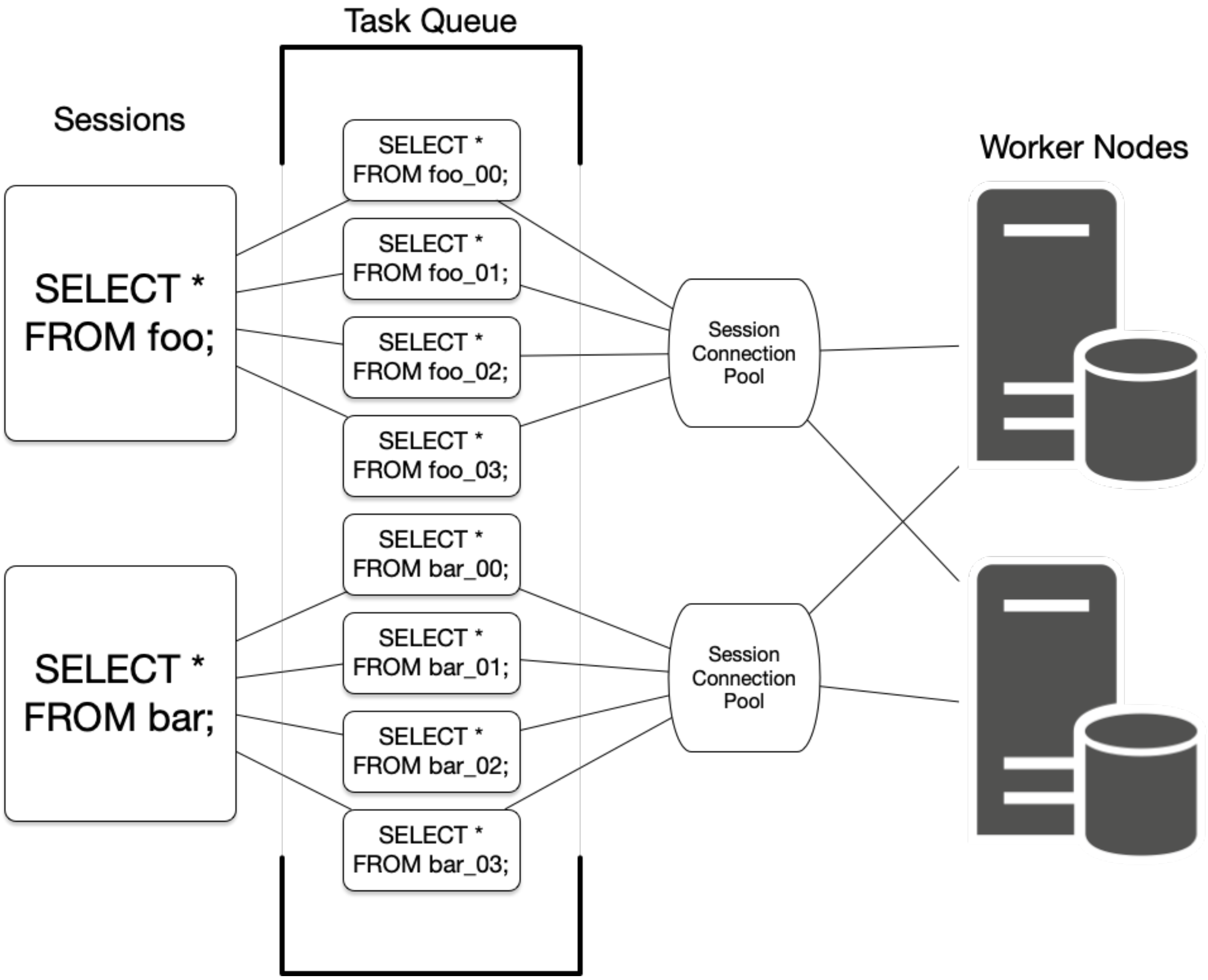
Queries reading or affecting shards spread evenly across many nodes are able to run at "real-time" speed. Note that the results of the query still need to pass back through the coordinator node, so the speedup is most apparent when the final results are compact, such as aggregate functions like counting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

Query Processing explains more about how queries are broken into fragments and how their execution is managed.

10.3 Query Execution

When executing multi-shard queries, Citus must balance the gains from parallelism with the overhead from database connections (network latency and worker node resource usage). To configure Citus' query execution for best results with your database workload, it helps to understand how Citus manages and conserves database connections between the coordinator node and worker nodes.

Citus transforms each incoming multi-shard query session into per-shard queries called tasks. It queues the tasks, and runs them once it's able to obtain connections to the relevant worker nodes. For queries on distributed tables `foo` and `bar`, here's a diagram of the connection management:



The coordinator node has a connection pool for each session. Each query (such as `SELECT * FROM foo` in the diagram) is limited to opening at most *citus.max_adaptive_executor_pool_size (integer)* simultaneous connections for its tasks per worker. That setting is configurable at the session level, for priority management.

It can be faster to execute short tasks sequentially over the same connection rather than establishing new connections for them in parallel. Long running tasks, on the other hand, benefit from more immediate parallelism.

To balance the needs of short and long tasks, Citus uses *citus.executor_slow_start_interval (integer)*. That setting specifies a delay between connection attempts for the tasks in a multi-shard query. When a query first queues tasks, the tasks can acquire just one connection. At the end of each interval where there are pending connections, Citus increases the number of simultaneous connections it will open. The slow start behavior can be disabled entirely by setting the GUC to 0.

When a task finishes using a connection, the session pool will hold the connection open for later. Caching the connection avoids the overhead of connection reestablishment between coordinator and worker. However, each pool will hold no more than *citus.max_cached_conns_per_worker (integer)* idle connections open at once, to limit idle connection resource usage in the worker.

Finally, the setting *citus.max_shared_pool_size (integer)* acts as a fail-safe. It limits the total connections per worker between all tasks.

For recommendations about tuning these parameters to match your workload, see *Connection Management*.

CHAPTER

ELEVEN

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.

11.1 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 a big table of events

11.2 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.

CHAPTER

TWELVE

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.

12.1 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*.

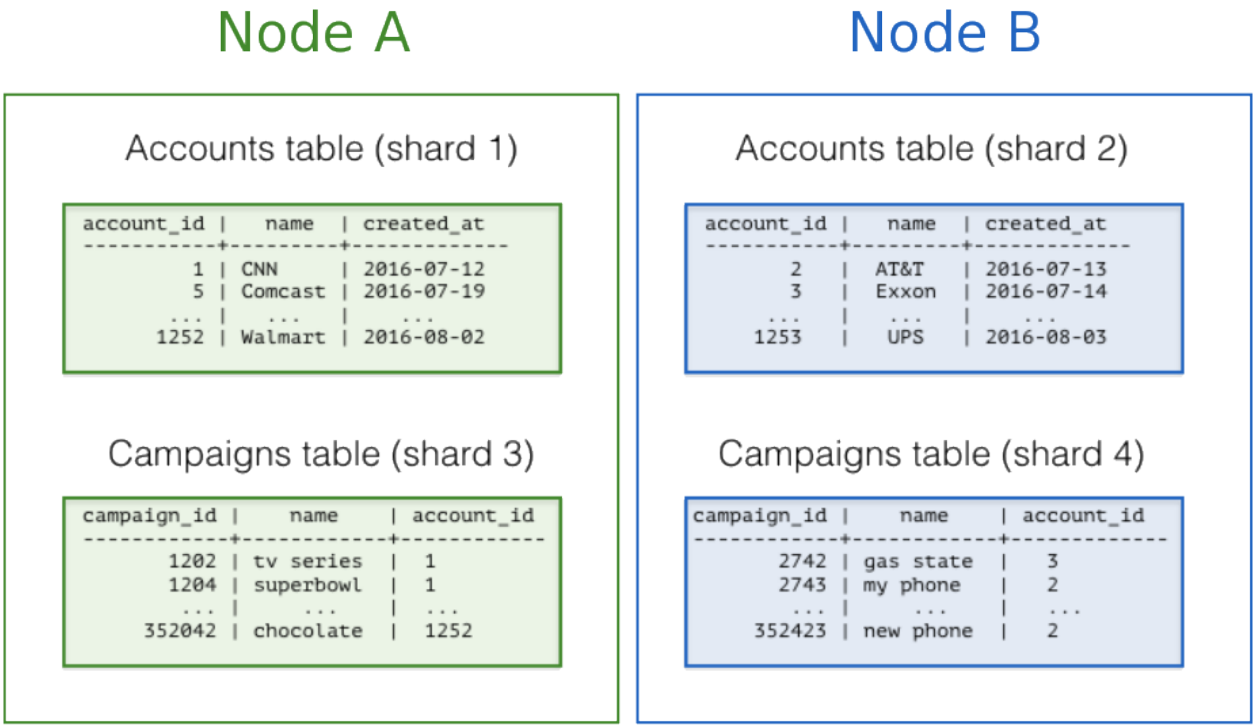
The following diagram illustrates co-location in the multi-tenant data model. It contains two tables, Accounts and Campaigns, each distributed by `account_id`. The shaded boxes represent shards, each of whose color represents which worker node contains it. Green shards are stored together on one worker 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.

12.1.1 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.

Read the *Multi-tenant Applications* guide for a detailed example of building this kind of application.



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
```

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
```

12.2 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.

12.2.1 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.

12.3 Timeseries Data

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

The most common mistake in modeling timeseries information in Citus 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.

12.3.1 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 Citus creates shards for the inherited tables.

Read the *Timeseries Data* guide for a detailed example of building this kind of application.

12.4 Table Co-Location

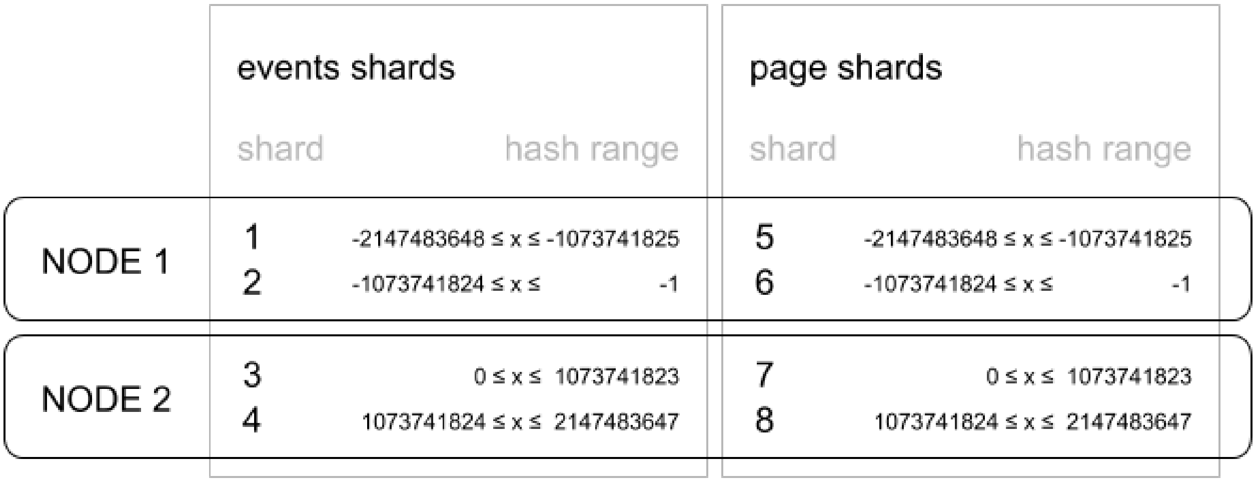
Relational databases are the first choice of data store for many applications due to their enormous flexibility and reliability. Historically the one criticism of relational databases is that they can run on only a single machine, which creates inherent limitations when data storage needs outpace server 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.

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

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

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



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

x = hash(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.

12.4.2 A practical example of co-location

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

```
CREATE TABLE event (
  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."

12.4.3 Using Regular PostgreSQL Tables

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

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

As long as the working set for this query fits in memory, this is an appropriate solution for many applications 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.

12.4.4 Distributing tables by ID

As the number of tenants and the data stored for each tenant grows, query times will typically go up as the working set no longer fits in memory or CPU becomes a bottleneck. In this case, we can shard the data across many nodes using Citus. The first and most important choice we need to make when sharding is the distribution column. Let's start with a naive choice of using `event_id` for the event table and `page_id` for the `page` table:

```
-- naively use event_id and page_id as distribution columns

SELECT create_distributed_table('event', 'event_id');
SELECT create_distributed_table('page', 'page_id');
```

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

Across all shards of the page table (Q1):

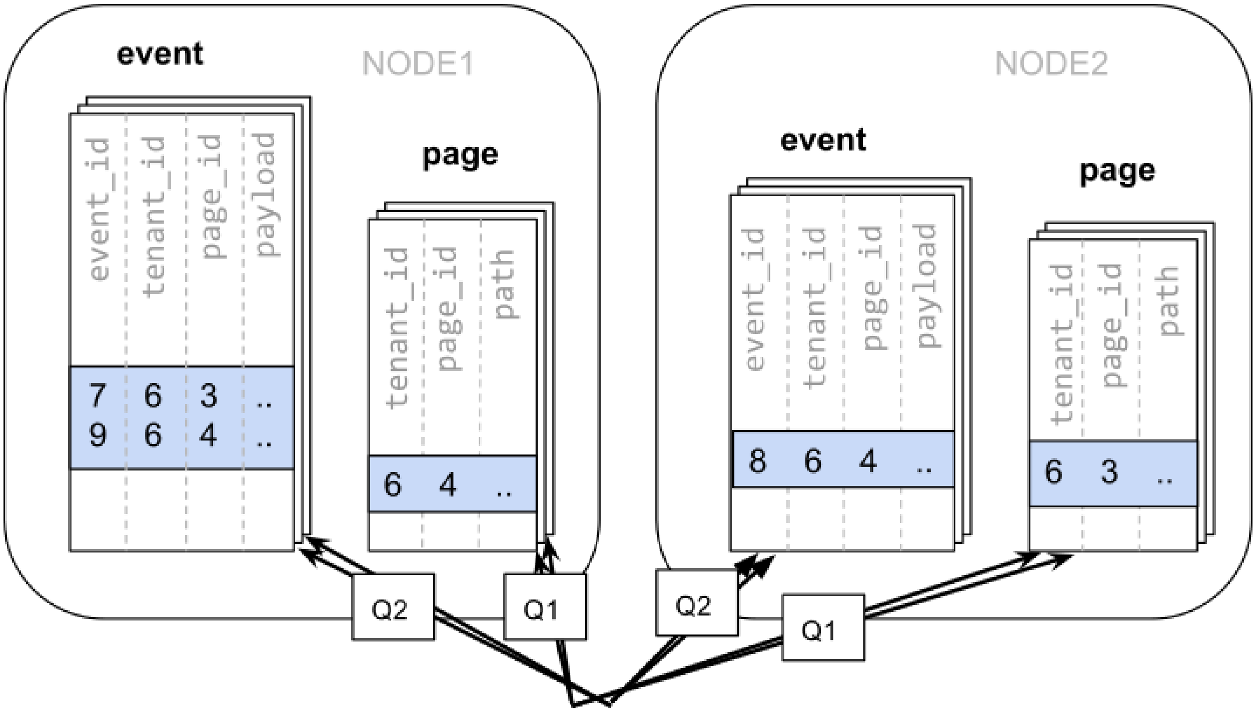
```
SELECT page_id FROM page WHERE path LIKE '/blog%' AND tenant_id = 6;
```

Across all shards of the event table (Q2):

```
SELECT page_id, count(*) AS count
FROM event
WHERE page_id IN (/*...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 Citus will do. However, this is only beneficial if the amount of work that the query does is substantially greater than the overhead of querying many shards. It's generally better to avoid doing such heavy lifting directly from the application, for example by *pre-aggregating* the data.

12.4.5 Distributing tables by tenant

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

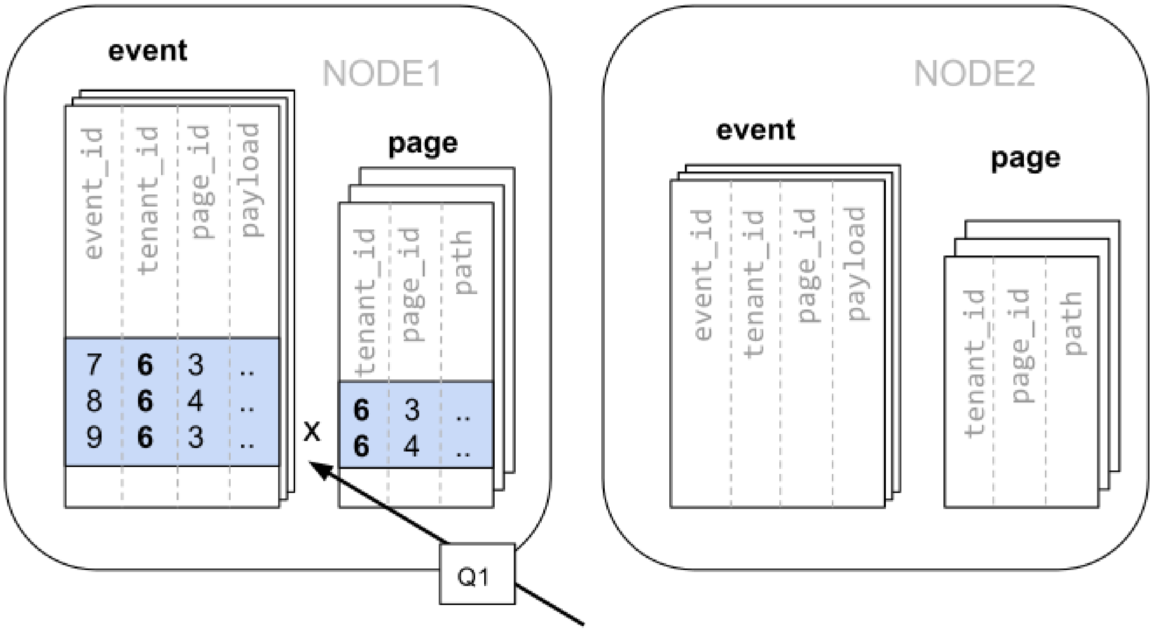
In Citus, rows with the same distribution column value are guaranteed to be on the same node. Each shard in a distributed table effectively has a set of co-located shards from other distributed tables that contain the same distribution column values (data for the same tenant). Starting over, we can create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

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

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

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

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



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

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

12.4.6 Co-location means better feature support

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

- Full SQL support for queries on a single set of co-located shards
- Multi-statement transaction support for modifications on a single set of co-located shards
- Aggregation through INSERT..SELECT
- Foreign keys
- Distributed outer joins
- Pushdown CTEs (requires PostgreSQL >=12)

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*.

12.4.7 Query Performance

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

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

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

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

CHAPTER

THIRTEEN

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.

13.1 Identify Distribution Strategy

13.1.1 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.

13.1.2 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:

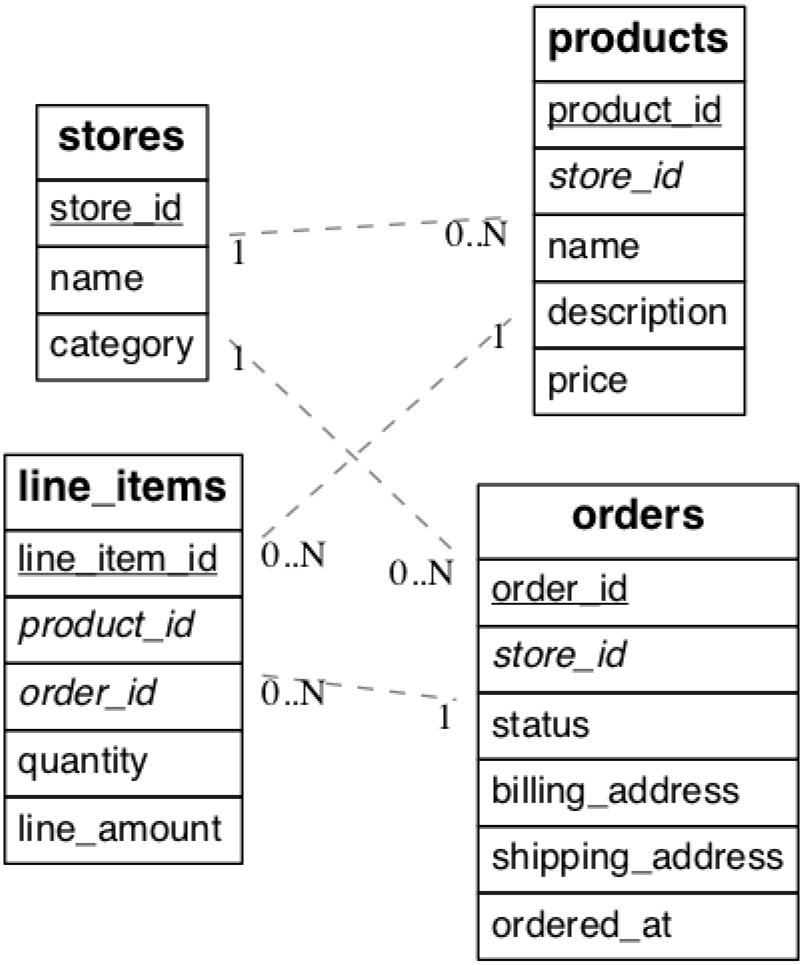


Fig. 1: (Underlined items are primary keys, italicized items are foreign keys.)

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.

13.2 Prepare Source Tables for Migration

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

13.2.1 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.

13.2.2 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;
```

Doing the whole table at once may cause too much load on the database and disrupt other queries. The backfill can done more slowly instead. One way to do that is to make a function that backfills small batches at a time, then call the function repeatedly with pg_cron.

```
-- the function to backfill up to one
-- thousand rows from line_items

CREATE FUNCTION backfill_batch()
RETURNS void LANGUAGE sql AS $$
  WITH batch AS (
    SELECT line_items_id, order_id
      FROM line_items
     WHERE store_id IS NULL
     LIMIT 10000
       FOR UPDATE
      SKIP LOCKED
  )
  UPDATE line_items AS li
     SET store_id = orders.store_id
    FROM batch, orders
   WHERE batch.line_item_id = li.line_item_id
     AND batch.order_id = orders.order_id;
```

(continues on next page)

(continued from previous page)

```
$$;

-- run the function every quarter hour
SELECT cron.schedule('*/15 * * * *', 'SELECT backfill_batch()');

-- ^^ note the return value of cron.schedule
```

Once the backfill is caught up, the cron job can be disabled:

```
-- assuming 42 is the job id returned
-- from cron.schedule

SELECT cron.unschedule(42);
```

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

13.3 Prepare Application for Citus

13.3.1 Set up Development Citus Cluster

When modifying the application to work with Citus, you'll need a database to test against. Follow the instructions to set up a *Single-Node Citus* of your choice.

Next dump a copy of the schema from your application's original database and restore the schema in the new development database.

```
# get schema from source db

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

# load schema into test db

psql postgres://user:pass@testhost:5432/db -f schema.sql
```

The schema should include a distribution key ("tenant id") in all tables you wish to distribute. Before pg_dumping the schema, be sure you have completed the step *Prepare Source Tables for Migration* from the previous section.

Include distribution column in keys

Citus *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.

Some of the libraries listed in the next section are able to help migrate the database schema to include the distribution column in keys. However, here is an example of the underlying SQL commands to turn the simple keys composite in the development database:

```
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);
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 from the previous section will look like this:

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

13.3.2 Add distribution key to queries

Once the distribution key is present on all appropriate tables, the application needs to include it in queries. The following steps should be done using a copy of the application running in a development environment, and testing against a Citus back-end. After the application is working with Citus we'll see how to migrate production data from the source database into a real Citus cluster.

- 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 Citus 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.

Cross-shard queries are supported, but in a multi-tenant application most queries should be targeted to a single node. For simple select, update, and delete queries this means that the *where* clause should filter by tenant id. Citus can then

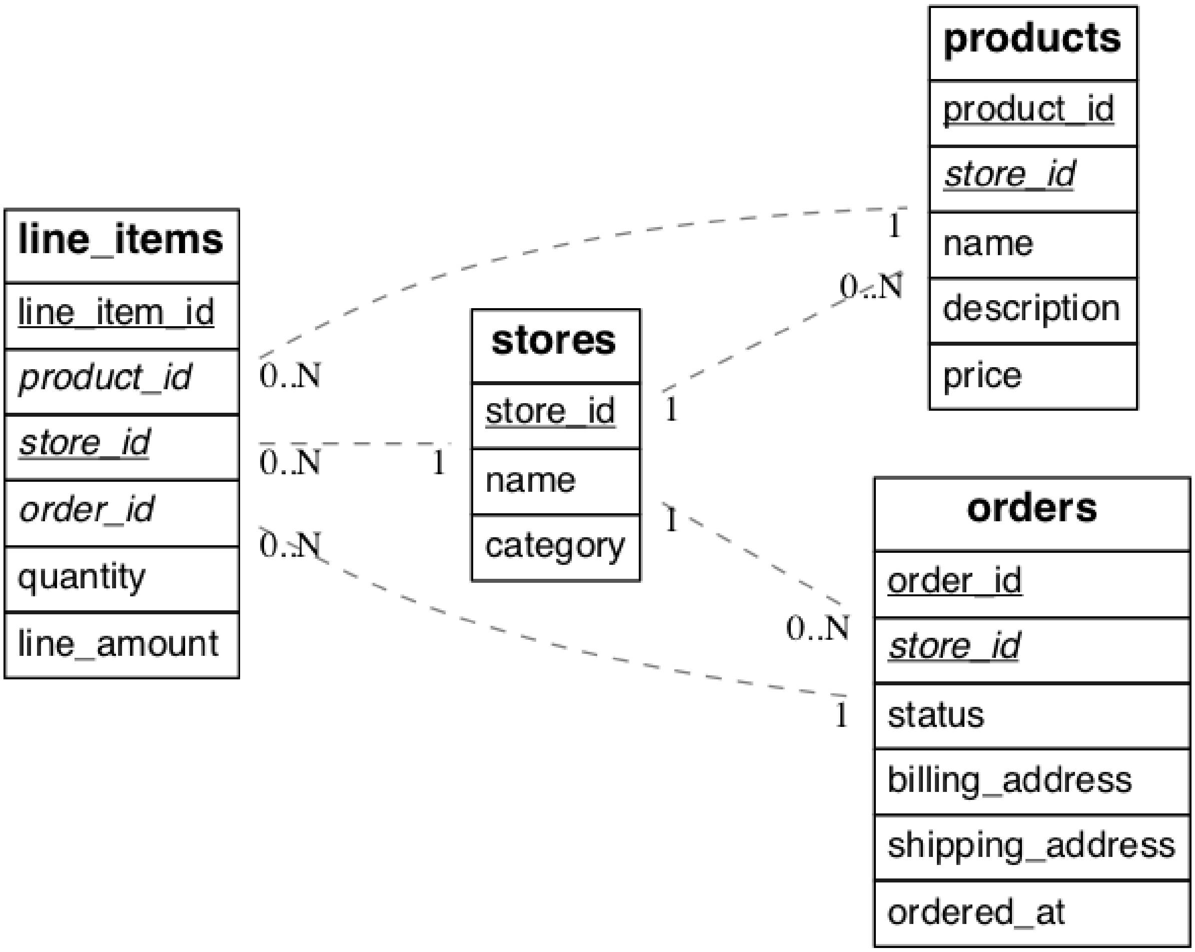


Fig. 2: (Underlined items are primary keys, italicized items are foreign keys.)

run these queries efficiently on a single node.

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

This section investigates how to migrate multi-tenant Rails applications to a Citus storage backend. We'll use the activerecord-multi-tenant Ruby gem for easier scale-out.

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

Preparing to scale-out a multi-tenant application

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

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

```
# typical pattern with multiple belongs_to relationships

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

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

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

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

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

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

2. Use UNIQUE constraints which include the tenant_id

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

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

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

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

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

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

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

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

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

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

Updating the Rails Application

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

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

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

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

Note: The library relies on the tenant id column to be present and non-null for all rows. However, it is often useful to have the library set the tenant id for *new* records, while backfilling missing tenant id values in existing records as a background task. This makes it easier to get started with activerecord-multi-tenant.

To support this, the library has a write-only mode, in which the tenant id column is not filtered in queries, but is set properly for new records. Include the following in a Rails initializer to enable it:

```
MultiTenant.enable_write_only_mode
```

Once you are ready to enforce tenancy, add a NOT NULL constraint to your tenant_id column and simply remove the initializer line.

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

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

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

  before_filter :set_customer_as_tenant

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

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

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

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

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

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

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

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

  def down
    drop_table :page_views
  end
end
```

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

Citus to scale out the data to multiple nodes.

Updating the Test Suite

If the test suite for your Rails application uses the `database_cleaner` gem to reset the test database between runs, be sure to use the “truncation” strategy rather than “transaction.” We have seen occasional failures during transaction rollbacks in the tests. The database_cleaner documentation has instructions for changing the cleaning strategy.

Continuous Integration

The easiest way to run a Citus cluster in continuous integration is by using the official Citus Docker containers. Here is how to do it on Circle CI in particular.

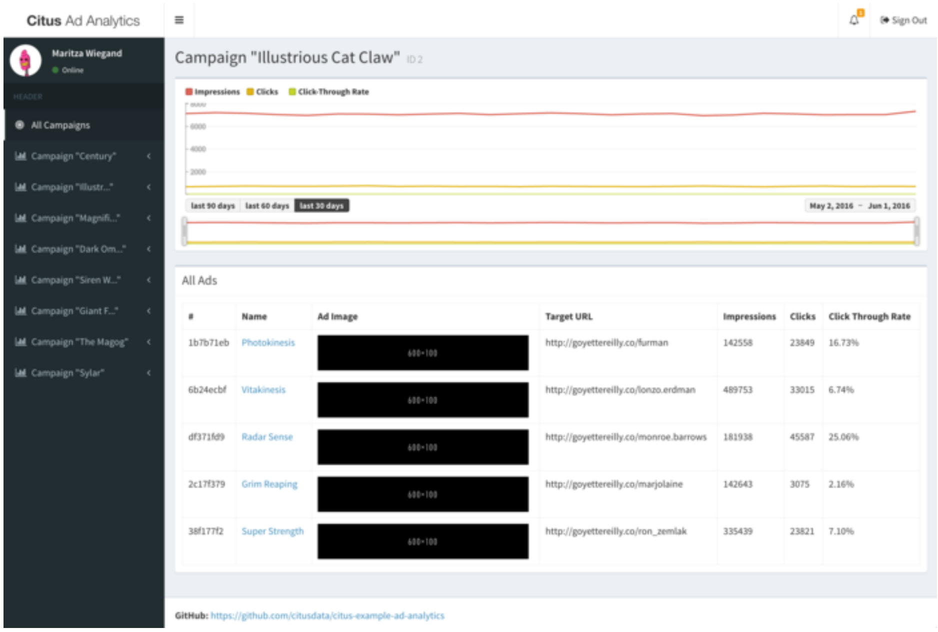
1. Copy https://github.com/citusdata/docker/blob/master/docker-compose.yml into the Rails project, and name it citus-docker-compose.yml.
2. Update the `steps:` section in `.circleci/config.yml`. This will start a coordinator and worker node:

```
steps:
  - setup_remote_docker:
      docker_layer_caching: true
  - run:
      name: Install Docker Compose
      command: |
        curl -L https://github.com/docker/compose/releases/download/1.19.0/docker-
↪compose-`uname -s`-`uname -m` > ~/docker-compose
        chmod +x ~/docker-compose
        mv ~/docker-compose /usr/local/bin/docker-compose

  - checkout

  - run:
      name: Starting Citus Cluster
      command: docker-compose -f citus-docker-compose.yml up -d
```

3. Have your test suite connect to the database in Docker, which will be on localhost:5432.

Example Application

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



Campaign "Illustrious Cat Claw" ID 2

All Ads

#	Name	Ad Image	Target URL	Impressions	Clicks	Click Through Rate
1b7b71eb	Photokinesis	600×100	http://goyettereilly.co/furman	142558	23849	16.73%
6b24ecbf	Vitakinesis	600×100	http://goyettereilly.co/lonzo.erdman	489753	33015	6.74%
df371fd9	Radar Sense	600×100	http://goyettereilly.co/monroe.barrows	181938	45587	25.06%
2c17f379	Grim Reaping	600×100	http://goyettereilly.co/marjolaine	142643	3075	2.16%
38f177f2	Super Strength	600×100	http://goyettereilly.co/ron_zemlak	335439	23821	7.10%

GitHub: https://github.com/citusdata/citus-example-ad-analytics

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

ASP.NET

In *Identify Distribution Strategy* we discussed the framework-agnostic database changes required for using Citus in the multi-tenant use case. The current section investigates how to build multi-tenant ASP.NET applications that work with a Citus storage backend.

Example App

To make this migration section concrete, let's consider a simplified version of StackExchange. For reference, the final result exists on Github.

Schema

We'll start with two tables:

```
CREATE TABLE tenants (
    id uuid NOT NULL,
    domain text NOT NULL,
    name text NOT NULL,
    description text NOT NULL,
    created_at timestamptz NOT NULL,
    updated_at timestamptz NOT NULL
);

CREATE TABLE questions (
    id uuid NOT NULL,
    tenant_id uuid NOT NULL,
```

(continues on next page)

(continued from previous page)

```
    title text NOT NULL,
    votes int NOT NULL,
    created_at timestamptz NOT NULL,
    updated_at timestamptz NOT NULL
);

ALTER TABLE tenants ADD PRIMARY KEY (id);
ALTER TABLE questions ADD PRIMARY KEY (id, tenant_id);
```

Each tenant of our demo application will connect via a different domain name. ASP.NET Core will inspect incoming requests and look up the domain in the `tenants` table. You could also look up tenants by subdomain (or any other scheme you want).

Notice how the `tenant_id` is also stored in the `questions` table. This will make it possible to *colocate* the data. With the tables created, use `create_distributed table` to tell Citus to shard on the tenant ID:

```
SELECT create_distributed_table('tenants', 'id');
SELECT create_distributed_table('questions', 'tenant_id');
```

Next include some test data.

```
INSERT INTO tenants VALUES (
    'c620f7ec-6b49-41e0-9913-08cfe81199af',
    'bufferoverflow.local',
    'Buffer Overflow',
    'Ask anything code-related!',
    now(),
    now());

INSERT INTO tenants VALUES (
    'b8a83a82-bb41-4bb3-bfaa-e923faab2ca4',
    'dboverflow.local',
    'Database Questions',
    'Figure out why your connection string is broken.',
    now(),
    now());

INSERT INTO questions VALUES (
    '347b7041-b421-4dc9-9e10-c64b8847fedf',
    'c620f7ec-6b49-41e0-9913-08cfe81199af',
    'How do you build apps in ASP.NET Core?',
    1,
    now(),
    now());

INSERT INTO questions VALUES (
    'a47ffcd2-635a-496e-8c65-c1cab53702a7',
    'b8a83a82-bb41-4bb3-bfaa-e923faab2ca4',
    'Using postgresql for multitenant data?',
    2,
    now(),
    now());
```

This completes the database structure and sample data. We can now move on to setting up ASP.NET Core.

ASP.NET Core project

If you don't have ASP.NET Core installed, install the .NET Core SDK from Microsoft. These instructions will use the dotnet CLI, but you can also use Visual Studio 2017 or newer if you are on Windows.

Create a new project from the MVC template with dotnet new:

```
dotnet new mvc -o QuestionExchange
cd QuestionExchange
```

You can preview the template site with dotnet run if you'd like. The MVC template includes almost everything you need to get started, but Postgres support isn't included out of the box. You can fix this by installing the Npgsql.EntityFrameworkCore.PostgreSQL package:

```
dotnet add package Npgsql.EntityFrameworkCore.PostgreSQL
```

This package adds Postgres support to Entity Framework Core, the default ORM and database layer in ASP.NET Core. Open the Startup.cs file and add these lines anywhere in the ConfigureServices method:

```
var connectionString = "connection-string";

services.AddEntityFrameworkNpgsql()
    .AddDbContext<AppDbContext>(options => options.UseNpgsql(connectionString));
```

You'll also need to add these declarations at the top of the file:

```
using Microsoft.EntityFrameworkCore;
using QuestionExchange.Models;
```

Replace connection-string with your Citus connection string. Mine looks like this:

```
Server=myformation.db.citusdata.com;Port=5432;Database=citus;Userid=citus;
↪Password=mypassword;SslMode=Require;Trust Server Certificate=true;
```

Note: You can use the Secret Manager to avoid storing your database credentials in code (and accidentally checking them into source control).

Next, you'll need to define a database context.

Adding Tenancy to App

Define the Entity Framework Core context and models

The database context class provides an interface between your code and your database. Entity Framework Core uses it to understand what your data schema looks like, so you'll need to define what tables are available in your database.

Create a file called AppDbContext.cs in the project root, and add the following code:

```
using System.Linq;
using Microsoft.EntityFrameworkCore;
using QuestionExchange.Models;
namespace QuestionExchange
```

(continues on next page)

(continued from previous page)

```
{
    public class AppDbContext : DbContext
    {
        public AppDbContext(DbContextOptions<AppDbContext> options)
            : base(options)
        {
        }

        public DbSet<Tenant> Tenants { get; set; }

        public DbSet<Question> Questions { get; set; }
    }
}
```

The two DbSet properties specify which C# classes to use to model the rows of each table. You'll create these classes next. Before you do that, add a new method below the Questions property:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    var mapper = new Npgsql.NpgsqlSnakeCaseNameTranslator();
    var types = modelBuilder.Model.GetEntityTypes().ToList();

    // Refer to tables in snake_case internally
    types.ForEach(e => e.Relational().TableName = mapper.TranslateMemberName(e.
↪Relational().TableName));

    // Refer to columns in snake_case internally
    types.SelectMany(e => e.GetProperties())
        .ToList()
        .ForEach(p => p.Relational().ColumnName = mapper.TranslateMemberName(p.
↪Relational().ColumnName));
}
```

C# classes and properties are PascalCase by convention, but your Postgres tables and columns are lowercase (and snake_case). The OnModelCreating method lets you override the default name translation and let Entity Framework Core know how to find the entities in your database.

Now you can add classes that represent tenants and questions. Create a Tenant.cs file in the Models directory:

```
using System;

namespace QuestionExchange.Models
{
    public class Tenant
    {
        public Guid Id { get; set; }

        public string Domain { get; set; }

        public string Name { get; set; }

        public string Description { get; set; }

```

(continues on next page)

(continued from previous page)

```
        public DateTimeOffset CreatedAt { get; set; }

        public DateTimeOffset UpdatedAt { get; set; }
    }
}
```

And a Question.cs file, also in the Models directory:

```
using System;

namespace QuestionExchange.Models
{
    public class Question
    {
        public Guid Id { get; set; }

        public Tenant Tenant { get; set; }

        public string Title { get; set; }

        public int Votes { get; set; }

        public DateTimeOffset CreatedAt { get; set; }

        public DateTimeOffset UpdatedAt { get; set; }
    }
}
```

Notice the Tenant property. In the database, the question table contains a tenant_id column. Entity Framework Core is smart enough to figure out that this property represents a one-to-many relationship between tenants and questions. You'll use this later when you query your data.

So far, you've set up Entity Framework Core and the connection to Citus. The next step is adding multi-tenant support to the ASP.NET Core pipeline.

Install SaasKit

SaasKit is an excellent piece of open-source ASP.NET Core middleware. This package makes it easy to make your Startup request pipeline tenant-aware, and is flexible enough to handle many different multi-tenancy use cases.

Install the SaasKit.Multitenancy package:

```
dotnet add package SaasKit.Multitenancy
```

SaasKit needs two things to work: a tenant model and a tenant resolver. You already have the former (the Tenant class you created earlier), so create a new file in the project root called CachingTenantResolver.cs:

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Caching.Memory;
```

(continues on next page)

(continued from previous page)

```
using Microsoft.Extensions.Logging;
using SaasKit.Multitenancy;
using QuestionExchange.Models;

namespace QuestionExchange
{
    public class CachingTenantResolver : MemoryCacheTenantResolver<Tenant>
    {
        private readonly AppDbContext _context;

        public CachingTenantResolver(
            AppDbContext context, IMemoryCache cache, ILoggerFactory loggerFactory)
             : base(cache, loggerFactory)
        {
            _context = context;
        }

        // Resolver runs on cache misses
        protected override async Task<TenantContext<Tenant>> ResolveAsync(HttpContext␣
↪context)
        {
            var subdomain = context.Request.Host.Host.ToLower();

            var tenant = await _context.Tenants
                .FirstOrDefaultAsync(t => t.Domain == subdomain);

            if (tenant == null) return null;

            return new TenantContext<Tenant>(tenant);
        }

        protected override MemoryCacheEntryOptions CreateCacheEntryOptions()
            => new MemoryCacheEntryOptions().SetAbsoluteExpiration(TimeSpan.
↪FromHours(2));

        protected override string GetContextIdentifier(HttpContext context)
            => context.Request.Host.Host.ToLower();

        protected override IEnumerable<string> GetTenantIdentifiers(TenantContext<Tenant>
↪ context)
            => new string[] { context.Tenant.Domain };
    }
}
```

The `ResolveAsync` method does the heavy lifting: given an incoming request, it queries the database and looks for a tenant matching the current domain. If it finds one, it passes a `TenantContext` back to SaasKit. All of tenant resolution logic is totally up to you - you could separate tenants by subdomains, paths, or anything else you want.

This implementation uses a tenant caching strategy so you don't hit the database with a tenant lookup on every incoming request. After the first lookup, tenants are cached for two hours (you can change this to whatever makes sense).

With a tenant model and a tenant resolver ready to go, open up the `Startup` class and add this line anywhere inside the `ConfigureServices` method:

```
services.AddMultitenancy<Tenant, CachingTenantResolver>();
```

Next, add this line to the `Configure` method, below `UseStaticFiles` but **above** `UseMvc`:

```
app.UseMultitenancy<Tenant>();
```

The `Configure` method represents your actual request pipeline, so order matters!

Update views

Now that all the pieces are in place, you can start referring to the current tenant in your code and views. Open up the `Views/Home/Index.cshtml` view and replace the whole file with this markup:

```
@inject Tenant Tenant
@model QuestionListViewModel

@{
    ViewData["Title"] = "Home Page";
}

<div class="row">
    <div class="col-md-12">
        <h1>Welcome to <strong>@Tenant.Name</strong></h1>
        <h3>@Tenant.Description</h3>
    </div>
</div>

<div class="row">
    <div class="col-md-12">
        <h4>Popular questions</h4>
        <ul>
            @foreach (var question in Model.Questions)
            {
                <li>@question.Title</li>
            }
        </ul>
    </div>
</div>
```

The `@inject` directive gets the current tenant from SaasKit, and the `@model` directive tells ASP.NET Core that this view will be backed by a new model class (that you'll create). Create the `QuestionListViewModel.cs` file in the Models directory:

```
using System.Collections.Generic;

namespace QuestionExchange.Models
{
    public class QuestionListViewModel
    {
    public IEnumerable<Question> Questions { get; set; }
    }
}
```

Query the database

The HomeController is responsible for rendering the index view you just edited. Open it up and replace the Index() method with this one:

```
public async Task<IActionResult> Index()
{
    var topQuestions = await _context
        .Questions
        .Where(q => q.Tenant.Id == _currentTenant.Id)
        .OrderByDescending(q => q.UpdatedAt)
        .Take(5)
        .ToArrayAsync();

    var viewModel = new QuestionListViewModel
    {
        Questions = topQuestions
    };

    return View(viewModel);
}
```

This query gets the newest five questions for this tenant (granted, there's only one question right now) and populates the view model.

> For a large application, you'd typically put data access code in a service or repository layer and keep it out of your controllers. This is just a simple example!

The code you added needs _context and _currentTenant, which aren't available in the controller yet. You can make these available by adding a constructor to the class:

```
public class HomeController : Controller
{
    private readonly AppDbContext _context;
    private readonly Tenant _currentTenant;

    public HomeController(AppDbContext context, Tenant tenant)
    {
        _context = context;
        _currentTenant = tenant;
    }

    // Existing code...
```

To keep the compiler from complaining, add this declaration at the top of the file:

```
using Microsoft.EntityFrameworkCore;
```

Test the app

The test tenants you added to the database were tied to the (fake) domains bufferoverflow.local and dboverflow.local. You'll need to edit your hosts file to test these on your local machine:

```
127.0.0.1 bufferoverflow.local
127.0.0.1 dboverflow.local
```

Start your project with dotnet run or by clicking Start in Visual Studio and the application will begin listening on a URL like localhost:5000. If you visit that URL directly, you'll see an error because you haven't set up any default tenant behavior yet.

Instead, visit http://bufferoverflow.local:5000 and you'll see one tenant of your multi-tenant application! Switch to http://dboverflow.local:5000 to view the other tenant. Adding more tenants is now a simple matter of adding more rows in the tenants table.

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.

Other (SQL 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'

```

(continues on next page)

(continued from previous page)

```
-- 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'
```

13.3.3 Enable Secure Connections

Clients should connect to Citus with SSL to protect information and prevent man-in-the-middle attacks.

13.3.4 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 Citus cluster and switch over the production app. We have techniques to minimize downtime for this step.

13.4 Migrate Production Data

At this time, having updated the database schema and application queries to work with Citus, you're ready for the final step. It's time to migrate data to the Citus 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.

13.4.1 Small Database Migration

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 from your development database:

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

2. Connect to the Citus 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 original production 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 Citus 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!

13.4.2 Big Database Migration

Larger environments can use tools like Citus Warp, Debezium, Striim or HVR for online replication. These tools allow you to stream changes from a PostgreSQL source database into our cloud_topic on Microsoft Azure. It's as if the application automatically writes to two databases rather than one, except with perfect transactional logic.

For this process we strongly recommend contacting us by opening a support request. To do the replication, 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 replication 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 your development database (the Citus database you used for locally testing the application). 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

Identify the IP address for the destination coordinator node. 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 Azure portal. An engineer will connect to your database to perform an intial database dump, open a replication slot, and begin the replication. We can include/exclude your choice of tables in the migration.

During the first stage of replication, 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 initial dump/restore is complete and replication begins, then the database will be able to archive unused WAL files again.

As the replication proceeds, **pay attention to disk usage on the source database.** If there is a data-type mismatch between the source and destination, or other unexpe cted schema change, the replication can stall. The replication slot can grow indefini tely on the source during a prolonged stall, leading to potential crashes.

Because of the potential for replication stalls, we strongly recommend minimizing sc hema changes while doing replication. If an invasive schema change is required, you will need to stop and try again.

Steps to make an invasive schema change:

1. Ask a support engineer to stop the replication.
2. Change the schema on the source database.
3. Change the schema on the destination database.
4. Begin again.

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.

CHAPTER

FOURTEEN

SQL REFERENCE

14.1 Creating and Modifying Distributed Objects (DDL)

14.1.1 Creating And Distributing Tables

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

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

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

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

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

This example would create a total of citus.shard_count number of shards where each shard owns a portion of a hash token space. Once the shards are created, this function saves all distributed metadata on the coordinator.

Each created shard is assigned a unique 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, you can change it to a reference table by running:

```
SELECT undistribute_table('table_name');
SELECT create_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');
```

(continues on next page)

(continued from previous page)

```
NOTICE:  Copying data from local table...
NOTICE:  copying the data has completed
DETAIL:  The local data in the table is no longer visible, but is still on disk.
HINT:  To remove the local data, run: SELECT truncate_local_data_after_distributing_
↪table($$public.series$$)
 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.

When distributing tables A and B, where A has a foreign key to B, distribute the key destination table B first. Doing it in the wrong order will cause an error:

```
ERROR:  cannot create foreign key constraint
DETAIL:  Referenced table must be a distributed table or a reference table.
```

If it's not possible to distribute in the correct order then drop the foreign keys, distribute the tables, and recreate the foreign keys.

After the tables are distributed, use the *truncate_local_data_after_distributing_table* function to remove local data. Leftover local data in distributed tables is inaccessible to Citus queries, and can cause irrelevant constraint violations on the coordinator.

When migrating data from an external database, such as from Amazon RDS to our cloud_topic, first create the Citus distributed tables via `create_distributed_table`, then copy the data into the table. Copying into distributed tables avoids running out of space on the coordinator node.

14.1.2 Co-Locating Tables

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

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

```
-- 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 `colocated_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.

14.1.3 Dropping Tables

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

```
DROP TABLE github_events;
```

14.1.4 Modifying Tables

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

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 Citus 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:  cannot execute ALTER TABLE command involving partition column
*/
```

As a workaround, you can consider *changing the distribution column*, updating it, and changing it back.

Adding/Removing Constraints

Using Citus allows you to continue to enjoy the safety of a relational database, including database constraints (see the PostgreSQL docs). Due to the nature of distributed systems, Citus will not cross-reference uniqueness constraints or referential integrity between worker nodes.

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.

Foreign keys may be created in these situations:

- between two local (non-distributed) tables,
- between two reference tables,
- between reference tables and local tables (by default enabled, via *citus.enable_local_reference_table_foreign_keys (boolean)*),
- between two *colocated* distributed tables when the key includes the distribution column, or
- as a distributed table referencing a *reference table*

Foreign keys from reference tables to distributed tables are not supported.

Citus supports all referential actions on foreign keys from local to reference tables, but does not support support `ON DELETE/UPDATE CASCADE` in the reverse direction (reference to local).

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.

```

(continues on next page)

(continued from previous page)

```
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;
```

Using NOT VALID Constraints

In some situations it can be useful to enforce constraints for new rows, while allowing existing non-conforming rows to remain unchanged. Citus supports this feature for CHECK constraints and foreign keys, using PostgreSQL's "NOT VALID" constraint designation.

For example, consider an application which stores user profiles in a *reference table*.

```
-- we're using the "text" column type here, but a real application
-- might use "citext" which is available in a postgres contrib module

CREATE TABLE users ( email text PRIMARY KEY );
SELECT create_reference_table('users');
```

In the course of time imagine that a few non-addresses get into the table.

```
INSERT INTO users VALUES
   ('foo@example.com'), ('hacker12@aol.com'), ('lol');
```

We would like to validate the addresses, but PostgreSQL does not ordinarily allow us to add a CHECK constraint that fails for existing rows. However, it *does* allow a constraint marked not valid:

```
ALTER TABLE users
ADD CONSTRAINT syntactic_email
CHECK (email ~
   '^[a-zA-Z0-9.!#$%&''*+/=?^_`{|}~-]+@[a-zA-Z0-9](?:[a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?(?:\.
↪[a-zA-Z0-9](?:[a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?)*$'
) NOT VALID;
```

This succeeds, and new rows are protected.

```
INSERT INTO users VALUES ('fake');

/*
ERROR:  new row for relation "users_102010" violates
        check constraint "syntactic_email_102010"
DETAIL:  Failing row contains (fake).
*/
```

Later, during non-peak hours, a database administrator can attempt to fix the bad rows and re-validate the constraint.

```
-- later, attempt to validate all rows
ALTER TABLE users
VALIDATE CONSTRAINT syntactic_email;
```

The PostgreSQL documentation has more information about NOT VALID and VALIDATE CONSTRAINT in the ALTER TABLE section.

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);
```

14.1.5 Types and Functions

Creating custom SQL types and user-defined functions propogates to worker nodes. However, creating such database objects in a transaction with distributed operations involves tradeoffs.

Citus parallelizes operations such as `create_distributed_table()` across shards using multiple connections per worker. Whereas, when creating a database object, Citus propagates it to worker nodes using a single connection per worker. Combining the two operations in a single transaction may cause issues, because the parallel connections will not be able to see the object that was created over a single connection but not yet committed.

Consider a transaction block that creates a type, a table, loads data, and distributes the table:

```
BEGIN;

-- type creation over a single connection:
CREATE TYPE coordinates AS (x int, y int);
CREATE TABLE positions (object_id text primary key, position coordinates);

-- data loading thus goes over a single connection:
SELECT create_distributed_table('positions', 'object_id');
\COPY positions FROM 'positions.csv'

COMMIT;
```

Prior to Citus 11.0, Citus would defer creating the type on the worker nodes, and commit it separately when creating the distributed table. This enabled the data copying in `create_distributed_table()` to happen in parallel. However, it also meant that the type was not always present on the Citus worker nodes – or if the transaction rolled back, the type would remain on the worker nodes.

With Citus 11.0, the default behaviour changes to prioritize schema consistency between coordinator and worker nodes. The new behavior has a downside: if object propagation happens after a parallel command in the same transaction, then the transaction can no longer be completed, as highlighted by the ERROR in the code block below:

```
BEGIN;
CREATE TABLE items (key text, value text);
-- parallel data loading:
SELECT create_distributed_table('items', 'key');
\COPY items FROM 'items.csv'
CREATE TYPE coordinates AS (x int, y int);

ERROR:  cannot run type command because there was a parallel operation on a distributed␣
↪table in the transaction
```

If you run into this issue, there are two simple workarounds:

1. Use set `citus.create_object_propagation` to `deferred` to return to the old object propagation behavior, in which case there may be some inconsistency between which database objects exist on different nodes.
2. Use set `citus.multi_shard_modify_mode` to `sequential` to disable per-node parallelism. Data load in the same transaction might be slower.

14.1.6 Manual Modification

Most DDL commands are auto-propagated. For any others, you can propagate the changes manually. See *Manual Query Propagation*.

14.2 Ingesting, Modifying Data (DML)

14.2.1 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, Citus determines the right shard to which the insert should be routed to. Then, the query is forwarded to the right shard, and the remote insert command is executed on all the replicas of that shard.

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'
```

(continues on next page)

(continued from previous page)

```
  ), (
    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, the easiest way to do *distributed rollups*.

In Citus there are three 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.

The second way of executing an `INSERT ... SELECT` statement is by repartitioning the results of the result set into chunks, and sending those chunks among workers to matching destination table shards. Each worker node can insert the values into local destination shards.

The repartitioning optimization can happen when the SELECT query doesn't require a merge step on the coordinator. It doesn't work with the following SQL features, which require a merge step:

- ORDER BY
- LIMIT
- OFFSET
- GROUP BY when distribution column is not part of the group key
- Window functions when partitioning by a non-distribution column in the source table(s)
- Joins between non-colocated tables (i.e. repartition joins)

When the source and destination tables are not colocated, and the repartition optimization cannot be applied, then Citus uses the third way of executing `INSERT ... SELECT`. It selects the results from worker nodes, and pulls the data up to the coordinator node. The coordinator redirects rows back down to the appropriate shard. Because all the data must pass through a single node, this method is not as efficient.

When in doubt about which method Citus is using, use the EXPLAIN command, as described in *PostgreSQL tuning*. When the target table has a very large shard count, it may be wise to disable repartitioning, see *citus.enable_repartitioned_insert_select (boolean)*.

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 (note that this data requires the database to have UTF8 encoding):

```
\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.

14.3 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. Citus users often wait for a certain time period after the end of day to run a query like this, to accommodate late arriving data.

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

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

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;
```

14.3.1 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, Citus 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, Citus 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.

14.3.2 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.

14.4 Querying Distributed Tables (SQL)

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

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

14.4.1 Aggregate Functions

Citus supports and parallelizes most aggregate functions supported by PostgreSQL, including custom user-defined aggregates. Aggregates execute using one of three methods, in this order of preference:

1. When the aggregate is grouped by a table's distribution column, Citus can push down execution of the entire query to each worker. All aggregates are supported in this situation and execute in parallel on the worker nodes. (Any custom aggregates being used must be installed on the workers.)

2. When the aggregate is *not* grouped by a table's distribution column, Citus can still optimize on a case-by-case basis. Citus has internal rules for certain aggregates like sum(), avg(), and count(distinct) that allows it to rewrite queries for *partial aggregation* on workers. For instance, to calculate an average, Citus obtains a sum and a count from each worker, and then the coordinator node computes the final average.

   Full list of the special-case aggregates:

   > avg, min, max, sum, count, array_agg, jsonb_agg, jsonb_object_agg, json_agg, json_object_agg, bit_and, bit_or, bool_and, bool_or, every, hll_add_agg, hll_union_agg, topn_add_agg, topn_union_agg, any_value, tdigest(double precision, int), tdigest_percentile(double precision, int, double precision), tdigest_percentile(double precision, int, double precision[]), tdigest_percentile(tdigest, double precision), tdigest_percentile(tdigest, double precision[]), tdigest_percentile_of(double precision, int, double precision), tdigest_percentile_of(double precision, int, double precision[]), tdigest_percentile_of(tdigest, double precision), tdigest_percentile_of(tdigest, double precision[])

3. Last resort: pull all rows from the workers and perform the aggregation on the coordinator node. When the aggregate is not grouped on a distribution column, and is not one of the predefined special cases, then Citus falls back to this approach. It causes network overhead, and can exhaust the coordinator's resources if the data set to be aggregated is too large. (It's possible to disable this fallback, see below.)

Beware that small changes in a query can change execution modes, causing potentially surprising inefficiency. For example `sum(x)` grouped by a non-distribution column could use distributed execution, while `sum(distinct x)` has to pull up the entire set of input records to the coordinator.

All it takes is one column to hurt the execution of a whole query. In the example below, if `sum(distinct value2)` has to be grouped on the coordinator, then so will `sum(value1)` even if the latter was fine on its own.

```sql
SELECT sum(value1), sum(distinct value2) FROM distributed_table;
```

To avoid accidentally pulling data to the coordinator, you can set a GUC:

```
SET citus.coordinator_aggregation_strategy TO 'disabled';
```

Note that disabling the coordinator aggregation strategy will prevent "type three" aggregate queries from working at all.

Count (Distinct) Aggregates

Citus supports count(distinct) aggregates in several ways. If the count(distinct) aggregate is on the distribution column, Citus can directly push down the query to the workers. If not, Citus 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 by simply running the below command from the coordinator

   ```
   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,
```

(continues on next page)

(continued from previous page)

```
    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.

```
-- run below command from coordinator, it will be propagated to the worker nodes as well
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.

```sql
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.

Percentile Calculations

Finding an exact percentile over a large number of rows can be prohibitively expensive, because all rows must be transferred to the coordinator for final sorting and processing. Finding an approximation, on the other hand, can be done in parallel on worker nodes using a so-called *sketch algorithm*. The coordinator node then combines compressed summaries into the final result rather than reading through the full rows.

A popular sketch algorithm for percentiles uses a compressed data structure called *t-digest*, and is available for PostgreSQL in the tdigest extension. Citus has integrated support for this extension.

Here's how to use t-digest in Citus:

1. Download and install the tdigest extension on all PostgreSQL nodes (the coordinator and all the workers). The tdigest extension github repository has installation instructions.
2. Create the tdigest extension within the database. Run the following command on the coordinator:

   ```sql
   CREATE EXTENSION tdigest;
   ```

   The coordinator will propagate the command to the workers as well.

When any of the aggregates defined in the extension are used in queries, Citus will rewrite the queries to push down partial tdigest computation to the workers where applicable.

T-digest accuracy can be controlled with the `compression` argument passed into aggregates. The trade-off is accuracy vs the amount of data shared between workers and the coordinator. For a full explanation of how to use the aggregates in the tdigest extension, have a look at the documentation on the official tdigest github repository.

14.4.2 Limit Pushdown

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

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

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

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

14.4.3 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.

14.4.4 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 Citus 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 Citus to prune away shard pairs which cannot produce matching join keys. The joins between remaining shard pairs are executed in parallel on the workers and then the results are returned to the coordinator.

Note: 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.

Reference tables can also join with tables local to the coordinator node.

Repartition joins

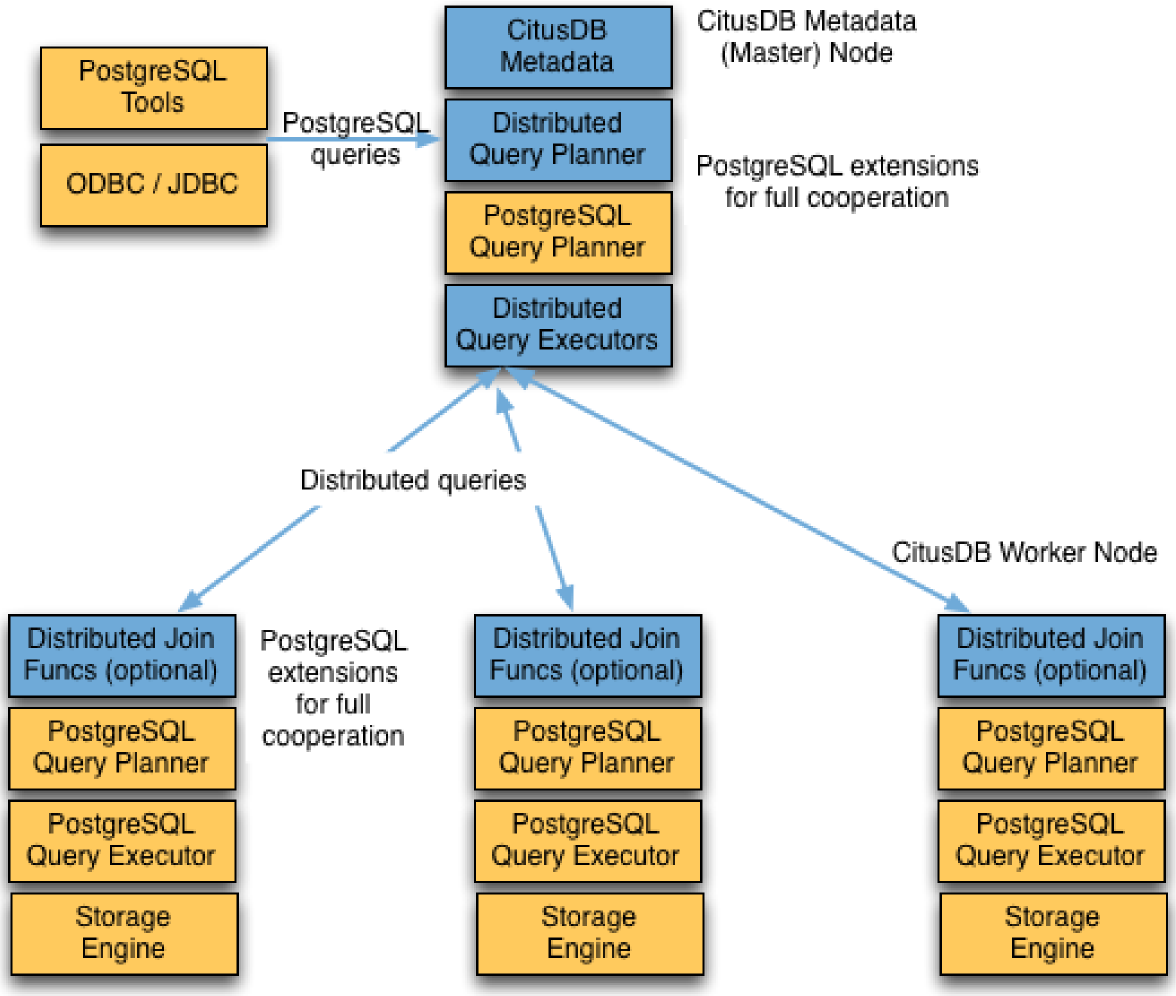
In some cases, you may need to join two tables on columns other than the distribution column. For such cases, Citus also allows joining on non-distribution key columns by dynamically repartitioning the tables for the query.

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

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

14.5 Query Processing

A Citus cluster consists of a coordinator instance and multiple worker instances. The data is sharded 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.


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

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.

14.5.1 Distributed Query Planner

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

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

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

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

14.5.2 Distributed Query Executor

Citus's distributed executor runs distributed query plans and handles failures. The executor is well suited for getting fast responses to queries involving filters, aggregations and co-located joins, as well as running single-tenant queries with full SQL coverage. It opens one connection per shard to the workers as needed 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.

Subquery/CTE Push-Pull Execution

If necessary Citus 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 Citus to support a greater variety of SQL constructs.

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 `page_views` table partitioned by `page_id`. To query the number of visitor hosts 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 hosts.

```
SELECT page_id, count(distinct host_ip)
FROM page_views
WHERE page_id IN (
  SELECT page_id
  FROM page_views
  GROUP BY page_id
  ORDER BY count(*) DESC
  LIMIT 20
)
GROUP BY page_id;
```

The executor would like to run a fragment of this query against each shard by page_id, counting distinct host_ips, 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 Citus 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 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 (Citus Adaptive)  (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
```

(continues on next page)

(continued from previous page)

```
              ->  Custom Scan (Citus Adaptive)  (cost=0.00..0.00 rows=0 width=0)
                Task Count: 32
                Tasks Shown: One of 32
                ->  Task
                  Node: host=localhost port=9701 dbname=postgres
                  ->  HashAggregate  (cost=54.70..56.70 rows=200 width=12)
                    Group Key: page_id
                    ->  Seq Scan on page_views_102008 page_views  (cost=0.00..43.47␣
↪rows=2247 width=4)
      Task Count: 32
      Tasks Shown: One of 32
      ->  Task
        Node: host=localhost port=9701 dbname=postgres
        ->  HashAggregate  (cost=84.50..86.75 rows=225 width=36)
          Group Key: page_views.page_id, page_views.host_ip
          ->  Hash Join  (cost=17.00..78.88 rows=1124 width=36)
            Hash Cond: (page_views.page_id = intermediate_result.page_id)
            ->  Seq Scan on page_views_102008 page_views  (cost=0.00..43.47 rows=2247␣
↪width=36)
            ->  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 (Citus Adaptive)  (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 (Citus Adaptive)  (cost=0.00..0.00 rows=0 width=0)
                Task Count: 32
                Tasks Shown: One of 32
                ->  Task
                  Node: host=localhost port=9701 dbname=postgres
                  ->  HashAggregate  (cost=54.70..56.70 rows=200 width=12)
```

(continues on next page)

(continued from previous page)

```
                    Group Key: page_id
                    ->  Seq Scan on page_views_102008 page_views  (cost=0.00..43.47␣
↪rows=2247 width=4)
.
```

Worker nodes run the above for each of the thirty-two shards (Citus 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=9701 dbname=postgres
        ->  HashAggregate  (cost=84.50..86.75 rows=225 width=36)
          Group Key: page_views.page_id, page_views.host_ip
          ->  Hash Join  (cost=17.00..78.88 rows=1124 width=36)
            Hash Cond: (page_views.page_id = intermediate_result.page_id)
.
```

Citus starts another executor job in this second subtree. It's going to count distinct hosts in page_views. 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 page_views_102008 page_views  (cost=0.00..43.47 rows=2247␣
↪width=36)
            ->  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 Citus executed the query in multiple steps with a distributed subplan, and how you can use EXPLAIN to learn about distributed query execution.

14.5.3 PostgreSQL planner and executor

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

14.6 Manual Query Propagation

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

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. Citus provides utility functions to propagate SQL to workers, shards, or colocated 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 Citus otherwise does not run natively. Use them carefully to avoid data inconsistency and deadlocks.

14.6.1 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.

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

To run on all nodes – both workers and the coordinator – use `run_command_on_all_nodes()`.

Note: This command shouldn't be used to create database objects on the workers, as doing so will make it harder to add worker nodes in an automated fashion.

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.

14.6.2 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)::regclass = '%s'::regclass
```

(continues on next page)

(continued from previous page)

```
        AND n.nspname NOT IN ('citus', 'pg_toast', 'pg_catalog')
    $cmd$
  );
```

A useful companion to `run_command_on_shards` is `run_command_on_colocated_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 synchronize them so that every time little_vals
-- are created, big_vals appear with double the value
--
-- First we make a trigger function, which will
-- take the destination table placement as an argument
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;

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

14.6.3 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!

14.7 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 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:

- SELECT … FOR UPDATE work in single-shard queries only
- TABLESAMPLE work in single-shard queries only
- Correlated subqueries are supported only when the correlation is on the *Distribution Column*.
- Outer joins between distributed tables are only supported on the *Distribution Column*
- Recursive CTEs work in single-shard queries only
- Grouping sets work in single-shard queries only
- The SQL MERGE command is supported in the following combinations of *Table Types*:

Target	Source	Support	Comments
Local	Local	Yes	
Local	Distributed	No	
Local	Reference	No	
Distributed	Local	No	
Distributed	Distributed	Yes	Only co-located tables.
Distributed	Reference	No	Feature in development.
Reference	N/A	No	Reference table as target is not allowed.

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.

14.7.1 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.

Work around limitations using CTEs

When a SQL query is unsupported, one way to work around it is using CTEs, which use what we call pull-push execution.

```
SELECT * FROM dist WHERE EXISTS (SELECT 1 FROM local WHERE local.a = dist.a);
/*
ERROR:  direct joins between distributed and local tables are not supported
HINT:  Use CTE's or subqueries to select from local tables and use them in joins
*/
```

To work around this limitation, you can turn the query into a router query by wrapping the distributed part in a CTE

```
WITH cte AS (SELECT * FROM dist)
SELECT * FROM cte WHERE EXISTS (SELECT 1 FROM local WHERE local.a = cte.a);
```

Remember that the coordinator will send the results of the 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*.

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 using grouping sets on a distributed table.

In our *real-time analytics tutorial* we created a table called `github_events`, distributed by the column `user_id`. Let's query it and find the earliest events for a preselected set of repos, grouped by combinations of event type and event publicity. A convenient way to do this is with grouping sets. However, as mentioned, this feature is not yet supported in distributed queries:

```
-- this won't work

  SELECT repo_id, event_type, event_public,
         grouping(event_type, event_public),
         min(created_at)
    FROM github_events
   WHERE repo_id IN (8514, 15435, 19438, 21692)
GROUP BY repo_id, ROLLUP(event_type, event_public);
```

```
ERROR:  could not run distributed query with GROUPING
HINT:  Consider using an equality filter on the distributed table's partition column.
```

There is a 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, event_type, event_public, created_at
    FROM github_events
       WHERE repo_id IN (8514, 15435, 19438, 21692)
    );

-- now run the aggregate locally

  SELECT repo_id, event_type, event_public,
         grouping(event_type, event_public),
         min(created_at)
    FROM results
GROUP BY repo_id, ROLLUP(event_type, event_public);
```

```
.
 repo_id |    event_type     | event_public | grouping |         min
---------+-------------------+--------------+----------+---------------------
    8514 | PullRequestEvent  | t            |        0 | 2016-12-01 05:32:54
    8514 | IssueCommentEvent | t            |        0 | 2016-12-01 05:32:57
   19438 | IssueCommentEvent | t            |        0 | 2016-12-01 05:48:56
   21692 | WatchEvent        | t            |        0 | 2016-12-01 06:01:23
   15435 | WatchEvent        | t            |        0 | 2016-12-01 05:40:24
   21692 | WatchEvent        |              |        1 | 2016-12-01 06:01:23
   15435 | WatchEvent        |              |        1 | 2016-12-01 05:40:24
    8514 | PullRequestEvent  |              |        1 | 2016-12-01 05:32:54
    8514 | IssueCommentEvent |              |        1 | 2016-12-01 05:32:57
   19438 | IssueCommentEvent |              |        1 | 2016-12-01 05:48:56
   15435 |                   |              |        3 | 2016-12-01 05:40:24
   21692 |                   |              |        3 | 2016-12-01 06:01:23
   19438 |                   |              |        3 | 2016-12-01 05:48:56
    8514 |                   |              |        3 | 2016-12-01 05:32:54
```

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

CHAPTER

FIFTEEN

CITUS API

15.1 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.

15.1.1 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 configuration value. If the table contains any rows, they are automatically distributed to worker nodes.

Arguments

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

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

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 with the same shard count. If you want to break this colocation later, you can use *update_distributed_table_colocation*. 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 you want to break this implicit colocation, you can use *update_distributed_table_colocation*.

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

shard_count: (Optional) the number of shards to create for the new distributed table. When specifying `shard_count` you can't specify a value of `colocate_with` other than `none`. To change the shard count of an existing table or colocation group, use the *alter_distributed_table* function.

Possible values for `shard_count` are between 1 and 64000. For guidance on choosing the optimal value, see *Shard Count*.

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 Objects (DDL)*.

truncate_local_data_after_distributing_table

Truncate all local rows after distributing a table, and prevent constraints from failing due to outdated local records. The truncation cascades to tables having a foreign key to the designated table. If the referring tables are not themselves distributed then truncation is forbidden until they are, to protect referential integrity:

```
ERROR:  cannot truncate a table referenced in a foreign key constraint by a local table
```

Truncating local coordinator node table data is safe for distributed tables because their rows, if they have any, are copied to worker nodes during distribution.

Arguments

table_name: Name of the distributed table whose local counterpart on the coordinator node should be truncated.

Return Value

N/A

Example

```
-- requires that argument is a distributed table
SELECT truncate_local_data_after_distributing_table('public.github_events');
```

undistribute_table

The undistribute_table() function undoes the action of *create_distributed_table* or *create_reference_table*. Undistributing moves all data from shards back into a local table on the coordinator node (assuming the data can fit), then deletes the shards.

Citus will not undistribute tables that have – or are referenced by – foreign keys, unless the *cascade_via_foreign_keys* argument is set to true. If this argument is false (or omitted), then you must manually drop the offending foreign key constraints before undistributing.

Arguments

table_name: Name of the distributed or reference table to undistribute.

cascade_via_foreign_keys: (Optional) When this argument set to "true," undistribute_table also undistributes all tables that are related to **table_name** through foreign keys. Use caution with this parameter, because it can potentially affect many tables.

Return Value

N/A

Example

This example distributes a `github_events` table and then undistributes it.

```
-- first distribute the table
SELECT create_distributed_table('github_events', 'repo_id');

-- undo that and make it local again
SELECT undistribute_table('github_events');
```

alter_distributed_table

The alter_distributed_table() function can be used to change the distribution column, shard count or colocation properties of a distributed table.

Arguments

table_name: Name of the distributed table that will be altered.

distribution_column: (Optional) Name of the new distribution column.

shard_count: (Optional) The new shard count.

colocate_with: (Optional) The table that the current distributed table will be colocated with. Possible values are `default`, `none` to start a new colocation group, or the name of another table with which to colocate.

cascade_to_colocated: (Optional) When this argument is set to "true", `shard_count` and `colocate_with` changes will also be applied to all of the tables that were previously colocated with the table, and the colocation will be preserved. If it is "false", the current colocation of this table will be broken.

Return Value

N/A

Example

```
-- change distribution column
SELECT alter_distributed_table('github_events', distribution_column:='event_id');

-- change shard count of all tables in colocation group
SELECT alter_distributed_table('github_events', shard_count:=6, cascade_to_
↪colocated:=true);

-- change colocation
SELECT alter_distributed_table('github_events', colocate_with:='another_table');
```

alter_table_set_access_method

The alter_table_set_access_method() function changes access method of a table (e.g. heap or *columnar*).

Arguments

table_name: Name of the table whose access method will change.

access_method: Name of the new access method.

Return Value

N/A

Example

```
SELECT alter_table_set_access_method('github_events', 'columnar');
```

remove_local_tables_from_metadata

The remove_local_tables_from_metadata() function removes local tables from Citus' metadata that no longer need to be there. (See *citus.enable_local_reference_table_foreign_keys (boolean)*.)

Usually if a local table is in Citus' metadata, there's a reason, such as the existence of foreign keys between the table and a reference table. However, if `enable_local_reference_foreign_keys` is disabled, Citus will no longer manage metadata in that situation, and unnecessary metadata can persist until manually cleaned.

Arguments

N/A

Return Value

N/A

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');
```

citus_add_local_table_to_metadata

The citus_add_local_table_to_metadata() function adds a local Postgres table into Citus metadata. A major use-case for this function is to make local tables on the coordinator accessible from any node in the cluster. This is mostly useful when running queries from other nodes. The data associated with the local table stays on the coordinator – only its schema and metadata are sent to the workers.

Note that adding local tables to the metadata comes at a slight cost. When you add the table, Citus must track it in the *Partition table*. Local tables that are added to metadata inherit the same limitations as reference tables (see *Creating and Modifying Distributed Objects (DDL)* and *SQL Support and Workarounds*).

If you *undistribute_table*, Citus will automatically remove the resulting local tables from metadata, which eliminates such limitations on those tables.

Arguments

table_name: Name of the table on the coordinator to be added to Citus metadata.

cascade_via_foreign_keys: (Optional) When this argument set to "true," `citus_add_local_table_to_metadata` adds other tables that are in a foreign key relationship with given table into metadata automatically. Use caution with this parameter, because it can potentially affect many tables.

Return Value

N/A

Example

This example informs the database that the nation table should be defined as a a coordinator-local table, accessible from any node:

```sql
SELECT citus_add_local_table_to_metadata('nation');
```

update_distributed_table_colocation

The update_distributed_table_colocation() function is used to update colocation of a distributed table. This function can also be used to break colocation of a distributed table. Citus will implicitly colocate two tables if the distribution column is the same type, this can be useful if the tables are related and will do some joins. If table A and B are colocated, and table A gets rebalanced, table B will also be rebalanced. If table B does not have a replica identity, the rebalance will fail. Therefore, this function can be useful breaking the implicit colocation in that case.

Note that this function does not move any data around physically.

Arguments

table_name: Name of the table colocation of which will be updated.

colocate_with: The table to which the table should be colocated with.

If you want to break the colocation of a table, you should specify `colocate_with => 'none'`.

Return Value

N/A

Example

This example shows that colocation of `table A` is updated as colocation of `table B`.

```
SELECT update_distributed_table_colocation('A', colocate_with => 'B');
```

Assume that `table A` and `table B` are colocated(possibily implicitly), if you want to break the colocation:

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

Now, assume that `table A`, `table B`, `table C` and `table D` are colocated and you want to colocate `table A` and `table B` together, and `table C` and `table D` together:

```
SELECT update_distributed_table_colocation('C', colocate_with => 'none');
SELECT update_distributed_table_colocation('D', colocate_with => 'C');
```

If you have a hash distributed table named `none` and you want to update its colocation, you can do:

```
SELECT update_distributed_table_colocation('"none"', colocate_with => 'some_other_hash_
↪distributed_table');
```

create_distributed_function

Propagates a function from the coordinator node to workers, and marks it for distributed execution. When a distributed function is called on the coordinator, Citus uses the value of the "distribution argument" to pick a worker node to run the function. Executing the function on workers increases parallelism, and can bring the code closer to data in shards for lower latency.

Note that the Postgres search path is not propagated from the coordinator to workers during distributed function execution, so distributed function code should fully-qualify the names of database objects. Also notices emitted by the functions will not be displayed to the user.

Arguments

function_name: Name of the function to be distributed. The name must include the function's parameter types in parentheses, because multiple functions can have the same name in PostgreSQL. For instance, `'foo(int)'` is different from `'foo(int, text)'`.

distribution_arg_name: (Optional) The argument name by which to distribute. For convenience (or if the function arguments do not have names), a positional placeholder is allowed, such as `'$1'`. If this parameter is not specified, then the function named by `function_name` is merely created on the workers. If worker nodes are added in the future the function will automatically be created there too.

colocate_with: (Optional) When the distributed function reads or writes to a distributed table (or more generally *Co-Locating Tables*), be sure to name that table using the `colocate_with` parameter. This ensures that each invocation of the function runs on the worker node containing relevant shards.

Return Value

N/A

Example

```
-- an example function which updates a hypothetical
-- event_responses table which itself is distributed by event_id
CREATE OR REPLACE FUNCTION
  register_for_event(p_event_id int, p_user_id int)
RETURNS void LANGUAGE plpgsql AS $fn$
BEGIN
  INSERT INTO event_responses VALUES ($1, $2, 'yes')
  ON CONFLICT (event_id, user_id)
  DO UPDATE SET response = EXCLUDED.response;
END;
$fn$;

-- distribute the function to workers, using the p_event_id argument
-- to determine which shard each invocation affects, and explicitly
-- colocating with event_responses which the function updates
SELECT create_distributed_function(
  'register_for_event(int, int)', 'p_event_id',
  colocate_with := 'event_responses'
);
```

alter_columnar_table_set

The alter_columnar_table_set() function changes settings on a *columnar table*. Calling this function on a non-columnar table gives an error. All arguments except the table name are optional.

To view current options for all columnar tables, consult this table:

```
SELECT * FROM columnar.options;
```

The default values for columnar settings for newly-created tables can be overridden with these GUCs:

- columnar.compression
- columnar.compression_level
- columnar.stripe_row_count
- columnar.chunk_row_count

Arguments

table_name: Name of the columnar table.

chunk_row_count: (Optional) The maximum number of rows per chunk for newly-inserted data. Existing chunks of data will not be changed and may have more rows than this maximum value. The default value is 10000.

stripe_row_count: (Optional) The maximum number of rows per stripe for newly-inserted data. Existing stripes of data will not be changed and may have more rows than this maximum value. The default value is 150000.

compression: (Optional) `[none|pglz|zstd|lz4|lz4hc]` The compression type for newly-inserted data. Existing data will not be recompressed or decompressed. The default and generally suggested value is zstd (if support has been compiled in).

compression_level: (Optional) Valid settings are from 1 through 19. If the compression method does not support the level chosen, the closest level will be selected instead.

Return Value

N/A

Example

```
SELECT alter_columnar_table_set(
  'my_columnar_table',
  compression => 'none',
  stripe_row_count => 10000);
```

create_time_partitions

The create_time_partitions() function creates partitions of a given interval to cover a given range of time.

Arguments

table_name: (regclass) table for which to create new partitions. The table must be partitioned on one column, of type date, timestamp, or timestamptz.

partition_interval: an interval of time, such as `'2 hours'`, or `'1 month'`, to use when setting ranges on new partitions.

end_at: (timestamptz) create partitions up to this time. The last partition will contain the point end_at, and no later partitions will be created.

start_from: (timestamptz, optional) pick the first partition so that it contains the point start_from. The default value is `now()`.

Return Value

True if it needed to create new partitions, false if they all existed already.

Example

```
-- create a year's worth of monthly partitions
-- in table foo, starting from the current time

SELECT create_time_partitions(
  table_name         := 'foo',
  partition_interval := '1 month',
  end_at             := now() + '12 months'
);
```

drop_old_time_partitions

The drop_old_time_partitions() function removes all partitions whose intervals fall before a given timestamp. In addition to using this function, you might consider *alter_old_partitions_set_access_method* to compress the old partitions with columnar storage.

Arguments

table_name: (regclass) table for which to remove partitions. The table must be partitioned on one column, of type date, timestamp, or timestamptz.

older_than: (timestamptz) drop partitions whose upper range is less than or equal to older_than.

Return Value

N/A

Example

```
-- drop partitions that are over a year old

CALL drop_old_time_partitions('foo', now() - interval '12 months');
```

alter_old_partitions_set_access_method

In a *Timeseries Data* use case, tables are often partitioned by time, and old partitions are compressed into read-only columnar storage.

Arguments

parent_table_name: (regclass) table for which to change partitions. The table must be partitioned on one column, of type date, timestamp, or timestamptz.

older_than: (timestamptz) change partitions whose upper range is less than or equal to older_than.

new_access_method: (name) either *'heap'* for row-based storage, or *'columnar'* for columnar storage.

Return Value

N/A

Example

```
CALL alter_old_partitions_set_access_method(
  'foo', now() - interval '6 months',
  'columnar'
);
```

15.1.2 Metadata / Configuration Information

citus_add_node

Note: This function requires database superuser access to run.

The citus_add_node() function registers a new node addition in the cluster in the Citus metadata table pg_dist_node. It also copies reference tables to the new node.

If running `citus_add_node` on a single-node cluster, be sure to run *citus_set_coordinator_host* first.

Arguments

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

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

groupid: A group of one primary server its secondary servers, relevant only for streaming replication. Be sure to set `groupid` to a value greater than zero, since zero is reserved for the coordinator node. The default is -1.

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

nodecluster: The cluster name. Default 'default'

Return Value

The nodeid column from the newly inserted row in *pg_dist_node*.

Example

```
select * from citus_add_node('new-node', 12345);
 citus_add_node
-----------------
              7
(1 row)
```

citus_update_node

Note: This function requires database superuser access to run.

The citus_update_node() function changes the hostname and port for a node registered in the Citus 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 citus_update_node(123, 'new-address', 5432);
```

citus_set_node_property

The citus_set_node_property() function changes properties in the Citus metadata table *pg_dist_node*. Currently it can change only the `shouldhaveshards` property.

Arguments

node_name: DNS name or IP address for the node.

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

property: the column to change in pg_dist_node, currently only shouldhaveshard is supported.

value: the new value for the column.

Return Value

N/A

Example

```
SELECT * FROM citus_set_node_property('localhost', 5433, 'shouldhaveshards', false);
```

citus_add_inactive_node

Note: This function requires database superuser access to run.

The citus_add_inactive_node function, similar to *citus_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

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

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

groupid: A group of one primary server and zero or more secondary servers, relevant only for streaming replication. Default -1

noderole: Whether it is ‘primary’ or ‘secondary’. Default ‘primary’

nodecluster: The cluster name. Default ‘default’

Return Value

The nodeid column from the newly inserted row in *pg_dist_node*.

Example

```
select * from citus_add_inactive_node('new-node', 12345);
 citus_add_inactive_node
--------------------------
                        7
(1 row)
```

citus_activate_node

Note: This function requires database superuser access to run.

The `citus_activate_node` function marks a node as active in the Citus metadata table `pg_dist_node` and copies reference tables to the node. Useful for nodes added via *citus_add_inactive_node*.

Arguments

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

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

Return Value

The nodeid column from the newly inserted row in *pg_dist_node*.

Example

```
select * from citus_activate_node('new-node', 12345);
 citus_activate_node
----------------------
                    7
(1 row)
```

citus_disable_node

Note: This function requires database superuser access to run.

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

Arguments

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

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

Return Value

N/A

Example

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

citus_add_secondary_node

Note: This function requires database superuser access to run.

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

Arguments

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

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

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

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

nodecluster: The cluster name. Default ‘default’

Return Value

The nodeid column for the secondary node, inserted row in *pg_dist_node*.

Example

```
select * from citus_add_secondary_node('new-node', 12345, 'primary-node', 12345);
 citus_add_secondary_node
---------------------------
                         7
(1 row)
```

citus_remove_node

Note: This function requires database superuser access to run.

The citus_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

nodename: DNS name of the node to be removed.

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

Return Value

N/A

Example

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

(1 row)
```

citus_get_active_worker_nodes

The citus_get_active_worker_nodes() function returns a list of active worker host names and port numbers.

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 citus_get_active_worker_nodes();
 node_name | node_port
-----------+-----------
 localhost |      9700
 localhost |      9702
 localhost |      9701

(3 rows)
```

citus_backend_gpid

The citus_backend_gpid() function returns the global process identifier (GPID) for the PostgreSQL backend serving the current session. A GPID encodes both a node in the Citus cluster, and the operating system process ID of PostgreSQL on that node.

Citus extends the PostgreSQL server signaling functions `pg_cancel_backend()` and `pg_terminate_backend()` so that they accept GPIDs. In Citus, calling these functions on one node can affect a backend running on another node.

Arguments

N/A

Return Value

An integer GPID, of the form (NodeId * 10,000,000,000) + ProcessId.

Example

```
SELECT citus_backend_gpid();
```

```
 citus_backend_gpid
--------------------
        10000002055
```

citus_check_cluster_node_health

Check connectivity between all nodes. If there are N nodes, this function checks all N^2 connections between them.

Arguments

N/A

Return Value

List of tuples where each tuple contains the following information:

from_nodename: DNS name of the source worker node

from_nodeport: Port on the source worker node on which the database server is listening

to_nodename: DNS name of the destination worker node

to_nodeport: Port on the destination worker node on which the database server is listening

result: Whether a connection could be established

Example

```
SELECT * FROM citus_check_cluster_node_health();
```

```
 from_nodename | from_nodeport | to_nodename | to_nodeport | result
---------------+---------------+-------------+-------------+--------
 localhost     |          1400 | localhost   |        1400 | t
 localhost     |          1400 | localhost   |        1401 | t
 localhost     |          1400 | localhost   |        1402 | t
 localhost     |          1401 | localhost   |        1400 | t
 localhost     |          1401 | localhost   |        1401 | t
 localhost     |          1401 | localhost   |        1402 | t
 localhost     |          1402 | localhost   |        1400 | t
 localhost     |          1402 | localhost   |        1401 | t
 localhost     |          1402 | localhost   |        1402 | t

(9 rows)
```

citus_set_coordinator_host

This function is required when adding worker nodes to a Citus cluster which was created initially as a *single-node cluster*. When the coordinator registers a new worker, it adds a coordinator hostname from the value of *citus.local_hostname (text)*, which is by default `localhost`. The worker would attempt to connect to `localhost` to talk to the coordinator, which is obviously wrong.

Thus, the system administrator should call `citus_set_coordinator_host` before calling *citus_add_node* in a single-node cluster.

Arguments

host: DNS name of the coordinator node.

port: (Optional) The port on which the coordinator lists for PostgreSQL connections. Defaults to `current_setting('port')`.

node_role: (Optional) Defaults to `primary`.

node_cluster: (Optional) Defaults to `default`.

Return Value

N/A

Example

```
-- assuming we're in a single-node cluster

-- first establish how workers should reach us
SELECT citus_set_coordinator_host('coord.example.com', 5432);

-- then add a worker
SELECT * FROM citus_add_node('worker1.example.com', 5432);
```

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 (deprecated), 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. Must be 'h' (hash).

part_key: Distribution column for the table.

part_replica_count: (Deprecated) 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  |                  1 |   ␣
↪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-distributed tables as well as reference tables.

Arguments

table_name: The distributed table.

distribution_value: The value of the distribution column.

Return Value

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

Example

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

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

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

15.1.3 Cluster Management And Repair Functions

citus_move_shard_placement

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.

Note: Citus Community edition supports all shard transfer modes as of version 11.0!

Return Value

N/A

Example

```sql
SELECT citus_move_shard_placement(12345, 'from_host', 5432, 'to_host', 5432);
```

citus_rebalance_start

The citus_rebalance_start() function moves table shards to make them evenly distributed among the workers. It begins a background job to do the rebalancing, and returns immediately.

The rebalancing process 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.

Every shard is assigned a cost when determining whether shards are “evenly distributed.” By default each shard has the same cost (a value of 1), so distributing to equalize the cost across workers is the same as equalizing the number of shards on each. The constant cost strategy is called “by_shard_count” and is the default rebalancing strategy.

The default strategy is appropriate under these circumstances:

1. The shards are roughly the same size
2. The shards get roughly the same amount of traffic
3. Worker nodes are all the same size/type
4. Shards haven’t been pinned to particular workers

If any of these assumptions don't hold, then the default rebalancing can result in a bad plan. In this case you may customize the strategy, using the `rebalance_strategy` parameter.

It's advisable to call *get_rebalance_table_shards_plan* before running citus_rebalance_start, to see and verify the actions to be performed.

Arguments

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

drain_only: (Optional) When true, move shards off worker nodes who have `shouldhaveshards` set to false in *Worker node table*; move no other shards.

rebalance_strategy: (Optional) the name of a strategy in *Rebalancer strategy table*. If this argument is omitted, the function chooses the default strategy, as indicated in the table.

Return Value

N/A

Example

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

```
SELECT citus_rebalance_start();
NOTICE:  Scheduling...
NOTICE:  Scheduled as job 1337.
DETAIL:  Rebalance scheduled as background job 1337.
HINT:  To monitor progress, run: SELECT details FROM citus_rebalance_status();
```

citus_rebalance_status

The *citus_rebalance_start* function returns immediately, while the rebalance continues as a background job. The``citus_rebalance_status()`` function allows you to monitor the progress of this rebalance.

Example

To get general information about the rebalance, you can select all columns from the status. This shows the basic state of the job:

```
SELECT * FROM citus_rebalance_status();
```

```
.
 job_id |   state    | job_type  |           description            |          started_at  ␣
↪          |           finished_at            | details
```

(continues on next page)

(continued from previous page)

```
--------+----------+-----------+---------------------------------+----------------------
↪--------+-------------------------------+-----------
      4 | running  | rebalance | Rebalance colocation group 1    | 2022-08-09 21:57:27.
↪833055+02 | 2022-08-09 21:57:27.833055+02 | { ... }
```

Rebalancer specifics live in the `details` column, in JSON format:

```
SELECT details FROM citus_rebalance_status();
```

```
{
    "phase": "copy",
    "phase_index": 1,
    "phase_count": 3,
    "last_change":"2022-08-09 21:57:27",
    "colocations": {
        "1": {
            "shard_moves": 30,
            "shard_moved": 29,
            "last_move":"2022-08-09 21:57:27"
        },
        "1337": {
            "shard_moves": 130,
            "shard_moved": 0
        }
    }
}
```

citus_rebalance_stop

This function cancels a rebalance in progress, if any.

Arguments

N/A

Return value

N/A

citus_rebalance_wait

This function blocks until a running rebalance is complete. If no rebalance is in progress when `citus_rebalance_wait()` is called, then the function returns immediately.

The function can be useful for scripts or benchmarking.

Arguments

N/A

Return value

N/A

rebalance_table_shards

> **Warning:** The `rebalance_table_shards()` function is deprecated. As of Citus v11.2, use *citus_rebalance_start* instead.

get_rebalance_table_shards_plan

Output the planned shard movements of *citus_rebalance_start* without performing them. While it's unlikely, get_rebalance_table_shards_plan can output a slightly different plan than what a citus_rebalance_start call with the same arguments will do. This could happen because they are not executed at the same time, so facts about the cluster – e.g. disk space – might differ between the calls.

Arguments

A superset of the arguments for citus_rebalance_start: relation, threshold, max_shard_moves, excluded_shard_list, and drain_only. See documentation of that function for the arguments' meaning.

Return Value

Tuples containing these columns:

- **table_name**: The table whose shards would move
- **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

get_rebalance_progress

Note: Citus v11.2 introduces the *citus_rebalance_status* function, whose output is easier to understand than that of `get_rebalance_progress`.

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 `citus_rebalance_start()`.

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 of the shard 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
- **source_shard_size**: Size of the shard on the source node in bytes
- **target_shard_size**: Size of the shard on the target node in bytes

Example

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

```
────────────────────────────────────────────────────────────────────────────────────────────
│ sessionid │ table_name │ shardid │ shard_size │  sourcename   │ sourceport │ ␣
↪targetname   │ targetport │ progress │ source_shard_size │ target_shard_size │
├───────────────────────────────────────────────────────────────────────────────────────────
│      7083 │ foo        │  102008 │    1204224 │ n1.foobar.com │       5432 │ n4.foobar.
↪com │       5432 │        0 │           1204224 │                 0 │
│      7083 │ foo        │  102009 │    1802240 │ n1.foobar.com │       5432 │ n4.foobar.
↪com │       5432 │        0 │           1802240 │                 0 │
│      7083 │ foo        │  102018 │     614400 │ n2.foobar.com │       5432 │ n4.foobar.
↪com │       5432 │        1 │            614400 │            354400 │
```

(continues on next page)

(continued from previous page)

```
│       7083 │ foo          │  102019 │       8192 │ n3.foobar.com │       5432 │ n4.foobar.
↪com │       5432 │        2 │                 0 │          8192 │
└──────────────────────────────────────────────────────────────────────────────────────────
```

citus_add_rebalance_strategy

Append a row to the `pg_dist_rebalance_strategy`.

Arguments

For more about these arguments, see the corresponding column values in *Rebalancer strategy table*.

name: identifier for the new strategy

shard_cost_function: identifies the function used to determine the "cost" of each shard

node_capacity_function: identifies the function to measure node capacity

shard_allowed_on_node_function: identifies the function which determines which shards can be placed on which nodes

default_threshold: a floating point threshold that tunes how precisely the cumulative shard cost should be balanced between nodes

minimum_threshold: (Optional) a safeguard column that holds the minimum value allowed for the threshold argument of citus_rebalance_start(). Its default value is 0

Return Value

N/A

citus_set_default_rebalance_strategy

Update the *Rebalancer strategy table* table, changing the strategy named by its argument to be the default chosen when rebalancing shards.

Arguments

name: the name of the strategy in pg_dist_rebalance_strategy

Return Value

N/A

Example

```
SELECT citus_set_default_rebalance_strategy('by_disk_size');
```

citus_remote_connection_stats

The citus_remote_connection_stats() function shows the number of active connections to each remote node.

Arguments

N/A

Example

```
SELECT * from citus_remote_connection_stats();
```

```
.
    hostname    | port | database_name | connection_count_to_node
----------------+------+---------------+--------------------------
 citus_worker_1 | 5432 | postgres      |                        3
(1 row)
```

citus_drain_node

The citus_drain_node() function moves shards off the designated node and onto other nodes who have `shouldhaveshards` set to true in *Worker node table*. This function is designed to be called prior to removing a node from the cluster, i.e. turning the node's physical server off.

Arguments

nodename: The hostname name of the node to be drained.

nodeport: The port number of the node to be drained.

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.

Note: Citus Community edition supports all shard transfer modes as of version 11.0!

rebalance_strategy: (Optional) the name of a strategy in *Rebalancer strategy table*. If this argument is omitted, the function chooses the default strategy, as indicated in the table.

Return Value

N/A

Example

Here are the typical steps to remove a single node (for example ‘10.0.0.1’ on a standard PostgreSQL port):

1. Drain the node.

```
SELECT * from citus_drain_node('10.0.0.1', 5432);
```

2. Wait until the command finishes

3. Remove the node

When draining multiple nodes it’s recommended to use *citus_rebalance_start* instead. Doing so allows Citus to plan ahead and move shards the minimum number of times.

1. Run this for each node that you want to remove:

```
SELECT * FROM citus_set_node_property(node_hostname, node_port, 'shouldhaveshards',␣
↪false);
```

2. Drain them all at once with *citus_rebalance_start*:

```
SELECT * FROM citus_rebalance_start(drain_only := true);
```

3. Wait until the draining rebalance finishes

4. Remove the nodes

isolate_tenant_to_new_shard

Note: Citus Community edition includes the isolate_tenant_to_new_shard function as of version 11.0!

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                  │
└────────────────────────────┘
```

15.2 Citus Tables and Views

15.2.1 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 :- 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 :- * postgresql streaming replication: ‘s’ * two-phase commit (for reference tables): ‘t’

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

(continues on next page)

(continued from previous page)

```
---------------+------------+------------------------------------------------------------
↪------------------------------------------------------------+--------------+----------
 github_events | h          | {VAR :varno 1 :varattno 4 :vartype 20 :vartypmod -1␣
↪:varcollid 0 :varlevelsup 0 :varnoold 1 :varoattno 4 :location -1} |           2 | s
 (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. In case of hash distributed tables, they are hash token ranges assigned to that shard. These statistics are used for pruning away unrelated shards during SELECT queries.

Name	Type	Description
logicalrelid	regclass	Distributed table to which this shard belongs. This value references the relfilenode column in the pg_class system catalog table.
shardid	bigint	Globally unique identifier assigned to this shard.
shardstorage	char	Type of storage used for this shard. Different storage types are discussed in the table below.
shardminvalue	text	For hash distributed tables, minimum hash token value assigned to that shard (inclusive).
shardmaxvalue	text	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
```

(continues on next page)

(continued from previous page)

```
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 cstore_fdw tables)
FOREIGN	'f'	Indicates that shard stores foreign data. (Used by distributed file_fdw tables)

Shard information view

In addition to the low-level shard metadata table described above, Citus provides a `citus_shards` view to easily check:

- Where each shard is (node, and port),
- What kind of table it belongs to, and
- Its size

This view helps you inspect shards to find, among other things, any size imbalances across nodes.

```
SELECT * FROM citus_shards;
```

```
.
 table_name | shardid | shard_name   | citus_table_type | colocation_id | nodename  |␣
↪nodeport | shard_size
------------+---------+--------------+------------------+---------------+-----------+----
↪------+------------
 dist       |  102170 | dist_102170  | distributed      |            34 | localhost |   ␣
↪ 9701 |   90677248
 dist       |  102171 | dist_102171  | distributed      |            34 | localhost |   ␣
↪ 9702 |   90619904
```

(continues on next page)

(continued from previous page)

```
dist       |  102172 | dist_102172  | distributed      |            34 | localhost |   ␣
↪ 9701 |   90701824
dist       |  102173 | dist_102173  | distributed      |            34 | localhost |   ␣
↪ 9702 |   90693632
ref        |  102174 | ref_102174   | reference        |             2 | localhost |   ␣
↪ 9701 |       8192
ref        |  102174 | ref_102174   | reference        |             2 | localhost |   ␣
↪ 9702 |       8192
dist2      |  102175 | dist2_102175 | distributed      |            34 | localhost |   ␣
↪ 9701 |     933888
dist2      |  102176 | dist2_102176 | distributed      |            34 | localhost |   ␣
↪ 9702 |     950272
dist2      |  102177 | dist2_102177 | distributed      |            34 | localhost |   ␣
↪ 9701 |     942080
dist2      |  102178 | dist2_102178 | distributed      |            34 | localhost |   ␣
↪ 9702 |     933888
```

The colocation_id refers to the *colocation group*. For more info about citus_table_type, see *Table Types*.

Shard placement table

The pg_dist_placement table tracks the location of shards on worker nodes. Each 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
placementid	bigint	Unique auto-generated identifier for each individual placement.
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 hash distributed tables, zero.
groupid	int	Identifier used to denote a group of one primary server and zero or more secondary servers.

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

Note: As of Citus 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
```

(continues on next page)

(continued from previous page)

```
   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.

Worker node table

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

Name	Type	Description
nodeid	int	Auto-generated identifier for an individual node.
groupid	int	Identifier used to denote a group of one primary server and zero or more secondary servers. By default it is the same as the nodeid.
nodename	text	Host Name or IP Address of the PostgreSQL worker node.
nodeport	int	Port number on which the PostgreSQL worker node is listening.
noderack	text	(Optional) Rack placement information for the worker node.
hasmetadata	boolean	Reserved for internal use.
isactive	boolean	Whether the node is active accepting shard placements.
noderole	text	Whether the node is a primary or secondary
nodecluster	text	The name of the cluster containing this node
metadatasynced	boolean	Reserved for internal use.
shouldhaveshards	boolean	If false, shards will be moved off node (drained) when rebalancing, nor will shards from new distributed tables be placed on the node, unless they are colocated with shards already there

```
SELECT * from pg_dist_node;
 nodeid | groupid | nodename  | nodeport | noderack | hasmetadata | isactive | noderole␣
↪| nodecluster | metadatasynced | shouldhaveshards
--------+---------+-----------+----------+----------+-------------+----------+----------
↪+-------------+----------------+------------------
      1 |       1 | localhost |    12345 | default  | f           | t        | primary ␣
↪| default     | f              | t
      2 |       2 | localhost |    12346 | default  | f           | t        | primary ␣
↪| default     | f              | t
      3 |       3 | localhost |    12347 | default  | f           | t        | primary ␣
↪| default     | f              | t
(3 rows)
```

Distributed object table

The citus.pg_dist_object table contains a list of objects such as types and functions that have been created on the coordinator node and propagated to worker nodes. When an administrator adds new worker nodes to the cluster, Citus automatically creates copies of the distributed objects on the new nodes (in the correct order to satisfy object dependencies).

Name	Type	Description
classid	oid	Class of the distributed object
objid	oid	Object id of the distributed object
objsubid	integer	Object sub id of the distributed object, e.g. attnum
type	text	Part of the stable address used during pg upgrades
object_names	text[]	Part of the stable address used during pg upgrades
object_args	text[]	Part of the stable address used during pg upgrades
distribution_argument_index	integer	Only valid for distributed functions/procedures
colocationid	integer	Only valid for distributed functions/procedures

"Stable addresses" uniquely identify objects independently of a specific server. Citus tracks objects during a PostgreSQL upgrade using stable addresses created with the pg_identify_object_as_address() function.

Here's an example of how `create_distributed_function()` adds entries to the `citus.pg_dist_object` table:

```
CREATE TYPE stoplight AS enum ('green', 'yellow', 'red');

CREATE OR REPLACE FUNCTION intersection()
RETURNS stoplight AS $$
DECLARE
        color stoplight;
BEGIN
        SELECT *
          FROM unnest(enum_range(NULL::stoplight)) INTO color
         ORDER BY random() LIMIT 1;
        RETURN color;
END;
$$ LANGUAGE plpgsql VOLATILE;

SELECT create_distributed_function('intersection()');
```

(continues on next page)

(continued from previous page)

```
-- will have two rows, one for the TYPE and one for the FUNCTION
TABLE citus.pg_dist_object;
```

```
-[ RECORD 1 ]---------------+------
classid                     | 1247
objid                       | 16780
objsubid                    | 0
type                        |
object_names                |
object_args                 |
distribution_argument_index |
colocationid                |
-[ RECORD 2 ]---------------+------
classid                     | 1255
objid                       | 16788
objsubid                    | 0
type                        |
object_names                |
object_args                 |
distribution_argument_index |
colocationid                |
```

Citus tables view

The citus_tables view shows a summary of all tables managed by Citus (distributed and reference tables). The view combines information from Citus metadata tables for an easy, human-readable overview of these table properties:

- *Table type*
- *Distribution column*
- *Colocation group* id
- Human-readable size
- Shard count
- Owner (database user)
- Access method (heap or *columnar*)

Here's an example:

```
SELECT * FROM citus_tables;
```

```
──────────────────────────────────────────────────────────────────────────────────────────────────
│ table_name │ citus_table_type │ distribution_column │ colocation_id │ table_size │␣
↪shard_count │ table_owner │ access_method │
├─────────────────────────────────────────────────────────────────────────────────────────────────
│ foo.test   │ distributed      │ test_column         │             1 │ 0 bytes    │     ␣
↪     32 │ citus       │ heap          │
│ ref        │ reference        │ <none>              │             2 │ 24 GB      │     ␣
↪      1 │ citus       │ heap          │
```

(continues on next page)

(continued from previous page)

```
│ test          │ distributed        │ id                    │              1 │ 248 TB    │      ␣
↪     32 │ citus         │ heap            │
└──────────────────────────────────────────────────────────────────────────────────────────────────────────
```

Time partitions view

Citus provides UDFs to manage partitions for the *Timeseries Data* use case. It also maintains a `time_partitions` view to inspect the partitions it manages.

Columns:

- **parent_table** the table which is partitioned
- **partition_column** the column on which the parent table is partitioned
- **partition** the name of a partition table
- **from_value** lower bound in time for rows in this partition
- **to_value** upper bound in time for rows in this partition
- **access_method** `heap` for row-based storage, and `columnar` for columnar storage

```
SELECT * FROM time_partitions;
```

```
──────────────────────────────────────────────────────────────────────────────────────────────────────────
│      parent_table        │ partition_column │                partition                 │ ␣
↪   from_value         │       to_value       │ access_method │
├──────────────────────────────────────────────────────────────────────────────────────────────────────────
│ github_columnar_events │ created_at         │ github_columnar_events_p2015_01_01_0000 │␣
↪2015-01-01 00:00:00 │ 2015-01-01 02:00:00 │ columnar      │
│ github_columnar_events │ created_at         │ github_columnar_events_p2015_01_01_0200 │␣
↪2015-01-01 02:00:00 │ 2015-01-01 04:00:00 │ columnar      │
│ github_columnar_events │ created_at         │ github_columnar_events_p2015_01_01_0400 │␣
↪2015-01-01 04:00:00 │ 2015-01-01 06:00:00 │ columnar      │
│ github_columnar_events │ created_at         │ github_columnar_events_p2015_01_01_0600 │␣
↪2015-01-01 06:00:00 │ 2015-01-01 08:00:00 │ heap          │
└──────────────────────────────────────────────────────────────────────────────────────────────────────────
```

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, 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. (Deprecated)
distributioncolumntype	oid	The type of the distribution column for all tables in this co-location group.
distributioncolumncollation	oid	The collation of the distribution column for all tables in this co-location group.

```
SELECT * from pg_dist_colocation;
  colocationid | shardcount | replicationfactor | distributioncolumntype |␣
↪distributioncolumncollation
 --------------+------------+-------------------+------------------------+---------------
↪--------------
             2 |         32 |                 1 |                     20 |             ␣
↪             0
  (1 row)
```

Rebalancer strategy table

This table defines strategies that *citus_rebalance_start* can use to determine where to move shards.

Name	Type	Description
name	name	Unique name for the strategy
default_strategy	boolean	Whether *citus_rebalance_start* should choose this strategy by default. Use *citus_set_default_rebalance_strategy* to update this column
shard_cost_function	regproc	Identifier for a cost function, which must take a shardid as bigint, and return its notion of a cost, as type real
node_capacity_function	regproc	Identifier for a capacity function, which must take a nodeid as int, and return its notion of node capacity as type real
shard_allowed_on_node_function	regproc	Identifier for a function that given shardid bigint, and nodeidarg int, returns boolean for whether the shard is allowed to be stored on the node
default_threshold	float4	Threshold for deeming a node too full or too empty, which determines when the citus_rebalance_start should try to move shards
minimum_threshold	float4	A safeguard to prevent the threshold argument of citus_rebalance_start() from being set too low
improvement_threshold	float4	Determines when moving a shard is worth it during a rebalance. The rebalancer will move a shard when the ratio of the improvement with
		the shard move to the improvement without crosses the threshold. This is most useful with the by_disk_size strategy.

A Citus installation ships with these strategies in the table:

```
SELECT * FROM pg_dist_rebalance_strategy;
```

```
-[ RECORD 1 ]-------------------+-----------------------------------
name                            | by_shard_count
default_strategy                | t
shard_cost_function             | citus_shard_cost_1
node_capacity_function          | citus_node_capacity_1
shard_allowed_on_node_function  | citus_shard_allowed_on_node_true
default_threshold               | 0
minimum_threshold               | 0
improvement_threshold           | 0
-[ RECORD 2 ]-------------------+-----------------------------------
name                            | by_disk_size
default_strategy                | f
shard_cost_function             | citus_shard_cost_by_disk_size
node_capacity_function          | citus_node_capacity_1
shard_allowed_on_node_function  | citus_shard_allowed_on_node_true
default_threshold               | 0.1
minimum_threshold               | 0.01
improvement_threshold           | 0.5
```

The default strategy, `by_shard_count`, assigns every shard the same cost. Its effect is to equalize the shard count across nodes. The other predefined strategy, `by_disk_size`, assigns a cost to each shard matching its disk size in bytes plus that of the shards that are colocated with it. The disk size is calculated using `pg_total_relation_size`, so it includes indices. This strategy attempts to achieve the same disk space on every node. Note the threshold of 0.1 – it prevents unnecessary shard movement caused by insigificant differences in disk space.

Creating custom rebalancer strategies

Here are examples of functions that can be used within new shard rebalancer strategies, and registered in the *Rebalancer strategy table* with the *citus_add_rebalance_strategy* function.

- Setting a node capacity exception by hostname pattern:

```
-- example of node_capacity_function

CREATE FUNCTION v2_node_double_capacity(nodeidarg int)
    RETURNS real AS $$
    SELECT
        (CASE WHEN nodename LIKE '%.v2.worker.citusdata.com' THEN 2.0::float4 ELSE␣
↪1.0::float4 END)
    FROM pg_dist_node where nodeid = nodeidarg
    $$ LANGUAGE sql;
```

- Rebalancing by number of queries that go to a shard, as measured by the *Query statistics table*:

```
-- example of shard_cost_function

CREATE FUNCTION cost_of_shard_by_number_of_queries(shardid bigint)
    RETURNS real AS $$
    SELECT coalesce(sum(calls)::real, 0.001) as shard_total_queries
```

(continues on next page)

(continued from previous page)

```
    FROM citus_stat_statements
    WHERE partition_key is not null
        AND get_shard_id_for_distribution_column('tab', partition_key) = shardid;
$$ LANGUAGE sql;
```

- Isolating a specific shard (10000) on a node (address ‘10.0.0.1’):

```
-- example of shard_allowed_on_node_function

CREATE FUNCTION isolate_shard_10000_on_10_0_0_1(shardid bigint, nodeidarg int)
    RETURNS boolean AS $$
    SELECT
        (CASE WHEN nodename = '10.0.0.1' THEN shardid = 10000 ELSE shardid != 10000␣
↪END)
    FROM pg_dist_node where nodeid = nodeidarg
    $$ LANGUAGE sql;

-- The next two definitions are recommended in combination with the above function.
-- This way the average utilization of nodes is not impacted by the isolated shard.
CREATE FUNCTION no_capacity_for_10_0_0_1(nodeidarg int)
    RETURNS real AS $$
    SELECT
        (CASE WHEN nodename = '10.0.0.1' THEN 0 ELSE 1 END)::real
    FROM pg_dist_node where nodeid = nodeidarg
    $$ LANGUAGE sql;
CREATE FUNCTION no_cost_for_10000(shardid bigint)
    RETURNS real AS $$
    SELECT
        (CASE WHEN shardid = 10000 THEN 0 ELSE 1 END)::real
    $$ LANGUAGE sql;
```

Query statistics table

Note: The citus_stat_statements view is part of Citus Community edition as of version 11.0!

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.

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 *executor* used: adaptive, 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:

```
-[ RECORD 1 ]-+----------------------------------------------
queryid       | -909556869173432820
userid        | 10
dbid          | 13340
query         | insert into foo select generate_series($1,$2)
executor      | insert-select
partition_key |
calls         | 1
-[ RECORD 2 ]-+----------------------------------------------
queryid       | 3919808845681956665
userid        | 10
dbid          | 13340
query         | select count(*) from foo;
executor      | adaptive
partition_key |
calls         | 1
-[ RECORD 3 ]-+----------------------------------------------
queryid       | 5351346905785208738
userid        | 10
dbid          | 13340
query         | select * from foo where id = $1
executor      | adaptive
partition_key | 42
calls         | 1
```

Caveats:

- The stats data is not replicated, and won't survive database crashes or failover
- 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

Tenant-level query statistics view

The `citus_stat_tenants` view augments the *Query statistics table* with information about how many queries each tenant is running. Tracing queries to originating tenants helps, among other things, for deciding when to do *Tenant Isolation*.

This view counts recent single-tenant queries happening during a configurable time period. The tally of read-only and total queries for the period increases until the current period ends. After that, the counts are moved to last period's statistics, which stays constant until expiration. The period length can be set in seconds using `citus.stats_tenants_period`, and is 60 seconds by default.

The view displays up to `citus.stat_tenants_limit` rows (by default 100). It counts only queries filtered to a single tenant, ignoring queries that apply to multiple tenants at once.

Name	Type	Description
nodeid	int	Node ID from *Worker node table*
colocation_id	int	ID of *colocation group*
tenant_attribute	text	Value in the distribution column identifying tenant
read_count_in_this_period	int	Number of read (SELECT) queries for tenant in period
read_count_in_last_period	int	Number of read queries one period of time ago
query_count_in_this_period	int	Number of read/write queries for tenant in time period
query_count_in_last_period	int	Number of read/write queries one period of time ago
cpu_usage_in_this_period	double	Seconds of CPU time spent for this tenant in period
cpu_usage_in_last_period	double	Seconds of CPU time spent for this tenant last period

Tracking tenant level statistics adds overhead, and by default is disabled. To enable it, set `citus.stat_tenants_track` to `'all'`.

Example:

Suppose we have a distributed table called `dist_table`, with distribution column `tenant_id`. Then we make some queries:

```sql
INSERT INTO dist_table(tenant_id) VALUES (1);
INSERT INTO dist_table(tenant_id) VALUES (1);
INSERT INTO dist_table(tenant_id) VALUES (2);

SELECT count(*) FROM dist_table WHERE tenant_id = 1;
```

The tenant-level statistics will reflect the queries we just made:

```sql
SELECT tenant_attribute, read_count_in_this_period,
       query_count_in_this_period, cpu_usage_in_this_period
  FROM citus_stat_tenants;
```

```
 tenant_attribute | read_count_in_this_period | query_count_in_this_period | cpu_usage_
↪in_this_period
------------------+---------------------------+----------------------------+------------
↪-------------
 1                |                         1 |                          3 |            ␣
↪    0.000883
 2                |                         0 |                          1 |            ␣
↪    0.000144
```

Distributed Query Activity

In some situations, 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 Citus 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_stat_activity**: shows the distributed queries that are executing on all nodes. A superset of `pg_stat_activity`, usable wherever the latter is.
- **citus_dist_stat_activity**: the same as `citus_stat_activity` but restricted to distributed queries only, and excluding Citus fragment queries.
- **citus_lock_waits**: Blocked queries throughout the cluster.

The first two views include all columns of pg_stat_activity plus the global PID of the worker that initiated the query.

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

```
-- run in one session
-- (with a pg_sleep so we can see it)

SELECT count(*), pg_sleep(3) FROM users_table;
```

We can see the query appear in `citus_dist_stat_activity`:

```
-- run in another session

SELECT * FROM citus_dist_stat_activity;

-[ RECORD 1 ]----+-------------------------------------------
global_pid       | 10000012199
nodeid           | 1
is_worker_query  | f
datid            | 13724
datname          | postgres
pid              | 12199
leader_pid       |
usesysid         | 10
usename          | postgres
application_name | psql
client_addr      |
client_hostname  |
client_port      | -1
backend_start    | 2022-03-23 11:30:00.533991-05
xact_start       | 2022-03-23 19:35:28.095546-05
query_start      | 2022-03-23 19:35:28.095546-05
state_change     | 2022-03-23 19:35:28.09564-05
wait_event_type  | Timeout
wait_event       | PgSleep
state            | active
backend_xid      |
backend_xmin     | 777
query_id         |
query            | SELECT count(*), pg_sleep(3) FROM users_table;
backend_type     | client backend
```

The citus_dist_stat_activity view hides internal Citus fragment queries. To see those, we can use the more detailed citus_stat_activity view. For instance, the previous count(*) query requires information from all shards. Some of the information is in shard users_table_102039, which is visible in the query below.

```
SELECT * FROM citus_stat_activity;

-[ RECORD 1 ]----+------------------------------------------------------------------
global_pid       | 10000012199
nodeid           | 1
is_worker_query  | f
datid            | 13724
datname          | postgres
pid              | 12199
leader_pid       |
usesysid         | 10
usename          | postgres
application_name | psql
client_addr      |
client_hostname  |
client_port      | -1
backend_start    | 2022-03-23 11:30:00.533991-05
xact_start       | 2022-03-23 19:32:18.260803-05
query_start      | 2022-03-23 19:32:18.260803-05
state_change     | 2022-03-23 19:32:18.260821-05
wait_event_type  | Timeout
wait_event       | PgSleep
state            | active
backend_xid      |
backend_xmin     | 777
query_id         |
query            | SELECT count(*), pg_sleep(3) FROM users_table;
backend_type     | client backend
-[ RECORD 2 ]----------+------------------------------------------------------------
↪-------------------------
global_pid       | 10000012199
nodeid           | 1
is_worker_query  | t
datid            | 13724
datname          | postgres
pid              | 12725
leader_pid       |
usesysid         | 10
usename          | postgres
application_name | citus_internal gpid=10000012199
client_addr      | 127.0.0.1
client_hostname  |
client_port      | 44106
backend_start    | 2022-03-23 19:29:53.377573-05
xact_start       |
query_start      | 2022-03-23 19:32:18.278121-05
state_change     | 2022-03-23 19:32:18.278281-05
wait_event_type  | Client
wait_event       | ClientRead
```

(continues on next page)

(continued from previous page)

```
state            | idle
backend_xid      |
backend_xmin     |
query_id         |
query            | SELECT count(*) AS count FROM public.users_table_102039 users WHERE true
backend_type     | client backend
```

The query field shows rows being counted in shard 102039.

Here are examples of useful queries you can build using citus_stat_activity:

```
-- active queries' wait events

SELECT query, wait_event_type, wait_event
  FROM citus_stat_activity
 WHERE state='active';

-- active queries' top wait events

SELECT wait_event, wait_event_type, count(*)
  FROM citus_stat_activity
 WHERE state='active'
 GROUP BY wait_event, wait_event_type
 ORDER BY count(*) desc;

-- total internal connections generated per node by Citus

SELECT nodeid, count(*)
  FROM citus_stat_activity
 WHERE is_worker_query
 GROUP BY nodeid;
```

The next view is citus_lock_waits. To see how it 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_gpid                          | 10000011981
blocking_gpid                         | 10000011979
blocked_statement                     | UPDATE numbers SET j = 3 WHERE i = 1;
current_statement_in_blocking_process | UPDATE numbers SET j = 2 WHERE i = 1;
waiting_nodeid                        | 1
blocking_nodeid                       | 1
```

In this example the queries originated on the coordinator, but the view can also list locks between queries originating on workers.

15.2.2 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 part of Citus Community edition as of version 11.0!

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

Name	Type	Description
nodeid	integer	Node id from *Worker node table*, or 0, or -1
rolename	name	Postgres role
authinfo	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 part of Citus Community edition as of version 11.0!

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 *Worker node table*
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');
```

15.3 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';
```

15.3.1 General configuration

citus.max_background_task_executors_per_node (integer)

Determines how many background tasks can be executed in parallel at a given time. For instance, these tasks are for shard moves from/to a node. When increasing this value, you'll often also want to increase `citus.max_background_task_executors` and max_worker_processes.

- Default: 1
- Minimum: 1
- Maximum: 128

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_connection_timeout (integer)

The `citus.node_connection_timeout` GUC sets the maximum duration (in milliseconds) to wait for connection establishment. Citus raises an error if the timeout elapses before at least one worker connection is established. This GUC affects connections from the coordinator to workers, and workers to each other.

- Default: thirty seconds
- Minimum: ten milliseconds
- Maximum: one hour

```
-- set to 60 seconds
ALTER DATABASE foo
SET citus.node_connection_timeout = 60000;
```

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 specific subset of the allowed options, namely:

- application_name
- connect_timeout
- gsslib†
- keepalives
- keepalives_count
- keepalives_idle
- keepalives_interval
- krbsrvname†
- sslcompression
- sslcrl
- sslmode (defaults to "require" as of Citus 8.1)
- sslrootcert
- tcp_user_timeout

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

The `node_conninfo` setting takes effect only on newly opened connections. To force all connections to use the new settings, make sure to reload the postgres configuration:

```
SELECT pg_reload_conf();
```

Warning: Citus versions prior to 9.2.4 require a full database restart to force all connections to use the new setting.

citus.local_hostname (text)

Citus nodes need occasionally to connect to themselves for systems operations. By default, they use the address `localhost` to refer to themselves, but this can cause problems. For instance, when a host requires `sslmode=verify-full` for incoming connections, adding `localhost` as an alternative hostname on the SSL certificate isn't always desirable – or even feasible.

`citus.local_hostname` selects the hostname a node uses to connect to itself. The default value is `localhost`.

```sql
ALTER SYSTEM SET citus.local_hostname TO 'mynode.example.com';
```

citus.show_shards_for_app_name_prefixes (text)

By default, Citus hides shards from the list of tables PostgreSQL gives to SQL clients. It does this because there are multiple shards per distributed table, and the shards can be distracting to the SQL client.

The citus.show_shards_for_app_name_prefixes GUC allows shards to be displayed for selected clients that want to see them. Its default value is `''`.

```sql
-- show shards to psql only (hide in other clients, like pgAdmin)

SET citus.show_shards_for_app_name_prefixes TO 'psql';

-- also accepts a comma separated list

SET citus.show_shards_for_app_name_prefixes TO 'psql,pg_dump';
```

15.3.2 Query Statistics

citus.stat_statements_purge_interval (integer)

Note: This GUC is now part of the Citus Community edition as of version 11.0!

Sets the frequency at which the maintenance daemon removes records from *citus_stat_statements* that are unmatched in `pg_stat_statements`. This configuration value sets the time interval between purges in seconds, with a default value of 10. A value of 0 disables the purges.

```sql
SET citus.stat_statements_purge_interval TO 5;
```

This parameter is effective on the coordinator and can be changed at runtime.

citus.stat_statements_max (integer)

Note: This GUC is now part of the Citus Community edition as of version 11.0!

The maximum number of rows to store in *citus_stat_statements*. Defaults to 50000, and may be changed to any value in the range 1000 - 10000000. Note that each row requires 140 bytes of storage, so setting stat_statements_max to its maximum value of 10M would consume 1.4GB of memory.

Changing this GUC will not take effect until PostgreSQL is restarted.

citus.stat_statements_track (enum)

Note: This GUC is now part of the Citus Community edition as of version 11.0!

Recording statistics for *citus_stat_statements* requires extra CPU resources. When the database is experiencing load, the administrator may wish to disable statement tracking. The `citus.stat_statements_track` GUC can turn tracking on and off.

- **all**: (default) Track all statements.
- **none**: Disable tracking.

15.3.3 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 are committed in a single round. Data may be lost if a commit fails after COPY succeeds on all placements (rare).

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.

citus.replicate_reference_tables_on_activate (boolean)

Reference table shards must be placed on all nodes which have distributed tables. By default, reference table shards are copied to a node at node activation time, that is, when such functions as *citus_add_node* or *citus_activate_node* are called. However, node activation might be an inconvenient time to copy the placements, because it can take a long time when there are large reference tables.

You can defer reference table replication by setting the `citus.replicate_reference_tables_on_activate` GUC to 'off'. Reference table replication will then happen when we create new shards on the node. For instance, when calling *create_distributed_table*, *create_reference_table*, or when the shard rebalancer moves shards to the new node.

The default value for this GUC is 'on'.

citus.metadata_sync_mode (enum)

Note: Requires superuser access to change.

This GUC determines how Citus synchronizes *metadata* across nodes. By default Citus updates all metadata in a single transaction for consistency. However, PostgreSQL has a hard memory limit related to cache invalidations, and Citus metadata syncing for a large cluster can fail from memory exhaustion.

As a workaround, Citus provides an optional nontransactional sync mode which uses a series of smaller transactions. While this mode works in limited memory, there's a possibility of transactions failing and leaving metadata in an inconsistency state. To help with this potential problem, nontransactional metadata sync is designed as an idempotent action, so you can re-run it repeatedly if needed.

There are two values for this GUC:

- **transactional:** (Default) Synchronize all metadata in a single transaction.
- **nontransactional:** Synchronize metadata using multiple small transactions.

Examples:

```sql
-- to add a new node and sync nontransactionally

SET citus.metadata_sync_mode TO 'nontransactional';
SELECT citus_add_node(<ip>, <port>);

-- to manually (re)sync

SET citus.metadata_sync_mode TO 'nontransactional';
SELECT start_metadata_sync_to_all_nodes();
```

We advise trying transactional mode first, and switching to nontransactional only if a memory failure occurs.

15.3.4 Planner Configuration

citus.local_table_join_policy (enum)

This GUC determines how Citus moves data when doing a join between local and distributed tables. Customizing the join policy can help reduce the amount of data sent between worker nodes.

Citus will send either the local or distributed tables to nodes as necessary to support the join. Copying table data is referred to as a "conversion." If a local table is converted, then it will be sent to any workers that need its data to perform the join. If a distributed table is converted, then it will be collected in the coordinator to support the join. The citus planner will send only the necessary rows doing a conversion.

There are four modes available to express conversion preference:

- **auto:** (Default) Citus will convert either all local or all distributed tables to support local and distributed table joins. Citus decides which to convert using a heuristic. It will convert distributed tables if they are joined using a constant filter on a unique index (such as a primary key). This ensures less data gets moved between workers.
- **never:** Citus will not allow joins between local and distributed tables.
- **prefer-local:** Citus will prefer converting local tables to support local and distributed table joins.
- **prefer-distributed:** Citus will prefer converting distributed tables to support local and distributed table joins. If the distributed tables are huge, using this option might result in moving lots of data between workers.

For example, assume `citus_table` is a distributed table distributed by the column `x`, and that `postgres_table` is a local table:

```sql
CREATE TABLE citus_table(x int primary key, y int);
SELECT create_distributed_table('citus_table', 'x');

CREATE TABLE postgres_table(x int, y int);

-- even though the join is on primary key, there isn't a constant filter
-- hence postgres_table will be sent to worker nodes to support the join
SELECT * FROM citus_table JOIN postgres_table USING (x);

-- there is a constant filter on a primary key, hence the filtered row
-- from the distributed table will be pulled to coordinator to support the join
SELECT * FROM citus_table JOIN postgres_table USING (x) WHERE citus_table.x = 10;

SET citus.local_table_join_policy to 'prefer-distributed';
-- since we prefer distributed tables, citus_table will be pulled to coordinator
-- to support the join. Note that citus_table can be huge.
SELECT * FROM citus_table JOIN postgres_table USING (x);

SET citus.local_table_join_policy to 'prefer-local';
-- even though there is a constant filter on primary key for citus_table
-- postgres_table will be sent to necessary workers because we are using 'prefer-local'.
SELECT * FROM citus_table JOIN postgres_table USING (x) WHERE citus_table.x = 10;
```

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)

Note: This GUC is applicable for queries against *Reference Tables*.

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.

15.3.5 Intermediate Data Transfer

citus.binary_worker_copy_format (boolean)

Use the binary copy format to transfer intermediate data between workers. During large table joins, Citus may have to dynamically repartition and shuffle data between different workers. For Postgres 13 and lower, the default for this setting is `false`, which means text encoding is used to transfer this data. For Postgres 14 and higher, the default is `true`. Setting this parameter is `true` instructs the database to use PostgreSQL's binary serialization format to transfer data. The 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.max_intermediate_result_size (integer)

The maximum size in KB of intermediate results for CTEs that are unable to be pushed down to worker nodes for execution, and for 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.

15.3.6 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*.

citus.enable_local_reference_table_foreign_keys (boolean)

This setting, enabled by default, allows foreign keys to be created between reference and local tables. For the feature to work, the coordinator node must be registered with itself, using *citus_add_node*.

Note that foreign keys between reference tables and local tables come at a slight cost. When you create the foreign key, Citus must add the plain table to Citus' metadata, and track it in *Partition table*. Local tables that are added to metadata inherit the same limitations as reference tables (see *Creating and Modifying Distributed Objects (DDL)* and *SQL Support and Workarounds*)..

If you drop the foreign keys, Citus will automatically remove such local tables from metadata, which eliminates such limitations on those tables.

citus.enable_change_data_capture (boolean)

This setting, disabled by default, causes Citus to alter the `wal2json` and `pgoutput` logical decoders to work with distributed tables. Specifically, it rewrites the names of shards (e.g. `foo_102027`) in decoder output to the base names of the distributed tables (e.g. `foo`). It also avoids publishing duplicate events during tenant isolation and shard split/move/rebalance operations.

For an example of using this GUC, see *Change Data Capture (CDC)*.

15.3.7 Executor Configuration

General

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.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;
```

citus.propagate_set_commands (enum)

Determines which SET commands are propagated from the coordinator to workers. The default value for this parameter is 'none'.

The supported values are:

- **none:** no SET commands are propagated.
- **local:** only SET LOCAL commands are propagated.

citus.enable_repartition_joins (boolean)

Ordinarily, attempting to perform *Repartition joins* with the adaptive executor will fail with an error message. However, setting `citus.enable_repartition_joins` to true allows Citus to perform the join. The default value is false.

citus.enable_repartitioned_insert_select (boolean)

By default, an INSERT INTO … SELECT statement that cannot be pushed down will attempt to repartition rows from the SELECT statement and transfer them between workers for insertion. However, if the target table has too many shards then repartitioning will probably not perform well. The overhead of processing the shard intervals when determining how to partition the results is too great. Repartitioning can be disabled manually by setting `citus.enable_repartitioned_insert_select` to false.

citus.enable_binary_protocol (boolean)

Setting this parameter to true instructs the coordinator node to use PostgreSQL's binary serialization format (when applicable) to transfer data with workers. Some column types do not support binary serialization.

Enabling this parameter is mostly useful when the workers must return large amounts of data. Examples are when a lot of rows are requested, the rows have many columns, or they use big types such as `hll` from the postgresql-hll extension.

The default value is `true` for Postgres versions 14 and higher. For Postgres versions 13 and lower the default is `false`, which means all results are encoded and transferred in text format.

citus.max_shared_pool_size (integer)

Specifies the maximum number of connections that the coordinator node, across all simultaneous sessions, is allowed to make per worker node. PostgreSQL must allocate fixed resources for every connection and this GUC helps ease connection pressure on workers.

Without connection throttling, every multi-shard query creates connections on each worker proportional to the number of shards it accesses (in particular, up to #shards/#workers). Running dozens of multi-shard queries at once can easily hit worker nodes' `max_connections` limit, causing queries to fail.

By default, the value is automatically set equal to the coordinator's own `max_connections`, which isn't guaranteed to match that of the workers (see the note below). The value -1 disables throttling.

Note: There are certain operations that do not obey citus.max_shared_pool_size, most importantly repartition joins. That's why it can be prudent to increase the max_connections on the workers a bit higher than max_connections on the coordinator. This gives extra space for connections required for repartition queries on the workers.

citus.max_adaptive_executor_pool_size (integer)

Whereas *citus.max_shared_pool_size (integer)* limits worker connections across all sessions, `max_adaptive_executor_pool_size` limits worker connections from just the *current* session. This GUC is useful for:

- Preventing a single backend from getting all the worker resources
- Providing priority management: designate low priority sessions with low max_adaptive_executor_pool_size, and high priority sessions with higher values

The default value is 16.

citus.executor_slow_start_interval (integer)

Time to wait in milliseconds between opening connections to the same worker node.

When the individual tasks of a multi-shard query take very little time, they can often be finished over a single (often already cached) connection. To avoid redundantly opening additional connections, the executor waits between connection attempts for the configured number of milliseconds. At the end of the interval, it increases the number of connections it is allowed to open next time.

For long queries (those taking >500ms), slow start might add latency, but for short queries it's faster. The default value is 10ms.

citus.max_cached_conns_per_worker (integer)

Each backend opens connections to the workers to query the shards. At the end of the transaction, the configured number of connections is kept open to speed up subsequent commands. Increasing this value will reduce the latency of multi-shard queries, but will also increase overhead on the workers.

The default value is 1. A larger value such as 2 might be helpful for clusters that use a small number of concurrent sessions, but it's not wise to go much further (e.g. 16 would be too high).

citus.force_max_query_parallelization (boolean)

Simulates the deprecated and now nonexistent real-time executor. This is used to open as many connections as possible to maximize query parallelization.

When this GUC is enabled, Citus will force the adaptive executor to use as many connections as possible while executing a parallel distributed query. If not enabled, the executor might choose to use fewer connections to optimize overall query execution throughput. Internally, setting this true will end up using one connection per task.

One place where this is useful is in a transaction whose first query is lightweight and requires few connections, while a subsequent query would benefit from more connections. Citus decides how many connections to use in a transaction based on the first statement, which can throttle other queries unless we use the GUC to provide a hint.

```
BEGIN;
-- add this hint
SET citus.force_max_query_parallelization TO ON;

-- a lightweight query that doesn't require many connections
SELECT count(*) FROM table WHERE filter = x;

-- a query that benefits from more connections, and can obtain
-- them since we forced max parallelization above
SELECT ... very .. complex .. SQL;
COMMIT;
```

The default value is false.

Explain output

citus.explain_all_tasks (boolean)

By default, Citus shows the output of a single, arbitrary task when running EXPLAIN on a distributed query. In most cases, the explain output will be similar across tasks. 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.

citus.explain_analyze_sort_method (enum)

Determines the sort method of the tasks in the output of EXPLAIN ANALYZE. The default value of citus.explain_analyze_sort_method is `execution-time`.

The supported values are:

- **execution-time:** sort by execution time.
- **taskId:** sort by task id.

CHAPTER

SIXTEEN

EXTERNAL INTEGRATIONS

16.1 Change Data Capture (CDC)

Note: Logical decoding for distributed tables is a preview feature in Citus v11.3.

16.1.1 Differences from single-node PostgreSQL

PostgreSQL provides logical decoding to export events of interest from its write-ahead log (WAL) to other programs or databases. The WAL keeps track of all committed data transactions; it is the authority on all data changes happening on a PostgreSQL instance. By subscribing to a logical replication slot, an application can get real-time notification of the changes.

Consuming change events from a Citus cluster rather than single-node PostgreSQL adds two sets of challenges:

1. **Client subscription complexity.** Any logical decoding publication must be created in duplicate across all nodes. An interested application must subscribe to all of them. Furthermore, events within each publication are guaranteed to be emitted in the order they happen in the node, but this guarantee does not hold for events arriving from multiple nodes.

2. **Shard complexity.** (Automatically handled by Citus v11.3+ when the *citus.enable_change_data_capture (boolean)* GUC is enabled.) Citus nodes store shards as physical tables. For example, a distributed table called `foo` is stored across multiple tables with names such as `foo_102027`, `foo_102028`, etc. The underlying shard names should be translated back to the conceptual name of the distributed table for CDC. Also administrative actions like moving shards between worker nodes generates WAL events, but these spurious events should be hidden from CDC.

If the *citus.enable_change_data_capture (boolean)* GUC is enabled, Citus automatically handles the first set of issues, normalizing shard names and suppressing events from shard creation and movement. The second set of issues, however, are the client's responsibility.

16.1.2 Logical replication of distributed tables to PostgreSQL tables

Postgres' CDC with logical decoding forms the basis of logical replication, which allows you to replicate table changes from one PostgreSQL server to another. As an example, let's subscribe to changes from a distributed table in a Citus cluster, and populate a regular table in a single-node PostgreSQL server.

First, enable logical decoding on all Citus nodes. In the postgresql.conf file of each of them, configure the WAL and replication slots:

```
# add to postgresql.conf on all nodes

# include enough information in WAL for logical decoding

wal_level = logical

# we need at least one replication slot, but leave headroom

max_replication_slots = 10

# enable Citus' logical decoding improvements for dist tables

citus.enable_change_data_capture = 'on'
```

After changing the configuration files, restart the PostgreSQL server in each Citus node.

Create an example distributed table on the coordinator, and a publication to share changes from the table.

```
-- example distributed table

CREATE TABLE items (
  key TEXT PRIMARY KEY,
  value TEXT
);

SELECT create_distributed_table('items', 'key');

-- citus will propagate to workers, creating publications there too

CREATE PUBLICATION items_pub FOR TABLE items;

-- fill with sample data

INSERT INTO items SELECT generate_series(1,10), 'a value';
```

Note: Be careful *not* to create a publication for all tables.

```
-- don't do this!
CREATE PUBLICATION FOR ALL TABLES;
```

Creating publications for all tables includes Citus' own *metadata tables*, which can cause problems.

Check that the worker nodes have the correct hostname set for the coordinator, and adjust with *citus_set_coordinator_host* if necessary. Then, create logical replication slots on all nodes by running the following on the coordinator:

```
-- create replication slots on all nodes, and choose the pgoutput format

SELECT * FROM run_command_on_all_nodes($$
  SELECT pg_create_logical_replication_slot('cdc_slot', 'pgoutput', false)
$$);
```

Finally, on the single node PostgreSQL server, create the `items` table, and set up subscriptions to read rows from all Citus nodes.

```
-- on the standalone PostgreSQL server

-- match the table structure we have on Citus

CREATE TABLE items (
  key TEXT PRIMARY KEY,
  value TEXT
);

-- subscribe to coordinator

CREATE SUBSCRIPTION subc
  CONNECTION 'dbname=<dbname> host=<coordinator host> user=<username> port=<coordinator port>'
  PUBLICATION items_pub
  WITH (copy_data=true,create_slot=false,slot_name='cdc_slot');

-- subscribe to each worker (assuming there are two workers)

CREATE SUBSCRIPTION subw0
  CONNECTION 'dbname=<dbname> host=<worker1 host> user=<user> port=<worker1 port>'
  PUBLICATION items_pub
  WITH (copy_data=false,create_slot=false,slot_name='cdc_slot');

CREATE SUBSCRIPTION subw1
  CONNECTION 'dbname=<dbname> host=<worker2 host> user=<user> port=<worker2 port>'
  PUBLICATION items_pub
  WITH (copy_data=false,create_slot=false,slot_name='cdc_slot');
```

Note: The `copy_data` argument should be set to true on the coordinator subscription, to copy any existing data to the CDC client. However, `copy_data` should be false for worker subscriptions, otherwise it'll result in duplicate data in the CDC client and cause replication errors.

16.1.3 Logical decoding caveats

- **Support for only the pgoutput and wal2json formats, so far.** You can currently use these two decoders (if installed). Other decoders will not have the distributed table name normalizing applied to them
- **Replication slots need to be created separately on each node.** While the initial creation is easy with *run_command_on_all_nodes*, there is still extra work when adding a new node. You need to create a replication slot (or subscription) before rebalancing shards.
- **Cross-node changes arriving out of order.** Changes happening on the same node always arrive in the same order, but since your client will listen to each node separately, there are no guarantees regarding the order of changes happening across different nodes.
- **Distributed table modification restriction.** Using *alter_distributed_table* will break the replication stream.
- **No columnar tables.** logical decoding does not work with *Columnar Storage*.
- **The need for a consumer.** When doing logical decoding (with or without Citus), be sure there are subscribers consuming publications, or else the WAL data will accumulate and cause problems for the publishing database.

16.2 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
```

(continues on next page)

(continued from previous page)

```
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

```

(continues on next page)

(continued from previous page)

```
# 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 │
└─────────
```

16.2.1 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 our cloud_topic, 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.

16.3 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 Citus 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 sparkcitus
```

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

```
// add this to build.sbt

name := "sparkcitus"
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 `sparkcitus/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
```

(continues on next page)

(continued from previous page)

```
        .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},
```

(continues on next page)

(continued from previous page)

```
{"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 Citus:

```
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 sparkcitus 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:sparkcitus> 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"

```

(continues on next page)

(continued from previous page)

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

This uses the CopyHelper to ingest the ionformation. 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.

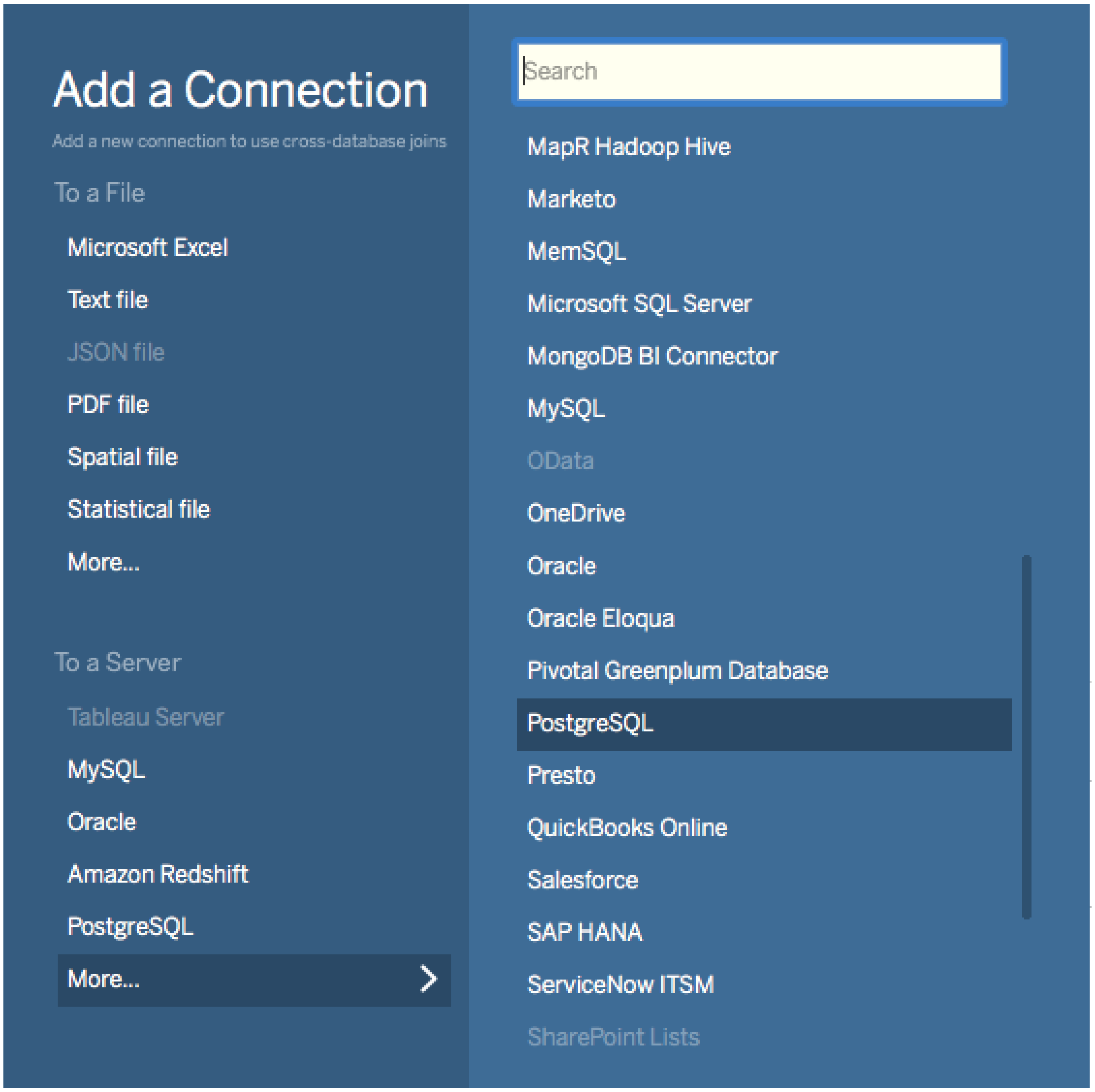
A more complicated, but robust, approach is to use the custom Spark partitioner spark-citus so that partitions match up exactly with Citus shards. This allows running transactions directly on worker nodes which can rollback on read failure. See the presentation linked in that repository for more information.

16.4 Business Intelligence with Tableau

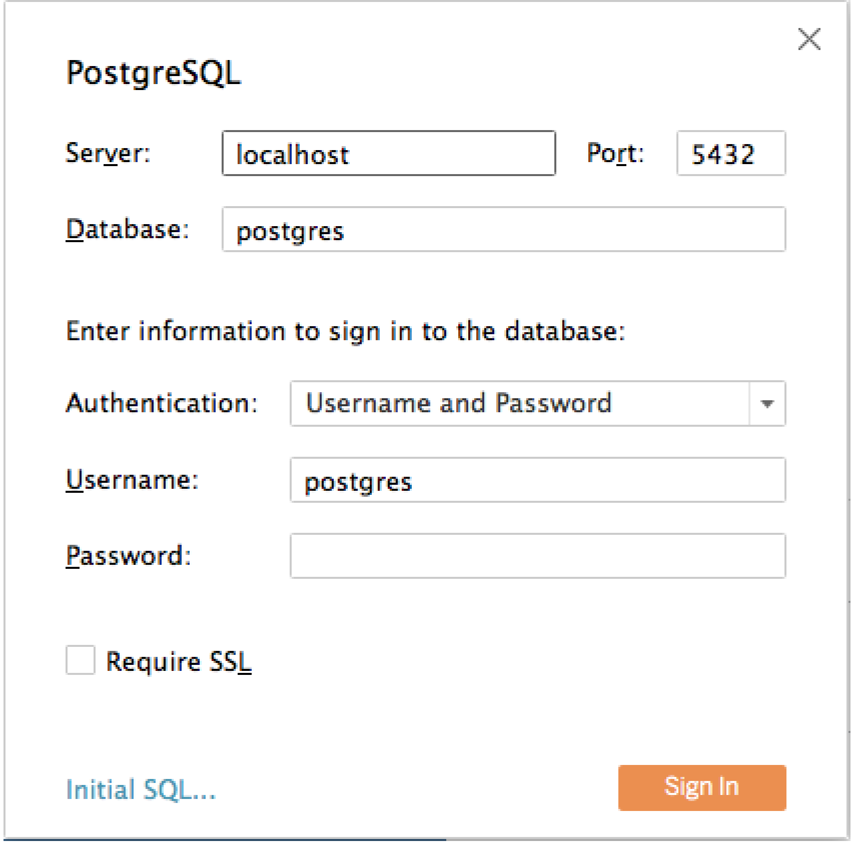
Tableau is a popular business intelligence and analytics tool for databases. Citus and Tableau provide a seamless experience for performing ad-hoc reporting or analysis.

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 Citus cluster. (Note if you're connecting to our cloud_topic you must select "Require SSL.")



PostgreSQL

Server: localhost

Port: 5432

Database: postgres

Enter information to sign in to the database:

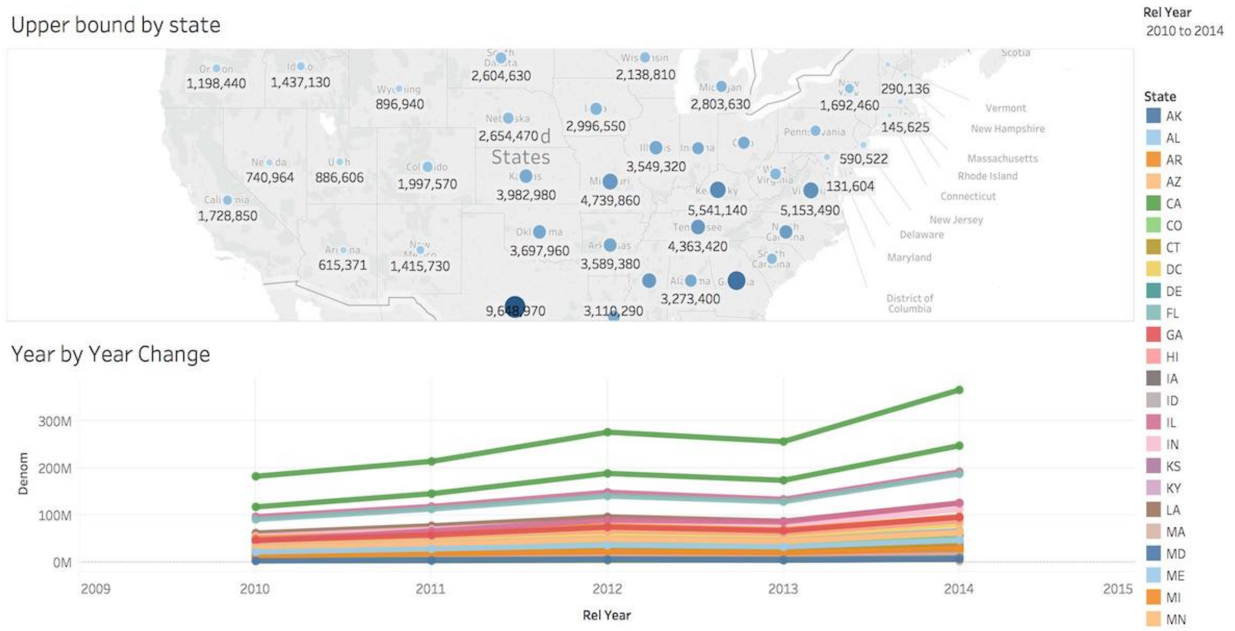
Authentication: Username and Password

Username: postgres

Password:

□ Require SSL

- 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. Citus can compute aggregations over this range in human real-time.


Upper bound by state
Rel Year
2010 to 2014
1,198,440
1,437,130
896,940
2,604,630
2,138,810
2,803,630
1,692,460
290,136
145,625
2,654,470
2,996,550
States
740,964
886,606
1,997,570
3,982,980
3,549,320
590,522
1,728,850
4,739,860
5,541,140
131,604
5,153,490
3,697,960
4,363,420
615,371
1,415,730
3,589,380
3,273,400
9,648,970
3,110,290
Vermont
New Hampshire
Massachusetts
Rhode Island
Connecticut
New Jersey
Delaware
Maryland
District of Columbia
Scotia
Year by Year Change
Denom
300M
200M
100M
0M
2009
2010
2011
2012
2013
2014
2015
Rel Year
State
AK
AL
AR
AZ
CA
CO
CT
DC
DE
FL
GA
HI
IA
ID
IL
IN
KS
KY
LA
MA
MD
ME
MI
MN

CHAPTER

SEVENTEEN

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 Community edition 11.0+ supports fully online shard rebalancing. We discuss briefly the functions provided by the shard rebalancer 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.

17.1 Choosing Cluster Size

This section explores configuration settings for running a cluster in production.

17.1.1 Shard Count

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. If you decide to change the shard count of a table after distributing, you can use the *alter_distributed_table* function.

Multi-Tenant SaaS Use-Case

The optimal choice varies depending on your access patterns for the data. For instance, in the *Multi-Tenant SaaS 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 Citus 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)`.

17.2 Initial Hardware Size

The size of a cluster, in terms of number of nodes and their hardware capacity, is easy to change. (Scaling on our cloud_topic 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.

17.2.1 Multi-Tenant SaaS Use-Case

For those migrating to Citus 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).

17.2.2 Real-Time Analytics Use-Case

Total cores: when working data fits in RAM, you can expect a linear performance improvement on Citus 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 Citus. 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.

17.3 Scaling the cluster

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

17.3.1 Add a worker

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

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

```
SELECT * from citus_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.

Note: If your cluster has very large reference tables, they can slow down the addition of a node. In this case, consider the *citus.replicate_reference_tables_on_activate (boolean)* GUC.

Also, new nodes synchronize Citus' *metadata* upon creation. By default, the sync happens inside a single transaction for consistency. However, in a big cluster with large amounts of metadata, the transaction can run out of memory and fail. If you encounter this situation, you can choose a non-transactional metadata sync mode with the *citus.metadata_sync_mode (enum)* GUC.

17.3.2 Rebalance Shards without Downtime

Note: Starting in version 11.0, Citus Community edition now supports non-blocking reads *and* writes during rebalancing.

If you want to move existing shards to a newly added worker, Citus provides a *citus_rebalance_start* function to make it easier. This function will distribute shards evenly among the workers.

The function is configurable to rebalance shards according to a number of strategies, to best match your database workload. See the function reference to learn which strategy to choose. Here's an example of rebalancing shards using the default strategy:

```
SELECT citus_rebalance_start();
```

Many products, like multi-tenant SaaS applications, cannot tolerate downtime, and on our managed service, 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.

Parallel Rebalancing

This operation carries out multiple shard moves in a sequential order by default. There are some cases where you may prefer to rebalance faster at the expense of using more resources such as network bandwidth. In those situations, customers are able to configure a rebalance operation to perform a number of shard moves in parallel.

The *citus.max_background_task_executors_per_node (integer)* GUC allows tasks such as shard rebalancing to operate in parallel. You can increase it from its default value (1) as desired to boost parallelism.

```
ALTER SYSTEM SET citus.max_background_task_executors_per_node = 2;
SELECT pg_reload_conf();

SELECT citus_rebalance_start();
```

What are the typical use cases?

- Scaling out faster when adding new nodes to the cluster
- Rebalancing the cluster faster to even out the utilization of nodes

Corner Cases and Gotchas

`citus.max_background_task_executors_per_node` value limits the number of parallel task executors in general. Also, shards in the same colocation group will always move sequentially so parallelism may be limited by the number of colocation groups.

How it Works

Citus's 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. Citus 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. Here's how to fix it.

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.

17.3.3 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.

17.4 Dealing With Node Failures

In this subsection, we discuss how you can deal with node failures without incurring any downtime on your Citus cluster.

17.4.1 Worker Node Failures

Citus uses PostgreSQL streaming replication, allowing it to tolerate worker-node failures. This option 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 our cloud_topic which is pre-configured for replication and high-availability.

17.4.2 Coordinator Node Failures

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

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

17.5 Tenant Isolation

Note: Starting in version 11.0, Citus Community edition includes tenant isolation functionality!

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 provides 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 citus_shards
 WHERE shardid = 102240;

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

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

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

17.6 Viewing Query Statistics

Note: Starting in version 11.0, Citus Community edition now includes the citus_stat_statements view!

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_statments. 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 our cloud_topic 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:

```
-[ RECORD 1 ]-+----------------------------------------------
queryid       | -6844578505338488014
userid        | 10
dbid          | 13340
query         | SELECT count(*) FROM foo;
executor      | adaptive
partition_key |
calls         | 1
-[ RECORD 2 ]-+----------------------------------------------
queryid       | 185453597994293667
userid        | 10
dbid          | 13340
query         | insert into foo select generate_series($1,$2)
```

(continues on next page)

(continued from previous page)

```
executor      | insert-select
partition_key |
calls         | 1
-[ RECORD 3 ]-+-----------------------------------------------
queryid       | 1301170733886649828
userid        | 10
dbid          | 13340
query         | SELECT * FROM foo WHERE id = $1
executor      | adaptive
partition_key | 42
calls         | 1
```

We can see that Citus uses the adaptive executor most commonly to run queries. This executor fragments the query into constituent queries to run on relevant nodes, 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. 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.

17.6.1 Tenant-level Statistics

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 scoped to a single tenant?"

```
SELECT sum(calls),
       partition_key IS NOT NULL AS single_tenant
FROM citus_stat_statements
GROUP BY 2;
```

```
.
 sum | single_tenant
-----+---------------
   2 | f
   1 | t
```

In a multi-tenant database, for instance, we would expect the vast majority of queries to be single tenant. Seeing too many multi-tenant queries may indicate that queries do not have the proper filters to match a tenant, and are using unnecessary resources.

To investigate which tenants in particular are most active, you can use the *Tenant-level query statistics view*.

17.6.2 Statistics Expiration

The pg_stat_statements view limits the number of statements it tracks, and the duration of its records. Because citus_stat_statements tracks a strict subset of the queries in pg_stat_statements, a choice of equal limits for the two views would cause a mismatch in their data retention. Mismatched records can cause joins between the views to behave unpredictably.

There are three ways to help synchronize the views, and all three can be used together.

1. Have the maintenance daemon periodically sync the citus and pg stats. The GUC `citus.stat_statements_purge_interval` sets time in seconds for the sync. A value of 0 disables periodic syncs.

2. Adjust the number of entries in citus_stat_statements. The `citus.stat_statements_max` GUC removes old entries when new ones cross the threshold. The default value is 50K, and the highest allowable value is 10M. Note that each entry costs about 140 bytes in shared memory so set the value wisely.
3. Increase `pg_stat_statements.max`. Its default value is 5000, and could be increased to 10K, 20K or even 50K without much overhead. This is most beneficial when there is more local (i.e. coordinator) query workload.

Note: Changing `pg_stat_statements.max` or `citus.stat_statements_max` requires restarting the PostgreSQL service. Changing `citus.stat_statements_purge_interval`, on the other hand, will come into effect with a call to pg_reload_conf().

17.7 Resource Conservation

17.7.1 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.

To customize the timeout per query, use `SET LOCAL` in a transaction:

```
BEGIN;
-- this limit applies to just the current transaction
SET LOCAL statement_timeout TO 300000;

-- ...
COMMIT;
```

17.8 Security

17.8.1 Connection Management

Note: Since Citus version 8.1.0 (released 2018-12-17) the traffic between the different nodes in the cluster is encrypted for NEW installations. This is done by using TLS with self-signed certificates. This means that this **does not protect against Man-In-The-Middle attacks.** This only protects against passive eavesdropping on the network.

Clusters originally created with a Citus version before 8.1.0 do not have any network encryption enabled between nodes (even if upgraded later). To set up self-signed TLS on on this type of installation follow the steps in official postgres documentation together with the citus specific settings described here, i.e. changing `citus.node_conninfo` to `sslmode=require`. This setup should be done on coordinator and workers.

When Citus nodes communicate with one another they consult 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:

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

ALTER SYSTEM SET citus.node_conninfo =
  'sslrootcert=/path/to/citus-ca.crt sslcrl=/path/to/citus-ca.crl sslmode=verify-full';
```

There is a whitelist of parameters that the GUC accepts, see the *node_conninfo* reference for details. As of Citus 8.1, the default value for node_conninfo is `sslmode=require`, which prevents unencrypted communication between nodes. If your cluster was originally created before Citus 8.1 the value will be `sslmode=prefer`. After setting up self-signed certificates on all nodes it's recommended to change this setting to `sslmode=require`.

After changing this setting it is important to reload the postgres configuration. Even though the changed setting might be visible in all sessions, the setting is only consulted by Citus when new connections are established. When a reload signal is received, Citus marks all existing connections to be closed which causes a reconnect after running transactions have been completed.

```sql
SELECT pg_reload_conf();
```

Warning: Citus versions before 9.2.4 require a restart for existing connections to be closed.

For these versions a reload of the configuration does not trigger connection ending and subsequent reconnecting. Instead the server should be restarted to enforce all connections to use the new settings.

```sql
-- only superusers can access this table

-- add a password for user jdoe
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.

```sql
-- update user jdoe to use certificate authentication
UPDATE pg_dist_authinfo
SET authinfo = 'sslcert=/path/to/user.crt sslkey=/path/to/user.key'
WHERE nodeid = 123 AND rolename = 'jdoe';
```

This changes the user from using a password to use a certificate and keyfile while connecting to node 123 instead. Make sure the user certificate is signed by a certificate that is trusted by the worker you are connecting to and authentication settings on the worker allow for certificate based authentication. Full documentation on how to use client certificates can be found in the postgres libpq documentation.

Changing `pg_dist_authinfo` does not force any existing connection to reconnect.

17.8.2 Setup Certificate Authority signed certificates

This section assumes you have a trusted Certificate Authority that can issue server certificates to you for all nodes in your cluster. It is recommended to work with the security department in your organization to prevent key material from being handled incorrectly. This guide covers only Citus specific configuration that needs to be applied, not best practices for PKI management.

For all nodes in the cluster you need to get a valid certificate signed by the *same Certificate Authority*. The following **machine specific** files are assumed to be available on every machine:

- `/path/to/server.key`: Server Private Key
- `/path/to/server.crt`: Server Certificate or Certificate Chain for Server Key, signed by trusted Certificate Authority.

Next to these machine specific files you need these cluster or CA wide files available:

- `/path/to/ca.crt`: Certificate of the Certificate Authority
- `/path/to/ca.crl`: Certificate Revocation List of the Certificate Authority

Note: The Certificate Revocation List is likely to change over time. Work with your security department to set up a mechanism to update the revocation list on to all nodes in the cluster in a timely manner. A reload of every node in the cluster is required after the revocation list has been updated.

Once all files are in place on the nodes, the following settings need to be configured in the Postgres configuration file:

```
# the following settings allow the postgres server to enable ssl, and
# configure the server to present the certificate to clients when
# connecting over tls/ssl
ssl = on
ssl_key_file = '/path/to/server.key'
ssl_cert_file = '/path/to/server.crt'

# this will tell citus to verify the certificate of the server it is connecting to
citus.node_conninfo = 'sslmode=verify-full sslrootcert=/path/to/ca.crt sslcrl=/path/to/
↪ca.crl'
```

After changing, either restart the database or reload the configuration to apply these changes. A restart is required if a Citus version below 9.2.4 is used. Also, adjusting *citus.local_hostname (text)* may be required for proper functioning with `sslmode=verify-full`.

Depending on the policy of the Certificate Authority used you might need or want to change `sslmode=verify-full` in `citus.node_conninfo` to `sslmode=verify-ca`. For the difference between the two settings please consult the official postgres documentation.

Lastly, to prevent any user from connecting via an un-encrypted connection, changes need to be made to `pg_hba.conf`. Many Postgres installations will have entries allowing `host` connections which allow SSL/TLS connections as well as plain TCP connections. By replacing all `host` entries with `hostssl` entries, only encrypted connections will be allowed to authenticate to Postgres. For full documentation on these settings take a look at the pg_hba.conf file documentation on the official Postgres documentation.

Note: When a trusted Certificate Authority is not available, one can create their own via a self-signed root certificate. This is non-trivial and the developer or operator should seek guidance from their security team when doing so.

To verify the connections from the coordinator to the workers are encrypted you can run the following query. It will show the SSL/TLS version used to encrypt the connection that the coordinator uses to talk to the worker:

```
SELECT run_command_on_workers($$
  SELECT version FROM pg_stat_ssl WHERE pid = pg_backend_pid()
$$);
```

```
 ────────────────────────────
│   run_command_on_workers   │
 ────────────────────────────
│  (localhost,9701,t,TLSv1.2) │
│  (localhost,9702,t,TLSv1.2) │
 ────────────────────────────
(2 rows)
```

17.8.3 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 and a ssl/tls connection to nodes in the local
# network. The following ranges correspond to 24, 20, and 16-bit blocks
# in Private IPv4 address spaces.
hostssl    all             all             10.0.0.0/8              md5

# Require passwords and ssl/tls connections when the host connects to
# itself as well.
hostssl    all             all             127.0.0.1/32            md5
hostssl    all             all             ::1/128                 md5
```

The coordinator node needs to know roles' passwords in order to communicate with the workers. Our cloud_topic keeps track of that kind of information for you. 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.

17.8.4 Row-Level Security

Note: Starting in version 11.0, Citus Community edition now supports row-level security for distributed tables.

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`. Then run the following as an administrator:

```sql
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.

```sql
-- 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:  new row violates row-level security policy for table "events_102055"
*/
```

17.9 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 is one thing to keep in mind. While including other extensions in shared_preload_libraries, you should make sure that Citus is the first extension.

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.

17.10 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 Citus extension within that database, and register the workers in the coordinator database.

Connect to each of the worker nodes and run:

```
-- on every worker node

CREATE DATABASE newbie;
\c newbie
CREATE EXTENSION citus;
```

Then, on the coordinator:

```
CREATE DATABASE newbie;
\c newbie
CREATE EXTENSION citus;

SELECT * from citus_add_node('node-name', 5432);
SELECT * from citus_add_node('node-name2', 5432);
-- ... for all of them
```

Now the new database will be operating as another Citus cluster.

CHAPTER

EIGHTEEN

TABLE MANAGEMENT

18.1 Determining Table and Relation Size

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

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

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

```sql
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 │
└───────────────┴───────┘
```

18.2 Vacuuming Distributed Tables

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

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

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

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

```
VACUUM my_distributed_table;
```

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

18.3 Analyzing Distributed Tables

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

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

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

```
ANALYZE my_distributed_table;
```

Citus propagates the ANALYZE command to all worker node placements.

18.4 Columnar Storage

Citus 10 introduces append-only columnar table storage for analytic and data warehousing workloads. When columns (rather than rows) are stored contiguously on disk, data becomes more compressible, and queries can request a subset of columns more quickly.

18.4.1 Usage

To use columnar storage, specify `USING columnar` when creating a table:

```sql
CREATE TABLE contestant (
    handle TEXT,
    birthdate DATE,
    rating INT,
    percentile FLOAT,
    country CHAR(3),
    achievements TEXT[]
) USING columnar;
```

You can also convert between row-based (heap) and columnar storage.

```sql
-- Convert to row-based (heap) storage
SELECT alter_table_set_access_method('contestant', 'heap');

-- Convert to columnar storage (indexes will be dropped)
SELECT alter_table_set_access_method('contestant', 'columnar');
```

Citus converts rows to columnar storage in “stripes” during insertion. Each stripe holds one transaction’s worth of data, or 150000 rows, whichever is less. (The stripe size and other parameters of a columnar table can be changed with the *alter_columnar_table_set* function.)

For example, the following statement puts all five rows into the same stripe, because all values are inserted in a single transaction:

```sql
-- insert these values into a single columnar stripe

INSERT INTO contestant VALUES
  ('a','1990-01-10',2090,97.1,'XA','{a}'),
  ('b','1990-11-01',2203,98.1,'XA','{a,b}'),
  ('c','1988-11-01',2907,99.4,'XB','{w,y}'),
  ('d','1985-05-05',2314,98.3,'XB','{}'),
  ('e','1995-05-05',2236,98.2,'XC','{a}');
```

It’s best to make large stripes when possible, because Citus compresses columnar data separately per stripe. We can see facts about our columnar table like compression rate, number of stripes, and average rows per stripe by using *VACUUM VERBOSE*:

```sql
VACUUM VERBOSE contestant;
```

```
INFO:  statistics for "contestant":
storage id: 10000000000
total file size: 24576, total data size: 248
compression rate: 1.31x
```

(continues on next page)

(continued from previous page)

```
total row count: 5, stripe count: 1, average rows per stripe: 5
chunk count: 6, containing data for dropped columns: 0, zstd compressed: 6
```

The output shows that Citus used the zstd compression algorithm to obtain 1.31x data compression. The compression rate compares a) the size of inserted data as it was staged in memory against b) the size of that data compressed in its eventual stripe.

Because of how it's measured, the compression rate may or may not match the size difference between row and columnar storage for a table. The only way truly find that difference is to construct a row and columnar table that contain the same data, and compare.

18.4.2 Measuring compression

Let's create a new example with more data to benchmark the compression savings.

```
-- first a wide table using row storage
CREATE TABLE perf_row(
  c00 int8, c01 int8, c02 int8, c03 int8, c04 int8, c05 int8, c06 int8, c07 int8, c08␣
↪int8, c09 int8,
  c10 int8, c11 int8, c12 int8, c13 int8, c14 int8, c15 int8, c16 int8, c17 int8, c18␣
↪int8, c19 int8,
  c20 int8, c21 int8, c22 int8, c23 int8, c24 int8, c25 int8, c26 int8, c27 int8, c28␣
↪int8, c29 int8,
  c30 int8, c31 int8, c32 int8, c33 int8, c34 int8, c35 int8, c36 int8, c37 int8, c38␣
↪int8, c39 int8,
  c40 int8, c41 int8, c42 int8, c43 int8, c44 int8, c45 int8, c46 int8, c47 int8, c48␣
↪int8, c49 int8,
  c50 int8, c51 int8, c52 int8, c53 int8, c54 int8, c55 int8, c56 int8, c57 int8, c58␣
↪int8, c59 int8,
  c60 int8, c61 int8, c62 int8, c63 int8, c64 int8, c65 int8, c66 int8, c67 int8, c68␣
↪int8, c69 int8,
  c70 int8, c71 int8, c72 int8, c73 int8, c74 int8, c75 int8, c76 int8, c77 int8, c78␣
↪int8, c79 int8,
  c80 int8, c81 int8, c82 int8, c83 int8, c84 int8, c85 int8, c86 int8, c87 int8, c88␣
↪int8, c89 int8,
  c90 int8, c91 int8, c92 int8, c93 int8, c94 int8, c95 int8, c96 int8, c97 int8, c98␣
↪int8, c99 int8
);

-- next a table with identical columns using columnar storage
CREATE TABLE perf_columnar(LIKE perf_row) USING COLUMNAR;
```

Fill both tables with the same large dataset:

```
INSERT INTO perf_row
  SELECT
    g % 00500, g % 01000, g % 01500, g % 02000, g % 02500, g % 03000, g % 03500, g %␣
↪04000, g % 04500, g % 05000,
    g % 05500, g % 06000, g % 06500, g % 07000, g % 07500, g % 08000, g % 08500, g %␣
↪09000, g % 09500, g % 10000,
    g % 10500, g % 11000, g % 11500, g % 12000, g % 12500, g % 13000, g % 13500, g %␣
↪14000, g % 14500, g % 15000,
```

(continues on next page)

(continued from previous page)

```
    g % 15500, g % 16000, g % 16500, g % 17000, g % 17500, g % 18000, g % 18500, g %␣
↪19000, g % 19500, g % 20000,
    g % 20500, g % 21000, g % 21500, g % 22000, g % 22500, g % 23000, g % 23500, g %␣
↪24000, g % 24500, g % 25000,
    g % 25500, g % 26000, g % 26500, g % 27000, g % 27500, g % 28000, g % 28500, g %␣
↪29000, g % 29500, g % 30000,
    g % 30500, g % 31000, g % 31500, g % 32000, g % 32500, g % 33000, g % 33500, g %␣
↪34000, g % 34500, g % 35000,
    g % 35500, g % 36000, g % 36500, g % 37000, g % 37500, g % 38000, g % 38500, g %␣
↪39000, g % 39500, g % 40000,
    g % 40500, g % 41000, g % 41500, g % 42000, g % 42500, g % 43000, g % 43500, g %␣
↪44000, g % 44500, g % 45000,
    g % 45500, g % 46000, g % 46500, g % 47000, g % 47500, g % 48000, g % 48500, g %␣
↪49000, g % 49500, g % 50000
  FROM generate_series(1,50000000) g;

INSERT INTO perf_columnar
  SELECT
    g % 00500, g % 01000, g % 01500, g % 02000, g % 02500, g % 03000, g % 03500, g %␣
↪04000, g % 04500, g % 05000,
    g % 05500, g % 06000, g % 06500, g % 07000, g % 07500, g % 08000, g % 08500, g %␣
↪09000, g % 09500, g % 10000,
    g % 10500, g % 11000, g % 11500, g % 12000, g % 12500, g % 13000, g % 13500, g %␣
↪14000, g % 14500, g % 15000,
    g % 15500, g % 16000, g % 16500, g % 17000, g % 17500, g % 18000, g % 18500, g %␣
↪19000, g % 19500, g % 20000,
    g % 20500, g % 21000, g % 21500, g % 22000, g % 22500, g % 23000, g % 23500, g %␣
↪24000, g % 24500, g % 25000,
    g % 25500, g % 26000, g % 26500, g % 27000, g % 27500, g % 28000, g % 28500, g %␣
↪29000, g % 29500, g % 30000,
    g % 30500, g % 31000, g % 31500, g % 32000, g % 32500, g % 33000, g % 33500, g %␣
↪34000, g % 34500, g % 35000,
    g % 35500, g % 36000, g % 36500, g % 37000, g % 37500, g % 38000, g % 38500, g %␣
↪39000, g % 39500, g % 40000,
    g % 40500, g % 41000, g % 41500, g % 42000, g % 42500, g % 43000, g % 43500, g %␣
↪44000, g % 44500, g % 45000,
    g % 45500, g % 46000, g % 46500, g % 47000, g % 47500, g % 48000, g % 48500, g %␣
↪49000, g % 49500, g % 50000
  FROM generate_series(1,50000000) g;

VACUUM (FREEZE, ANALYZE) perf_row;
VACUUM (FREEZE, ANALYZE) perf_columnar;
```

For this data, you can see a compression ratio of better than 8X in the columnar table.

```
SELECT pg_total_relation_size('perf_row')::numeric/
       pg_total_relation_size('perf_columnar') AS compression_ratio;
```

```
.
 compression_ratio
--------------------
 8.0196135873627944
```

(continues on next page)

(continued from previous page)

```
(1 row)
```

18.4.3 Example

Columnar storage works well with table partitioning. For an example, see *Archiving with Columnar Storage*.

18.4.4 Gotchas

- Columnar storage compresses per stripe. Stripes are created per transaction, so inserting one row per transaction will put single rows into their own stripes. Compression and performance of single row stripes will be worse than a row table. Always insert in bulk to a columnar table.
- If you mess up and columnarize a bunch of tiny stripes, there is no way to repair the table. The only fix is to create a new columnar table and copy data from the original in one transaction:

```
BEGIN;
CREATE TABLE foo_compacted (LIKE foo) USING columnar;
INSERT INTO foo_compacted SELECT * FROM foo;
DROP TABLE foo;
ALTER TABLE foo_compacted RENAME TO foo;
COMMIT;
```

- Fundamentally non-compressible data can be a problem, although it can still be useful to use columnar so that less is loaded into memory when selecting specific columns.
- On a partitioned table with a mix of row and column partitions, updates must be carefully targeted or filtered to hit only the row partitions.
  - If the operation is targeted at a specific row partition (e.g. *UPDATE p2 SET i = i + 1*), it will succeed; if targeted at a specified columnar partition (e.g. *UPDATE p1 SET i = i + 1*), it will fail.
  - If the operation is targeted at the partitioned table and has a WHERE clause that excludes all columnar partitions (e.g. *UPDATE parent SET i = i + 1 WHERE timestamp = '2020-03-15'*), it will succeed.
  - If the operation is targeted at the partitioned table, but does not exclude all columnar partitions, it will fail; even if the actual data to be updated only affects row tables (e.g. *UPDATE parent SET i = i + 1 WHERE n = 300*).

18.4.5 Limitations

Future versions of Citus will incrementally lift the current limitations:

- Append-only (no UPDATE/DELETE support)
- No space reclamation (e.g. rolled-back transactions may still consume disk space)
- Support for hash and btree indices only
- No index scans, or bitmap index scans
- No tidscans
- No sample scans
- No TOAST support (large values supported inline)

- No support for ON CONFLICT statements (except DO NOTHING actions with no target specified).
- No support for tuple locks (SELECT … FOR SHARE, SELECT … FOR UPDATE)
- No support for serializable isolation level
- Support for PostgreSQL server versions 12+ only
- No support for foreign keys, unique constraints, or exclusion constraints
- No support for logical decoding
- No support for intra-node parallel scans
- No support for AFTER … FOR EACH ROW triggers
- No UNLOGGED columnar tables
- No TEMPORARY columnar tables

CHAPTER

NINETEEN

UPGRADING CITUS

19.1 Upgrading Citus Versions

Upgrading the Citus version requires first obtaining the new Citus extension and then installing it in each of your database instances. 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.

19.1.1 Patch Version Upgrade

To upgrade a Citus version to its latest patch, issue a standard upgrade command for your package manager. Assuming version 11.3 is currently installed on Postgres 15:

Ubuntu or Debian

```
sudo apt-get update
sudo apt-get install --only-upgrade postgresql-15-citus-11.3
sudo service postgresql restart
```

Fedora, CentOS, or Red Hat

```
sudo yum update citus113_15
sudo service postgresql-15 restart
```

19.1.2 Major and Minor Version Upgrades

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

Note: Starting at version 8.1, new Citus nodes expect and require encrypted inter-node communication by default, whereas nodes upgraded to 8.1 from an earlier version preserve their earlier SSL settings. Be careful when adding a new Citus 8.1 (or newer) node to an upgraded cluster that does not yet use SSL. The *adding a worker* section covers that situation.

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.

Step 1. Update Citus Package

If upgrading both Citus and Postgres, always be sure to upgrade the Citus extension first, and the PostgreSQL version second (see *Upgrading PostgreSQL version from 14 to 15*). Here is how to do a Citus upgrade from 10.2 to 11.3 on Postgres 14:

Ubuntu or Debian

```
sudo apt-get update
sudo apt-get install postgresql-14-citus-11.3
sudo service postgresql restart
```

Fedora, CentOS, or Red Hat

```
# Fedora, CentOS, or Red Hat
sudo yum swap citus102_14 citus113_14
sudo service postgresql-14 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
ALTER EXTENSION citus UPDATE;

-- you should see the upgraded Citus version
SELECT * FROM citus_version();

-- if upgrading to Citus 11.x or later,
-- run on the coordinator node
CALL citus_finish_citus_upgrade();
```

Note: If upgrading to Citus 11.x from an earlier major version, the citus_finish_citus_upgrade() procedure will make sure that all worker nodes have the right schema and metadata. It may take several minutes to run, depending on how much metadata needs to be synced.

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.

19.2 Upgrading PostgreSQL version from 14 to 15

Note: Do not attempt to upgrade *both* Citus and Postgres versions at once. If both upgrades are desired, upgrade Citus first.

Also, if you're running Citus 10.0 or 10.1, don't upgrade your Postgres version. Upgrade to at least Citus 10.2 and then perform the Postgres upgrade.

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

Existing data directory (e.g. /opt/pgsql/14/data) `export OLD_PG_DATA=/opt/pgsql/14/data`

Existing PostgreSQL installation path (e.g. /usr/pgsql-14) `export OLD_PG_PATH=/usr/pgsql-14`

New data directory after upgrade `export NEW_PG_DATA=/opt/pgsql/15/data`

New PostgreSQL installation path `export NEW_PG_PATH=/usr/pgsql-15`

19.2.1 For Every Node

1. Back up Citus metadata in the old coordinator node.

```
-- run this on the coordinator and worker nodes

SELECT citus_prepare_pg_upgrade();
```

2. Configure the new database instance to use Citus.

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

```
shared_preload_libraries = 'citus'
```

- **DO NOT CREATE** Citus extension
- **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 Citus 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 on new coordinator node.

   ```
   -- run this on the coordinator and worker nodes

   SELECT citus_finish_pg_upgrade();
   ```

CHAPTER

TWENTY

QUERY PERFORMANCE TUNING

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

20.1 Table Distribution and Shards

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

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

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

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

20.2 PostgreSQL tuning

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

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

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

Here is an example of explaining the plan for a particular example query. We use the VERBOSE flag to see the actual queries which were sent to the worker nodes.

```
EXPLAIN VERBOSE
 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: remote_scan.minute
  ->  HashAggregate  (cost=0.00..0.00 rows=0 width=0)
    Group Key: remote_scan.minute
    ->  Custom Scan (Citus Adaptive)  (cost=0.00..0.00 rows=0 width=0)
      Task Count: 32
      Tasks Shown: One of 32
      ->  Task
        Query: SELECT date_trunc('minute'::text, created_at) AS minute, sum(((payload␣
↪OPERATOR(pg_catalog.->>) 'distinct_size'::text))::integer) AS num_commits FROM github_
↪events_102042 github_events WHERE (event_type OPERATOR(pg_catalog.=) 'PushEvent
↪'::text) GROUP BY (date_trunc('minute'::text, created_at))
        Node: host=localhost port=5433 dbname=postgres
        ->  HashAggregate  (cost=93.42..98.36 rows=395 width=16)
          Group Key: date_trunc('minute'::text, created_at)
          ->  Seq Scan on github_events_102042 github_events  (cost=0.00..88.20 rows=418␣
↪width=503)
            Filter: (event_type = 'PushEvent'::text)
(13 rows)
```

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

```
->  Custom Scan (Citus Adaptive)  (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, database, and the query that was sent to the worker so you can connect to the worker directly and try the query if desired:

```
Tasks Shown: One of 32
->  Task
  Query: SELECT date_trunc('minute'::text, created_at) AS minute, sum(((payload␣
↪OPERATOR(pg_catalog.->>) 'distinct_size'::text))::integer) AS num_commits FROM github_
↪events_102042 github_events WHERE (event_type OPERATOR(pg_catalog.=) 'PushEvent
↪'::text) GROUP BY (date_trunc('minute'::text, created_at))
  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)
```

(continues on next page)

(continued from previous page)

```
  ->  Seq Scan on github_events_102042 github_events  (cost=0.00..88.20 rows=418␣
↪width=503)
    Filter: (event_type = 'PushEvent'::text)
```

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

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

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

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

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

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

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

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

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

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

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

```
Sort  (cost=0.00..0.00 rows=0 width=0)
  Sort Key: remote_scan.minute
  ->  HashAggregate  (cost=0.00..0.00 rows=0 width=0)
    Group Key: remote_scan.minute
    ->  Custom Scan (Citus Adaptive)  (cost=0.00..0.00 rows=0 width=0)
      Task Count: 32
      Tasks Shown: All
      ->  Task
        Node: host=localhost port=5433 dbname=postgres
        ->  HashAggregate  (cost=93.42..98.36 rows=395 width=16)
          Group Key: date_trunc('minute'::text, created_at)
          ->  Seq Scan on github_events_102042 github_events  (cost=0.00..88.20␣
↪rows=418 width=503)
            Filter: (event_type = 'PushEvent'::text)
      ->  Task
        Node: host=localhost port=5434 dbname=postgres
        ->  HashAggregate  (cost=103.21..108.57 rows=429 width=16)
          Group Key: date_trunc('minute'::text, created_at)
          ->  Seq Scan on github_events_102043 github_events  (cost=0.00..97.47␣
↪rows=459 width=492)
            Filter: (event_type = 'PushEvent'::text)
      --
      -- ... repeats for all 32 tasks
      --     alternating between workers one and two
      --     (running in this case locally on ports 5433, 5434)
      --

(199 rows)
```

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

Note: Note that when citus.explain_all_tasks is enabled, EXPLAIN plans are retrieved sequentially, which may take a long time for EXPLAIN ANALYZE.

Citus, by default, sorts tasks by execution time in descending order. If *citus.explain_all_tasks* is disabled, then Citus shows the single longest-running task. Please note that this functionality can be used only with EXPLAIN ANALYZE, since regular EXPLAIN doesn't execute the queries, and therefore doesn't know any execution times. To change the sort order, you can use *citus.explain_analyze_sort_method (enum)*.

20.3 Scaling Out Performance

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

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

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

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

20.4 Distributed Query Performance Tuning

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

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

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

20.4.1 General

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

Subquery/CTE Network Overhead

In the best case Citus 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, Citus 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.
```

The size of intermediate results and their destination is available in EXPLAIN ANALYZE output:

```
EXPLAIN ANALYZE
WITH deleted_rows AS (
  DELETE FROM page_views WHERE tenant_id IN (3, 4) RETURNING *
), viewed_last_week AS (
  SELECT * FROM deleted_rows WHERE view_time > current_timestamp - interval '7 days'
)
SELECT count(*) FROM viewed_last_week;
```

```
Custom Scan (Citus Adaptive)  (cost=0.00..0.00 rows=0 width=0) (actual time=570.076..570.
↪077 rows=1 loops=1)
  ->  Distributed Subplan 31_1
        Subplan Duration: 6978.07 ms
        Intermediate Data Size: 26 MB
        Result destination: Write locally
        ->  Custom Scan (Citus Adaptive)  (cost=0.00..0.00 rows=0 width=0) (actual␣
↪time=364.121..364.122 rows=0 loops=1)
              Task Count: 2
              Tuple data received from nodes: 0 bytes
              Tasks Shown: One of 2
              ->  Task
                    Tuple data received from node: 0 bytes
                    Node: host=localhost port=5433 dbname=postgres
                    ->  Delete on page_views_102016 page_views  (cost=5793.38..49272.28␣
↪rows=324712 width=6) (actual time=362.985..362.985 rows=0 loops=1)
                          ->  Bitmap Heap Scan on page_views_102016 page_views ␣
↪(cost=5793.38..49272.28 rows=324712 width=6) (actual time=362.984..362.984 rows=0␣
↪loops=1)
                                Recheck Cond: (tenant_id = ANY ('{3,4}'::integer[]))
                                ->  Bitmap Index Scan on view_tenant_idx_102016  (cost=0.
↪00..5712.20 rows=324712 width=0) (actual time=19.193..19.193 rows=325733 loops=1)
                                      Index Cond: (tenant_id = ANY ('{3,4}'::integer[]))
                        Planning Time: 0.050 ms
```

(continues on next page)

(continued from previous page)

```
                        Execution Time: 363.426 ms
        Planning Time: 0.000 ms
        Execution Time: 364.241 ms
 Task Count: 1
 Tuple data received from nodes: 6 bytes
 Tasks Shown: All
 ->  Task
       Tuple data received from node: 6 bytes
       Node: host=localhost port=5432 dbname=postgres
       ->  Aggregate  (cost=33741.78..33741.79 rows=1 width=8) (actual time=565.008..565.
↪008 rows=1 loops=1)
             ->  Function Scan on read_intermediate_result intermediate_result  (cost=0.
↪00..29941.56 rows=1520087 width=0) (actual time=326.645..539.158 rows=651466 loops=1)
                   Filter: (view_time > (CURRENT_TIMESTAMP - '7 days'::interval))
           Planning Time: 0.047 ms
           Execution Time: 569.026 ms
Planning Time: 1.522 ms
Execution Time: 7549.308 ms
```

In the above EXPLAIN ANALYZE output, you can see the following information about the intermediate results:

```
Intermediate Data Size: 26 MB
Result destination: Write locally
```

It tells us how large the intermediate results were, and where the intermediate results were written to. In this case, they were written to the node coordinating the query execution, as specified by "Write locally". For some other queries it can also be of the following format:

```
Intermediate Data Size: 26 MB
Result destination: Send to 2 nodes
```

Which means the intermediate result was pushed to 2 worker nodes and it involved more network traffic.

When using CTEs, or joins between CTEs and distributed tables, you can avoid push-pull execution by following these rules:

- Tables should be colocated
- The CTE queries should not require any merge steps (e.g., LIMIT or GROUP BY on a non-distribution key)
- Tables and CTEs should be joined on distribution keys

Also PostgreSQL 12 or above allows Citus to take advantage of *CTE inlining* to push CTEs down to workers in more circumstances. The inlining behavior can be controlled with the `MATERIALIZED` keyword – see the PostgreSQL docs for details.

20.4.2 Advanced

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

Connection Management

When executing multi-shard queries, Citus must balance the gains from parallelism with the overhead from database connections. The *Query Execution* section explains the steps of turning queries into worker tasks and obtaining database connections to the workers.

- Set *citus.max_adaptive_executor_pool_size (integer)* to a low value like 1 or 2 for transactional workloads with short queries (e.g. < 20ms of latency). For analytical workloads where parallelism is critical, leave this setting at its default value of 16.
- Set *citus.executor_slow_start_interval (integer)* to a high value like 100ms for transactional workloads comprised of short queries that are bound on network latency rather than parallelism. For analytical workloads, leave this setting at its default value of 10ms.
- The default value of 1 for *citus.max_cached_conns_per_worker (integer)* is reasonable. A larger value such as 2 might be helpful for clusters that use a small number of concurrent sessions, but it's not wise to go much further (e.g. 16 would be too high). If set too high, sessions will hold idle connections and use worker resources unnecessarily.
- Set *citus.max_shared_pool_size (integer)* to match the max_connections setting of your *worker* nodes. This setting is mainly a fail-safe.

Task Assignment Policy

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

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

Intermediate Data Transfer Format

On Postgres 13 and lower, Citus defaults to transfering intermediate query data between workers in textual format. For certain data types, like hll or hstore arrays, the cost of serializing and deserializing data can be high. In such cases, using the binary format to transfer intermediate data can improve query performance. You can enable the *citus.binary_worker_copy_format (boolean)* configuration option to use the binary format.

Binary protocol

In some cases, a large part of query time is spent in sending query results from workers to the coordinator. This mostly happens when queries request many rows (such as `select * from table`), or when result columns use big types (like `hll` or `tdigest` from the postgresql-hll and tdigest extensions).

In those cases it can be beneficial to set `citus.enable_binary_protocol` to `true`, which will change the encoding of the results to binary, rather than using text encoding. Binary encoding significantly reduces bandwidth for types that have a compact binary representation, such as `hll`, `tdigest`, `timestamp` and `double precision`.

For Postgres 14 and higher, the default for this setting is already `true`. So explicitly enabling it for those Postgres versions has no effect.

20.5 Scaling Out Data Ingestion

Citus lets you scale out data ingestion to very high rates, but there are several trade-offs to consider in terms of application integration, throughput, and latency. In this section, we discuss different approaches to data ingestion, and provide guidelines for expected throughput and latency numbers.

20.5.1 Real-time Insert and Updates

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

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

Insert Throughput

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

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

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

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

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

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

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 INSERTs. This is because pgbench creates a primary key index for UPDATE statements and an UPDATE incurs more work on the worker nodes. It's also worth noting an additional differences between INSERT and UPDATEs.

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.

Insert and Update: Throughput Checklist

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

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

Insert and Update: Latency

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

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

(continues on next page)

(continued from previous page)

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

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

20.5.2 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
```

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

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

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

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

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

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

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

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

Note: 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.

CHAPTER

TWENTYONE

USEFUL DIAGNOSTIC QUERIES

21.1 Finding which shard contains data for a specific tenant

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

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

```
SELECT 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 │
└─────────┴────────────┴─────────────┴───────────┴──────────┴─────────────┘
```

21.2 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      │
│_______________
```

21.3 Detecting locks

This query will run across all worker nodes and identify locks, how long they've been open, and the offending queries:

```
SELECT * FROM citus_lock_waits;
```

For more information, see *Distributed Query Activity*.

21.4 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_table`:

```
SELECT shardid, table_name, shard_size
FROM citus_shards
WHERE table_name = 'my_table';
```

Example output:

```
.
 shardid | table_name | shard_size
---------+------------+------------
  102170 | my_table   |   90177536
  102171 | my_table   |   90177536
  102172 | my_table   |   91226112
  102173 | my_table   |   90177536
```

This query uses the *Shard information view*.

21.5 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 table_name, table_size
  FROM citus_tables;
```

Example output:

```
───────────────────────────
│  table_name    │ table_size │
│───────────────────────────
│ github_users   │ 39 MB      │
│ github_events  │ 98 MB      │
│───────────────────────────
```

There are other ways to measure distributed table size, as well. See *Determining Table and Relation Size*.

21.6 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)::regclass = '%s'::regclass
    ORDER BY
```

(continues on next page)

(continued from previous page)

```
      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##0
↪"} │
│   102010 │ t       │                                                                 ␣
↪    │
│   102011 │ t       │                                                                 ␣
↪    │
└────────────────────────────────────────────────────────────────────────────────────────
```

21.7 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 │
└───────────────
```

21.8 Viewing system queries

21.8.1 Active queries

The citus_stat_activity view shows which queries are currently executing. You can filter to find the actively executing ones, along with the process ID of their backend:

```
SELECT global_pid, query, state
  FROM citus_stat_activity
 WHERE state != 'idle';
```

21.8.2 Why are queries waiting

We can also query to see the most common reasons that non-idle queries that are waiting. For an explanation of the reasons, check the PostgreSQL documentation.

```
SELECT wait_event || ':' || wait_event_type AS type, count(*) AS number_of_occurences
  FROM pg_stat_activity
 WHERE state != 'idle'
GROUP BY wait_event, wait_event_type
ORDER BY number_of_occurences DESC;
```

Example output when running `pg_sleep` in a separate query concurrently:

```
┌─────────────────┬──────────────────────┐
│      type       │ number_of_occurences │
├─────────────────┼──────────────────────┤
│                 │                    1 │
│ PgSleep:Timeout │                    1 │
└─────────────────┴──────────────────────┘
```

21.9 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:

```
-- on coordinator
SELECT 100 * (sum(idx_blks_hit) - sum(idx_blks_read)) / sum(idx_blks_hit) AS index_hit_
↪rate
  FROM pg_statio_user_indexes;

-- on workers
SELECT nodename, result as index_hit_rate
FROM run_command_on_workers($cmd$
  SELECT 100 * (sum(idx_blks_hit) - sum(idx_blks_read)) / sum(idx_blks_hit) AS index_hit_
↪rate
    FROM pg_statio_user_indexes;
$cmd$);
```

Example output:

```
┌───────────┬────────────────┐
│ nodename  │ index_hit_rate │
├───────────┼────────────────┤
│ 10.0.0.16 │ 96.0           │
│ 10.0.0.20 │ 98.0           │
└───────────┴────────────────┘
```

21.10 Cache hit rate

Most applications typically access a small fraction of their total data at once. Postgres keeps frequently accessed data in memory to avoid slow reads from disk. You can see statistics about it in the pg_statio_user_tables view.

An important measurement is what percentage of data comes from the memory cache vs the disk in your workload:

```
-- on coordinator
SELECT
  sum(heap_blks_read) AS heap_read,
  sum(heap_blks_hit)  AS heap_hit,
  100 * sum(heap_blks_hit) / (sum(heap_blks_hit) + sum(heap_blks_read)) AS cache_hit_rate
FROM
  pg_statio_user_tables;

-- on workers
SELECT nodename, result as cache_hit_rate
FROM run_command_on_workers($cmd$
  SELECT
    100 * sum(heap_blks_hit) / (sum(heap_blks_hit) + sum(heap_blks_read)) AS cache_hit_
↪rate
  FROM
    pg_statio_user_tables;
$cmd$);
```

Example output:

```
┌───────────┬──────────┬──────────────────┐
│ heap_read │ heap_hit │  cache_hit_rate  │
├───────────┼──────────┼──────────────────┤
│         1 │      132 │ 99.2481203007518796 │
└───────────┴──────────┴──────────────────┘
```

If you find yourself with a ratio significantly lower than 99%, then you likely want to consider increasing the cache available to your database

CHAPTER

TWENTYTWO

COMMON ERROR MESSAGES

22.1 Could not receive query results

Caused when the the coordinator node is unable to connect to a worker.

```
SELECT 1 FROM companies WHERE id = 2928;
```

```
ERROR:  connection to the remote node localhost:5432 failed with the following error:␣
↪could not connect to server: Connection refused
        Is the server running on host "localhost" (127.0.0.1) and accepting
        TCP/IP connections on port 5432?
```

22.1.1 Resolution

To fix, check that the worker is accepting connections, and that DNS is correctly resolving.

22.2 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. Citus 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;
```

(continues on next page)

(continued from previous page)

```
UPDATE lockme SET x = 3 WHERE id = 2;
                                       UPDATE lockme SET x = 4 WHERE id = 1;
```

```
ERROR:  canceling the transaction since it was involved in a distributed deadlock
```

22.2.1 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.

22.3 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.

22.3.1 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).

22.4 SSL error: certificate verify failed

As of Citus 8.1, nodes are required talk to one another using SSL by default. If SSL is not enabled on a Postgres server when Citus is first installed, the install process will enable it, which includes creating and self-signing an SSL certificate.

However, if a root certificate authority file exists (typically in `~/.postgresql/root.crt`), then the certificate will be checked unsuccessfully against that CA at connection time. The Postgres documentation about SSL support warns:

> For backward compatibility with earlier versions of PostgreSQL, if a root CA file exists, the behavior of sslmode=require will be the same as that of verify-ca, meaning the server certificate is validated against the CA. Relying on this behavior is discouraged, and applications that need certificate validation should always use verify-ca or verify-full.

22.4.1 Resolution

Possible solutions are to sign the certificate, turn off SSL, or remove the root certificate. Also a node may have trouble connecting to itself without the help of *citus.local_hostname (text)*.

22.5 Could not connect to any active placements

When all available worker connection slots are in use, further connections will fail.

```
WARNING:  connection error: hostname:5432
ERROR:  could not connect to any active placements
```

22.5.1 Resolution

This error happens most often when copying data into Citus 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.

22.6 Remaining connection slots are reserved for non-replication superuser connections

This occurs when PostgreSQL runs out of available connections to serve concurrent client requests.

22.6.1 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. (Our cloud_topic has a built-in PgBouncer instance.)

22.7 PgBouncer cannot connect to server

In a self-hosted Citus cluster, this error indicates that the coordinator node is not responding to PgBouncer.

22.7.1 Resolution

Try connecting directly to the server with psql to ensure it is running and accepting connections.

22.8 Relation *foo* is not distributed

This error no longer occurs in the current version of Citus. It was caused by attempting to join local and distributed tables in the same query.

22.8.1 Resolution

Upgrade to Citus 10.0 or higher.

22.9 Unsupported clause type

This error no longer occurs in the current version of Citus. 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
```

22.9.1 Resolution

Upgrade to Citus 7.2 or higher.

22.10 Cannot open new connections after the first modification command within a transaction

This error no longer occurs in the current version of Citus 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
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
```

22.10.1 Resolution

Upgrade to Citus 7.2 or higher.

22.11 Cannot create uniqueness constraint

As a distributed system, Citus 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:  creating unique indexes on non-partition columns is currently unsupported
```

Enforcing uniqueness on a non-distribution column would require Citus to check every shard on every INSERT to validate, which defeats the goal of scalability.

22.11.1 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.

22.12 Function create_distributed_table does not exist

```
SELECT create_distributed_table('foo', 'id');
/*
ERROR:  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.
*/
```

22.12.1 Resolution

When basic *Citus Utility Functions* are not available, check whether the Citus 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.

22.13 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 shards. 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 timestamptz 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 timestamptz
UPDATE foo SET ... WHERE foo_timestamp < now();
```

```
ERROR:  STABLE functions used in UPDATE queries cannot be called with column references
```

In this case the comparison operator < between timestamp and timestamptz is not immutable.

22.13.1 Resolution

Avoid stable functions on columns in a distributed UPDATE statement. In particular, whenever working with times use timestamptz rather than timestamp. Having a time zone in timestamptz makes calculations immutable.

CHAPTER

TWENTYTHREE

FREQUENTLY ASKED QUESTIONS

23.1 Can I create primary keys on distributed tables?

Currently Citus imposes primary key constraint only if the distribution column is a part of the primary key. This assures that the constraint needs to be checked only on one shard to ensure uniqueness.

23.2 How do I add nodes to an existing Citus cluster?

On the Citus managed service with Azure Cosmos DB for PostgreSQL (formerly known as Hyperscale (Citus)), adding nodes is quite easy: all it takes is a few clicks in the Azure portal. With Citus open source, you can add nodes manually by calling the *citus_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, the new node 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 provides an open source shard rebalancer utility. You can find more information in the *Rebalance Shards without Downtime* section.

23.3 How does Citus handle failure of a worker node?

Citus uses PostgreSQL's streaming replication to replicate the entire worker-node as-is. It replicates 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.

23.4 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*.

23.5 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

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.

23.6 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 co-location group, 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 SaaS 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

```sql
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*.

23.7 How do I change the shard count for a hash partitioned table?

Citus has a function called *alter_distributed_table* that can change the shard count of a distributed table.

23.8 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*.

23.9 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.

23.10 How do I create database roles, functions, extensions etc in a Citus cluster?

Certain commands, when run on the coordinator node, do not get propagated to the workers:

- `CREATE ROLE/USER`
- `CREATE DATABASE`
- `ALTER ... SET SCHEMA`
- `ALTER TABLE ALL IN TABLESPACE`
- `CREATE TABLE` (see *Table Types*)

For the other types of objects above, create them explicitly on all nodes. Citus provides a function to execute queries across all workers:

```
SELECT run_command_on_workers($cmd$
  /* the command to run */
  CREATE ROLE ...
$cmd$);
```

Learn more in *Manual Query Propagation*. Also note that even after manually propagating CREATE DATABASE, Citus must still be installed there. See *Creating a New Database*.

In the future Citus will automatically propagate more kinds of objects. The advantage of automatic propagation is that Citus will automatically create a copy on any newly added worker nodes (see *Distributed object table* for more about that.)

23.11 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 *citus_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 citus_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.

23.12 Which shard contains data for a particular tenant?

Citus provides UDFs and metadata tables to determine the mapping of a distribution column value to a particular shard, and the shard placement on a worker node. See *Finding which shard contains data for a specific tenant* for more details.

23.13 I forgot the distribution column of a table, how do I find it?

The Citus coordinator node metadata tables contain this information. See *Finding the distribution column for a table*.

23.14 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*.

23.15 Why does pg_relation_size report zero bytes for a distributed table?

The data in distributed tables lives on the worker nodes (in shards), not on the coordinator. A true measure of distributed table size is obtained as a sum of shard sizes. Citus provides helper functions to query this information. See *Determining Table and Relation Size* to learn more.

23.16 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';
```

23.17 Can I run Citus on Microsoft Azure?

Yes, Citus is available as a managed service with Azure Cosmos DB for PostgreSQL, formerly known as Hyperscale (Citus) in Azure Database for PostgreSQL.

23.18 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.

23.19 How does cstore_fdw work with Citus?

The cstore_fdw extension is no longer needed on PostgreSQL 12 and above, because *Columnar Storage* is now implemented directly in Citus. Unlike cstore_fdw, Citus' columnar tables support transactional semantics, replication, and pg_upgrade. Citus' query parallelization, seamless sharding, and HA benefits combine powerfully with the superior compression and I/O utilization of columnar storage for large dataset archival and reporting.

23.20 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.

CHAPTER

TWENTYFOUR

RELATED ARTICLES

24.1 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.

24.1.1 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 off these are trade-offs.

24.1.2 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.

24.1.3 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
- 2n - 1 rollup tables to cover all combinations in n columns

24.1.4 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 Citus 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
```

24.1.5 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;
```

(continues on next page)

(continued from previous page)

```
 hour | distinct_push_count
------+---------------------
    5 |    6206.61586498546
    6 |    9517.80542100396
    7 |    10370.4087640166
    8 |    7067.26073810357
(4 rows)
```

24.1.6 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 2n - 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.

24.1.7 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 Citus, 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.

24.2 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 a many of them out there, but today we're just going to focus on one, HyperLogLog or HLL. HLL is very successfull 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 distict 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.

24.2.1 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.

24.2.2 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.

24.2.3 Hands on with HLL

Note: This section mentions the Citus Cloud service. We are no longer onboarding new users to Citus Cloud on AWS. If you're new to Citus, the good news is, Citus is still available to you: as open source, and in the cloud on Microsoft Azure, as a fully-integrated deployment option in Azure Database for PostgreSQL.

See cloud_topic.

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 simply running the below command from the coordinator;

```
CREATE EXTENSION hll;
```

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 ares feasible for you.

24.3 Postgres Parallel Indexing in Citus

(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. Citus makes creating and maintaining indexes that much faster through parallelization.

Citus can be used to distribute PostgreSQL tables across many machines. One of the many advantages of Citus 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. Citus 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 Citus 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 Citus 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.

```sql
-- 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.

24.4 Real-time Event Aggregation at Scale Using Postgres with Citus

(Copy of original publication)

Note: This article mentions the Citus Cloud service. We are no longer onboarding new users to Citus Cloud on AWS. If you're new to Citus, the good news is, Citus is still available to you: as open source, and in the cloud on Microsoft Azure, as a fully-integrated deployment option in Azure Database for PostgreSQL.

See cloud_topic.

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. Citus 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;
```

(continues on next page)

(continued from previous page)

```
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;

  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.

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.

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,
```

(continues on next page)

(continued from previous page)

```
    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, and many other features, Citus can not only horizontally scale your multi-tenant database, but can also unify many different parts of your data pipeline into one platform.

24.5 How Distributed Outer Joins on PostgreSQL with Citus 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 Citus.

Let's say we have two tables, customer and purchase:

```
customer table:
 customer_id |      name
-------------+------------------
           1 | Corra Ignacio
```

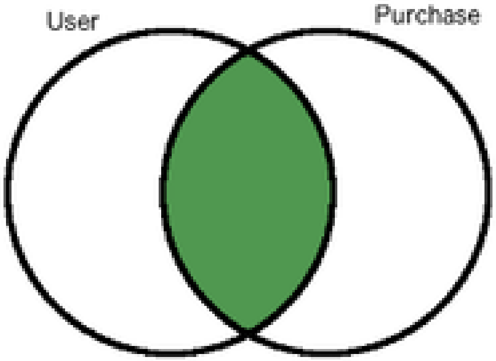
(continues on next page)

(continued from previous page)

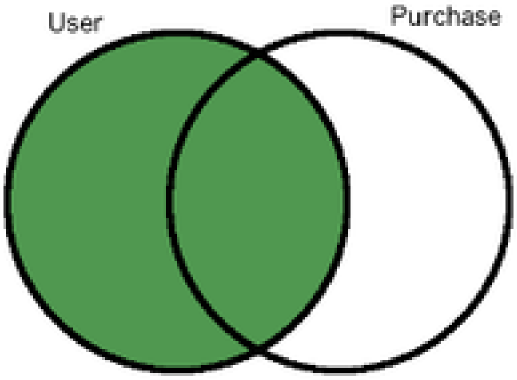
```
          3 | Warren Brooklyn
          2 | Jalda Francis

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:

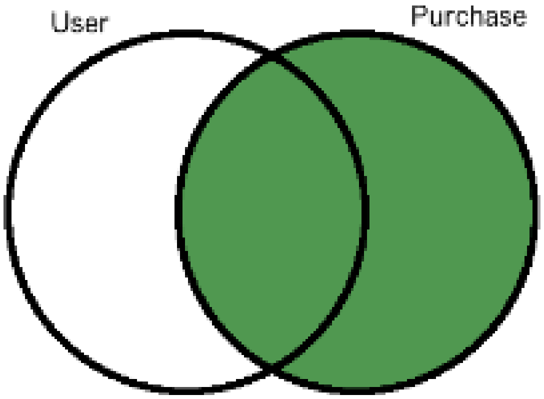
```
SELECT customer.name, purchase.comment
FROM customer JOIN purchase ON customer.customer_id = purchase.customer_id
ORDER BY purchase.comment;

     name      |         comment
---------------+------------------------
 Corra Ignacio | Comfortable
 Jalda Francis | Good Read, cheap price
 Corra Ignacio | Nice to Have!
```



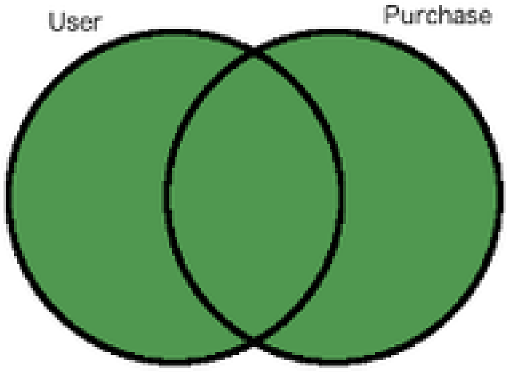
```
SELECT customer.name, purchase.comment
FROM customer INNER JOIN purchase ON customer.customer_id = purchase.customer_id
ORDER BY purchase.comment;


     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 |
```

24.5.1 Distributed Outer Joins with Citus

The Citus 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, Citus 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;
```

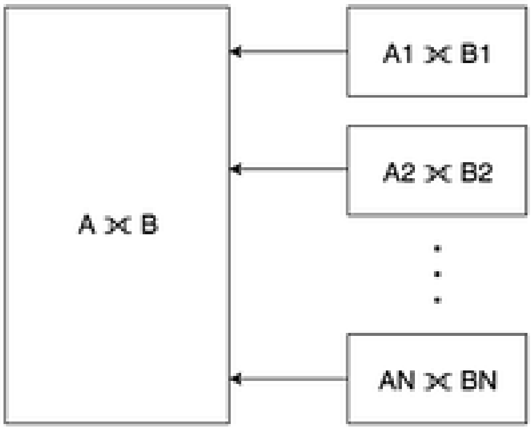
The following queries are not supported out of the box:

```
-- 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 Citus 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 Citus 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 Citus uses this plan to generate a distributed plan where various checks about Citus’ 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 Citus 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 Citus simply takes care of you. If you're sharding on some shared id, as is common in certain use cases, then Citus 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 Citus or learning how more works we encourage you to join the conversation with us on Slack.

24.6 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)

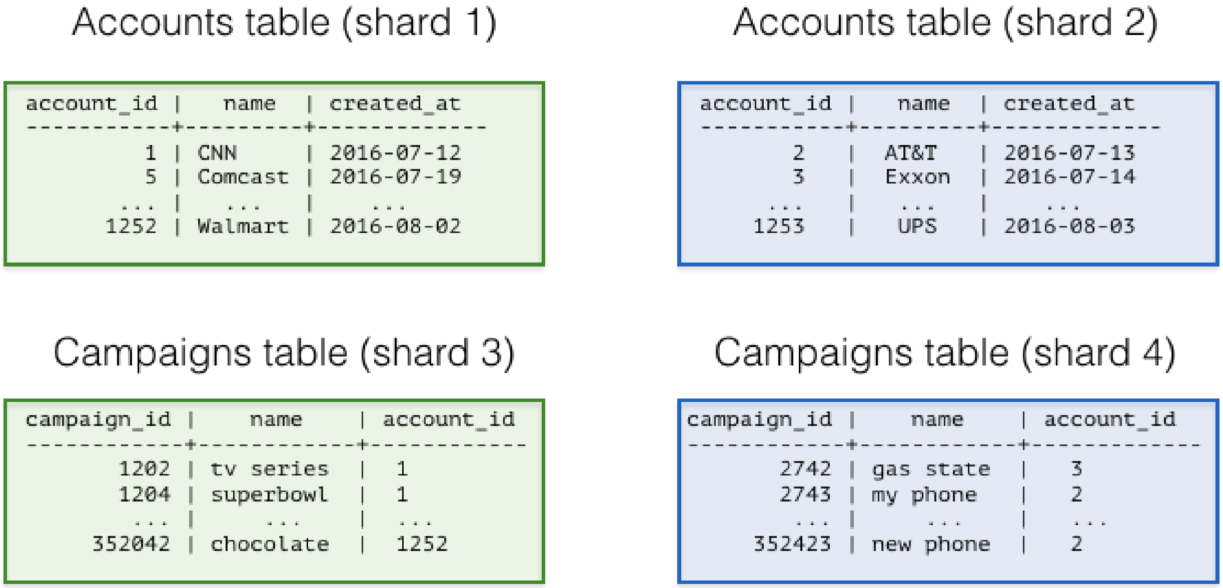
The option you pick has implications on scalability, how you handle data that varies across tenants, isolation, and ease-of-maintenance. And these implications have been discussed in detail across many StackOverflow questions and database articles. So, what is the best solution?

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.



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
```

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.

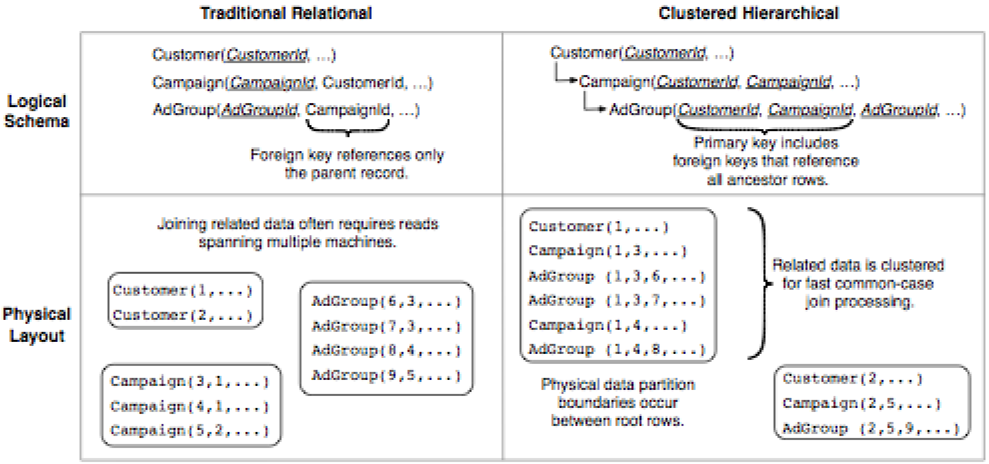

Traditional Relational
Clustered Hierarchical
Logical Schema
Customer(CustomerId, ...)
Campaign(CampaignId, CustomerId, ...)
AdGroup(AdGroupId, CampaignId, ...)
Foreign key references only the parent record.
Customer(CustomerId, ...)
Campaign(CustomerId, CampaignId, ...)
AdGroup(CustomerId, CampaignId, AdGroupId, ...)
Primary key includes foreign keys that reference all ancestor rows.
Physical Layout
Joining related data often requires reads spanning multiple machines.
Customer(1,...)
Customer(2,...)
AdGroup(6,3,...)
AdGroup(7,3,...)
AdGroup(8,4,...)
AdGroup(9,5,...)
Campaign(3,1,...)
Campaign(4,1,...)
Campaign(5,2,...)
Customer(1,...)
Campaign(1,3,...)
AdGroup (1,3,6,...)
AdGroup (1,3,7,...)
Campaign(1,4,...)
AdGroup (1,4,8,...)
Related data is clustered for fast common-case join processing.
Physical data partition boundaries occur between root rows.
Customer(2,...)
Campaign(2,5,...)
AdGroup (2,5,9,...)

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?

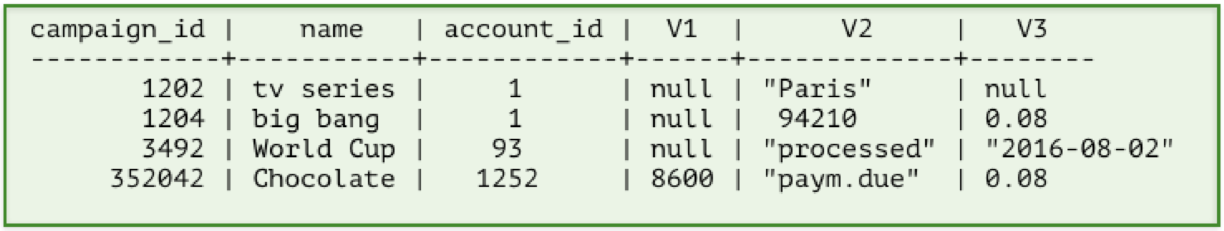
When you shard your tables for multi-tenancy, then you're having your database do the work for you. The database will either ensure that an Alter Table goes through across all shards, or it will roll it back.

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.

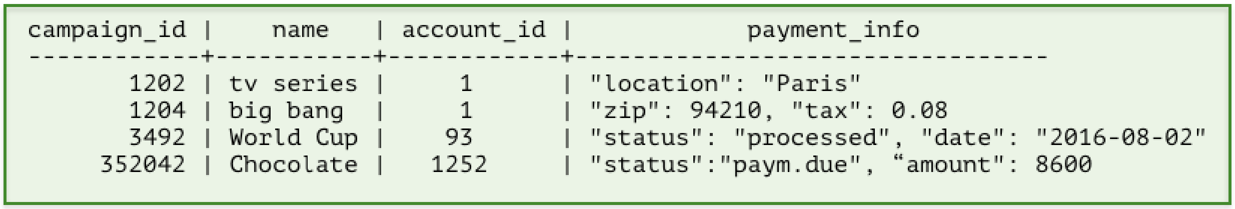
In this database model, your tables have a preset collection of custom columns, labeled in this image as V1, V2, and



```
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
```

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.



```
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
```

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 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 Citus.

24.7 Building a Scalable Postgres Metrics Backend using the Citus Extension

(Copy of original publication)

From nearly the beginning of the Citus 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 Citus 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.

24.7.1 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:

```
citus=> 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 Citus 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.

24.7.2 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 Citus 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 Citus 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:

```
citus=> 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"}
```

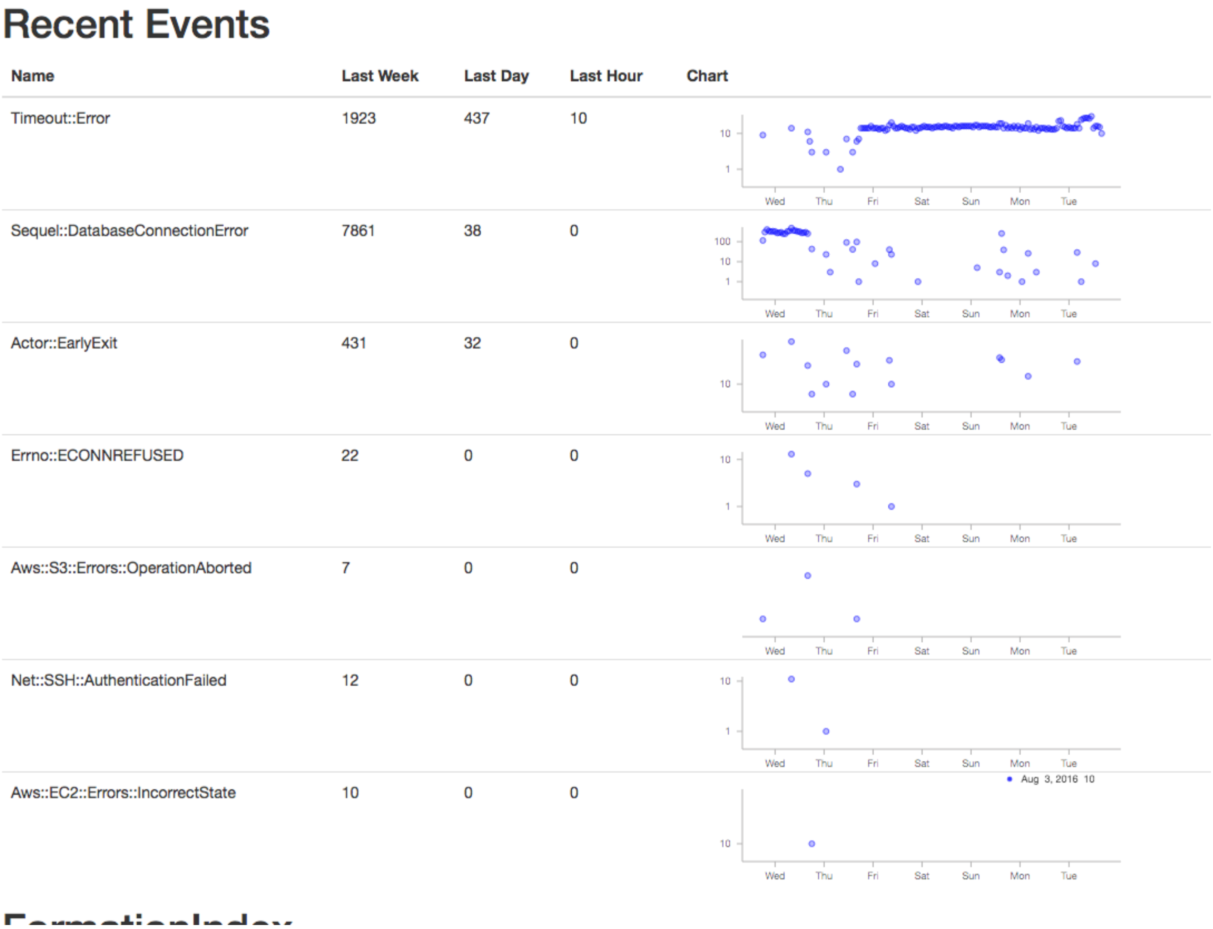
(continues on next page)

(continued from previous page)

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



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	

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.

24.8 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 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 Citus 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.

24.8.1 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,
```

(continues on next page)

(continued from previous page)

```
  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.

24.8.2 Multi-tenancy and co-location, a perfect pair

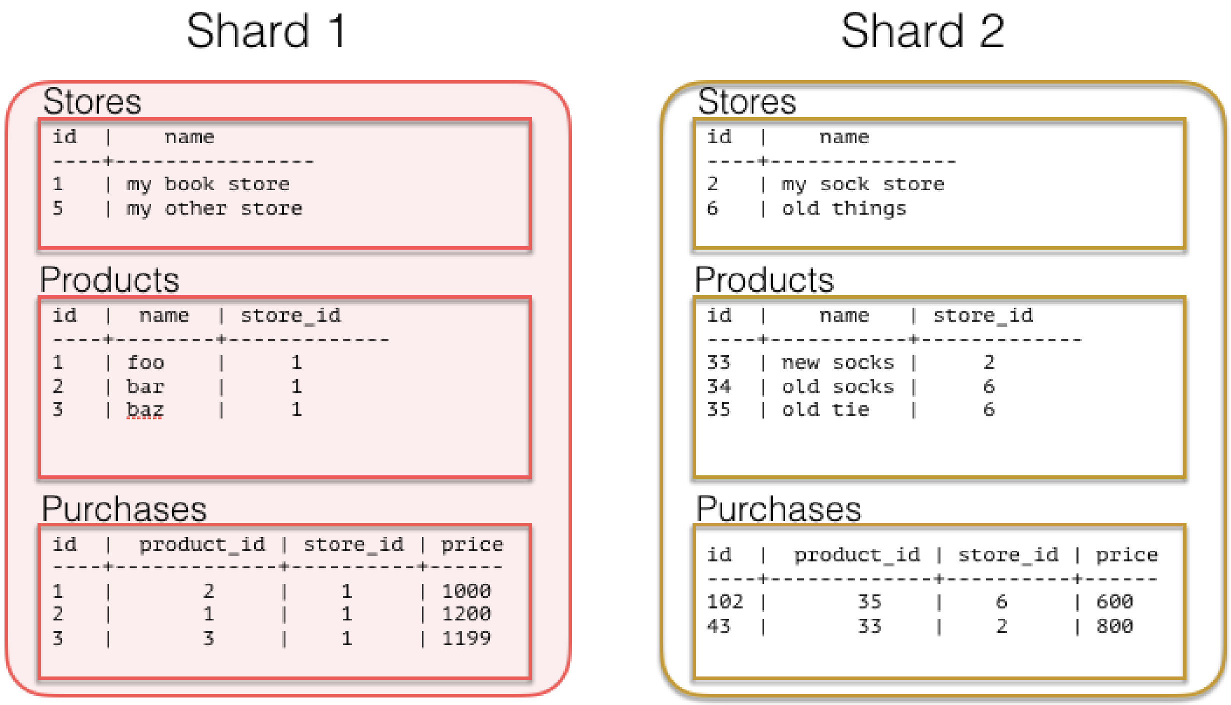
Co-locating data within the same physical instance avoids sending data over the network during joins. This can result in much faster operations. With Citus there are a number of ways to move your data around so you can join and query it in a flexible manner, but for this class of multi-tenant SaaS apps 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



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
```

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
```

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.

24.8.3 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.

24.9 Sharding Postgres with Semi-Structured Data and Its Performance Implications

(Copy of original publication)

If you're looking at Citus its 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.

24.9.1 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.

24.9.2 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.

24.9.3 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;
```

24.9.4 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.

24.10 Scalable Real-time Product Search using PostgreSQL with Citus

(Copy of original publication)

Note: This article mentions the Citus Cloud service. We are no longer onboarding new users to Citus Cloud on AWS. If you're new to Citus, the good news is, Citus is still available to you: as open source, and in the cloud on Microsoft Azure, as a fully-integrated deployment option in Azure Database for PostgreSQL.

See cloud_topic.

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 Citus, 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 Citus, with the goal of showcasing several powerful features.

We start by setting up a *multi-node Citus cluster on EC2* using 4 m3.2xlarge instances as workers. An even easier way to get started is to use Citus Cloud, which gives you a managed Citus 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:

```
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 Citus, we call the functions for *Creating and Modifying Distributed Objects (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');
$function$;
```

After setting up the function, we define a GIN index on it, which speeds up text searches on the product table.

```
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 Citus can be used to load the data into the product table. COPY for hash-partitioned tables is currently available in the latest version of Citus and in Citus 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 |                                         to_
↪json
------------+------------------------+-------+-------------------------------------------
↪--------------------------------------------
    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.