Query Performance and Tuning

Last updated by | Shital Modi | Nov 17, 2020 at 8:12 AM PST

Query Performance and Tuning

Tuesday, April 30, 2019
11:33 AM

Query Performance Tuning

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

Table Distribution and Shards

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

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

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

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

PostgreSQL tuning

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

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

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

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

```
EXPLAIN
 SELECT date_trunc('minute', created_at) AS minute,
        sum((payload->>'distinct_size')::int) AS num_commits
   FROM github_events
  WHERE event_type = 'PushEvent'
  GROUP BY minute
  ORDER BY minute;
Sort  (cost=0.00..0.00 rows=0 width=0)
  Sort Key: minute
  ->  HashAggregate  (cost=0.00..0.00 rows=0 width=0)
    Group Key: minute
```

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

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

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

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

```
Tasks Shown: One of 32
->  Task
  Node: host=localhost port=5433 dbname=postgres
```

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

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

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

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

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

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

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

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

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

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

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

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

```
EXPLAIN
 SELECT date_trunc('minute', created_at) AS minute,
        sum((payload->>'distinct_size')::int) AS num_commits
   FROM github_events
  WHERE event_type = 'PushEvent'
  GROUP BY minute
  ORDER BY minute;
 Sort  (cost=0.00..0.00 rows=0 width=0)
   Sort Key: minute
   ->  HashAggregate  (cost=0.00..0.00 rows=0 width=0)
     Group Key: minute
     ->  Custom Scan (Citus Real-Time)  (cost=0.00..0.00 rows=0 width=0)
       Task Count: 32
       Tasks Shown: All
       ->  Task
         Node: host=localhost port=5433 dbname=postgres
         ->  HashAggregate  (cost=93.42..98.36 rows=395 width=16)
           Group Key: date_trunc('minute'::text, created_at)
           ->  Seq Scan on github_events_102042 github_events  (cost=0.00..88.20 rows=418 width=503)
             Filter: (event_type = 'PushEvent'::text)
       ->  Task
         Node: host=localhost port=5434 dbname=postgres
         ->  HashAggregate  (cost=103.21..108.57 rows=429 width=16)
           Group Key: date_trunc('minute'::text, created_at)
           ->  Seq Scan on github_events_102043 github_events  (cost=0.00..97.47 rows=459 width=492)
             Filter: (event_type = 'PushEvent'::text)
       --
       -- ... repeats for all 32 tasks
       --     alternating between workers one and two
       --     (running in this case locally on ports 5433, 5434)
       --
(199 rows)
```

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

Note

Note that when citus.explain_all_tasks is enabled, EXPLAIN plans are retrieved sequentially, which may take a long time for EXPLAIN ANALYZE.

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.

Distributed Query Performance Tuning

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

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

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

General

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

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

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

Subquery/CTE Network Overhead

In the best case 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.
```

If this happens, consider whether you can move limits or filters inside CTEs/subqueries. For instance

```
-- It's slow to retrieve all rows and limit afterward
WITH cte_slow AS (SELECT * FROM users_table)
SELECT * FROM cte_slow LIMIT 10;
-- Limiting inside makes the intermediate results small
WITH cte_fast AS (SELECT * FROM users_table LIMIT 10)
SELECT * FROM cte_fast;
```

Advanced

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

Task Assignment Policy

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

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

Intermediate Data Transfer Format

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

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

Real Time Executor

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

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

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

Task Tracker Executor

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

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

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

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

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.

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 two additional differences between INSERT and UPDATEs.

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

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

Insert and Update: Throughput Checklist
When you're running the above pgbench benchmarks on a moderately sized Citus cluster, you can generally expect 10K-50K 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');
-- 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
```

Staging Data Temporarily

When loading data for temporary staging, consider using an unlogged table. These are tables which are not backed by the Postgres write-ahead log. This makes them faster for inserting rows, but not suitable for long term data storage. You can use an unlogged table as a place to load incoming data, prior to manipulating the data and moving it to permanent tables.

```
-- example unlogged table
CREATE UNLOGGED TABLE unlogged_table (
  key text,
  value text
);
-- its shards will be unlogged as well when
-- the table is distributed
SELECT create_distributed_table('unlogged_table', 'key');
-- ready to load data
```

Bulk Copy (250K - 2M/s)

Distributed tables support COPY from the 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.

Citus MX (50k/s-500k/s)

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

Citus MX is available in Citus Enterprise Edition and on Citus Cloud. For more information see Citus MX.