Citus: Distributed PostgreSQL for Data-Intensive Applications

Umur Cubukcu
umur.cubukcu@microsoft.com
Microsoft Corporation

Ozgun Erdogan
ozgun.erdogan@microsoft.com
Microsoft Corporation

Sumedh Pathak
sumedh.pathak@microsoft.com
Microsoft Corporation

Sudhakar Sannakkayala
sudhakar.sannakkayala@microsoft.com
Microsoft Corporation

Marco Slot
marco.slot@microsoft.com
Microsoft Corporation

ABSTRACT

Citus is an open source distributed database engine for PostgreSQL that is implemented as an extension. Citus gives users the ability to distribute data, queries, and transactions in PostgreSQL across a cluster of PostgreSQL servers to handle the needs of data-intensive applications. The development of Citus has largely been driven by conversations with companies looking to scale PostgreSQL beyond a single server and their workload requirements. This paper describes the requirements of four common workload patterns and how Citus addresses those requirements. It also shares benchmark results demonstrating the performance and scalability of Citus in each of the workload patterns and describes how Microsoft uses Citus to address one of its most challenging data problems.

CCS CONCEPTS

• **Information systems → Relational parallel and distributed DBMSs**.

KEYWORDS

postgresql, distributed database, relational database, database extension

ACM Reference Format:
Umur Cubukcu, Ozgun Erdogan, Sumedh Pathak, Sudhakar Sannakkayala, and Marco Slot. 2021. Citus: Distributed PostgreSQL for Data-Intensive Applications. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3448016.3457551

1 INTRODUCTION

PostgreSQL is one of the most popular open source database management systems [19]. It is highly versatile and used across different industries and areas as diverse as particle physics [23] and geospatial databases [18]. One of the defining characteristics of PostgreSQL is its extensibility [24], which enables developers to add new database functionality without forking from the original project. Many companies have leveraged the rich functionality and ecosystem of PostgreSQL to build advanced, successful applications. This in turn has created significant demand for PostgreSQL to scale beyond a single server.

Over the past decade, our team has developed an open source PostgreSQL extension (plug-in) called Citus [3], which turns PostgreSQL into a distributed database management system (DDBMS). The goal of Citus is to address the scalability needs within the PostgreSQL ecosystem. At an early stage, we started offering Citus as a product, which drove us to talk to over a thousand companies that were looking to scale out PostgreSQL. From these conversations, we learned that the need for scale often goes hand in hand with complex application logic that relies on many different relational database capabilities and on performant implementations of those capabilities. In addition, applications rely on a broad ecosystem of tools and extensions.

Traditionally, new DDBMSs that aimed to offer compatibility with an existing relational database system have followed one of three approaches: (i) Build the database engine from scratch and write a layer to provide over-the-wire SQL compatibility, (ii) Fork an open source database systems and build new features on top of it, or (iii) Provide new features through a layer that sits between the application and database, as middleware. For each of these approaches, the cost of keeping up with the ongoing developments in the core project over the decades-long lifecycle of a database management system is substantial, and often insurmountable. Most projects lag by many years in terms of compatibility with new PostgreSQL features, tools, and extensions.

Citus is the first distributed database that delivers its functionality through the PostgreSQL extension APIs. The extension APIs provide sufficient control over the behavior of PostgreSQL to integrate a sharding layer, a distributed query planner and executor, and distributed transactions in a way that is transparent to the application. Being an extension allows Citus to maintain compatibility with the latest PostgreSQL features and tools at negligible engineering cost. Moreover, Citus distributes data across regular PostgreSQL servers and sends queries over the regular PostgreSQL protocol. This means that Citus can utilize all the data access and storage capabilities offered by the underlying PostgreSQL servers, including advanced capabilities such as JSONB, lateral joins, GiST indexes, array types and other extensions.

Building a distributed PostgreSQL engine that is 100% compatible with a single server and scales in all scenarios without performance regressions is perhaps impossible, but also unnecessary. Not every application that benefits from PostgreSQL also benefits from scaling out. We found that the PostgreSQL applications that benefit from scaling out largely fall into 4 workload patterns, namely: Multi-tenant / SaaS, real-time analytics, high-performance CRUD,

SIGMOD '21, June 20–25, 2021, Virtual Event, China

ACM ISBN 978-1-4503-8343-1/21/06.
https://doi.org/10.1145/3448016.3457551

Scale requirements	MT	RA	HC	DW
Typical query latency	10ms	100ms	1ms	10s+
Typical query throughput	10k/s	1k/s	100k/s	10/s
Typical data size	1TB	10TB	1TB	10TB

Table 1: Scale requirements of workload patterns for distributed relational databases

and data warehousing. Each workload pattern requires a different combination of capabilities from the database. The requirements of these workload patterns have largely driven the development of Citus.

This paper shares what we learned from building and deploying Citus over the years. As such, the paper brings three main contributions. First, we describe the four workload patterns that we observed in customer conversations and their requirements in terms of scale and distributed database capabilities. Second, we describe the PostgreSQL extension APIs and how Citus uses them to implement a comprehensive distributed database system. Finally, the Citus architecture shows how we addressed the requirements of a broad range of data-intensive applications within a single distributed database system.

The remainder of this paper is organized as follows:

- Section 2 distills the high-level requirements of four PostgreSQL workload patterns that benefit from scaling out.
- Section 3 describes how Citus implements and scales distributed query planning and execution, distributed transactions, and other database operations.
- Section 4 presents benchmark results demonstrating that Citus can scale PostgreSQL in each of the four workload patterns.
- Section 5 shows how Citus is used in a data-intensive real-time analytics application at Microsoft.
- Section 6 describes related work in distributed relational database management systems.
- Section 7 concludes the paper and shares some of our future work.

2 WORKLOAD REQUIREMENTS

When talking to companies looking to scale out PostgreSQL about potential Citus adoption, we observed that almost all workloads we encountered followed four patterns: Multi-tenant (MT), real-time analytics (RA), high-performance CRUD (HC), and data warehousing (DW).

Table 1 gives an overview of the approximate scale requirements in each workload pattern. It is worth noting that the latency, throughput, and data size requirements vary significantly, which in practice means that each workload pattern requires a different combination of distributed database capabilities to achieve its notion of high performance at scale. We describe the four workload patterns and capabilities requested for each workload (*in italics*) in the remainder of this Section and give an overview in Table 2. In the Citus architecture section, we will further describe these capabilities and how Citus implements them.

Feature requirements	MT	RA	HC	DW
Distributed tables	Yes	Yes	Yes	Yes
Co-located distributed tables	Yes	Yes	Yes	Yes
Reference tables	Yes	Yes	Yes	Yes
Local tables	Some	Some		
Distributed transactions	Yes	Yes	Yes	Yes
Distributed schema changes	Yes	Yes	Yes	Yes
Query routing	Yes	Yes	Yes	
Parallel, distributed SELECT		Yes		Yes
Parallel, distributed DML		Yes		
Co-located distributed joins	Yes	Yes		Yes
Non-co-located distributed joins				Yes
Columnar storage		Some		Yes
Parallel bulk loading		Yes		Yes
Connection scaling			Yes	

Table 2: Workload patterns and required distributed relational database capabilities.

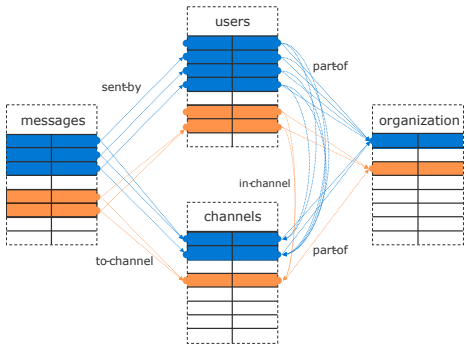


Figure 1: Data model for a simple multi-tenant messaging application with blue and orange representing different tenants

There are relatively complex relationships within the data model that are expressed through foreign keys and joins, but not across tenants. Hence multi-tenant applications can scale along the tenant dimension.

2.1 Multi-tenant

Multi-tenant applications serve many relatively independent tenants from a single backend deployment. A typical example is Software-as-a-Service (SaaS). Such applications often have complex OLTP workloads with many relationships, though tenants are relatively independent within the data model, as shown in the example in Figure 1. One of the benefits of the SaaS model that has helped make it successful is that the cost of adoption (i.e. adding a new tenant) is low for both user and application developer, which can lead to rapid growth. One of the challenges in scaling a multi-tenant workload is that the working set is relatively large due to the large number of independent tenants.

A traditional approach to scaling a relational database for a multi-tenant application is manual sharding. The data for each tenant is placed into its own database or schema (namespace). The application then needs to keep track of where each database or schema is placed, build infrastructure for moving data around, synchronize data and schema changes across potentially thousands of

databases, and use external systems to do analytics across tenants. This approach only scales up to the level of investment that the application developer is willing to make in building a distributed database management plane.

The alternative approach is to use a shared schema with tenant ID columns. To scale out this approach for multi-tenant applications in PostgreSQL, we identified several distributed database capabilities. These capabilities are essential to meet both the functional requirements of complex SaaS applications (including complex SQL, foreign keys, constraints, indexes) and the scale requirements (low query latency). Tables that contain tenant-specific data should all have a tenant ID column, such that they can be *distributed* and *co-located* by tenant ID. Co-location enforces that the same tenant ID is always on the same server, such that joins and foreign keys on the tenant ID can be performed locally. The database system should then be able to *route arbitrarily complex SQL queries* that filter by tenant ID to the appropriate server with minimal overhead to linearly scale query throughput. Attempts by the distributed query planner to analyze and break up the query will typically result in significant regressions from single server PostgreSQL. *Reference tables* that are replicated across servers are needed for (local) joins and foreign keys with tables that are shared across tenants. Applications may also perform analytics (e.g. *parallel queries* that do *co-located distributed joins* on the tenant ID) and transactions (incl. distributed *schema changes*) across all tenants.

In addition to distributed database capabilities, we also hear questions from customers that are specific to multi-tenant workloads. First, customers may need the flexibility to customize data for certain tenants. For these customers, we recommend adding new fields using the JSONB data type, which can be efficiently indexed through GIN or expression indexes. Second, customers may need control over tenant placement to avoid issues with noisy neighbors. For this, Citus provides features to view hotspots, to isolate a tenant onto its own server, and to provide fine-grained control over tenant placement [22].

Citus implements the combination of these capabilities required for multi-tenant workloads. As such, Citus provides a significantly simpler alternative to scaling out than the database-per-tenant approach.

2.2 Real-time Analytics

Real-time analytics applications provide interactive analytics or search on large streams of data with only minor delay. The main data stream typically consists of event data or time series data describing telemetry from devices or software. Use cases may include system monitoring, anomaly detection (e.g. fraud detection), behavioral analytics (e.g. funnel analysis, segmentation), geospatial analytics, and others. A common real-time analytics application is a multi-user analytics dashboard that visualizes aggregations of the data through charts. The database needs to be able to sustain a high write throughput to keep up with a stream of events, while the application issues hundreds of analytical queries per second. Queries should have sub second response times to be able to show results interactively, regardless of the data volume. Since the query set is known upfront, indexes, materialized views, and other data transformations (e.g. rollups) can be used to minimize response

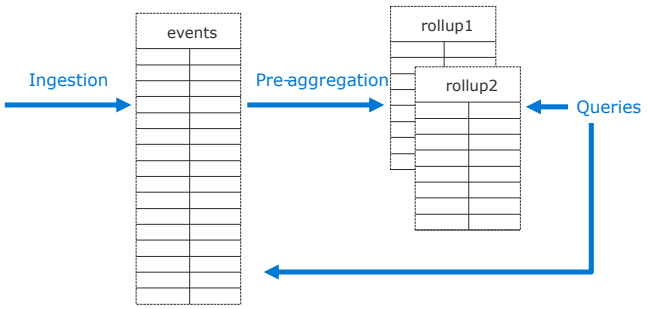


Figure 2: A simple real-time analytics pipeline

Data is ingested into a raw data table and then incrementally pre-aggregated into one or more rollups. The application may query both the rollups and the raw event data.

times. The database needs to be able to update these incrementally to quickly reflect new data in the application. An example of a typical real-time analytics pipeline is shown in Figure 2.

PostgreSQL has many powerful capabilities for building real-time analytics applications. Its heap storage format and COPY command allow very fast data ingestion, while MVCC allows analytical queries to run concurrently with writes. PostgreSQL also has a versatile set of data types and index types, including comprehensive support for arrays, JSON, and custom types. The only shortcoming of PostgreSQL for real-time analytics applications is that the data volume can easily exceed the capacity of a single server and most operations are single-threaded.

To scale real-time analytics workloads, the database system needs to be able to *distribute* tables across servers and support *parallel bulk loading* to keep up with the data volume. *Parallel, distributed DML*, in particular INSERT..SELECT, is needed to be able to incrementally pre-aggregate large volumes of incoming data into rollup tables. *Co-location* between the source table and the rollup table enables very fast INSERT..SELECT. Queries from the dashboard use *query routing* or *parallel, distributed SELECT* on the event data or the rollup tables to keep response times low. *Co-located distributed joins* are needed for advanced analytics (e.g. funnel queries).

Citus supports the capabilities required for building large-scale real-time analytics dashboards on PostgreSQL. Section 5 gives an example of a petabyte-scale real-time analytics use case at Microsoft.

2.3 High-performance CRUD

High-performance CRUD workloads involve many objects / documents that are modified in a relatively independent manner. The application primarily accesses the data through simple CRUD (create, read, update, delete) operations on a key, but may also issue more complex queries across objects. The objects typically follow an unstructured data format like JSON.

PostgreSQL is a popular choice for this type of workload because of its sophisticated JSON support. A large PostgreSQL server can handle hundreds of thousands of writes and millions of reads per second. Scalability problems can arise when the working set is large or when making incremental changes to large objects at a high rate. The PostgreSQL MVCC model requires writing a new copy and later reclaiming the space (auto-vacuuming), which can

cause performance degradation if auto-vacuuming cannot keep up. Another limitation of PostgreSQL is that it can only handle a limited number of (idle) connections, because of the process-per-connection architecture and the relatively high memory overhead of a process.

To scale high performance CRUD workloads, tables need to be *distributed* by key. CRUD operations can then be *routed* to a shard with very low planning overhead and use the primary key index within the shard, enabling high throughput and low latency. A significant benefit of sharding PostgreSQL tables in high performance CRUD workloads is that, apart from the ability to use more hardware, auto-vacuuming is parallelized across many cores. *Reference tables* are not strictly required, but can make objects significantly more compact through normalization, which is beneficial at scale. *Parallel, distributed SELECT and DML* are useful for performing scans and analytics across a large number of objects. To *scale the number of connections*, any server needs to be able to process distributed queries.

Citus meets all the requirements for high performance CRUD workloads, but still has some inherent limitations around connection scalability. We are working with the community to improve on connection scalability in PostgreSQL itself [20].

2.4 Data warehousing

Data warehousing applications combine data from different sources into a single database system to generate ad-hoc analytical reports. The application typically does not have low latency or high throughput requirements, but queries may need to scan very large amounts of data. The database needs to support very fast scans and be able to find efficient query plans for handwritten SQL involving arbitrary joins.

PostgreSQL generally lacks the scanning performance and parallelism to perform very large scans within a reasonable amount of time. On the other hand, its performance features, comprehensive SQL support, and ecosystem still make it an attractive option for analytics.

To scale data warehouse applications, scans need to be sped up through *parallel, distributed SELECT* and *columnar storage*. Distribution columns should be chosen to maximize the number of *co-located distributed joins*, but the database needs to support efficient *non-co-located joins* as well by reshuffling or broadcasting data over the network. The query optimizer needs to decide on a join order that minimizes network traffic.

At the time of writing, Citus meets most of the requirements for data warehouse applications, but has several limitations around non-co-located joins (e.g. correlated subqueries are unsupported), which limits its applicability in some data warehousing workloads.

2.5 Other workloads

Some workloads have not been a focus area of Citus, primarily because we rarely observed them among potential customers.

There are various workload patterns that PostgreSQL could address given its extensibility, but the ecosystem is centered around other database systems and tools. Examples include streaming analytics, Extract-Transform-Load (ETL), machine learning, and text search. We believe PostgreSQL can be successful in these areas, but it will take broader movements within developer communities to make PostgreSQL an important part of those ecosystems.

We also observed that complex, single-tenant OLTP workloads are less likely to run into the large working set problem that occur in multi-tenant OLTP workloads. Moreover, scaling out a complex, single-tenant OLTP application often lowers overall throughput, because network latency lowers per-connection throughput and results in locks being held for longer. This in turn lowers achievable concurrency.

Overall, the observations about the workload patterns described in this section have significantly influenced the Citus architecture.

3 CITUS ARCHITECTURE

In a Citus cluster, all servers run PostgreSQL with the Citus extension plus any number of other extensions installed. Citus uses the PostgreSQL extension APIs to change the behavior of the database in two ways: First, Citus replicates database objects such as custom types and functions to all servers. Second, Citus adds two new table types that can be used to take advantage of additional servers. The remainder of this section describes how Citus implements and scales the most important operations for Citus tables.

3.1 PostgreSQL extension APIs

A PostgreSQL extension consists of two parts: a set of SQL objects (e.g. metadata tables, functions, types) and a shared library that is loaded into PostgreSQL. All database modules within PostgreSQL are extensible, except for the parser. The main reason is that the parser code is generated at build time, while the extension infrastructure loads the shared library at run time. Keeping the parser non-extensible also forces syntactic interoperability between extensions.

Once a PostgreSQL extension is loaded, it can alter the behavior of PostgreSQL by setting certain hooks. Citus uses the following hooks:

User-defined functions (UDFs) are callable from SQL queries as part of a transaction and are primarily used to manipulate the Citus metadata and implement remote procedure calls.

Planner and executor hooks are global function pointers that allow an extension to provide an alternative query plan and execution method. After PostgreSQL parses a query, Citus checks if the query involves a Citus table. If so, Citus generates a plan tree that contains a CustomScan node, which encapsulates a distributed query plan.

CustomScan is an execution node in a PostgreSQL query plan that holds custom state and returns tuples via custom function pointers. The Citus CustomScan calls the distributed query executor, which sends queries to other servers and collects the results before returning them to the PostgreSQL executor.

Transaction callbacks are called at critical points in the lifecycle of a transaction (e.g. pre-commit, post-commit, abort). Citus uses these to implement distributed transactions.

Utility hook is called after parsing any command that does not go through the regular query planner. Citus uses this hook primarily to apply DDL and COPY commands that affect Citus tables.

Background workers run user-supplied code in separate processes. Citus uses this API to run a maintenance daemon. This

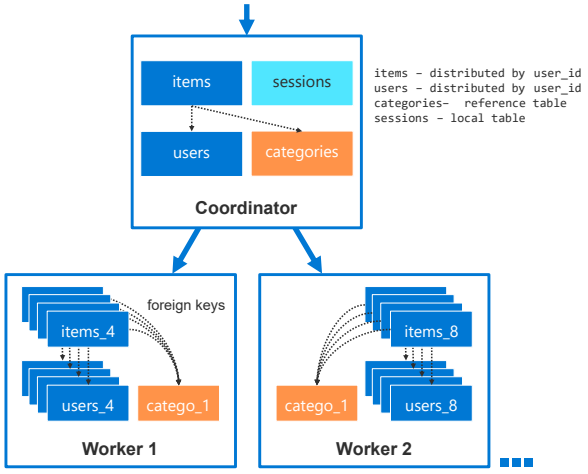


Figure 3: Example Citus deployment, showing the coordinator and two worker nodes.

daemon performs distributed deadlock detection, 2PC prepared transaction recovery, and cleanup.

Through these hooks, Citus can intercept any interaction between the client and the PostgreSQL engine that involves Citus tables. Citus can then replace or augment PostgreSQL's behavior.

3.2 Citus architecture diagram

Citus deployments typically have 1 coordinator and 0-n worker nodes, as shown in the Figure 3. The coordinator stores the metadata (catalogs) of the distributed tables and clients typically connect to the coordinator. A PostgreSQL server implicitly becomes the coordinator when the user adds a worker node via a Citus UDF. Worker nodes store the shards that contain the actual data. When the cluster is small, the coordinator itself can also be used as a worker node, so the smallest possible Citus cluster is a single server.

The benefit of using a single coordinator as the entry point is that PostgreSQL libraries and tools can interact with a Citus cluster as if it was a regular PostgreSQL server. Since the overhead of distributed queries is small compared to query execution, a large coordinator node can handle hundreds of thousands of transactions per second or ingest millions of rows per second via PostgreSQL's COPY command.

3.2.1 Scaling the coordinator node. The coordinator can become a scaling bottleneck in some cases, for example, when serving demanding high performance CRUD workloads. To resolve this bottleneck, Citus can distribute the distributed table metadata and all changes to it to all the nodes. In this case, each worker node assumes the role of coordinator for all distributed queries and transactions it receives. Clients should use a load balancing mechanism to divide connections over the workers. Since DDL commands modify distributed metadata, the application should continue to connect to the coordinator when issuing DDL commands. Since the volume of DDL commands is low, the coordinator no longer becomes a bottleneck.

Today, we recommend this more complex mode only when customers are familiar with PostgreSQL and have an actual scaling bottleneck. When each node can coordinate and also serve queries, each connection to the Citus cluster creates one or more connections within the cluster. Citus caches connections for higher performance, and this could lead to a connection scaling bottleneck within the cluster. We typically mitigate this issue by setting up connection pooling between the instances, via PgBouncer [10]. We are working to improve on connection scaling behavior in future versions of Citus and PostgreSQL itself [20].

3.3 Citus table types

Citus introduces two types of tables to PostgreSQL: Distributed tables and reference tables, without taking away the concept of regular ("local") PostgreSQL tables. Citus tables are initially created as regular PostgreSQL tables, and then they are converted by calling Citus-specific functions. After conversion, Citus intercepts all commands involving Citus tables in the relevant PostgreSQL hooks.

3.3.1 Distributed tables. Distributed tables are hash-partitioned along a distribution column into multiple logical shards with each shard containing a contiguous range of hash values. The advantage of hash-partitioning is that it enables co-location and reasonably well-balanced data without the need for frequent resharding. Range-partitioning is also available for some advanced use cases.

The create_distributed_table UDF converts a regular table to a distributed table by creating the shards on the workers and adding to the Citus metadata.

```
CREATE TABLE my_table (. . . );
SELECT create_distributed_table('my_table',
    'distribution_column');
```

Shards are placed on worker nodes in a round-robin fashion. A single worker node can contain multiple logical shards, such that the cluster can be rebalanced by moving individual shards between worker nodes.

3.3.2 Co-location. Citus ensures that the same range of (hash) values is always on the same worker node among distributed tables that are co-located. Relational operations (e.g., joins, foreign keys) that involve the distribution column of two or more co-located distributed tables can be performed without any network traffic by operating on pairs of co-located shards.

When creating a second distributed table, co-location can be specified using the colocate_with option:

```
CREATE TABLE other_table (. . . );
SELECT create_distributed_table('other_table',
   'distribution_column', colocate_with := 'my_table');
```

If no colocate_with option is specified, Citus automatically co-locates distributed tables based on the data types of the distribution columns. We found this to be helpful for users unfamiliar with distributed database concepts.

3.3.3 Reference tables. Reference tables are replicated to all nodes in a Citus cluster, including the coordinator. Joins between distributed tables and reference tables are implemented by joining each shard of the distributed table with the local replica of the reference table. Similarly, foreign keys from distributed tables to reference tables are enforced locally by creating regular foreign

keys between each of the shards of the distributed table and the local replica of the reference table.

Users create reference tables by calling create_reference_table:

```
CREATE TABLE dimensions (. . . );
SELECT create_reference_table('dimensions');
```

After conversion, writes to the reference table are replicated to all nodes and reads are answered directly by the coordinator or via load-balancing across worker nodes.

3.4 Data rebalancing

Most Citus clusters grow in data size and query volume. Certain worker nodes may also receive more load than others due to data distribution and query patterns. To enable an even distribution, Citus provides a shard rebalancer. By default, the rebalancer moves shards until it reaches an even number of shards across worker nodes. Alternatively, users can choose to rebalance based on data size or create a custom policy by defining cost, capacity, and constraint functions in SQL [7].

Most rebalance operations start with the customer changing their cluster size. The rebalancer then picks one shard and any shards that are co-located with it; and starts a shard move operation. To move shards, Citus creates a replica of the shards on a different node using PostgreSQL's logical replication. With logical replication, the shards in transit can continue to receive read and write queries. When the shard replicas have caught up with their source, Citus obtains write locks on the shards, waits for replication to complete, and updates distributed table metadata. From that point on, any new queries go to the new worker node. The last few steps typically only take a few seconds, hence there is minimal write downtime.

3.5 Distributed query planner

When a SQL query references a Citus table, the distributed query planner produces a PostgreSQL query plan that contains a CustomScan node, which contains the distributed query plan. A distributed query plan consists of a set of tasks (queries on shards) to run on the workers, and optionally a set of subplans whose results need to be broadcast or re-partitioned, such that their results can be read by subsequent tasks.

Citus needs to handle a wide range of workloads that require different query planning strategies to scale. Simple CRUD queries benefit from minimal planning overhead. Complex data warehousing queries on the other hand benefit from advanced query optimizations, which incur higher planning overhead. Over time, Citus evolved to have planners for different classes of queries. Figure 4 gives a basic example for each planner. We further describe those planners below.

Fast path planner handles simple CRUD queries on a single table with a single distribution column value. The planner extracts the distribution column value directly from a filter in the query and determines the shard that matches the value. The planner then rewrites the table name to the shard name to construct the query to run on the worker, which can be done with minimal CPU overhead. Hence, the fast path planner supports high throughput CRUD workloads.

Router planner handles arbitrarily complex queries that can be scoped to one set of co-located shards. The router planner checks

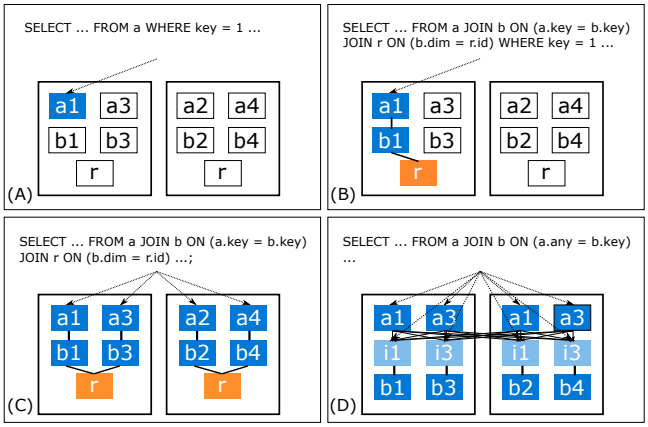


Figure 4: Citus planning examples: (A) Fast path planner showing single shard access, (B) Router planner showing single shard co-located- and reference table joins, (C) Logical pushdown planner showing multi-shard co-located- and reference table joins, (D) Logical join order planner showing one table being re-partitioned into intermediate results to perform a non-co-located join.

or infers whether all distributed tables have the same distribution column filter. If so, the table names in the query are rewritten to the names of the co-located shards that match the distribution column value. The router planner implicitly supports all SQL features that PostgreSQL supports since it will simply delegate the full query to another PostgreSQL server. Hence, the router planner enables multi-tenant SaaS applications to use all SQL features with minimal overhead.

Logical planner handles queries across shards by constructing a multi-relational algebra tree [15]. Multi-relational algebra formalizes two distributed execution primitives that are not available in PostgreSQL, to collect and repartition data. This difference influences the separation between router and logical planner.

The goal of the logical planner is to push as much of the computation to the worker nodes as possible before merging the results on the coordinator. We further distinguish between two logical planning strategies:

(1) **Logical pushdown planner** detects whether the join tree can be fully pushed down. This requires that all distributed tables have co-located joins between them and that subqueries do not require a global merge step (e.g. a GROUP BY must include the distribution column). If so, the planner can be largely agnostic to the SQL constructs being used within the join tree, since they are fully delegated to the worker nodes, and the distributed query plan becomes trivially parallel.

(2) **Logical join order planner** determines the optimal execution order for join trees involving non-co-located joins. It evaluates all possible join orders between distributed tables and subqueries using co-located joins, broadcast joins, and re-partition joins, and chooses the order that minimizes the network traffic. Broadcast joins and re-partition joins result in subplans with filters and projections pushed into the subplan.

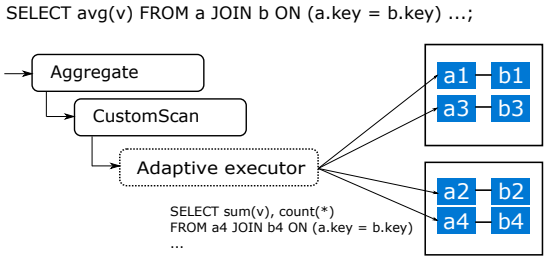


Figure 5: Call flow for the execution of a a simple analytical query that was planned by the logical pushdown planner.

For each query, Citus iterates over the four planners, from lowest to highest overhead. If a particular planner can plan the query, Citus uses it. This approach works well for two reasons: First, there is an order of magnitude difference between each workload's latency expectations as described in Table 1. Second, this difference also applies to each planner's overhead. As an example, a complex data warehousing query could take minutes to complete. In that case, the user does not mind paying a few milliseconds of overhead for the fast path, router, and pushdown planner.

3.6 Distributed query executor

A PostgreSQL query plan is structured as a tree of execution nodes that each have a function to return a tuple. Plans generated by the distributed query planner contain a CustomScan node that calls into the distributed query executor. In case the plan was generated via the fast path or router planner, the entire plan is a single CustomScan node since the execution is fully delegated to a single worker node. Plans generated by the logical planner may require a merge step (e.g. aggregation across shards). In that that case, there will be additional execution nodes above the CustomScan, which are handled by the regular PostgreSQL executor.

When the PostgreSQL executor calls into the CustomScan, Citus first executes subplans (if any) and then hands the execution to a component called the "adaptive executor". Figure 5 shows an example of the call flow for a simple analytical query.

3.6.1 Adaptive executor. The design of the adaptive executor is driven by the need to support a mixture of workloads and by the process-per-connection architecture of PostgreSQL. Some query plans will have a single task that needs to be routed to the right worker node with minimal overhead, while other query plans have many tasks that Citus runs in parallel by opening multiple connections per worker node. We found parallelizing queries via multiple connections to be more versatile and performant than the built-in parallel query capability in PostgreSQL that uses a fixed set of processes. The downside of opening multiple connections is the cost of connection establishment and overhead of extra processes, in particular when many distributed queries run concurrently.

The adaptive executor manages the parallelism vs. low latency trade-off using a technique we call "slow start". At the start of the query, the executor can use one connection to each worker (n=1). Every 10ms, the number of new connections that can be opened increases by one (n=n+1). If there are t pending tasks for a worker node that are not assigned to a specific connection, the executor increases the connection pool of that worker node by min(n,t) new connections. The reasoning behind slow start is that a simple in-memory index lookup typically takes less than a millisecond, so all tasks on a worker are typically executed before opening any additional connections. On the other hand, analytical queries often take hundreds of milliseconds or more, and the delayed connection establishment will barely be noticeable in the overall runtime.

While slow start increases the number of connections when tasks take a long time to complete, sometimes tasks take a long time because there are already many concurrent connections issuing queries to the worker node. Therefore, the executor also keeps track of the total number of connections to each worker node in shared memory to prevent it from exceeding a shared connection limit. When the counter reaches the limit, opening additional connections is avoided such that the overall number of outgoing connections remains at or below the limit. The implementation converges to a state where the available connection slots on the worker nodes are fairly distributed between the processes executing distributed queries on the coordinator.

When multiple connections per worker node are used, each connection will access a different subset of shards, and hold uncommitted writes and locks in case of a multi-statement transaction. For every connection, Citus therefore tracks which shards have been accessed to ensure that the same connection will be used for any subsequent access to the same set of co-located shards in the same transaction. When starting the execution of a statement, tasks are assigned to a connection if there was a prior access to the shards accessed within the transaction, and otherwise they are assigned to the general pool for the worker node. When a connection is ready, the executor first takes an assigned task from its queue, and otherwise takes a task from the general pool.

By combining slow start, the shared connection limit, and the task assignment algorithm, the adaptive executor can handle a variety of workload patterns, even when they run concurrently on a single database cluster, and support complex interactive transaction blocks without sacrificing parallelism.

3.7 Distributed transactions

Distributed transactions in Citus comprise a transaction on the coordinator, initiated by the client, and one or more transactions on worker nodes, initiated by the executor. For transactions that only involve a single worker node, Citus delegates responsibility to the worker node. For transactions that involve multiple nodes, Citus uses two-phase commit (2PC) for atomicity and implements distributed deadlock detection.

3.7.1 Single-node transactions. The simplest type of transaction Citus supports is a single statement transaction that goes to a single node (e.g. CRUD operations). In that case, there is no overhead to using a distributed table other than the extra round trip between coordinator and worker. When handling a multi-statement transaction in which all statements are routed to the same worker node (e.g. operations on a single tenant in a multi-tenant app), the coordinator simply sends commit/abort commands to that worker node from the commit/abort transaction callbacks. The worker node, by definition, provides the same transactional guarantees as a single PostgreSQL server.

3.7.2 Two-phase commit protocol. For transactions that write to multiple nodes, the executor opens transaction blocks on the worker nodes and performs a 2PC across them at commit time. PostgreSQL implements commands to prepare the state of a transaction in a way that preserves locks and survives restarts and recovery. This enables later committing or aborting the prepared transaction. Citus uses those commands to implement a full 2PC protocol.

When the transaction on the coordinator is about to commit, the pre-commit callback sends a "prepare transaction" over all connections to worker nodes with open transaction blocks. If successful, the coordinator writes a commit record for each prepared transaction to the Citus metadata and then the local transaction commits, which ensures the commit records are durably stored. In the post-commit and abort callbacks, the prepared transactions are committed or aborted on a best-effort basis.

When one or more prepared transactions fail to commit or abort, the commit record in the Citus metadata is used to determine the outcome of the transaction. A background daemon periodically compares the list of pending prepared transactions on each worker to the local commit records. If a commit record is present (read: visible) for a prepared transaction, the coordinator committed hence the prepared transaction must also commit. Conversely, if no record is present for a transaction that has ended, the prepared transaction must abort. When there are multiple coordinators, each coordinator performs 2PC recovery for the transactions it initiated. Since both commit records and prepared transactions are stored in the write-ahead log (which may be replicated, see Section 3.9), this approach is robust to failure of any of the nodes involved.

3.7.3 Distributed deadlocks. A challenge in multi-node transactions is the potential for distributed deadlocks, in particular between multi-statement transactions. To overcome this challenge, deadlock prevention or deadlock detection methods can be used. Deadlock prevention techniques such as wound-wait require a percentage of transactions to restart. PostgreSQL has an interactive protocol, which means results might be returned to the client before a restart occurs and the client is not expected to retry transactions. Hence, wound-wait is unsuitable for Citus. To maintain PostgreSQL compatibility, Citus therefore implements distributed deadlock detection, which aborts transactions when they get into an actual deadlock.

PostgreSQL already provides deadlock detection on a single node. Citus extends on this logic with a background daemon running on the coordinator node. This daemon detects distributed deadlocks by polling all worker nodes for the edges in their lock graph (*process a waits for process b*) every 2 seconds, and then merging all processes in the graph that participate in the same distributed transaction. If the resulting graph contains a cycle, then a cancellation is sent to the process belonging to the youngest distributed transaction in the cycle to abort the transaction.

Unless there is an actual deadlock, only a small number of transactions will be waiting for a lock in typical (distributed) database workloads, hence the overhead of distributed deadlock detection is small. When distributed deadlocks happen frequently, users are recommended to change the statement order in their transactions.

3.7.4 Multi-node transaction trade-offs. Multi-node transactions in Citus provide atomicity, consistency, and durability guarantees, but do not provide distributed snapshot isolation guarantees. A concurrent multi-node query could obtain a local MVCC snapshot before commit on one node, and after commit on another. Addressing this would require changes to PostgreSQL to make the snapshot manager extensible. In practice, we did not find a strong need for distributed snapshot isolation in the four workload patterns, and customers did not express a need for it yet. Most transactions in multi-tenant and CRUD applications are scoped to a single node, meaning they get isolation guarantees on that node. Analytical applications do not have strong dependencies between transactions and are hence more tolerant to relaxed guarantees.

Distributed snapshot isolation can be important in certain hybrid scenarios. However, existing distributed snapshot isolation techniques have a significant performance cost due to the need for additional network round trips or waiting for the clock, which increases response times and lowers achievable throughput. In the context of the synchronous PostgreSQL protocol, throughput is ultimately capped by #connections / response time. Since making a very large number of database connections is often impractical from the application perspective, low response time is the only way to achieve high throughput. Hence, we would likely make distributed snapshot isolation optional if we implement it in the future.

3.8 Specialized scaling logic

Apart from SELECT and DML commands that are handled via the distributed query planner and executor, there are several other important PostgreSQL capabilities for which Citus implements specialized scaling logic.

DDL commands in PostgreSQL are transactional, online operations. Citus preserves this property by taking the same locks as PostgreSQL and propagating the DDL commands to shards via the executor in a parallel, distributed transaction.

COPY commands append a CSV-formatted stream of data to a table. In PostgreSQL, this happens in a single thread, which also needs to update indexes and checks constraints. In case of Citus, the coordinator opens COPY commands for each of the shards and streams rows to the shards asynchronously, which means writes are partially parallelized across cores even with a single client.

INSERT..SELECT between distributed tables use one of 3 strategies: If the SELECT requires a merge step on the coordinator, the command is internally executed as a distributed SELECT and a COPY into the destination table. If there is no merge step, but the source and destination tables are not co-located, the INSERT..SELECT performs distributed re-partitioning of the SELECT result before inserting into the destination table. Otherwise, the INSERT..SELECT is performed directly on the co-located shards in parallel.

Stored procedures can be delegated to a worker node based on a distribution argument and a co-located distributed table to avoid network round trips between coordinator and worker nodes. The worker node can then perform most operations locally without network round trips, but it can also perform a distributed transaction across worker nodes when necessary.

3.9 High Availability and backups

High availability (HA) and backups for distributed database systems are complex topics that need to be looked at holistically from the

Workload	Benchmark
Multi-tenant	HammerDB TPC-C-based
Real-time analytics	Custom microbenchmarks
High-performance CRUD	YCSB
Data warehouse	Queries from TPC-H

Table 3: Benchmarks used for different workload patterns

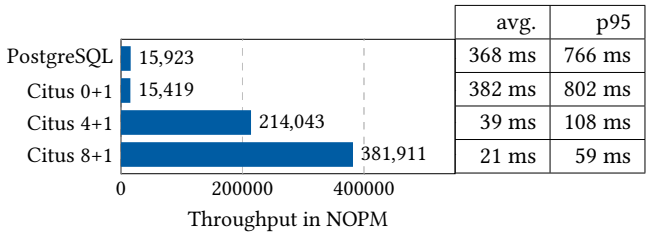


Figure 6: HammerDB TPC-C results with 250 vusers and 500 warehouses (~100GB) in "new order" transactions per minute (NOPM) and response times in milliseconds.

perspective of the user and the platform that runs the database. As this paper is primarily focused on the Citus extension, we only give a brief overview of the typical HA and backup approach for Citus clusters and leave further details to future papers.

HA in Citus is handled primarily at the server level using existing PostgreSQL replication. In an HA setup, each node in the cluster has one or more hot standby nodes and replicates its write-ahead log (WAL) using synchronous, asynchronous or quorum replication. When a node fails, the cluster orchestrator promotes a standby and updates the Citus metadata, DNS record, or virtual IP. The whole failover process takes 20-30 seconds, during which distributed transactions involving the node roll back. The orchestrator is typically part of the control plane in a managed service, but on-premises users can use the pg_auto_failover extension [9] to serve the same function.

Backups are also handled primarily at the server level by creating periodic disk snapshots or copies of the database directory and continuously archiving the WAL to remote storage in each server. Citus supports periodically creating a consistent restore point, which is a record in the WAL of each node. The restore point is created while blocking writes to the commit records table(s) on the coordinator(s), which prevents in-flight 2PC commits while creating the restore point. Restoring all servers to the same restore point guarantees that all multi-node transactions are either fully committed or aborted in the restored cluster, or can be completed by the coordinator through 2PC recovery on startup.

4 BENCHMARKS

This section presents benchmark results that compare the performance of Citus with one PostgreSQL server. With Citus, our goal has been to turn PostgreSQL into a distributed database. This goal included providing compatibility with PostgreSQL features and its ecosystem of libraries and tools. As such, when customers approached us, their primary performance baseline was single node PostgreSQL. The benchmarks here reflect that performance baseline for target workloads described in Section 2. Table 3 summarizes the relationship between workload patterns and benchmarks.

Each benchmark compares: **PostgreSQL**–A single PostgreSQL server, **Citus 0+1**–a single PostgreSQL server that uses Citus to shard data locally (coordinator also acts as worker), **Citus 4+1**–a Citus cluster with a coordinator and 4 worker nodes, and **Citus 8+1**–a Citus cluster with a coordinator and 8 worker nodes. All servers were Microsoft Azure virtual machines with 16 vcpus, 64 GB of memory, and network-attached disks with 7500 IOPS, running PostgreSQL 13 and Citus 9.5 with default settings. Benchmarks were run from a separate driver node.

Each benchmark is structured such that a single server cannot keep all the data in memory, but Citus 4+1 can, which demonstrates the often dramatic effect of scaling out memory along with CPU and I/O capacity. Citus 8+1 demonstrates the effect of scaling out only CPU and I/O capacity compared to Citus 4+1.

4.1 Multi-tenant benchmarks

To simulate a multi-tenant workload, we used the HammerDB [6] TPC-C-based workload, which is an OLTP benchmark that models an order processing system for warehouses derived from TPC-C [1]. The benchmark effectively models a multi-tenant OLTP workload in which warehouses are the tenants. Most tables have a warehouse ID column and most transactions only affect a single warehouse ID, which allows the workload to scale. Around ~7% of transactions span across multiple warehouses and are likely to be multi-node transactions in Citus.

We configured HammerDB 3.3 with 500 warehouses (~100GB of data), 250 virtual users (connections), a 1ms sleep time between transactions, and a 1 hour runtime, and ran it against each set up. In case of Citus, we converted the *items* table to a reference table and the remaining tables to co-located distributed tables with the warehouse ID column as the distribution column. Additionally, we configured Citus to delegate stored procedure calls to worker nodes based on the warehouse ID argument.

The New Orders Per Minute (NOPM) results obtained from running HammerDB against each set up are shown in Figure 6. On single server PostgreSQL and Citus 0+1 the data set does not fit in memory, which means that the amount of I/O is relatively high and bottlenecked on a single disk. Citus does not provide immediate performance benefits for OLTP workloads on the same hardware, hence Citus 0+1 is slightly slower than single server PostgreSQL due to (small) distributed query planning overhead. The main benefit of using Citus for OLTP workloads is that it can scale beyond a single server to ensure the working set fits in memory and sufficient I/O and CPU is available.

Throughput on Citus 4+1 is around 13 times higher than throughput on a single PostgreSQL server with only 5 times more hardware because the cluster can keep all data in memory. Hence, Citus 4+1 performs less I/O and becomes CPU bottlenecked. From 4 to 8 nodes, Citus shows slightly sublinear scalability. This is expected for the TPC-C-based workload due to the ~7% of transactions that span across nodes. The response time of these transactions is dominated by network round-trips for individual statements sent between nodes, which remains the same as the cluster scales out.

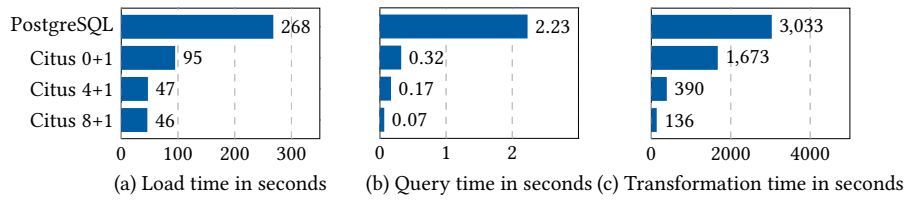


Figure 7: Real-time analytics microbenchmarks using ~100GB of GitHub Archive data: (a) Single session COPY, (b) Dashboard query using GIN index, (c) Data transformation using INSERT..SELECT

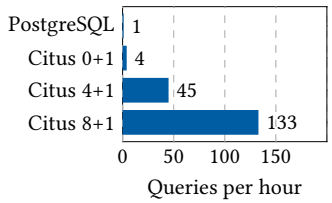
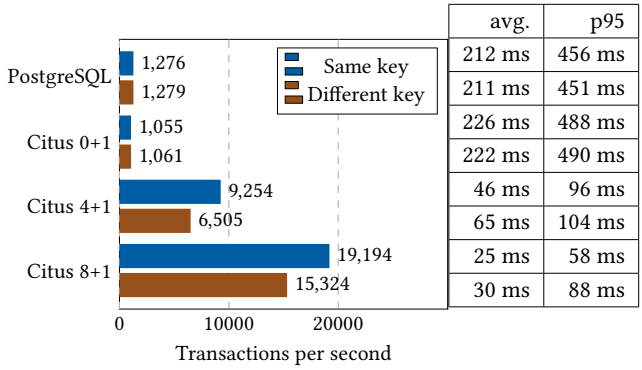


Figure 8: Data warehousing benchmark using queries from TPC-H at scale factor 100 (~135GB)



avg.	p95
212 ms	456 ms
211 ms	451 ms
226 ms	488 ms
222 ms	490 ms
46 ms	96 ms
65 ms	104 ms
25 ms	58 ms
30 ms	88 ms

Figure 9: Distributed transactions benchmark comparing two updates on using the same distribution key vs. different keys across two tables.

4.1.1 Distributed transaction performance. Multi-tenant applications mostly have single-tenant transactions, but there may be cross-tenant transactions such as the ones simulated by HammerDB. To get a more accurate measure of the overhead of 2PC, we created a synthetic benchmark using two tables of 50GB generated by the pgbench tool that comes with PostgreSQL. We then distributed and co-located these tables by key, and defined a simple multi-statement transaction:

```
UPDATE a1 SET v = v + :d WHERE key = :key1;
UPDATE a2 SET v = v - :d WHERE key = :key2;
```

We ran the transactions using pgbench for 1 hour with 250 connections. In one set of runs we used the same random value for both keys, such that these are two co-located updates. In another set of runs we used a different random value, which results in a 2PC when the keys are on different nodes. The results are displayed in Figure 9. The figure shows 2PC incurs a 20-30% performance penalty, but scales with the number of worker nodes.

4.2 Real-time analytics benchmark

There is not a standard real-time analytics benchmark, so we ran several microbenchmarks for the individual commands involved in real-time analytics. We used publicly available data from the GitHub archive [4] in JSON format and loaded data for January 2020 into a table defined as follows:

```
CREATE TABLE github_events (
 event_id text default md5(random()::text) primary key,
 data jsonb);
SELECT create_distributed_table('github_events',
 'event_id'); -- Citus only
CREATE INDEX text_search_idx ON github_events
USING GIN ((jsonb_path_query_array(data,
 '$.payload.commits[*].message')::text) gin_trgm_ops);
```

We used the pg_trgrm extension (included in PostgreSQL) to index the commit messages within the JSON data. The index makes queries for a substring in a commit message much faster, at the cost of increased write overhead. We created the index both on PostgreSQL and on the Citus clusters.

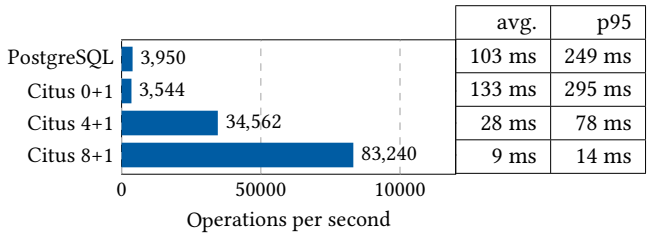
Our first microbenchmark measures ingestion performance in the presence of large indexes. We appended the first day of February 2020 (4.4GB of JSON data) using the COPY command. The average load times over 5 runs are shown in Figure 7(a). In this case, Citus 0+1 gives a speed up over PostgreSQL due to the partial parallelism described in Section 3.8. The Citus cluster with 4 worker nodes can speed up the COPY further due to the greater number of cores and I/O capacity. After that, the single COPY command becomes bottlenecked on a single core on the coordinator, hence increasing to 8 worker nodes does not provide additional speed up. To resolve this bottleneck, customers ingest data by running concurrent COPY commands.

Our second microbenchmark is a query that might be run by a dashboard: Compute the number of commits that contain the phrase "postgres" per day.

```
SELECT (data->>'created_at')::date,
  sum(jsonb_array_length(data->'payload'->'commits'))
FROM github_events
WHERE jsonb_path_query_array(data,
  '$.payload.commits[*].message')::text
  ILIKE '%postgres%' GROUP BY 1 ORDER BY 1 ASC;
```

The average runtime of the dashboard query over 5 runs, excluding the first to mitigate the variability caused by cache misses, is shown in Figure 7(b). The query only reads from memory and is largely bottlenecked on CPU, hence the greater parallelism provided by Citus enables the query to run faster, even on a single server.

Finally, real-time analytics often involves INSERT..SELECT queries to transform or pre-aggregate the data. For our third microbenchmark, we defined a data transformation step that extract commits from the GitHub push events. Average runtime over five runs is show in Figure 7(c). The parallelization significantly speeds up the INSERT..SELECT with a 96% reduction in runtime on Citus 8+1



	avg.	p95
PostgreSQL	103 ms	249 ms
Citus 0+1	133 ms	295 ms
Citus 4+1	28 ms	78 ms
Citus 8+1	9 ms	14 ms

Figure 10: YCSB Workload A results on 100M rows (~100GB) and response times for updates.

compared to a single PostgreSQL server, showing the ability of Citus to scale complex transactional data transformations.

4.3 High performance CRUD benchmark

The Yahoo Cloud Serving Benchmark (YCSB) [16] is designed to test high performance CRUD workloads on NoSQL databases. YCSB also has a JDBC driver that supports PostgreSQL. We ran workload A (50% reads, 50% updates) from YCSB on a table of 100M rows (~100GB) using 256 threads with uniform request distribution.

For this benchmark, the coordinator's CPU usage becomes a scaling bottleneck. Hence, we ran the benchmark with every worker node acting as coordinator and configured YCSB to load balance across all nodes. The results appear in Figure 10. The workload is largely I/O bound, hence throughput scales linearly with the higher I/O capacity when adding worker nodes. Single server Citus performs slightly worse than PostgreSQL due to the additional overhead of distributed query planning. On bigger clusters, the speed up is roughly proportional to the amount of I/O capacity with a small additional speed up due to data fitting in memory.

4.4 Data warehousing benchmark

A standard benchmark for data warehouses is TPC-H [2]. Queries in TPC-H do not have selective filters and therefore scan most of the data. Answering a TPC-H query quickly requires fast scanning and processing, which Citus achieves mainly through distributed parallelism and keeping more data in memory. At the time of writing, 4 of the 22 queries in TPC-H are not yet supported.

We used HammerDB to generate a TPC-H schema with scale factor 100 (~135GB) and distributed and co-located the *lineitem* and *orders* table by order key, and converted the smaller tables to reference tables to enable local joins. We then ran the 18 queries supported by Citus over a single session on each set-up. Figure 8 shows the number of queries per hour based on the completion time of the full set of queries over a single session.

Citus can achieve significant speeds up compared to PostgreSQL by efficiently utilizing all available cores. The fact that TPC-H queries scan all the data and the tables do not fully fit in memory, also means the single server is I/O bottlenecked while the Citus cluster is only CPU bottlenecked, which results in a two orders of magnitude speedup on the 8 node cluster compared to a single PostgreSQL server.

5 CITUS CASE STUDY: VENICEDB

Citus is used in many large-scale production systems that rely on a broad array of PostgreSQL and Citus capabilities to get the most out of their hardware. A good example of this is the VeniceDB project at Microsoft.

Microsoft uses Citus to analyze Windows measure data, which is derived from the telemetry coming in from hundreds of millions of Windows devices. Metrics are displayed on a real-time analytics dashboard called "Release Quality View" (RQV), which helps Windows engineering teams to assess the quality of the customer experience for each Windows release at the device grain. The RQV dashboard is a critical tool for Windows engineers, program managers, and executives, with hundreds of users per day.

The data store underlying RQV, code named VeniceDB, is powered by two >1000 core Citus clusters running on Microsoft Azure, which store over a petabyte of data. While many different distributed databases and data processing systems were evaluated for VeniceDB, only Citus could address the specific combination of requirements associated with the petabyte-scale VeniceDB workload, including:

- Sub second response times (p95) for >6M queries per day
- Ingest ~10TB of new measure data per day
- Show new measure data in RQV within 20 minutes
- Nested subqueries with high cardinality group by
- Advanced secondary indexes (e.g. partial indexes, GiST indexes) to efficiently find reports along various dimensions
- Advanced data types (e.g. arrays, HyperLogLog) to implement sophisticated analytical algorithms in SQL
- Row count reduction through incremental aggregation
- Atomic updates across nodes to cleanse bad data

In the Citus clusters, raw data is stored in the measures table, which is distributed by device ID and partitioned by time on disk using the built-in partitioning capability in PostgreSQL. The COPY command is used to parallelize the ingestion of incoming JSON data into the distributed table. Distributed INSERT..SELECT commands are used to perform device-level pre-aggregation of incoming data into several reports tables with various indexes. The reports tables are also distributed on device ID and co-located with the measures table such that Citus can fully parallelize the INSERT..SELECT.

Many of the queries from the RQV dashboard are of the form:

```
SELECT ..., avg(device_avg)
FROM (
  SELECT deviceid, ..., avg(metric) as device_avg
  FROM reports WHERE ...
  GROUP BY deviceid, <time period> , <other dimensions>
) AS subq
GROUP BY <time period>, <other dimensions>;
```

These queries filter by several dimensions (e.g. measure, time range, Windows build) to find a substantial subset of the data. The nested subquery first aggregates reports by device ID, which is needed to weigh overall averages by device rather than by the number of reports. There can be tens of millions of devices per query, which makes the GROUP BY deviceid challenging to compute efficiently. Since the subquery groups by the distribution column, the logical pushdown planner in Citus recognizes that it can push down the full subquery to all worker nodes to parallelize it. The

worker nodes then use index-only scans to read the data in device ID order and minimize the disk I/O and memory footprint of the GROUP BY. Finally, Citus distributes the outer aggregation step by calculating partial aggregates on the worker nodes and merging the partial aggregates on the coordinator to produce the final result.

At each step, VeniceDB uses a combination of advanced PostgreSQL and Citus capabilities to achieve maximum efficiency and scale on a single system.

6 RELATED WORK

Citus has architectural similarities with various other distributed database systems, but most systems focus only on a single workload pattern. In addition, Citus is unique in that it is a distributed RDBMS implemented as an extension of an existing RDBMS, which gives many benefits in terms of robustness, versatility, and compatibility with the open source ecosystem around PostgreSQL.

Vitess [12] is a sharding solution for MySQL. Like Citus, Vitess scales out an existing open source relational database. Unlike Citus, it is not an extension and therefore must be deployed separately from the database servers and requires additional application changes. Vitess is primarily optimized for multi-tenant and high performance CRUD workloads and has built-in connection pooling for scaling the number of connections. It has limited support for queries and transactions across shards, which makes it less applicable in other workload patterns.

Greenplum [5] and Redshift [21] are PostgreSQL-based data warehouses that are hence optimized for handling complex SQL queries that analyze large amounts of data with low concurrency. As a result, both systems today provide better per-core performance than Citus for long analytical queries. Greenplum and Redshift also use columnar storage for fast scans and implement joins by shuffling data over the network. Citus supports those primitives as well, but the Citus implementation is not as well-optimized yet. On the other hand, Citus can handle a mixture of transactional and analytical workloads, and can take advantage of the latest PostgreSQL features and extensions.

Aurora [27] can scale out the storage for a single PostgreSQL server for demanding OLTP workloads and fast recovery. Citus has a shared nothing architecture, which means storage scale out and fast recovery is achieved by dividing data across many database servers. The downside of a shared-nothing architecture is that the application needs to make additional data modelling decisions (choosing distributed tables), so it is not a drop-in replacement for applications built on a single server RDBMS. The advantages of a shared-nothing architecture over shared storage are the ability to combine the compute power of all servers and use advanced query parallelism. Also, Citus can be deployed in any environment.

Spanner [17], CockroachDB [25] and Yugabyte [13] have been developed with a focus on serializability for multi-node transactions. CockroachDB and Yugabyte support the PostgreSQL protocol as well, though significant functional limitations compared to PostgreSQL remain. A notable architectural difference between these systems and Citus is that they provide distributed snapshot isolation and use wound-wait rather than deadlock detection. In sections 3.7.4 and 3.7.3 we discussed the downsides of these techniques in the context of PostgreSQL compatibility and why we did not use them for Citus. One of the benefits of distributed snapshot isolation is that it avoids data modelling constraints. Citus users need to use co-location and reference tables to scope transactions to a single node in order to get full ACID guarantees. On the other hand, these techniques also enable efficient joins and foreign keys and we therefore find them to be essential for scaling complex relational database workloads.

TimescaleDB [11] is a PostgreSQL extension that optimizes PostgreSQL for time series data. It uses similar hooks as Citus to introduce the concept of a hypertable, which is automatically partitioned by time. Partitioning tables by time is useful for limiting index sizes to maintain high write performance for time series workloads, and for partition pruning which speeds up queries by time range. Citus and TimescaleDB are currently incompatible due to conflicting usages of PostgreSQL hooks, but Citus does work with pg_partman [8] which is a simpler time partitioning extension. Many real-time analytics applications that use Citus also use pg_partman on top of distributed tables, in which case the individual shards are locally partitioned to get both the benefits of distributed tables and time partitioning.

7 CONCLUSIONS AND FUTURE WORK

Citus is a distributed database engine for PostgreSQL that addresses the need for scalability in the PostgreSQL ecosystem. As an extension, Citus maintains long-term compatibility with the PostgreSQL project, including new features and tools. Rather than focusing on a particular workload, we designed Citus as a multi-purpose database system that can handle a broad variety of PostgreSQL workloads that need to scale beyond a single server. That way, users get the simplicity and flexibility of using a widely adopted, open source relational database system, at scale.

Much of our future work is around implementing support for any remaining PostgreSQL features that are not fully supported on distributed tables. These include non-co-located correlated subqueries, recursive CTEs, and logical replication between different table types. Increasingly, we are also seeing users with hybrid data models that keep small tables on a single server and then distribute only large tables. Automated data model optimization for these scenarios is another important area of future work. Finally, Citus is increasingly being used in more specialized workload patterns such as MobilityDB [14] and Kyrix-S [26]. There are many potential distributed query optimizations that can be implemented specifically for those workloads. We will explore making Citus itself extensible to iterate on those optimizations faster.

ACKNOWLEDGMENTS

Citus is the result of a collaborative effort that spans close to a decade. We would especially like to thank Andres Freund, Hadi Moshayedi, Jason Petersen, Metin Döşlü, Önder Kalacı, Murat Tuncer, Samay Sharma, and Utku Azman for their vast and numerous contributions over many years. We would also like to thank our reviewers for their invaluable comments in writing this paper, and Min Wei for his input on the VeniceDB section. Finally, we would like to thank the many users of Citus who provide the input and inspiration that continue to shape Citus into a practical distributed PostgreSQL solution.

REFERENCES

[1] 2010. *TPC Benchmark C: Standard Specification Revision 5.11.* Technical Report. Transaction Processing Performance Council (TPC).

[2] 2018. *TPC Benchmark H: Standard Specification Revision 2.18.0.* Technical Report. Transaction Processing Performance Council (TPC)).

[3] 2020. Citus Open Source Repo. https://github.com/citusdata/citus

[4] 2020. GitHub Archive. https://www.gharchive.org/

[5] 2020. Greenplum Database. https://greenplum.org/

[6] 2020. HammerDB. https://www.hammerdb.com/

[7] 2020. Hyperscale (Citus) documentation on Citus metadata. https://docs.microsoft.com/en-us/azure/postgresql/reference-hyperscale-metadata

[8] 2020. PG Partitioning Manager. https://github.com/pgpartman/pg_partman

[9] 2020. pg_auto_failover: Postgres extension and service for automated failover and high-availability. https://github.com/citusdata/pg_auto_failover

[10] 2020. PgBouncer: lightweight connection pooler for PostgreSQL. https://www.pgbouncer.org/

[11] 2020. Timescale: PostgreSQL for time series. https://www.timescale.com/

[12] 2020. Vitess: A database clustering system for horizontal scaling of MySQL. https://vitess.io/

[13] 2020. YugabyteDB. https://www.yugabyte.com/

[14] Mohamed Bakli, Mahmoud Sakr, and Esteban Zimanyi. 2019. Distributed moving object data management in MobilityDB. In *Proceedings of the 8th ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data*. 1–10.

[15] Stefano Ceri and Giuseppe Pelagatti. 1983. Correctness of query execution strategies in distributed databases. *ACM Transactions on Database Systems (TODS)* 8, 4 (1983), 577–607.

[16] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[17] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 1–22.

[18] P. Corti, T.J. Kraft, S.V. Mather, and B. Park. 2014. *PostGIS Cookbook*. Packt Publishing. https://books.google.nl/books?id=zCaxAgAAQBAJ

[19] DB-engines. 2020. PostgreSQL System Properties. https://db-engines.com/en/system/PostgreSQL

[20] Andres Freund. 2020. Analyzing the Limits of Connection Scalability in Postgres. https://techcommunity.microsoft.com/t5/azure-database-for-postgresql/analyzing-the-limits-of-connection-scalability-in-postgres/ba-p/1757266

[21] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. 2015. Amazon redshift and the case for simpler data warehouses. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 1917–1923.

[22] Craig Kerstiens. 2018. Introducing Landlord: per tenant stats in Postgres with Citus. https://www.citusdata.com/blog/2018/07/31/introducing-landlord-per-tenant-stats/

[23] Eamonn Maguire, Lukas Heinrich, and Graeme Watt. 2017. HEPData: a repository for high energy physics data. In *J. Phys. Conf. Ser*, Vol. 898. 52.

[24] Michael Stonebraker and Lawrence A Rowe. 1986. The design of POSTGRES. *ACM Sigmod Record* 15, 2 (1986), 340–355.

[25] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. 2020. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1493–1509.

[26] Wenbo Tao, Xinli Hou, Adam Sah, Leilani Battle, Remco Chang, and Michael Stonebraker. 2020. Kyrix-S: Authoring Scalable Scatterplot Visualizations of Big Data. *arXiv preprint arXiv:2007.15904* (2020).

[27] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1041–1052.