Benchmarking TimescaleDB vs. Cassandra for Time-Series Data

A comparison of two databases in terms of scale, data model, insert performance, and query performance

TIMESCALE

Overview

Time-series data is emerging in more and more applications, including IoT, DevOps, Finance, Retail, Logistics, Oil and Gas, Manufacturing, Automotive, Aerospace, SaaS, even machine learning and AI. If you are investing in a time-series database, that likely means you already have a meaningful amount of time-series data piling up quickly and need a place to store and analyze it.

With its simple data partitioning and distribution architecture, highly tunable consistency settings, and strong cluster management tooling, Cassandra is legendary for its scalability. Developers often forego the expressiveness of SQL for the intoxicating power of being able to add write nodes to a Cassandra cluster with a single command. Moreover, Cassandra's ability to provide sorted wide rows (more on this later) makes it a compelling use case for a scalable time-series data store.

Therefore, Cassandra's ease of use, staying power, and potential to handle time-series data well through its sequentially sorted wide rows make it a natural comparison to TimescaleDB. In this paper, we dive deeper into using Cassandra vs. TimescaleDB for time-series workloads by comparing scaling patterns, data model, insert performance, and query performance of each database.

Benchmarking setup

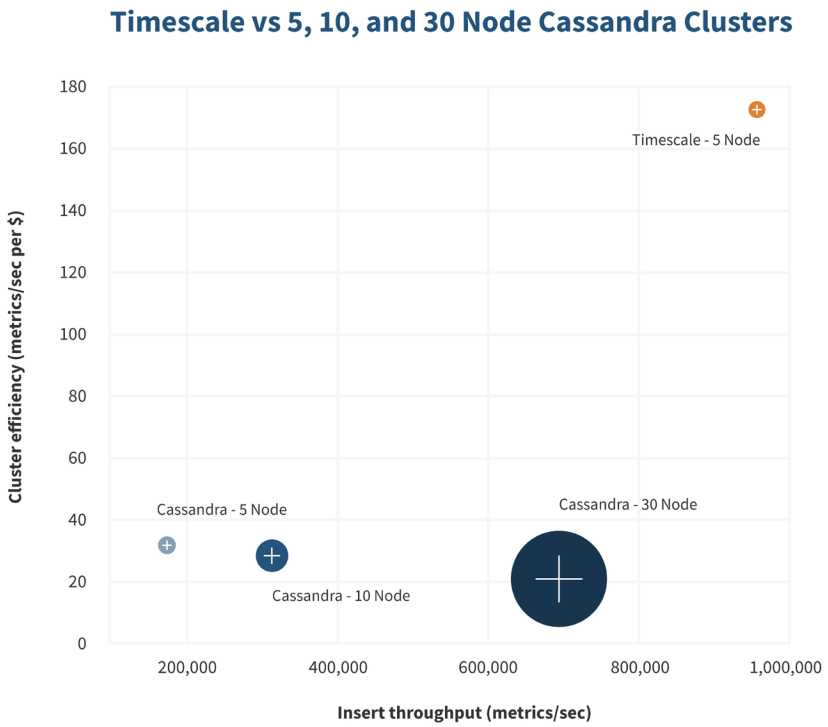
We start by comparing 5 node clusters for each database. Then, we benchmark a few different cluster configurations because the scaling properties of 5 node TimescaleDB and 5 node Cassandra are not perfectly analogous.

Here are the specs we used for these tests:

- 2 remote client machines, both on the same LAN as the databases
- Azure instances: Standard D8s v3 (8 vCPUs, 32 GB memory)
- 5 TimescaleDB nodes, 5/10/30 Cassandra nodes (as noted)
- 4 1-TB disks in a raid0 configuration (EXT4 filesystem)
- Dataset: 4,000 simulated devices generated 10 CPU metrics every 10 seconds for 3 full days (~100M reading intervals, ~1B metrics)
- For TimescaleDB, we set the chunk size to 12 hours, resulting in 6 total chunks (more here)

The results

Before moving forward, let's start with a visual preview of how 5 TimescaleDB nodes fare against various sizes of Cassandra clusters:



Now, let's look a bit further into how TimescaleDB and Cassandra achieve scalability.

Scaling patterns

Cassandra provides simple scale-out functionality through a combination of its data partitioning, virtual node abstraction, and internode gossip. Adding a node to the cluster redistributes the data in a manner transparent to the client and increases write throughput more-or-less linearly (actually somewhat sub-linear in our tests)[1]. Cassandra also provides tunable availability with its replication factor configuration, which determines the number of nodes that have a copy of a given piece of data.

PostgreSQL and TimescaleDB support scaling out reads by way of streaming replication. Each replica node can be used as a read node to increase read throughput. Although PostgreSQL does not natively provide scale-out write functionality, users can often get the additional throughput they need by using RAID disk arrays or leveraging the tablespace functionality provided by PostgreSQL.

In addition, unlike PostgreSQL, TimescaleDB allows users to (elastically) assign multiple tablespaces to a single hypertable if desired (e.g., multiple network-attached disks), creating the potential for massively scaling disk throughput on a single TimescaleDB instance. Moreover, as we'll see, the write performance a single TimescaleDB instance provides for time-series data is quite often more than sufficient for a production workload—and that's without some of the traditional NoSQL drawbacks that come with Cassandra.

Data model

Cassandra is a column family store. Data for a column family, which is roughly analogous to a table in relational databases, is stored as a set of unique keys. Each of these keys maps to a set of columns which each contain the values for a particular data entry. These key->column-set tuples are called "rows" (but should not be confused with rows in a relational database).

In Cassandra, data is partitioned across nodes based on the column family key (called the primary or partition key). Additionally, Cassandra allows for compound primary keys, where the first key in the key definition is the primary/partition key, and any additional keys are known as clustering keys. These clustering keys specify columns on which to sort the data for each row.

Let's take a look at how this plays out with the dataset we use for our benchmarks. We simulate a devops monitoring use case where 4,000 unique hosts report 10 CPU metrics every 10 seconds over the course of 3 days, resulting in a 100 million row dataset.

In our Cassandra model, this translates to us creating a column family like this:

```
CREATE TABLE measurements (
   series_id text,
   timestamp_ns bigint,
   value double,
PRIMARY KEY(series_id, timestamp_ns));
```

The primary key, series_id, is a combination of the host, day, and metric type in the format hostname#metric_type#day. This allows us to get around some of the query limitations of Cassandra discussed above, particularly the weak support for joins, indexes, and server side rollups. By encoding the host, metric type, and day into the primary key, we can quickly and easily access the subset of data we need and execute any further filtering, aggregation, and grouping more performantly on the client side.

We use timestamp_ns as our clustering key, which means that data for each row is ordered by timestamp as we insert it, providing optimal time range lookups. This is what a row of 3 values of the cpu_guest metric for a given host on a given day would look like.

Cassandra Time-series Data Model

host1#cpu_guest#2018-05-11	timestamp_ns: 1526069126290000000	timestamp_ns: 1526069136290000000	timestamp_ns: 1526069146290000000
	value: 60.8	value: 30.9	value: 39.8

Clustering Key

This is what we meant when we mentioned the wide row approach earlier. Each row contains multiple columns, which are themselves sets of key-value pairs. The number of columns for a given row grows as we insert more readings corresponding to that row's partition key. The columns are clustered by their timestamp, guaranteeing that each row will point to a sequentially sorted set of columns.

This ordered data is passed down to our custom client, which maintains a fairly involved client-side index to perform the filtering and aggregation that is not supported in a performant manner by Cassandra's secondary indexes. We maintain a data structure that essentially duplicates Cassandra's primary key->metrics mapping and performs filtering and aggregations as we add data from our Cassandra queries. The aggregations and rollups we do on the client side are very simple (min, max, avg, groupby, etc.), so the vast majority of the query time remains at the database level. (In other words, the client-side index works, but also takes a lot more work.)

Insert performance

Unlike TimescaleDB, Cassandra does not work well with large batch inserts. In fact, batching as a performance optimization is explicitly discouraged due to bottlenecks on the coordinator node if the transaction hits many partitions. Cassandra's default maximum batch size setting is very small at 5KB. Nonetheless, we found that a small amount of batching (batches of 100) actually did help significantly with insert throughput for our dataset, so we used a batch size of 100 for our benchmarks.

To give Cassandra a fair test against TimescaleDB, which allows for far larger batch sizes (we use 10,000 for our benchmarks), we ramped up the number of concurrent workers writing to Cassandra. While we used just 8 concurrent workers to maximize our write throughput on TimescaleDB, we used 1,800 concurrent workers (spread across multiple client machines) to max out our Cassandra throughput. We tested worker counts from 1 up to 1,800 before settling on 1,800 as the optimal number of workers for maximizing write throughput. Any number of workers above that caused unpredictable server side timeouts and negligible gains (in other words, the tradeoff of latency for throughput became unacceptable).

To avoid client-side bottlenecks (e.g., with data serialization, the client-side index, or network overhead), we used 2 client VMs, each using our Golang benchmarker with 900 goroutines writing concurrently. We attempted to get more throughput by spreading the client load across even more VMs, but we found no improvements beyond 2 boxes.

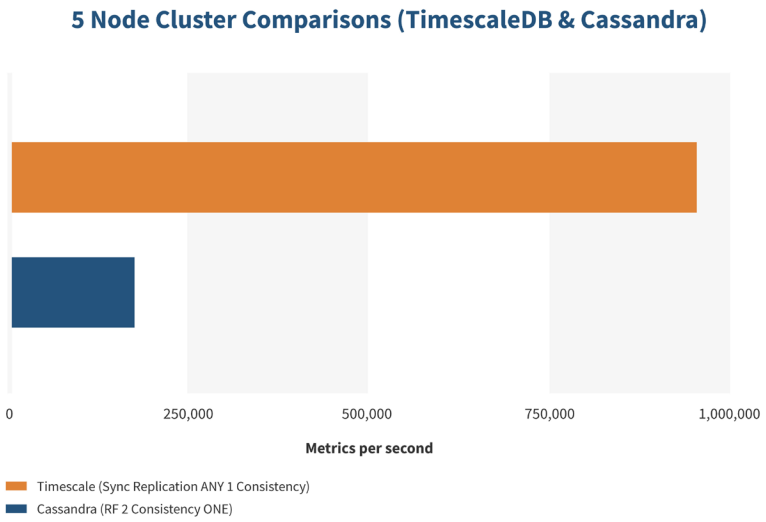
Since writes are sharded across nodes in Cassandra, its replication and consistency profile is a bit different than that of TimescaleDB. TimescaleDB writes all data to a single primary node which then replicates that data to any connected replicas through streaming replication. Cassandra, on the other hand, shards the writes across the cluster, so no single replica stores all the cluster's data. Instead, you define the replication factor for a given keyspace, which determines the number of nodes that will have a copy of each data item. You can further control the consistency of each write transaction on the client side by specifying how many nodes the client waits for the data to be written to. PostgreSQL and TimescaleDB similarly offer tunable consistency.

Given these significant differences, it's difficult to achieve a truly apples-to-apples comparison of a 5 node TimescaleDB cluster vs. a 5 node Cassandra cluster. We decided on comparing a TimescaleDB cluster with 1 primary and 4 read replicas, synchronous replication, and a consistency level of ANY 1 against a 5 node Cassandra cluster with Replication Factor set to 2 and a consistency level of ONE. In both cases, clients will wait on data to be copied to 1 replica. Eventually data will be copied to 2 nodes in the case of Cassandra, while data will be copied to all nodes in the case of TimescaleDB.

In theory, then, Cassandra should have an advantage in insert performance since writes will be sharded across multiple nodes. On the read side, TimescaleDB should have a small advantage for very hot sets of keys (given they may be more widely replicated), but the total read throughput of the two should be theoretically comparable.

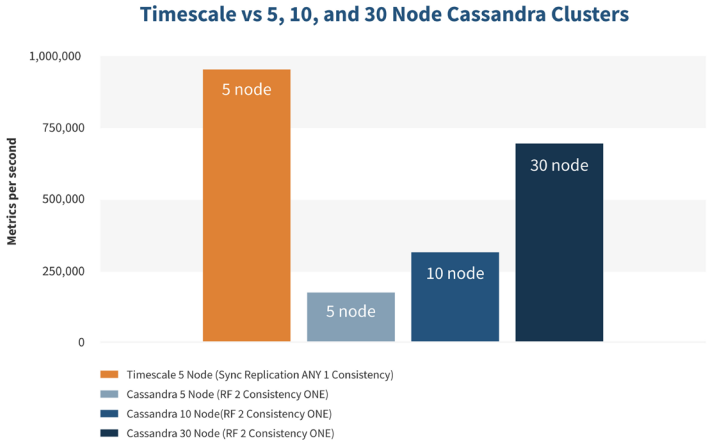
In practice, however, we find that TimescaleDB has an advantage over Cassandra in both reads and writes, and it's large.

Let's take a look at the insert rates for each cluster.



Despite Cassandra having the theoretical advantage of sharded writes, TimescaleDB exhibits 5.4x higher write performance than Cassandra. That actually understates the performance difference. Since TimescaleDB gets no write performance gains from adding extra nodes, we really only need a 3 node TimescaleDB cluster to achieve the same availability and write performance as our Cassandra cluster, making the real TimescaleDB performance multiplier closer to 7.6x.

This assumes that Cassandra scales perfectly linearly, which turns out to not quite be the case in our experience. We increased our Cassandra cluster to 10 then 30 nodes while keeping TimescaleDB at a cool 5 nodes.



Even a 30 node Cassandra cluster performs nearly 27% slower for inserts against a single TimescaleDB primary. With 3 TimescaleDB nodes—the maximum with TimescaleDB needed to provide the same availability as 30 node Cassandra with a Replication Factor of 2—we now see that Cassandra needs well over 10x (probably closer to 15x) the resources as TimescaleDB to achieve similar write rates.

For each node in these benchmarks, we paid for an Azure D8s v3 VM ($616.85/month) as well as 4 attached 1TB SSDs ($491.52/month). The minimum number of TimescaleDB nodes needed to achieve its write throughput and availability in the above chart is 3 ($3,325.11/month), while the minimum number of Cassandra nodes required to achieve its highest write throughput and availability in the above chart is 30 ($33,251.10/month). In other words, we paid $29,925.99 more for Cassandra to get 73% as much write throughput as TimescaleDB.

Put another way, TimescaleDB exhibits higher inserts at 10% of the cost of Cassandra.

Azure Costs for TimescaleDB (3 node) vs Cassandra (30 node)

	Cassandra (30 node)	TimescaleDB (3 node)	Ratio (Cassandra / TimescaleDB)
Azure Monthly Cost	$33,251.10	$3,325.11	10.00x
Write Throughput (metrics/s)	695,294.00	956,910.00	0.73x

TimescaleDB: Higher inserts at 1/10 the cost of Cassandra

Query performance

Cassandra is admittedly less celebrated than SQL databases for its strength with analytical queries, but we felt it was worth diving into a few types of queries that come up frequently with time-series datasets. For all queries, we used 4 concurrent clients per node per query. We measured both the mean query times and the total read throughput in queries per second.

Simple rollups: TimescaleDB competitive (up to 4x faster)

We'll start with the mean query time on a few single rollup (i.e., groupby) queries on time. We ran these queries in 1000 different permutations (i.e., random time ranges and hosts).

Mean Query Times for Simple Rollups with TimescaleDB and Cassandra (milliseconds per query)

	Cassandra	TimescaleDB	Ratio (Cassandra / TimescaleDB)
single-groupby(1,1)	18.70 ms	4.50 ms	4.15x
single-groupby(1,12)	30.80 ms	35.18 ms	0.88x
single-groupby(8,1)	46.30 ms	23.50 ms	1.97x

single-groupby(1,1) retrieves the max of 1 cpu metric per minute for 1 hour across 1 host
single-groupby(1,12) retrieves the max of 1 cpu metric per minute for 12 hours across 1 host
single-groupby(8,1) retrieves the max of 1 cpu metric per minute for 1 hours across 8 hosts

LOWER query times = BETTER performance

Cassandra holds its own here, but TimescaleDB is markedly better on 2 of the 3 simple rollup types and very competitive on the other. Even in simple time rollup queries where Cassandra's clustering keys should really shine, TimescaleDB's hypertables outperform.

Deducing the total read throughput of each database is fairly intuitive from the above chart, but let's take a look at the recorded QPS of each queryset just to make sure there are no surprises.

Read Throughput for Simple Rollups with TimescaleDB and Cassandra (queries per second)

	Cassandra	TimescaleDB	Ratio (Cassandra / TimescaleDB)
single-groupby(1,1)	970.87 q/s	3,333.33 q/s	0.29x
single-groupby(1,12)	625.00 q/s	543.48 q/s	1.15x
single-groupby(8,1)	434.78 q/s	833.33 q/s	0.52x

HIGHER queries per second = BETTER performance

When it comes to read throughput, TimescaleDB maintains its markedly better performance here.

Rollups on time and device: TimescaleDB 10x-47x faster

Bringing multiple rollups (across both time and device) into the mix starts to make both databases sweat, but TimescaleDB has a huge advantage over Cassandra, especially when it comes to rolling up multiple metrics. Given the lengthy mean read times here, we only ran 100 for each query type.

Mean Query Times for Rollups on Time and Device with Cassandra and TimescaleDB (milliseconds per query)

	Cassandra	TimescaleDB	Ratio (Cassandra / TimescaleDB)
double-groupby	280,320 ms	27,866 ms	10.01x
double-groupby-10	2,023,158 ms	43,045 ms	47.00x

'double-groupby' retrieves the AVG of one cpu metric per hour per device for 24 hours
'double-groupby-10' retrieves the AVG of 10 cpu metrics per hour per device for 24 hours

LOWER query times = BETTER performance

We see a similar story for read throughput.

Read Throughput for Rollups on Time and Device with Cassandra and TimescaleDB (queries per second)

	Cassandra	TimescaleDB	Ratio (Cassandra / TimescaleDB)
double-groupby	.124 q/s	.667 q/s	.186x
double-groupby-10	.009 q/s	.395 q/s	.023x

HIGHER queries per seconds = BETTER performance

Complex analytical queries: TimescaleDB 3100x-5800x faster

We also took a look at 2 slightly more complex queries that you commonly encounter in time-series analysis. The first ('lastpoint') is a query that retrieves the latest reading for every host in the dataset, even if you don't a priori know when it last communicated with the database[3]. The second ('groupby-orderby-limit') does a single rollup on time to get the MAX reading of a CPU metric on a per-minute basis for the last 5 intervals for which there are readings before a specified end time[4]. Each queryset was run 100 times.

Mean Read Times for Complex Queries with Cassandra and TimescaleDB (milliseconds per query)

	Cassandra	TimescaleDB	Ratio (Cassandra / TimescaleDB)
lastpoint	608,117.90 ms	103.53 ms	5873x
groupby-orderby-limit	239,257.60 ms	75.20 ms	3181x

'lastpoint' retrieves the latest reading for every host in the dataset
'groupby-orderby-limit' does a single rollup on time to get the MAX reading of a CPU metric on a per-minute basis for the last 5 intervals for which there are readings before a specified end time

And the read throughput.

Read Throughput for Rollups on Time and Device with Cassandra and TimescaleDB (queries per second)

	Cassandra	TimescaleDB	Ratio (Cassandra / TimescaleDB)
double-groupby	.124 q/s	.667 q/s	.186x
double-groupby-10	.009 q/s	.395 q/s	.023x

HIGHER queries per second = BETTER performance

For these queries, Cassandra is clearly not the right tool for the job. TimescaleDB can easily leverage hypertables to narrow the search space to a single chunk, using a per-chunk index on host and time to gather our data from there. Our multi-part primary key on Cassandra, on the other hand, provides no guarantee that all of the data in a given time range will even be on a single node. In practice, for queries like this that touch every host tag in the data set, we end up scanning most, if not all, of the nodes in a cluster and grouping on the client side.

Summary

As we see, 5 TimescaleDB nodes outperform a 30 node Cassandra cluster, with higher inserts, up to 5800x faster queries, 10% the cost, a much more flexible data model, and full SQL.

Cassandra's turnkey write scalability comes at a steep cost. For all but the simplest rollup queries, our benchmarks show TimescaleDB with a large advantage, with average query times anywhere from 10 to 5,873 times faster for common time-series queries. While Cassandra's clustered wide rows provide good performance for querying data for a single key, it quickly degrades for complex queries involving multiple rollups across many rows.

Additionally, while Cassandra makes it easy to add nodes to increase write throughput, it turns out you often just don't need to do that for TimescaleDB. With 10–15x the write throughput of Cassandra, a single TimescaleDB node with a couple of replicas for high availability is more than adequate for dealing with workloads that would require a 30+ node fleet of Cassandra instances to handle. However, Cassandra's scaling model does offer nearly limitless storage since adding more storage capacity is as simple as adding another node to the cluster. A single instance of TimescaleDB currently tops out around 50–100TB. If you need to store petabyte scale data and can't take advantage of retention policies or rollups, then massively clustered Cassandra might be the solution for you.

Cassandra was a pleasure to work with in terms of scaling out write throughput. But attaining that at the cost of per-node performance, the vibrant PostgreSQL ecosystem, and the expressiveness of full SQL simply does not seem worth it.

We try to be as open as we can about our data models, configurations, and methodologies so readers can raise any concerns they may have about our benchmarks and help us make them as accurate as possible. As a time-series database company we'll always be quite interested in evaluating the performance of other solutions.

Next steps

In this paper, we performed a detailed comparison of TimescaleDB and Cassandra. One of the worst mistakes a business can make is investing in a technology that will limit it in the future, let alone be the wrong fit today. That's why we encourage you to take a step back and analyze your stack before you find your database infrastructure crumbling to the ground.

If you are ready to get started with TimescaleDB, follow the instructions below. If you have any questions along the way, please contact us.

Where to go from here...

- Install TimescaleDB
- Join the Slack channel
- Reach out to hello@timescale.com and let us know how we can help

Additional resources

- Benchmark using the Time Series Benchmark Suite (TSBS)
- Follow our blog for database insights, engineering updates, tutorials, and more

Footnotes

1. We tested with up to 30 Cassandra nodes and saw performance that wasn't exactly linear but close enough for Cassandra's horizontal scalability to be compelling.
2. For example, a standard time range query like "SELECT timestamp_ns, value FROM %s WHERE series_id = host_12#usage_user#2016–01–02 AND timestamp_ns >= 1451692800000000000 AND timestamp_ns <= 1451692810000000000" would only work if timestamp_ns were a clustering key or a secondary index. Similarly, if we wanted to filter on value, we would need to define a clustering key or secondary index for it.
3. An example SQL query might help clarify this: "SELECT DISTINCT ON (t.hostname) * FROM tags t INNER JOIN LATERAL(SELECT * FROM cpu c WHERE c.tags_id = t.id ORDER BY time DESC LIMIT 1) AS b ON true ORDER BY t.hostname, b.time DESC".
4. And a bit more SQL for clarity: "SELECT date_trunc('minute', time) AS minute, max(usage_user) FROM cpu WHERE time < '2016–01–01 19:47:52.646325 -7:00' GROUP BY minute ORDER BY minute DESC LIMIT 5".