Benchmarking TimescaleDB vs. InfluxDB for Time-Series Data

How two time-series databases stack up in terms of data model, query language, reliability, performance, ecosystem and operational management

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. You may also recognize that the survival of your business will depend on the database you choose.

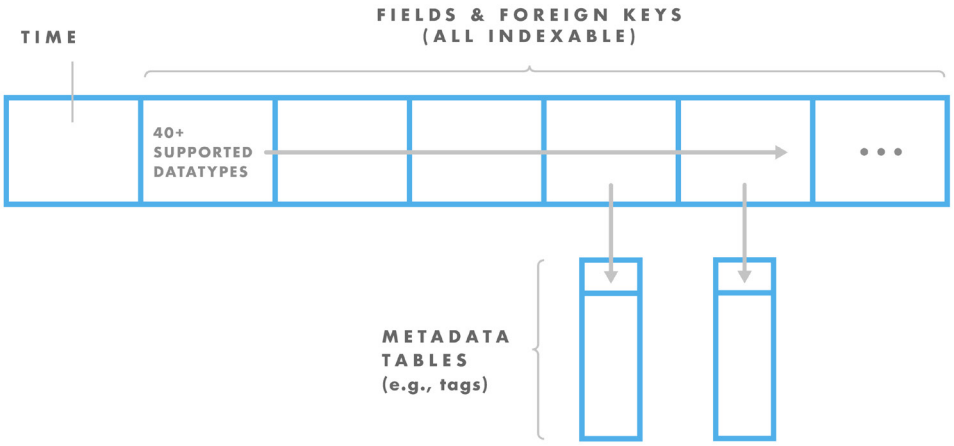
In this paper, we compare two leading time-series databases, TimescaleDB and InfluxDB, to help developers choose the time-series database best suited for their needs. Typically database comparisons focus on performance benchmarks. Yet performance is just a part of the overall picture; it doesn't matter how good benchmarks are if a database can't be used in production because it has an incompatible data model or query language, or if it lacks reliability. With that in mind, we begin by comparing TimescaleDB and InfluxDB across three qualitative dimensions including data model, query language, and reliability, before diving deeper with performance benchmarks. We then round out with a comparison across database ecosystem, and operational management.

Data model

Relational data model

TimescaleDB is a relational database, while InfluxDB is more of a custom, NoSQL, non-relational database. This means TimescaleDB relies on the relational data model, commonly found in PostgreSQL, MySQL, SQL Server, Oracle, etc. On the other hand, InfluxDB has developed its own custom data model, which, for the purpose of this comparison, is called the tagset data model.

With the relational model in TimescaleDB, each time-series measurement is recorded in its own row, with a time field followed by any number of other fields, which can be floats, ints, strings, booleans, arrays, JSON blobs, geospatial dimensions, date/time/timestamps, currencies, binary data, or even more complex data types. One can create indexes on any one field (standard indexes) or multiple fields (composite indexes), or on expressions like functions, or even limit an index to a subset of rows (partial index). Any of these fields can be used as a foreign key to secondary tables, which can then store additional metadata.



Advantages

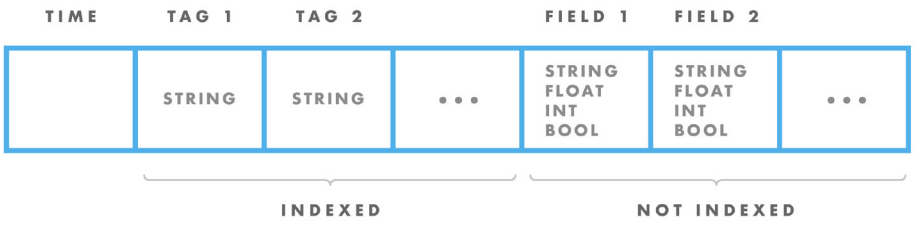
- Uses a flexible, narrow or wide table depending on how much data/metadata to record per reading
- Has many indexes to speed up queries or few indexes to reduce disk usage
- Denormalized metadata within the measurement row, or normalized metadata that lives in a separate table, can be updated at any time
- A rigid schema validates input types or a schemaless JSON blob to increase iteration speed
- Check constraints validate inputs, for example, checking for uniqueness or non-null values

Disadvantages

- To get started, one needs to generally choose a schema and explicitly decide whether or not have indexes

Tagset data model

With the InfluxDB tagset data model, each measurement has a timestamp, and an associated set of tags (tagset) and set of fields (fieldset). The fieldset represents the actual measurement reading values, while the tagset represents the metadata to describe the measurements. Field data types are limited to floats, ints, strings, and booleans, and cannot be changed without rewriting the data. Tagset values are indexed while fieldset values are not. Also, tagset values are always represented as strings, and cannot be updated.



Note: Abstract representation shown, not actual on disk organization.

Advantages

- If data naturally fits the tagset model, then it is easy to get started since as you don't have to worry about creating schemas or indexes

Disadvantages

- The model is quite rigid and limited, with no ability to create additional indexes, indexes on continuous fields (e.g., numerics), update metadata after the fact, enforce data validation, etc.
- The model may feel "schemaless", but there is actually an underlying schema that is auto-created from the input data, which may differ from the desired schema

Data model summary

If your data fits perfectly within the tagset data model, and you don't expect that to change in the future, then you should consider using InfluxDB as this model is easier to get started with. However, the relational model is more versatile and offers more functionality, flexibility, and control. This is especially important as your application evolves, and when planning your system, you should consider both its current and future needs.

Query language

From the beginning, TimescaleDB has firmly existed at the SQL end of the spectrum, fully embracing the language from day 1, and later further extending it to simplify time-series analysis. In contrast, InfluxDB began with a "SQL-like" query language (called InfluxQL), placing it in the middle of the spectrum, and has recently made a marked move towards the "custom" end with its new Flux query language.

At a high-level, here's how the two language syntaxes compare, using the computation of an exponential moving average as an example:

TimescaleDB (SQL):

```
SELECT time, (memUsed / procTotal /
1000000) as value
     FROM measurements
     WHERE time > now() - '1 hour';
```

InfluxDB (Flux):

```
// Memory used (in bytes)
memUsed = from(bucket: "telegraf/
autogen")
  |> range(start: -1h)
  |> filter(fn: (r) =>
    r._measurement == "mem" and
    r._field == "used"
   )

// Total processes running
procTotal = from(bucket: "telegraf/
autogen")
  |> range(start: -1h)
  |> filter(fn: (r) =>
    r._measurement == "processes" and
    r._field == "total"
   )

// Join memory used with total processes
and calculate
// the average memory (in MB) used for
running processes.
join(
    tables: {mem:memUsed,
proc:procTotal},
    on: ["_time", "_stop", "_start",
"host"]
  )
  |> map(fn: (r) => ({
    _time: r._time,
    _value: (r._value_mem / r._value_
proc) / 1000000
  })
)
```

Query language summary

While Flux may make some tasks easier, there are significant trade-offs to adopting a custom query language like it. New query languages introduce significant overhead, reduce readability, force a greater learning curve onto new users, and possess a scarcity of compatible tools. Additionally, they may not even be a viable option: rebuilding a system and re-educating a company to write and read a new query language is often not practically possible. Particularly if the company already is using SQL-compatible tools on top of the database for visualization.

Reliability

At its start, InfluxDB sought to completely write an entire database in Go. With its 0.9 release, they completely rewrote the backend storage engine (the earlier versions of Influx were going in the direction of a pluggable backend with LevelDB, RocksDB, or others). There are benefits from this approach, yet these design decisions have significant implications that affect reliability. First, InfluxDB has to implement the full suite of fault-tolerance mechanisms, including replication, high availability, and backup/restore. Second, InfluxDB is responsible for its on-disk reliability, e.g., to make sure all its data structures are both durable and resist data corruption across failures (and even failures during the recovery of failures).

Due to its architectural decisions, TimescaleDB instead relies on the 25+ years of hard, careful engineering work that the entire PostgreSQL community has done to build a rock-solid database that can support truly mission-critical applications.

Additionally, given TimescaleDB's design, it's able to leverage the full complement of tools that the PostgreSQL ecosystem offers and and all of these are available in open-source: streaming replication for high availability and read-only replicas, pg_dump and pg_recovery for full database snapshots, pg_basebackup and log shipping / streaming for incremental backups and arbitrary point-in-time recovery, WAL-E for continuous archiving to cloud storage, and robust COPY FROM and COPY TO tools for quickly importing/exporting data with a variety of formats.

InfluxDB, on the hand, had to build all these tools from scratch and it doesn't offer many of these capabilities. It initially offered replication and high availability in its open source, but subsequently pulled this capability out of open source and into its enterprise product. Its backup tools have the ability to perform a full snapshot and recover to this point-in-time, and only recently added some support for a manual form of incremental backups. Its ability to easily and safely export large volumes of data is also quite limited.

Databases need to provide strong on-disk reliability and durability, so that once a database has committed to storing a write, it is safely persisted to disk. In fact, for very large data volumes, the same argument even applies to indexing structures, which could otherwise take hours or days to recover; there's good reason that file systems have moved from painful fsck recovery to journaling mechanisms.

TimescaleDB does not change the lowest levels of PostgreSQL storage, nor interfere with the proper function of its write-ahead log. (The WAL ensures that as soon a write is accepted, it gets written to an on-disk log to ensure safety and durability, even before the data is written to its final location and all its indexes are safely updated.) These data structures are critical for ensuring consistency and atomicity; they prevent data from becoming lost or corrupted, and ensure safe recovery. Alternatively, InfluxDB had to design and implement all this functionality itself from scratch. This is a notoriously hard problem in databases that typically takes many years or even decades to get correct.

Reliability summary

These challenges and problems are not unique to InfluxDB, and every developer of a reliable, stateful service must grapple with them. Every database goes through a period when it sometimes loses data because it's really hard to get all the corner cases right. PostgreSQL went through this period in the 1990s, while InfluxDB still needs to figure out the kinks and has many years of dedicated engineering effort in store to catch up.

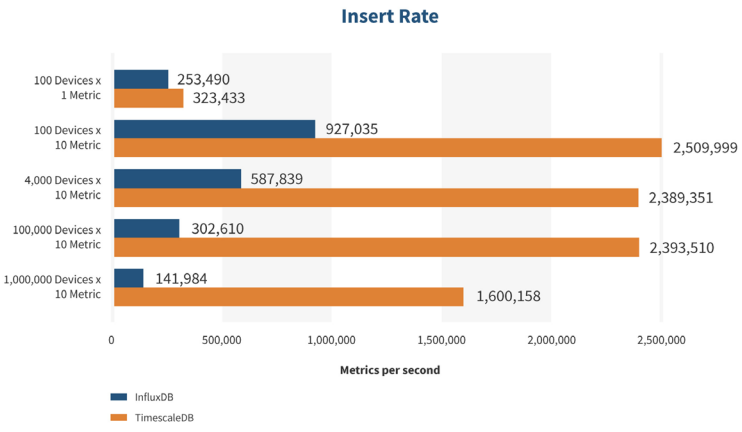
Performance

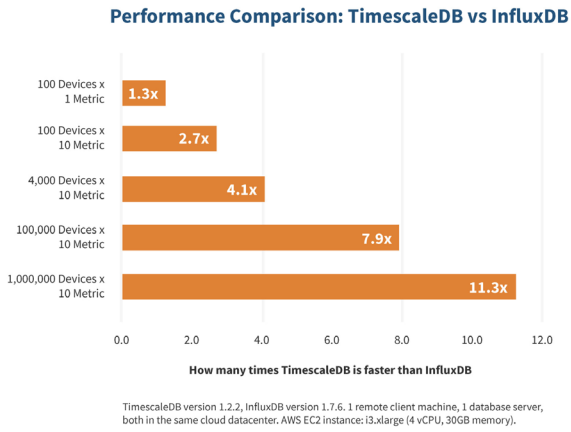
Below is a quantitative comparison of the two databases across a variety of insert and read workloads. Given how common high-cardinality datasets are within time-series, we first look at how TimescaleDB and InfluxDB handle this issue.

Insert performance

For insert performance, we used the following setup for each database:

- TimescaleDB version 1.2.2, InfluxDB version 1.7.6
- 1 remote client machine, 1 database server, both in the same cloud datacenter
- AWS EC2 instance: i3.xlarge (4 vCPU, 30GB memory)
- 4 1-TB disks in a raid0 configuration (EXT4 filesystem)
- Both databases were given all available memory
- Dataset: 100–1,000,000 simulated devices generated 1–10 CPU metrics every 10 seconds for ~100M reading intervals, ~1B metrics (1 month interval for 100 devices; 3 days for 4,000; 3 hours for 100,000; 3 minutes for 1,000,000), generated with the Time Series Benchmark Suite (TSBS).
- Schemas used for TimescaleDB (1) and InfluxDB (2)
- 10K batch size was used for both on inserts
- For TimescaleDB, we set the chunk size depending on the data volume, aiming for 10-15 chunks (more here)
- For InfluxDB, we enabled the TSI (time series index)





Insert performance summary

- For workloads with extremely low cardinality, the databases are comparable with TimescaleDB outperforming InfluxDB by 30%
- As cardinality increases, InfluxDB insert performance drops off dramatically faster than that with TimescaleDB
- For workloads with high cardinality, TimescaleDB outperforms InfluxDB by over 11x
- If your insert performance is far below these benchmarks (e.g., if it is 2,000 rows / second), then insert performance will not be your bottleneck

Note: These metrics are measured in terms of rows per second (in the case of InfluxDB, defined as a collection of metrics recorded at the same time). If you are collecting multiple metrics per row, then the total number of metrics per second can be much higher. For example, in our [4,000 devices x 10 metrics] test, you would multiply [rows per second] by [10], resulting in 1.44M metrics/sec for TimescaleDB and 0.56M metrics/sec for InfluxDB.

Read latency

For benchmarking read latency, we used the following setup for each database:

(The read latency setup varies from the previous one focused on inserts because this was conducted earlier in time.)

- TimescaleDB version 0.10.1, InfluxDB version 1.5.2
- 1 remote client machine, 1 database server, both in the same cloud datacenter
- Azure instance: Standard DS4 v2 (8 vCPU, 28 GB memory)
- 4 1-TB disks in a raid0 configuration (EXT4 filesystem)
- Both databases were given all available memory
- Dataset: 100–4,000 simulated devices generated 1–10 CPU metrics every 10 seconds for 3 full days (~100M reading intervals, ~1B metrics)
- 10K batch size was used for both on inserts
- For TimescaleDB, we set the chunk size to 12 hours, resulting in 6 total chunks (more here)
- For InfluxDB, we enabled the TSI (time series index)

On read (i.e., query) latency, the results are more complex. Unlike inserts which primarily vary on cardinality size (and perhaps also batch size), the universe of possible queries is essentially infinite, especially with a language as powerful as SQL. Often the best way to benchmark read latency is to do it with the actual queries you plan to execute. For this case, we use a broad set of queries to mimic the most common query patterns.

The results are below, using the same workloads we used for inserts. Latencies in this chart are all shown as milliseconds, with an additional column showing the relative performance of TimescaleDB compared to InfluxDB (highlighted in orange when TimescaleDB is faster, in blue when InfluxDB is faster).

QUERY PERFORMANCE
measured in milliseconds

	100 DEVICES x 1 METRIC			100 DEVICES x 10 METRICS			4,000 DEVICES x 10 METRICS		
SIMPLE ROLLUPS[1]	**InfluxDB**	**TimescaleDB**	**InfluxDB / TimescaleDB**	**InfluxDB**	**TimescaleDB**	**InfluxDB / TimescaleDB**	**InfluxDB**	**TimescaleDB**	**InfluxDB / TimescaleDB**
single-groupby-1-1-1	3.28	2.32	141.1%	3.10	2.22	139.6%	3.19	4.54	70.3%
single-groupby-1-1-12	13.59	9.19	147.9%	13.16	9.68	136.0%	12.88	43.85	29.4%
single-groupby-1-8-1	12.30	15.89	77.4%	6.93	16.32	42.5%	6.83	27.93	24.5%
single-groupby-5-1-1	N/A	N/A	N/A	6.78	2.78	243.9%	6.64	4.79	138.6%
single-groupby-5-1-12	N/A	N/A	N/A	45.60	14.22	320.7%	43.55	47.99	90.7%
single-groupby-5-8-1	N/A	N/A	N/A	24.26	19.29	125.8%	23.42	28.57	82.0%
DOUBLE ROLLUPS[2]									
double-groupby-1	133.76	241.02	55.5%	82.35	194.57	42.3%	3438.00	5938.40	57.9%
double-groupby-5	N/A	N/A	N/A	386.38	252.64	152.9%	16580.34	8061.28	205.7%
double-groupby-all	N/A	N/A	N/A	754.40	284.66	265.0%	33354.96	10568.49	315.6%
THRESHOLDS[3]									
high-cpu-1	13.14	6.71	195.8%	29.32	12.00	244.3%	31.25	46.44	67.3%
high-cpu-all	1040.82	223.55	465.6%	2525.88	693.47	364.2%	121561.86	26617.25	465.7%
COMPLEX QUERIES									
lastpoint	466.04	4.93	9453.1%	307.52	5.78	5320.4%	2113.05	275.51	767.0%
groupby-orderby-limit	8542.55	3.38	252738.2%	8151.25	3.44	236954.9%	46593.98	58.27	79962.2%

[1] Queries are in this form: single-groupby - [number of metrics] - [number of devices] - [number of hours]

[2] Queries are in this form: double-groupby - [number of metrics] (for all devices, for 12 hours)

[3] Queries are in this form: high-cpu - [number of devices] (for random window of time)

- InfluxDB better
- InfluxDB slightly better
- TimescaleDB slightly better
- TimescaleDB better
- TimescaleDB much better

Read latency performance summary

- For simple queries the results vary quite a bit; there are some where one database is clearly better than the other, while others depend on the cardinality of your dataset. The difference here is often in the range of single-digit to double-digit milliseconds
- When aggregating one metric, InfluxDB outperforms TimescaleDB. However, when aggregating multiple metrics, TimescaleDB outperforms InfluxDB.
- When selecting rows based on a threshold, TimescaleDB outperforms InfluxDB, except in the one case of computing threshold on one device with a high cardinality dataset
- For complex queries, TimescaleDB vastly outperforms InfluxDB, and supports a broader range of query types; the difference here is often in the range of seconds to tens of seconds.

Stability issues during benchmarking

It is worth noting that we had several operational issues benchmarking InfluxDB as our datasets grew, even with TSI enabled. In particular, as we experimented with higher cardinality data sets (100K+ tags), we ran into trouble with both inserts and queries on InfluxDB (but not TimescaleDB). While we were able to insert batches of 10K into InfluxDB at lower cardinalities, once we got·to 1M cardinality we would experience timeouts and errors with batch sizes that large. We had to cut our batches down to 1–5K and use client side code to deal with the backpressure incurred at higher cardinalities. We had to force our client code to sleep for up to 20 seconds after requests received errors writing the batches. With TimescaleDB, we were able to write large batches size at higher cardinality without issue.

Starting at 100K cardinality, we also experienced problems with some of our read queries on InfluxDB. Our InfluxDB HTTP connection would error out with a cryptic ‘End of File’ message. When we investigated the InfluxDB server we found out that InfluxDB had consumed all available memory to run the query and subsequently crashed with an Out of Memory error. Since PostgreSQL helpfully allows us to limit system memory usage with settings like shared_buffers and work_mem, this generally was not an issue for TimescaleDB even at higher cardinalities.

High-cardinality datasets

InfluxDB and the TSI

High-cardinality datasets are a significant weakness for InfluxDB. This is because of how the InfluxDB developers have architected their system, starting with their Time-series Index (TSI).

The InfluxDB TSI is a home-grown log-structured merge tree based system comprised of various data structures, including hashmaps and bitsets. This includes: an in-memory log ("LogFile") that gets periodically flushed to disk when it exceeds a threshold (5MB) and compacted to an on-disk memory-mapped index ("IndexFile"); a file ("SeriesFile") that contains a set of all series keys across the entire database. (Described here in their documentation.)

The performance of the TSI is limited by the complex interactions of all of these data structures.

The design decisions behind the TSI also leads to several other limitations with performance implications:

- Their total cardinality limit, according to the InfluxDB documentation, is around 30 million (although based on the graph above, InfluxDB starts to perform poorly well before that), or far below what is often required in time-series use cases like IoT.
- InfluxDB indexes tags but not fields, which means that queries that filter on fields can not perform better than full scans. For example, if one wanted to search for all rows where there was no free memory (e.g, something like, `SELECT * FROM sensor_data WHERE mem_free = 0`), one could not do better than a full linear scan (i.e., O(n) time) to identify the relevant data points.
- The set of columns included in the index is completely fixed and immutable. Changing what columns in your data are indexed (tagged) and what things are not requires a full rewrite of your data.
- InfluxDB is only able to index discrete, and not continuous, values due to its reliance on hashmaps. For example, to search all rows where temperature was greater than 90 degrees (e.g., something like SELECT * FROM sensor_data WHERE temperature > 90), one would again have to fully scan the entire dataset.
- Your cardinality on InfluxDB is affected by your cardinality across all time, even if some fields/values are no longer present in your dataset. This is because the SeriesFile stores all series keys across the entire dataset.

TimescaleDB and B-trees

In contrast, TimescaleDB is a relational database that relies on a proven data structure for indexing data: the B-tree. This decision leads to its ability to scale to high cardinalities.

First, TimescaleDB partitions your data by time, with one B-tree mapping time-segments to the appropriate partition ("chunk"). All of this partitioning happens behind the scenes and is hidden from the user, who is able to access a virtual table ("hypertable") that spans all of their data across all partitions.

Next, TimescaleDB allows for the creation of multiple indexes across your dataset (e.g., for equipment_id, sensor_id, firmware_version, site_id). These indexes are then created on every chunk, by default in the form of a B-tree. (One can also create indexes using any of the built-in PostgreSQL index types: Hash, GiST, SP-GiST, GIN, and BRIN.)

This approach has a few benefits for high-cardinality datasets:

- The simpler approach leads to a clearer understanding of how the database performs. As long as the indexes and data for the dataset we want to query fit inside memory, which is something that can be tuned, cardinality becomes a non-issue.
- In addition, since the secondary indexes are scoped at the chunk level, the indexes themselves only get as large as the cardinality of the dataset for that range of time.
- You have control over which columns to index, including the ability to create compound indexes over multiple columns. You can also add or delete indexes anytime you want, for example if your query workloads change. Unlike in InfluxDB, changing your indexing structure in TimescaleDB does not require you to rewrite the entire history of your data.
- You can create indexes on discrete and continuous fields, particularly because B-trees work well for a comparison using any of the following operators: `<, <=, =, >=, >, BETWEEN, IN, IS NULL, IS NOT NULL`. Our example queries from above (`SELECT * FROM sensor_data WHERE mem_free = 0` and `SELECT * FROM sensor_data WHERE temperature > 90`) will run in logarithmic, or O(log n), time.
- The other supported index types can come in handy in other scenarios, e.g., GIST indexes for "nearest neighbor" searches.

Ecosystem

Below is a non-exhaustive list of 1st party (e.g., the components of the InfluxData TICK stack) and 3rd party tools that connect with either database, to show the relative difference in the two database ecosystems. To reflect the popularity of the open source projects, we included the number of GitHub stars they had as of publication in parentheses, e.g., Apache Kafka (9k+). For many of the unofficial projects for InfluxDB, for example, the unofficial supporting project was often very early (very few stars) or inactive (no updates in months or years).

CATEGORY	THIRD-PARTY TOOL	InfluxDB Support	TimescaleDB Support
Visualization	Tableau	● (8)	●●●
	Grafana (23k)	●●●	●●●
	PowerBI	● (8)	●●●
	Mode	–	●●●
	Chronograf (706)	●●●	–
Message Bus	Apache Kafka (9k+)	●● (459)	●●●
Data Processing	Apache Spark (18k+)	● (19)	●●●
	Apache Flink (4k+)	●● [1] (71)	●●●
	Kapacitor (1468)	●●●	–
Monitoring	Prometheus (18k+)	●●●	●●●
	Zabbix	● (38)	●●●
	Telegraf (5k+)	●●●	●●●
Web framework + ORM	Rails (40k+)	● [2] (96)	●●●
ORM	SQLAlchemy (2k+)	● (18)	●●●
	Hibernate (3k+)	●● [3] (521)	●●●

[1] Project has less than 100 Github stars, but is part of official Apache project.
[2] Only for storing application metrics.
[3] No official Hibernate support, but has Java support.

● Unofficial support, project has < 100 Github stars
●● Unofficial support, project has > 100 Github stars
●●● Official support

Ecosystem summary

The database can only do so much, which is when one typically turns to the broader 3rd party ecosystem for additional capabilities. This is when the size and scope of the ecosystem make a large difference. TimescaleDB's approach of embracing SQL pays large dividends, as it allows TimescaleDB to speak with any tool that speaks SQL. In contrast, the non-SQL strategy chosen by InfluxDB isolates the database, and limits how InfluxDB can be used by its developers.

Operational management

Even if a database satisfies all the above needs, it still needs to work, and someone needs to operate it. Operational management requirements typically boil down to these categories: high availability, resource consumption (memory, disk, cpu), and general tooling.

High availability

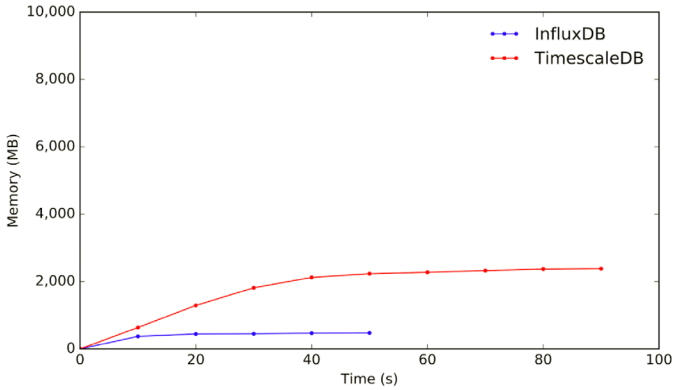
No matter how reliable the database, at some point your node will go down: hardware errors, disk errors, or some other unrecoverable issue. At that point, you will want to ensure you have a standby available for failover with no loss of data.

TimescaleDB supports high availability via PostgreSQL streaming replication. At one point, open source InfluxDB offered high availability via InfluxDB-relay, but it appears to be a dormant project (last update November 2016). Today InfluxDB HA is only offered by their enterprise version.

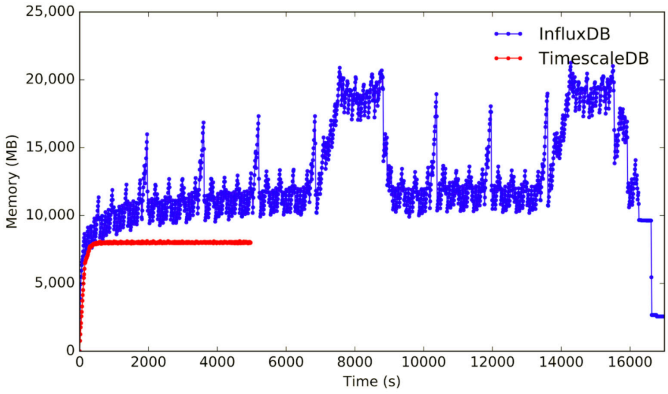
Resource consumption

For memory utilization, cardinality again plays a large role. Below are some graphs using the same workloads as before for measuring insert performance.

At low cardinality (100 devices sending one metric), InfluxDB requires less memory than TimescaleDB:



Both databases are inserting the same volume of data but take different amounts of time, which is why both line plots above and below don't end at the same time. However, as cardinality increases (100,000 devices sending 10 metrics), InfluxDB memory consumption far outpaces that of TimescaleDB (and with more volatility):



There was no way to limit total memory consumed by the InfluxDB TSI. At higher cardinalities, InfluxDB would run out of memory on inserts, which would lead to the database crashing and restarting. InfluxDB, like most databases that use a column-oriented approach, offers significantly better on-disk compression than PostgreSQL and TimescaleDB. With the dataset used for the performance benchmarks, here's how the two databases fared at the varying cardinalities:

- **100 devices x 1 metric x 30 days:** InfluxDB (12MB) vs. TimescaleDB (700MB) = 59x
- **100 devices x 10 metrics x 30 days:** InfluxDB (113MB) vs. TimescaleDB (1400MB) = 12x
- **4,000 devices x 10 metrics x 3 days:** InfluxDB (769MB) vs. TimescaleDB (5900MB) = 8x

Note: Disk size benchmarks were run using ZFS. Numbers do not include WAL size, as that is configurable by the user.

If minimizing disk storage is a primary requirement for your workload, then this is a big difference, and you may want to consider InfluxDB. However, as we saw earlier, depending on your workload InfluxDB may also require much more memory. Given that memory is typically 100x-1000x more expensive than disk, trading off high disk usage for lower memory may be worthwhile for certain workloads.

TimescaleDB also allows one to elastically scale the number of disks associated with a hypertable without any data migration, which is another option to offset the higher disk consumption, particularly in SAN or cloud contexts. Users have scaled a single TimescaleDB node to 10s of TB using this method. The other cost of InfluxDB's better on-disk compression is that it required the developers to rewrite the backend storage engine from scratch, which raises reliability challenges.

General tooling

When operating TimescaleDB, one inherits all of the battle-tested tools that exist in the PostgreSQL ecosystem: pg_dump and pg_restore for backup/restore, HA/failover tools like Patroni, load balancing tools for clustering reads like Pgpool, etc. Since TimescaleDB looks and feels like PostgreSQL, there are minimal operational learning curves. For operating InfluxDB, one is limited to the tools that the Influx team has built: backup, restore, internal monitoring, etc.

Next steps

In this paper, we performed a detailed comparison of TimescaleDB and InfluxDB. 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