Time Series Database Evaluation

Performance and Data Model Analysis of GridDB, QuestDB and TimescaleDB using the TSBS DevOps Benchmark.

Version 1.0

April 19, 2023

Fixstars Solutions, Inc.

Table Of Contents

Executive Summary

In this Time Series Benchmark Suite Evaluation of GridDB, QuestDB, and TimescaleDB, the three time series databases are compared by evaluating their ingestion or load performance as well as their query performance using five of the available queries in TSBS that highlight different use cases that could be used in a real world analysis of time series data.

Background

Toshiba's GridDB is a highly scalable, in-memory time series database with NoSQL and SQL interfaces. It uses a unique Key-Container data model allowing data from individual devices to be separated. The NoSQL API allows developers to write multiple rows or execute multiple queries from multiple containers at once.

QuestDB is a high performance open source SQL database for time series data that features columnar data storage and a variety of compatible APIs including InfluxDB and PostgreSQL line protocols.

TimescaleDB is a time-series SQL database providing fast analytics, scalability, with automated data management that uses PostgreSQL as its storage engine.

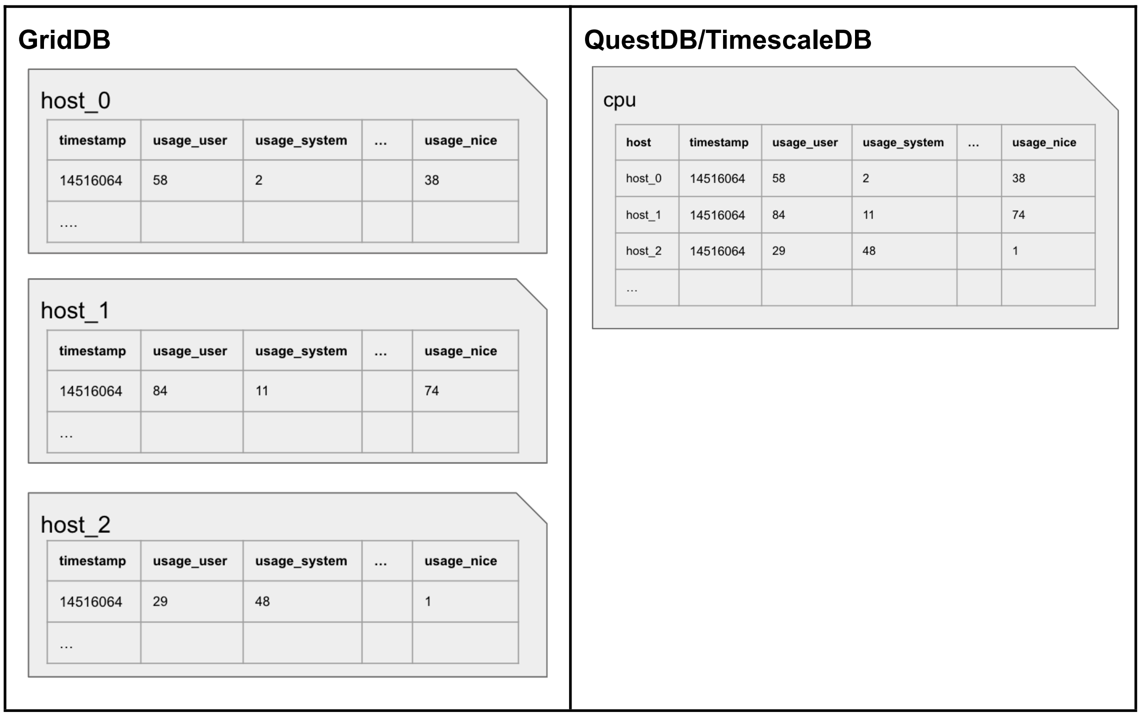
The Time Series Benchmark Suite (TSBS) is a set of applications to generate data and queries for a variety of use cases and time series databases. In this comparison, the cpu-only subset of TSBS's DevOps use case was used. The cpu-only data set is 10 CPU metrics such as usage_user, usage_system, etc for every host. To test how cardinality affects each database, 100, 1,000, 10,000, and 100,000 hosts were evaluated.

TSBS uses batches to load data which may not reflect real world use cases where data needs to be ingested in near real time. Batching of data is often done on per device level instead of grouping data from all devices together. Doing this simplifies the data producer and ingestion application optimizing the non-database components of the ingestion pipeline by reducing the number of messages sent.

Data Modelling

There is one significant difference between the databases being evaluated and that is that QuestDB and TimescaleDB use a tabular data model and GridDB uses a Key-Container data model. All data for QuestDB and TimescaleDB is inserted into one table while GridDB puts data for each host into separate tables (or containers in GridDB terminology).

The following diagrams show how GridDB and QuestDB/TimescaleDB represent the TSBS Devops dataset in their respective data models:



GridDB

host_0

timestamp	usage_user	usage_system	...	usage_nice
14516064	58	2		38
....				

host_1

timestamp	usage_user	usage_system	...	usage_nice
14516064	84	11		74
...				

host_2

timestamp	usage_user	usage_system	...	usage_nice
14516064	29	48		1
...				

QuestDB/TimescaleDB

cpu

host	timestamp	usage_user	usage_system	...	usage_nice
host_0	14516064	58	2		38
host_1	14516064	84	11		74
host_2	14516064	29	48		1
...					

This Key-Container data model has both advantages and disadvantages, it makes queries for a single container blazingly fast while queries that need to fetch data from multiple containers can be relatively slow if the query execution time is significantly faster than the time required to set up the query. Likewise, for high cardinality data sets, ingesting data can be inefficient in cases where only a small number of rows are inserted into each container.

QuestDB and TimescaleDB both use a tabular data model that makes creating relational queries spanning multiple devices simple to create and understand for anyone familiar with SQL. All three databases use a wide column format, where there are multiple data points per row for each timestamp. It is theoretically possible to use a narrow database schema for the TSBS DevOps data set but since the data is consistent, meaning that the metrics are always for the same point in time, the wide format is more efficient.

While it is possible to use GridDB with the same data model as QuestDB and TimescaleDB, it is generally best practice to not do so. The advantages and disadvantages of the

Key-Container data model will be discussed as each TSBS result is examined further. Likewise, it is possible to use separate tables for each device in a relational database like QuestDB or TimescaleDB, but it typically is not done as it minimizes some of the flexibility in creating queries.

Test Environment

The final evaluation was run on a single Standard D8s v3 (8 vcpus, 32 GiB memory) instance on Microsoft Azure using a RockyLinux 7.1 image. A 1 TB Premium SSD LRS disk was used to store the input data and database storage. The database and application (load or run_queries) were both run on the same instance.

Input scale was evaluated with 100, 1,000, 10,000, and 100,000 number of input servers. The 100,000 host evaluation used 1 month of data points per host 10 seconds apart while the other evaluations used 4 months of data points per host. A batch size or the number of records written to each database at one time of 1,000 records was used for ingestion as it was ideal for all three databases except the 10,000 and 10,000 host data sets with GridDB where a batch size of 10,000 proved to have higher performance. The batch size is set via a tsbs_load command line argument.

As the system had 8 cores, 8 workers (or writer threads) were used for QuestDB and TimescaleDB. GridDB used 10 workers to optimize the internal data structures and write patterns. Using 10 workers slightly degraded QuestDB and TimescaleDB performance. The number of workers is via a tsbs_load command line argument.

GridDB Community version 4.6 and its NoSQL API was used for the evaluation. A Java analog which read the same format input files was developed for GridDB ingestion while a TSBS Go run_queries program was used.

QuestDB version 6.5.3 was used with the original TSBS go applications that use Influx Line protocol for ingestion and REST based queries.

TimescaleDB version 2.9.2 was used along with PostgresSQL 12.13. Unmodified TSBS Go applications that use the PGX PostgresSQL driver were used. postgresql.conf was modified with the recommendations generated but the timescaledb-tune utility.

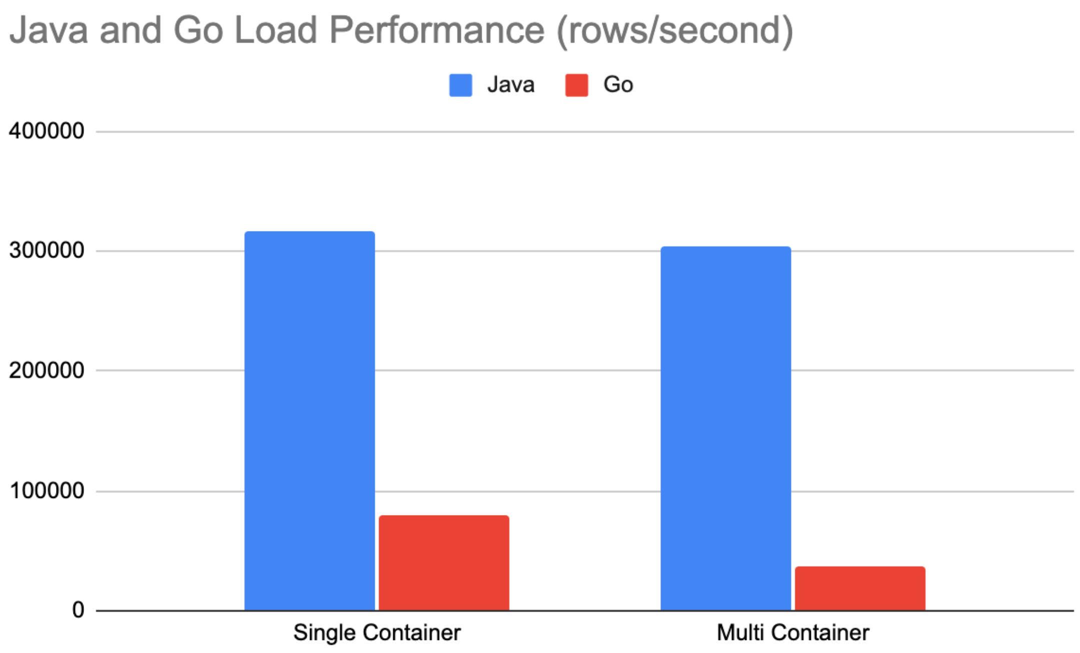
TSBS was fetched from their Git repository in late 2022 and compiled with go 1.19.2.

Benchmark Modifications

Java vs. Go

Early on when developing the GridDB implementation of TSBS, it was noted that load performance was not as expected and Fixstars suspected that GridDB's Go Driver could be the bottleneck. A Java equivalent implementation of the tsbs_load for GridDB was created and used instead.

Java is the native interface for GridDB while the other language bindings use SWIG and a series of abstractions to offer an API in multiple languages including Go, Python, NodeJS and others.



After profiling both the Java and Go versions of tsbs_load, their no-op version which scanned the input files and built data structures but did not perform writes had similar performance of approximately 500,000 rows per second but once actual GridDB writes were introduced, the Go performance fell to less than 100,000 rows pers second.

Data Ingestion Patterns

There are several options on ingesting time series data into the database and by default TSBS interleaves each record by hostname as so:

Hostname	Timestamp	Data
host_0	March 1 2016 10:00:00	…
host_1	March 1 2016 10:00:00	…
host_2	March 1 2016 10:00:00	…
…	March 1 2016 10:00:00	…
host_9	March 1 2016 10:00:00	…
host_0	March 1 2016 10:00:10	…
host_1	March 1 2016 10:00:10	…
…	March 1 2016 10:00:10	…

host_9	March 1 2016 10:00:10	…

Another time series ingestion design pattern would rearrange the data so there are multiple records for one host and then multiple records for another host. This mimics the scenario where each host would send a set of metrics that span a period of time rather than just a set of metrics for a specific timestamp to reduce communication overhead. The following example mimics data where each sequence is one minute or six rows long:

Hostname	Timestamp	Data
host_0	March 1 2016 10:00:00	…
host_0	March 1 2016 10:00:10	…
host_0	…	…
host_0	March 1 2016 10:00:50	…
host_1	March 1 2016 10:00:00	…
host_1	…	…
host_1	March 1 2016 10:00:50	…
…	…	…
host_9	March 1 2016 10:00:00	…
host_9	…	…
host_9	Marc 1 2016 10:00:50	…
host_0	March 1 2016 10:01:00	…
…	…	…

When using multiple containers with GridDB, ingestion is faster if data is not perfectly interleaved as the input size changes as it makes writes to each container more efficient.

With perfectly interleaved data and a batch size of 10,000, watch batch write has the following characteristics:

Number of Hosts	Containers Written	Rows Per Container
1,000	1,000	10
10,000	10,000	1
100,000	10,000	1

As the batch size is 10,000, for 1,000 hosts the load application can iterate through 10,000 rows adding 10 for each host to the array written to GridDB. For 10,000 hosts, each batch would include one row for every host. With 100,000 hosts, the first batch would include one row for hosts 0 - 9,999, the second batch would include one row for hosts 10,000 - 19,999 and so on. The eleventh batch would start again from the beginning and include a row for hosts 0 - 9,999.

Continuing with batch size of 10,000 and de-interleaving the data so there is 10 records in sequence for every host, for each batch GridDB writes fewer containers with more rows in every container:

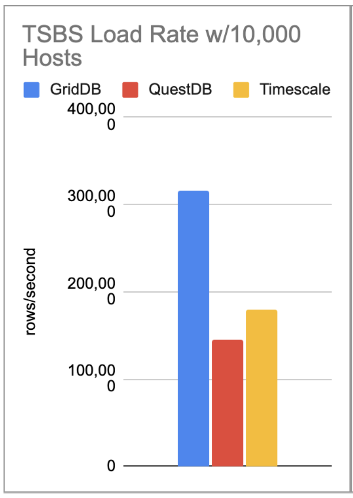
Number of Hosts	**Containers Written**	**Rows Per Container**
1,000	1,000	10
10,000	1,000	10
100,000	1,000	10

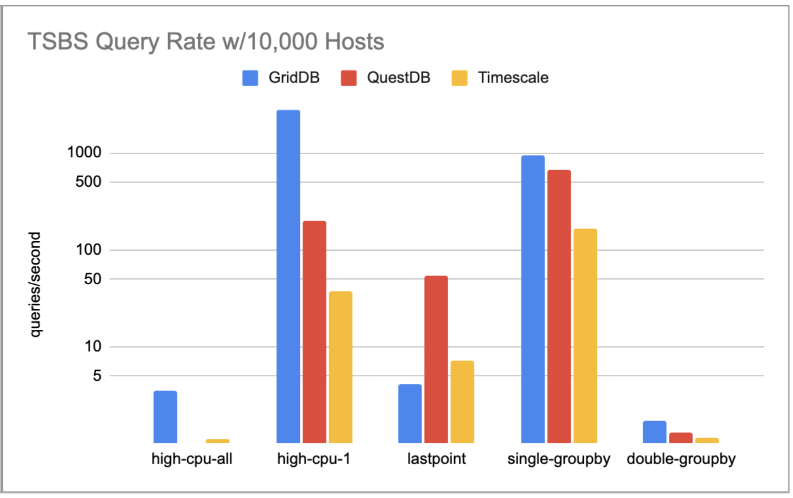
To explain further, the first batch would include ten rows for hosts 0 - 999, the second batch would include 1000 - 1999 and so on. With 10,000 hosts, the tenth batch would include hosts 0 - 999 again.

Optimizing these write patterns improved GridDB's load performance by approximately 10% but this de-interleaved input data was not used in the final evaluation of GridDB to show a fair comparison with QuestDB and TimescaleDB. When designing a data ingestion system for GridDB, the latter data pattern should be used to get the best GridDB load performance.

Results

In general, GridDB was the fastest of the three databases although it required more tuning. The relative ingestion and querying performance of the 10,000 host data set is summarized below.

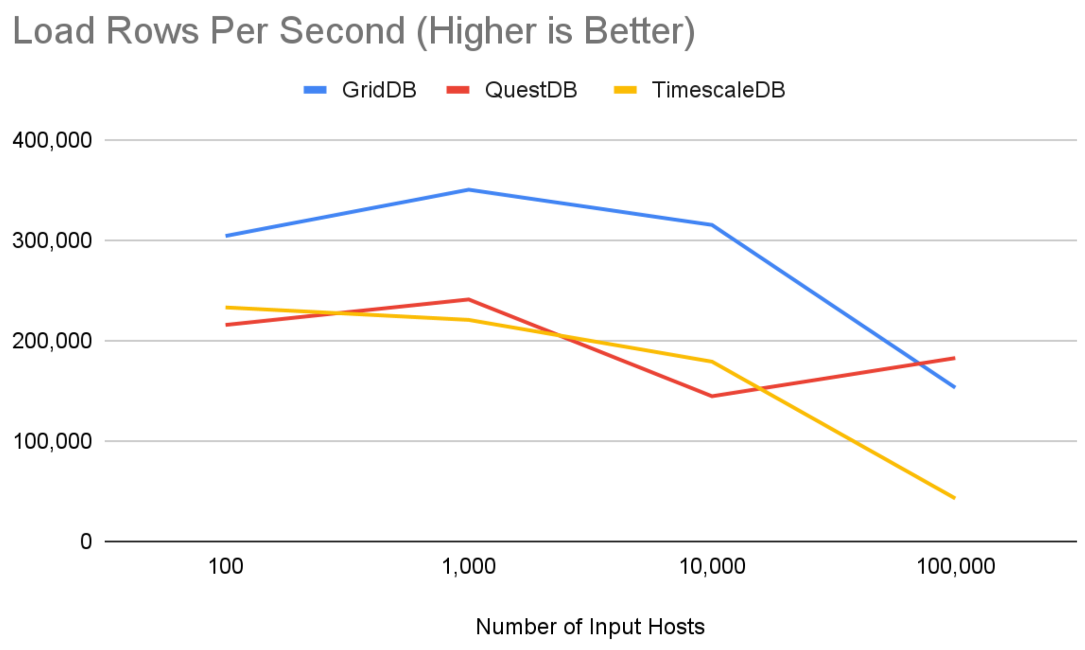




Load

For loading TSBS data, a batch size of 10,000 was used for GridDB while 1,000 was used for

QuestDB and Timescale which is what we found to be optimal. 8 workers were used for QuestDB and TimescaleDB to match the number of CPU cores while GridDB used 10 threads so the same thread would write the same container in the lower scale tests.



It should be noted that QuestDB's performance was much better using 4 workers and unlike GridDB, most of its CPU load was on the server and not the client. Its greater performance with 4 threads may not be typical in a real world application where there are large number of applications writing to it at the same time.

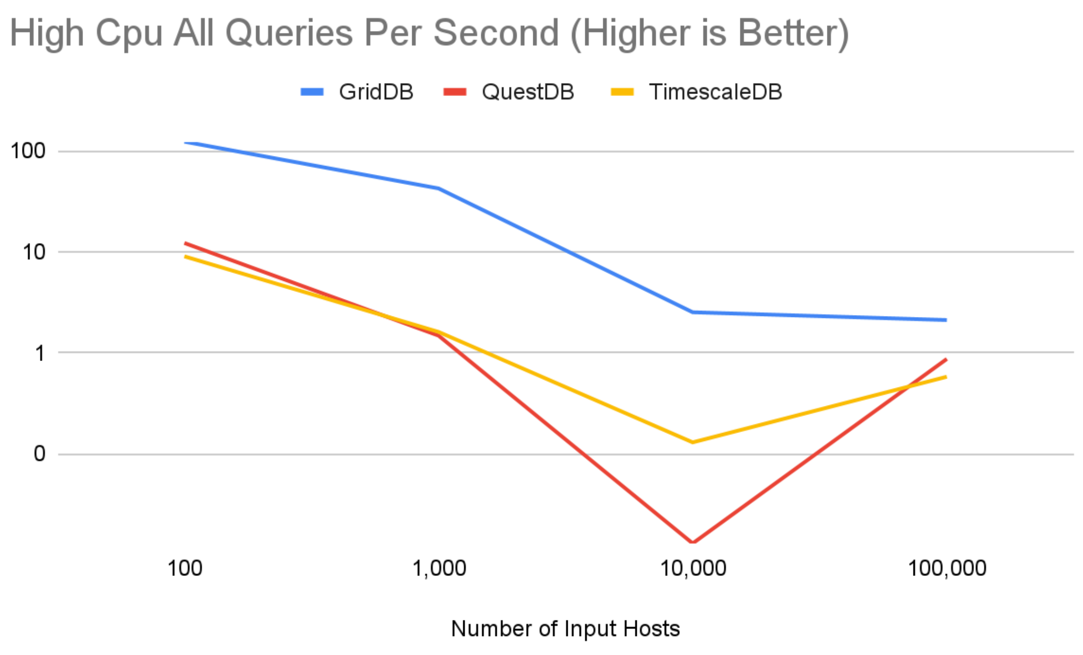
# of Hosts	GridDB	QuestDB	TimescaleDB	QuestDB (4 workers)
100	304,118	215,556	232,987	427,935
1,000	350,161	240,893	220,551	379,323
10,000	315,071	144,563	178,996	408,734
100,000	153,093	182,581	42,795	295,550

Loaded Rows Per Second.

High Cpu All Query

high-cpu-all is very similar to high-cpu-1, but instead of querying just a single host, it queries all hosts.

With GridDB, the same query is used, but it is executed on all host containers like lastpoint but since each single query itself is more complex than lastpoint, the query setup time for each container does not impact performance as it does with lastpoint.



# of Hosts	GridDB	QuestDB	TimescaleDB
100	122.9	12.2	9.0
1,000	42.4	1.5	1.6
10,000	2.5	0.0	0.1
100,000	2.1	0.9	0.6

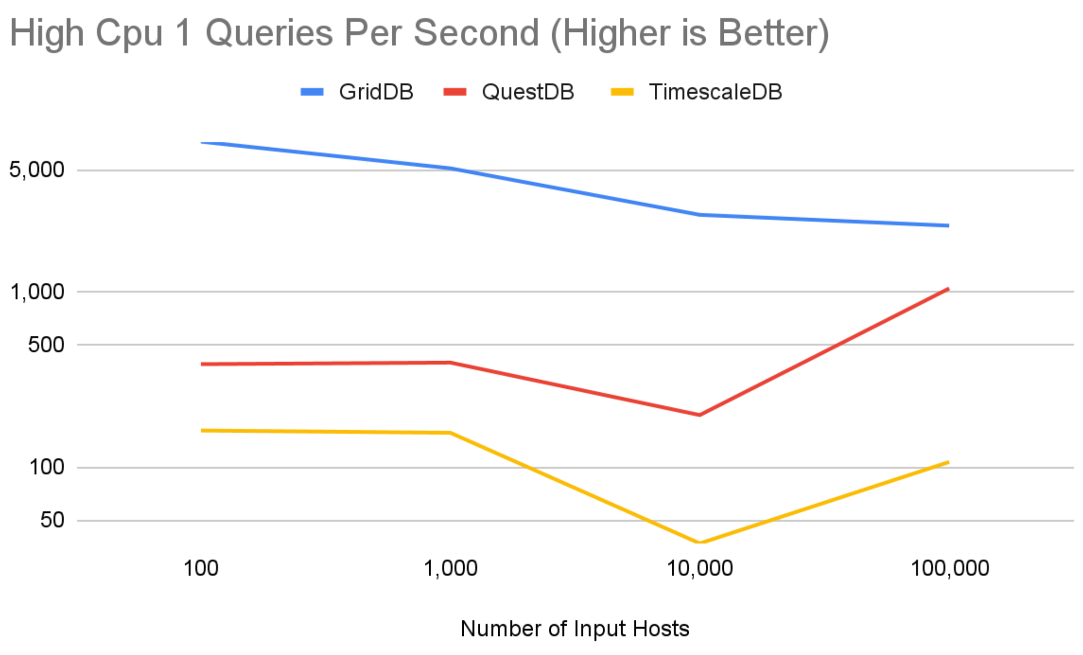
High Cpu All Queries Per Second.

High Cpu 1 Query

The high-cpu-1 query returns all entries for a single host where a metric is above a certain threshold. With QuestDB, it is implemented as `SELECT * FROM cpu WHERE usage_user > 90.0 AND hostname IN ('%s') AND timestamp >= '%s' AND timestamp < '%s'`` `.

In TimescaleDB, it is implemented as `SELECT * FROM cpu WHERE usage_user > 90.0 and time >= '%s' AND time < '%s' AND hostname IN (%s)`.

With GridDB, the query `select * where usage_user > 90 and timestamp > TIMESTAMP('%s') and timestamp < TIMESTAMP('%s')` is run on the specified hosts. The max-cpu-1 query demonstrates the Key-Container data model's biggest strength as the query is only run on a single container with much fewer rows in it compared to a single table.



# of Hosts	GridDB	QuestDB	TimescaleDB
100	7220.5	387.8	162.1
1,000	5086.8	395.8	157.3
10,000	2759.7	198.6	36.8
100,000	2393.2	1049.0	107.2

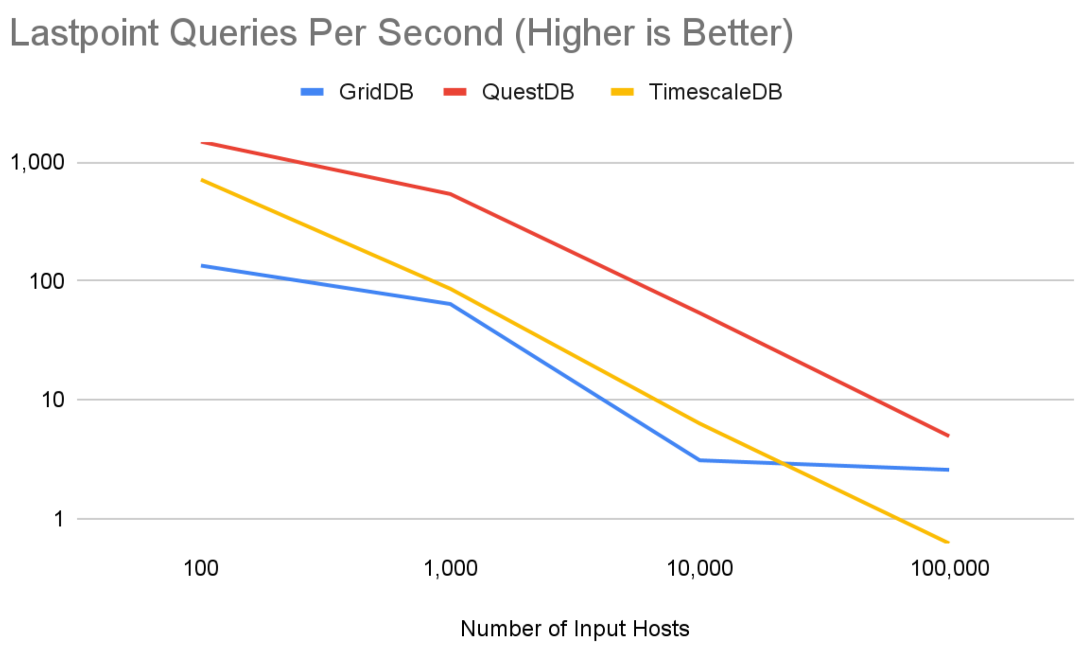
High Cpu 1 Queries Per Second.

Lastpoint Query

The lastpoint query fetches the most recent record for each host.

QuestDB used `SELECT * FROM cpu latest by hostname` while TimescaleDB used `SELECT DISTINCT ON (hostname) * FROM cpu ORDER BY hostname, time DESC.`

Lastpoint demonstrates GridDB's greatest weakness as it uses a very simple query that must be executed on every container. The `select time_prev(*, now())` TQL query was executed on every host container.



# of Hosts	GridDB	QuestDB	TimescaleDB
100	134.8	1488.0	713.1
1,000	64.1	539.3	86.1
10,000	3.1	53.8	6.3
100,000	2.6	5.0	0.6

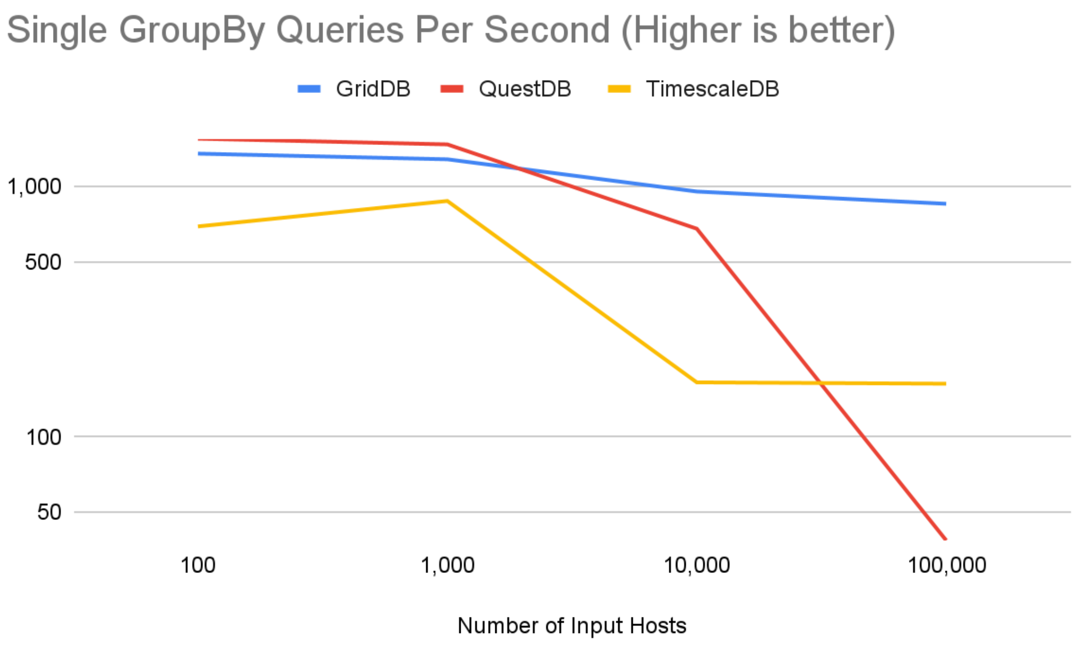
Lastpoint Queries Per Second.

Single GroupBy Query

The single groupby query performs a simple aggregate on one metric for 1 host, every minute for 1 hour.

QuestDB uses the following query, `SELECT timestamp, %s FROM cpu WHERE hostname IN ('%s') AND timestamp >= '%s' AND timestamp < '%s' SAMPLE BY 1m` while Timescale uses `SELECT %s AS minute, %s FROM cpu WHERE %s AND time >= '%s' AND time < '%s'  GROUP BY minute ORDER BY minute ASC`.

On GridDB, it is implemented with one query for each time interval and executed on one host container with the following query, `select max(%s) where timestamp < TO_TIMESTAMP_MS(%d)  and timestamp > TO_TIMESTAMP_MS(%d)`.



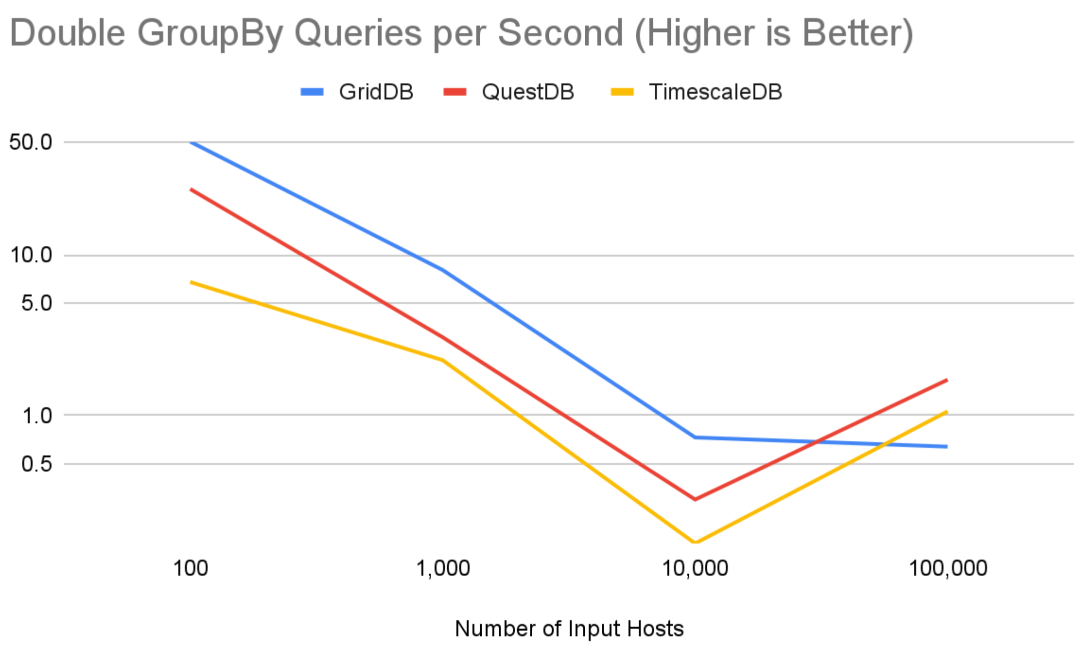
# of Hosts	GridDB	QuestDB	TimescaleDB
100	1351.3	1550.6	692.4
1,000	1282.7	1471.1	874.7
10,000	954.0	678.0	165.2
100,000	852.4	38.7	163.2

Single GroupBy Queries Per Second.

Double GroupBy Query

Like the single-groupby query, double-groupby performs a simple aggregate on one metric for a set of time periods over a longer duration. However, instead of querying just one host, all hosts are queried.

As with high-cpu-all, GridDB executes a set of n queries where n is the number of time periods on all host containers.



# of Hosts	GridDB	QuestDB	TimescaleDB
100	50.6	25.7	6.8
1,000	8.0	3.1	2.2
10,000	0.7	0.3	0.2
100,000	0.6	1.7	1.1

Double GroupBy Queries Per Second.

Conclusion

All three databases were able to ingest a 100,000 hosts worth of DevOps data in real time with 10 seconds between each data point making them all an effective ingestion option for this use case. QuestDB and TimescaleDB's slow query performance is concerning if real time analysis is required, but they would perform adequately in scenarios where analysis is performed as a nightly batch job.

While QuestDB shows tremendous ingestion performance under ideal conditions especially with a data set with high cardinality, it falls off significantly without the preferred number of workers. QuestDB is recommended for applications with extremely high cardinality where query performance is not as important.

GridDB also shows that an application must be carefully tuned to attain peak performance and shows outstanding query performance when application queries are able to fit its Key-Container data model.

Meanwhile, with TimescaleDB being built on top of PostgreSQL, its familiarity to many developers and administrators make it suitable for many applications despite its lower performance.