Evaluating the performance of NoSQL and Time Series databases using TSBS

A Project Report

Presented to

Dr. Chris Pollett

Department of Computer Science

San José State University

In Partial Fulfillment

Of the Requirements for the

Class CS 297

By

Aarsh Patel

May 2023

ABSTRACT

Time series are measurements or events that are tracked, monitored, down-sampled, and aggregated over time. The use of time series data has increased recently as time series data is essential in many domains such as finance, IoT, and scientific research, which requires efficient storage and retrieval of large amounts of data over time. Time series data is used for real-time monitoring, analytics, and forecasting. Many time-series databases are developed with a focus on storing such time-series data. Traditional NoSQL databases like MongoDB and Cassandra can also store time series data. This research aims to benchmark different time series and NoSQL databases using a benchmarking suite called Time Series Benchmarking Suite (TSBS). TSBS supports many time series and NoSQL databases. This project aims to evaluate the performance of four databases (3 time series and MongoDB) against various queries. Metrics like data storage footprint and read and write performance of databases will be the basis of the research question on how traditional NoSQL databases perform against time series databases to store time series data.

Keywords: Time Series Data, TSBS, Time Series Database Suite, NoSQL, MongoDB, TimescaleDB, InfluxDB, QuestDB

TABLE OF CONTENTS

I. INTRODUCTION 1

II. DELIVERABLE 1 3

III. DELIVERABLE 2 5

IV. DELIVERABLE 3 7

V. DELIVERABLE 4 12

VI. CONCLUSION 16

REFERENCES 18

SCREENSHOT OF EXPERIMENTS 18

I. INTRODUCTION

A time-indexed set of data points is called "time-series data." These data points, which track change over time, typically consist of successive measurements taken from the same source during a specified time interval [Paul,1]. In many fields, including finance, IoT, and scientific research, where adequate long-term data storage and retrieval are crucial, time series data is becoming increasingly significant. Time series data is becoming increasingly important, which necessitates the development of effective mechanisms for storage and retrieval that can deal with the massive volume of time series data. In recent years, numerous databases like InfluxDB, TimescaleDB, and QuestDB have been created to store time series data. NoSQL databases like MongoDB and Cassandra have also become viable choices for storing time series data.

The database for storing time series data will vary depending on the volume of data, the data structure, the types of queries, and the required performance. Benchmarking is a tried-and-true method for assessing how well various database systems perform in multiple scenarios. This research compares the performance of time series and NoSQL databases for storing time series data through a benchmarking suite called Time Series Database Suite (TSBS). InfluxDB, TimescaleDB, QuestDB, and MongoDB are the databases whose performance is assessed against various queries. Metrics like data storage footprint and read and write performance of databases will be the basis of the research question on how traditional NoSQL databases perform against time series databases when storing time series data.

The project is divided into four deliverables, each detailed in different sections of this report. Deliverable 1 involves researching time series data, its advantages, and time series databases and their properties. Deliverable 2 focuses on finding a benchmarking suite and finalizing the suite and the databases. Deliverable 3 involves studying the benchmarking suite, including the different steps to execute the benchmarking. Also, the study of the databases and their features is included in this deliverable. Deliverable 4 is the implementation of the benchmark with a small dataset with decided databases and queries, analyzing the results, and answering the research question. Finally, a conclusion is offered that summarizes my benchmarking results and responds to the project's research topic.

II. DELIVERABLE 1

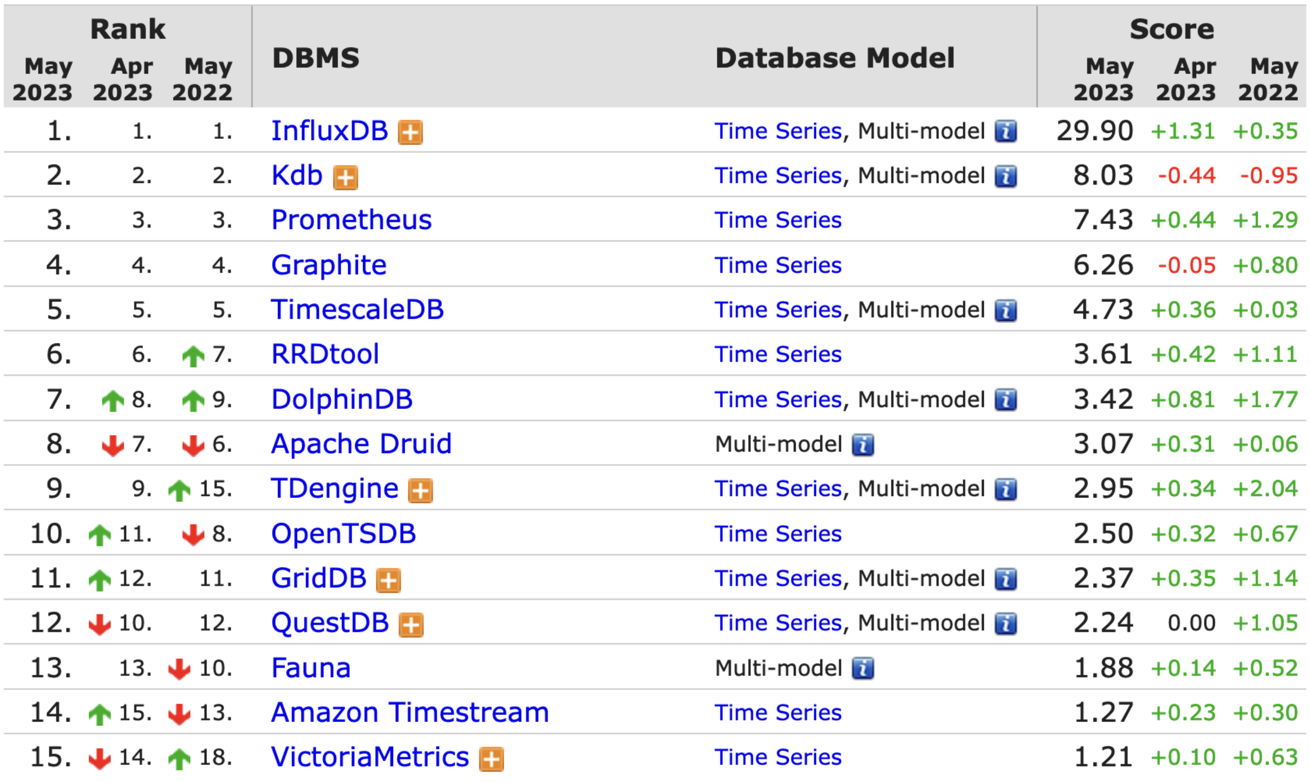
The goal of the first deliverable is to do a study on time-series data and databases to get an idea of their importance in the real world. This was achieved by researching time series data, databases, and their uses. As time series databases are used for benchmarking, it is crucial to understand them, their desired properties, and current database rankings. This deliverable was necessary for learning the basics of time series data and databases.

Time series data is critical, as discussed earlier. The main distinction between time series data and regular data is that time series data is primarily questioned across time. These days, the bulk of businesses produce an absurdly huge stream of measurements and events, which shows the importance of time series data and necessitates the need for databases specifically designed for storing time series data [1]. Time series data is used in domains like DevOps, IoT, and real-time analytics to enable efficient and effective monitoring and analysis. Time series data is used in DevOps monitoring [1] to keep track of the performance and overall health of systems and applications. A time-series database is used by developers and operations teams to track the system over time and spot trends or problems. This database collects and saves metrics like CPU utilization, memory consumption, network latency, and disk I/O. In IoT, the data generated by sensors and devices is stored in a time series database [1], enabling real-time analysis and monitoring of devices and their environments. Time series data are used in real-time analytics to analyze data streams and produce insights or forecasts.

With such high demand and use cases, time series data must be stored in specialized databases, as relational databases have scalability problems and cannot efficiently store data with time stamps. A time series database (TSDB) is tailored for time series data and explicitly designed to handle timestamped metrics, events, or measurements. A TSDB enables users to store, update, destroy, and organize [Naqvi, Yfantidou, and Zimanyi , 2] different time series data and is designed to handle large amounts of data.

Time-series databases have some required properties that make them suitable for storing and querying time-series data. The first property is that TSDB should be able to group data based on periods, making access to data and querying faster [2]. Another property is that the query language used in the database should make writing queries easier. As time-series data is continuously generated, the database should be designed to be highly available and scalable [2], as it should be able to handle large amounts of data with ease. Also, the database should be user-friendly and support many standard functions and operations common among time series databases, such as retention policies, aggregation, and range queries.

Many TSBS has been developed as the popularity of time series databases has increased recently. The database should be chosen depending on the requirements, ease of usage, and cost. A database ranking site called db-engines ranks databases based on popularity. Figure 1 shows the current rankings of time series databases. This will be helpful in Deliverable 2 when I have to finalize the benchmarking suite and the databases I will use for benchmarking.



Rank			DBMS	Database Model	Score		
May 2023	Apr 2023	May 2022			May 2023	Apr 2023	May 2022
1.	1.	1.	InfluxDB	Time Series, Multi-model	29.90	+1.31	+0.35
2.	2.	2.	Kdb	Time Series, Multi-model	8.03	-0.44	-0.95
3.	3.	3.	Prometheus	Time Series	7.43	+0.44	+1.29
4.	4.	4.	Graphite	Time Series	6.26	-0.05	+0.80
5.	5.	5.	TimescaleDB	Time Series, Multi-model	4.73	+0.36	+0.03
6.	6.	↑ 7.	RRDtool	Time Series	3.61	+0.42	+1.11
7.	↑ 8.	↑ 9.	DolphinDB	Time Series, Multi-model	3.42	+0.81	+1.77
8.	↓ 7.	↓ 6.	Apache Druid	Multi-model	3.07	+0.31	+0.06
9.	9.	↑ 15.	TDengine	Time Series, Multi-model	2.95	+0.34	+2.04
10.	↑ 11.	↓ 8.	OpenTSDB	Time Series	2.50	+0.32	+0.67
11.	↑ 12.	11.	GridDB	Time Series, Multi-model	2.37	+0.35	+1.14
12.	↓ 10.	12.	QuestDB	Time Series, Multi-model	2.24	0.00	+1.05
13.	13.	↓ 10.	Fauna	Multi-model	1.88	+0.14	+0.52
14.	↑ 15.	↓ 13.	Amazon Timestream	Time Series	1.27	+0.23	+0.30
15.	↓ 14.	↑ 18.	VictoriaMetrics	Time Series	1.21	+0.10	+0.63

Figure 1: TSBS rankings for May 2023

III. DELIVERABLE 2

Deliverable 2 involves finding a benchmarking suite for executing the benchmarking of different databases. Before finalizing the benchmarking suite, it is necessary to understand it, the databases it supports, and the metrics we can perform and evaluate, as these features determine if the suite is a good choice for benchmarking. My findings for this deliverable are reported in the following paragraph.

A benchmark suite is a group of tests or benchmarks created to evaluate a specific system's effectiveness, efficiency, and performance [Struckov, Yufa, Visheratin, and Nasonov, 3]. Several use cases from the real world are frequently included in the benchmark suite. Benchmark suites are used to examine and compare the effectiveness of various systems or components, enabling unbiased and quantitative evaluations. For databases, the benchmarking suite is used to compare the
performance of different databases' scalability, read/write performance, CPU, and memory usage.

The first benchmark I decided to use was the TS benchmark. The GitHub for the benchmarking is https://github.com/dbiir/TS Benchmark. This suite supported 4-time series databases. The steps to complete the test were generating the data, importing the data into the databases, configuring the parameters, and running the tests. The building of the benchmarking suite and downloading the databases locally were successful. However, generating the data and loading part got errors as a load.data file is missing from the original code. The error is shown in Figure 2. There is an issue raised on GitHub regarding that, but no update or solution to the bug is out yet, which resulted in finding a new benchmarking suite to perform the benchmarking.

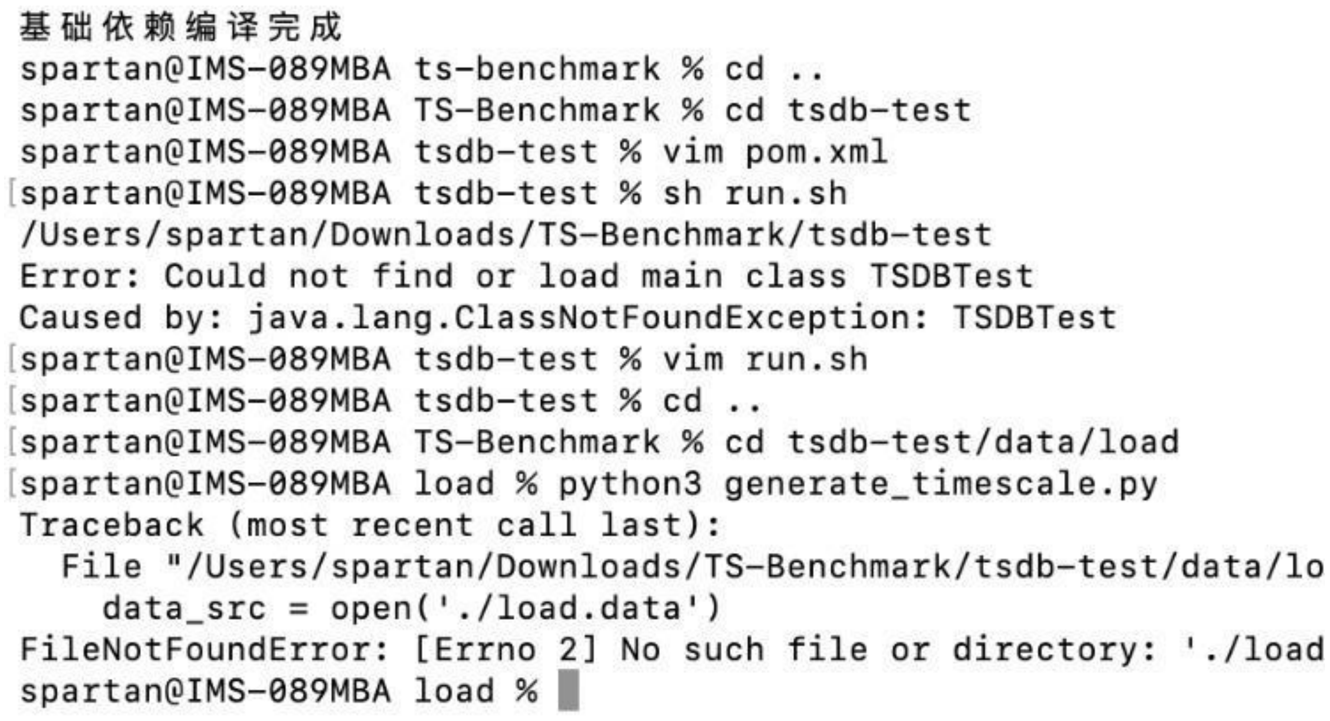

```
基础依赖编译完成
spartan@IMS-089MBA ts-benchmark % cd ..
spartan@IMS-089MBA TS-Benchmark % cd tsdb-test
spartan@IMS-089MBA tsdb-test % vim pom.xml
[spartan@IMS-089MBA tsdb-test % sh run.sh
/Users/spartan/Downloads/TS-Benchmark/tsdb-test
Error: Could not find or load main class TSDBTest
Caused by: java.lang.ClassNotFoundException: TSDBTest
[spartan@IMS-089MBA tsdb-test % vim run.sh
[spartan@IMS-089MBA tsdb-test % cd ..
[spartan@IMS-089MBA TS-Benchmark % cd tsdb-test/data/load
[spartan@IMS-089MBA load % python3 generate_timescale.py
Traceback (most recent call last):
  File "/Users/spartan/Downloads/TS-Benchmark/tsdb-test/data/lo
    data_src = open('./load.data')
FileNotFoundError: [Errno 2] No such file or directory: './load
spartan@IMS-089MBA load % 
```

Figure 2: TS-benchmark error

I found another benchmarking suite called Time Series Benchmarking Suite (TSBS). This suite supports many time series and NoSQL databases like MongoDB and Cassandra. The GitHub repository for the suite is https://github.com/timescale/tsbs.The suite is relatively new, and more updates are given periodically. I selected 4 databases for the benchmarking and 3 time series databases from the rankings provided by the DB-engines website [DB-Engines ranking ,4]. As shown in Figure 1, InfluxDB, TimescaleDB, and QuestDB are the top time-series databases supported by the benchmarking suite, so these three will serve as the time-series database, and MongoDB will be the NoSQL database. So, in this deliverable I used two benchmarking suite, TS-Benchmark and TSBS and finalized TSBS as the suite I will use for benchmarking.

IV. DELIVERABLE 3

This deliverable analyzes the features of NoSQL databases and the four databases selected for benchmarking. Also, studying the benchmarking code and exploring its features is included in this deliverable. It is essential to look at the features of NoSQL databases before learning MongoDB and other time-series databases.

NoSQL (Not Only SQL) databases are a database management system [Han, Le, and Du , 5] that provides a flexible and scalable approach for handling large volumes of unstructured data. NoSQL databases address conventional relational databases' scalability, performance, and data flexibility issues. They provide attributes including high availability, flexible schemas, and horizontal scalability. NoSQL has BASE properties as opposed to the ACID properties of relational databases. First, regarding ACID properties, ACID stands for atomicity, consistency, isolation, and durability. These properties ensure trustworthy transaction processing in a database system [Abramova and Bernardino ,6]. If any component of an ACID transaction fails, the entire transaction is rolled back, leaving the database in a consistent state. This ensures that database operations are carried out as an indivisible unit. For NoSQL databases, BASE stands for Basically Available, Soft State, and Eventually Consistent. BASE is a set of guidelines that some NoSQL databases [6] use to design and operate, especially those that place high availability and scalability over precise consistency. The immediate consistency guarantee offered by ACID transactions is replaced by eventual consistency in BASE properties, as in the BASE database, the modifications spread over time, and the system finally converges to a consistent state.

There is a theorem called the CAP theorem. CAP stands for Consistency, Availability, and Partition Tolerance (CAP), which says it is impossible to concurrently satisfy all three properties in a distributed database system [5]. Partition tolerance ensures that the system keeps working even with network partitions. Consistency describes how all nodes in the system have the same data simultaneously, and availability describes how every request receives a response.

The CAP Theorem states that a distributed system must decide between consistency and availability when a network partition occurs. The next part of this deliverable is about the distinguished properties of NOSQL that make them different from traditional databases and an overview of the types of NOSQL databases.

Some of the features of NoSQL databases are scalability, flexibility, and high performance. NoSQL databases can scale horizontally, handling large amounts of data [5] and high traffic loads by adding more hardware. NoSQL databases provide flexibility by offering a flexible schema that helps accommodate unstructured or changing data models without predefined schemas. NoSQL databases are optimized for read- and write-intensive workloads, thus providing faster data processing. The common types of NoSQL databases are column-based, document-based, key-value paired-based, and graph-based. Column-based databases are optimized for queries over large datasets and store columns of data together instead of rows [5]. Here, each row can have different columns. E.g., Cassandra, Amazon DynamoDB. In document-based databases, each key is paired in a document-like data structure, such as tree data, consisting of maps and scalar values [5]. E.g., MongoDB. In a key-value pair database, every item is stored as a key and its value [5]. E.g., CouchDB. Lastly, in the graph-based database, data can be represented as a graph [6], for example, social networks. E.g., Neo4J, Infinite Graph.

This section briefly overviews the features of the four databases that will be benchmarked. The first database is MongoDB. MongoDB is a NoSQL database developed in C++. It is a non-relational database that supports complex data types, like BJSON [5] data structures, to store complex data types. The data is stored as documents, which can be nested and contain arrays and other complex data types. MongoDB supports indexing and aggregation, enabling efficient data retrieval and processing. Also, replication and sharding [5] enable high availability, scalability, and fault tolerance. The second database is InfluxDB. InfluxDB is an open-source schema-less time series database written in the Go programming language. It supports an SQL-like query language [2] and supports sharding, which stores data in groups, thus optimizing data retrieval and querying. The third database is TimescaleDB, which is an open-source time series database written in C and extends PostgreSQL.

TimescaleDB supports standard SQL queries, and its table structures, called hyper table, provide easier access to data and efficient querying. TimescaleDB also supports data compression and replication, [Timescale docs ,7] which helps in efficient data storage. The last database is QuestDB, which is an open-source relational time-series database optimized for speed and low latency. It supports SQL querying and indexing and provides full ACID compliance [8] and transaction support. QuestDB also provides a columnar storage format and has a built-in time series extension that provides time-series functionality. I am using my Mac M1 for executing the benchmarking. I used Brew to install and configure the databases. Figure 3 shows the status of brew services, showing the successful installation of the databases and services running on Mac.

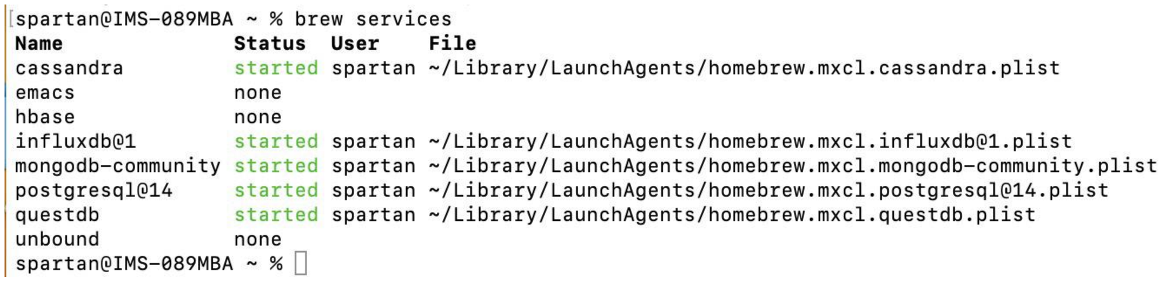

```
spartan@IMS-089MBA ~ % brew services
Name              Status  User    File
cassandra         started spartan ~/Library/LaunchAgents/homebrew.mxcl.cassandra.plist
emacs             none
hbase             none
influxdb@1        started spartan ~/Library/LaunchAgents/homebrew.mxcl.influxdb@1.plist
mongodb-community started spartan ~/Library/LaunchAgents/homebrew.mxcl.mongodb-community.plist
postgresql@14     started spartan ~/Library/LaunchAgents/homebrew.mxcl.postgresql@14.plist
questdb           started spartan ~/Library/LaunchAgents/homebrew.mxcl.questdb.plist
unbound           none
spartan@IMS-089MBA ~ %
```

Figure 3: Database running instance.

The next part of this deliverable is to look at the benchmarking suite and the different steps required for performing the benchmarking. The suite is a collection of GO programs and supports various use cases like CPU-only, DevOps, and IoT. There are different time series and NoSQL databases supported, as well as many queries to evaluate the write performances of the databases. The benchmarking steps are data generation, data loading, query generation, and query execution. The first step is data generation. The data can be generated according to the requirements, as we can specify parameters like the use case, PRNG seed, number of devices, time duration, and target database. The data is generated randomly, but it is deterministic if we supply the same PRNG value for each database. The following are the available parameters:

1. Use case (CPU-only, DevOps, or IoT)
2. PRNG seed for deterministic generation, E.g:123
3. The number of devices, E.g., 4000 (This will determine the size of the dataset)
4. A start time, E.g., 2023-04-01T00:00:00Z
5. An end time, E.g., 2023-04-02T00:00:00Z
6. The time between each reading, E.g:10s
7. Target Database, E.g., mongo (name determined for MongoDB)

The following is the data generation code of an example dataset I generated for MongoDB for a day with a scale = 100

```
tsbs_generate_data--use-case="cpu-only" --seed=123 --scale=100 --timestamp-
start="2023-04-01T00:00:00Z" --timestamp-end="2023-04-02T00:00:00Z" --log-
interval="10s" --format="mongo" /Users/spartan/tmp/mongo-data
```

The second step is loading the data into the database. The mongo-data file generated above can be loaded in MongoDB using a specific loader file provided in the benchmark. The following is an example code of MongoDB's data loading of the above generated file.

```
tsbs_load_mongo --file=/Users/spartan/tmp/mongo-data --document-per-event=true --
meta-field-index= “” --timeseries-collection=true --workers=10
```

The third step is query generation. This is done using tssb_generate_queries, and the same parameters should be used as the data loading as the generated queries should match the data. So, I used the same use case, PRNG, number of devices, and start date. The end date should be kept one second longer than the generated date. We should also define the number of queries and their type. So, for the same example of MongoDB, the code is as follows:

```
tsbs_generate_queries --use-case= “cpu-only” --seed=123 --scale=100 \
  --timestamp-start="2023-04-01T00:00:00Z" \ --timestamp-end="2023-04-02T00:00:01Z" \
  --queries=1000 --query-type="single-groupby-1-1-1" --format= “mongo” \ >
```

/Users/spartan/tmp/mongo-queries-breakdown-single-groupby-1-1-1

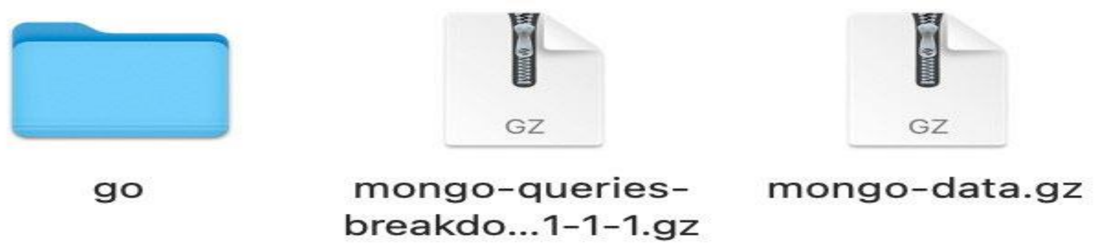


Figure 4: Data generation and query file.

Figure 4 shows the data and query generated file in my Mac. The last step is query execution. After the database is set up correctly, data and queries are generated, and the data is loaded into the target database, we can use the tsbs_run_queries program to successfully run the target queries with the generated data for a particular database. We can specify the number of workers needed by setting the worker parameter. For the same example of MongoDB, the query can be executed with the following code:

tsbs_run_queries_mongo --file=/Users/spartan/tmp/mongo-queries-breakdown-single-groupby-1-1-1 --workers=1

V. DELIVERABLE 4

Deliverable 4 involves the implementation of a benchmark with the four databases. The performance of the databases is evaluated by metrics like data storage footprint and read and write performance. For read performance, I ran four queries from the list of available queries. The four queries are as follows:

1. double-groupby-5: This query does multiple group-by by time and host_id. Returns the average of 5 metrics per host per day
2. cpu-max-all-8: This query finds the maximum value for all metrics for 1 hour for 8 hosts.
3. lastpoint: This query finds the latest reading for every device in the dataset.
4. groupby-orderby-limit: This query does a single rollup on time to get the MAX reading of a CPU metric per minute for the last 5 intervals for which there are readings before a specified end time that is randomly selected.

I decided to use a bigger dataset compared to the example dataset, as shown in Deliverable 3, with a scale of 1000 and a time duration of 3 days. I created the dataset and uploaded it to the databases, and then I calculated the read performance by developing and executing the four queries. The screenshots of the data loading and query implementation are included at the end of this report. After implementing MongoDB with the default Naive method of document-by-store, the query performance was worse than other time series databases. So, I used a better-recommended approach to store time series data in MongoDB, which is to aggregate the time series data into groups based on time. E.g., for each device, a document is created every hour. So, this contains a matrix with 60 times 60 (minutes and seconds) as the data is updated constantly. This data for an hour is stored as one document for a particular device. The document is updated accordingly when a reading is done, so there is no need to make a new document for every reading. This method resulted in much better performance. The results of the benchmarking are as follows:

Data Storage Footprint Results:

MongoDB: 23.72 GB

TimescaleDB: 5.62 GB

InfluxDB: 8.97 GB

QuestDB: 8.97 GB

Data Loading (Write Performance) Results:

MongoDB (Naive): 246 sec

MongoDB (Recommended): 123 sec

TimescaleDB: 59 sec

InfluxDB: 2.33 sec

QuestDB: 20.68 sec

Query Execution (Read Performance) Results:

	double-groupby-5	cpu-max-all-8	lastpoint	groupby-orderby-limit
MongoDB (Naive)	118	0.86	66.79	239
MongoDB (Recommended)	16.39	14.48	19.44	13.47
TimescaleDB	30.94	1.05	0.35	0.24
InfluxDB	27.84	0.44	11.60	83.66
QuestDB	7.97	N/A	0.32	0.16

The first metric is the data footprint. Analyzing the results shows that TimescaleDB wins as it uses a compression technique to compress the generated data. InfluxDB and QuestDB have the same data size because there is a similarity in their storage structures, as both use SQL tables for storing data, resulting in similar sizes. Both MongoDB methods generated 24 GB of data because keeping each piece of data in a different document and chunking will result in the same data size.

Analyzing the results of data loading, i.e., write performance, InfluxDB is the fastest when loading data as it uses a storage engine called the Time-Structured Merge Tree (TSM) [2], designed to write data quickly and compactly. Also, InfluxDB works very well with datasets with low cardinality, so it wins in write performance as the data used in this benchmark is relatively small. QuestDB is designed for high performance and offers several features to optimize write performance, such as vectorization and zero-garbage collection. TimescaleDB offers chunking and indexing [7], so grouping data help import data quickly, but it is slower than QuestDB. MongoDB does not support any special features for write performance, so the Mongo-naive approach takes minutes to insert the data. But changing 'document-per-event' to "false" helps the data to be grouped together, resulting in a lower time than MongoDB naive.

The last performance evaluation metric is read performance, i.e., query execution. Before discussing the results of time series databases, it is good to compare the performance of the two MongoDB implementations. We can see from the result that the recommended method has a considerable difference in efficiency. The data loading took almost half the time (246 vs. 123) due to aggregation, and the data can be quickly inserted into the database. This grouped data can also be filtered quickly, as we do not have to go to every document for query execution. So fewer documents result in the recommended method outperforming the naive approach in most queries by a significant margin.

TimescaleDB supports SQL and has a wide range of features like Group By functions and JOINS, which makes TimescaleDB a good choice for storing and querying time series data. TimescaleDB uses a unique hyper table concept [3] to partition data across time, which allows for efficient querying and analysis of time series data. In my benchmarking, I have used a chunk size of 12 hours, and the total duration of the data is 3 days. So, six hyper tables will be created for 12 hours, and all the data will be divided and stored according to the time stamp. As a result, TimescaleDB performed better than MongoDB and InfluxDB for queries like lastpoint and groupby, as grouping data in chunks makes it faster and more efficient to query. But the performance was worse for double-groupby queries and comparatively bad for cpu-max queries. But overall, it beats most of the databases in the benchmarking.

The TSM Tree storage engine of InfluxDB helps store the data by data points in chronological order. This data is organized in sorted key-value pairs, with each pair corresponding to a specific timestamp [2]. Depending on the query, the data can be retrieved for the time range by looping over the key-value points. But InfluxDB does not allow joins, which makes queries using joins perform worse as the code must be made without joins to get the desired result. Also, InfluxDB works only with recent data (depending on the retention rate), so grouping can be done with recent times only, which makes it slow for groupby queries. As a result, InfluxDB performs worse for the groupby-orderby queries, so it must take a longer route to loop the data and get the results. It also took InfluxDB more time to find the latest data reading as it performed worse than TimescaleDB and QuestDB for the lastpoitnt query.

The last database is QuestDB. Like TimescaleDB, QuestDB also uses SQL and other operations designed explicitly for aggregation and querying time series data, like ASOF JOIN, SAMPLE BY, LATEST ON, and AGGREGATES [Introduction: Questdb , 8]. Also, in QuestDB another important advantage of QuestDB is that the data is stored in columns and not rows, and fields like timestamps are accessed by columns, so querying is fast. As a result, QuestDB outperforms other databases because of its unique features. The 'Latest On' operator lets Quest DB win in the last-point query. QuestDB wins by a big ratio in other groupby queries, showing how the storage and partitioning system and other features designed especially for data series data make QuestDB a better choice for time series data.

VI. CONCLUSION

This project aims to benchmark MongoDB against time series data to answer how NoSQL databases compare against traditional time series databases. Deliverable 1 introduced me to time series data and helped me understand its uses, properties, and time series databases. Deliverable 2 focused on finalizing a benchmarking suite and the databases I will benchmark. The databases were finalized after looking at the current rankings of time series databases and those supported in the benchmark. Deliverable 3 involved familiarizing myself with the NoSQL databases, MongoDB, and the three time-series databases. This deliverable also included studying the benchmarking tool and steps and conducting the benchmarking with an example dataset and query, thus helping me understand the suite. Deliverable 4 involved benchmarking with 4 databases and 4 queries, analyzing the results, and providing a conclusion based on the metrics of data storage footprint and read and write performance.

The conclusion of the benchmarking is provided in this paragraph. All four databases have their own advantages and disadvantages, so choosing the suitable database for time series data depends on factors like data size, simplicity, time, and resources. While MongoDB is the least performant, it is a good choice if the developers have expertise in NoSQL databases and are familiar with MongoDB. InfluxDB is not open source, so for multi-node requirements, it can be costly. Also, it will take time to learn InfluxDB, as it is not a simple database. TimescaleDB is an optimal choice as it supports SQL and has better read and write performance, as seen in this benchmarking. QuestDB also uses SQL and other special operators and is specially designed for time series data, making it an optimal choice based on other databases' benchmark results and performance. Comparing the NoSQL database, i.e., MongoDB, with specific time series databases, we can conclude that it is only an excellent choice to use as a time series database if the current requirements in the company already have a NoSQL database as other non-time-based storage. So, suppose the application already uses MongoDB for non-time series data storage. In that case, using MongoDB for time-series data can be a good option, so the developers do not have to spend time learning a new database. Also, if the developers are proficient in SQL queries, then it is better to use time series databases like TimescaleDB and QuestDB, as they support SQL series and perform better than other databases.

REFERENCES

[1] D. Paul“Time Series Database (TSDB) guide: Influxdb,” InfluxData, 09-Feb-2023. [Online]. Available: https://www.influxdata.com/time-series-database/. [Accessed: 06-May-2023].

[2] S. N. Z. Naqvi, S. Yfantidou, and E. Zimanyi, “Time series databases and influxdb.” [Online]. Available: https://jira.lsstcorp.org/secure/attachment/37574/influxdb_2017.pdf. [Accessed: 06-May-2023].

[3] A. Struckov, S. Yufa, A. A. Visheratin, and D. Nasonov, “Evaluation of modern tools and techniques for storing time-series data,” Procedia Computer Science, vol. 156, pp. 19–28, 2019. doi: 10.1016/j.procs.2019.08.125

[4] “DB-Engines ranking,” DB-Engines. [Online]. Available: https://db-engines.com/en/ranking/time+series+dbms/all. [Accessed: 06-May-2023].

[5] J. Han, H. E, G. Le, and J. Du, “Survey on NoSQL database,” 2011 6th International Conference on Pervasive Computing and Applications, pp. 363–366, 2011.

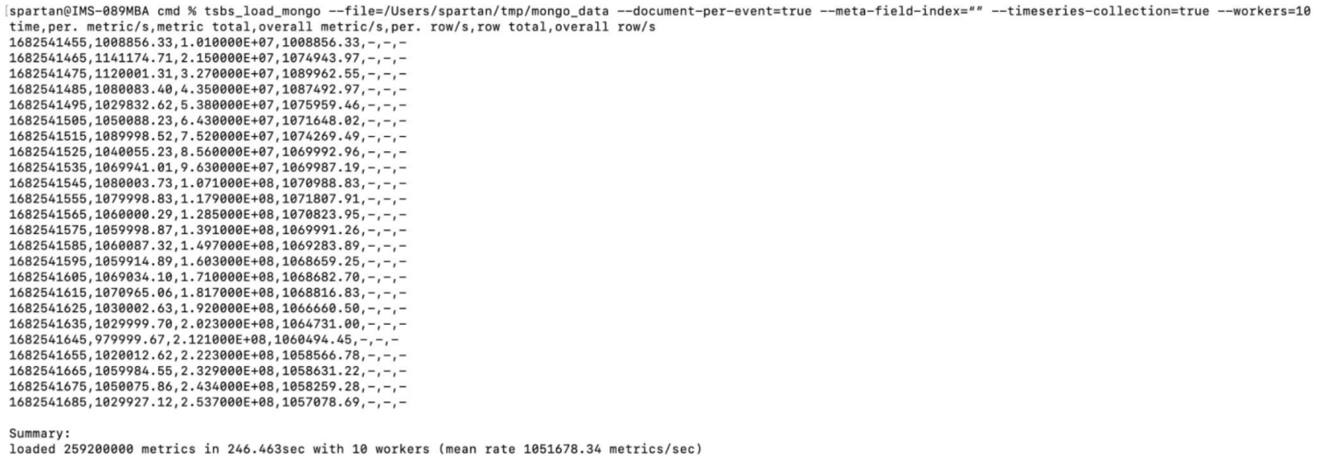
[6] V. Abramova and J. Bernardino, “NoSQL databases: MongoDB vs Cassandra,” Proceedings of the International C* Conference on Computer Science and Software Engineering, pp. 14–22, 2013.

[7] “Timescale docs,” TimescaleDB - Timeseries database for PostgreSQL. [Online]. Available: https://docs.timescale.com/. [Accessed: 06-May-2023].

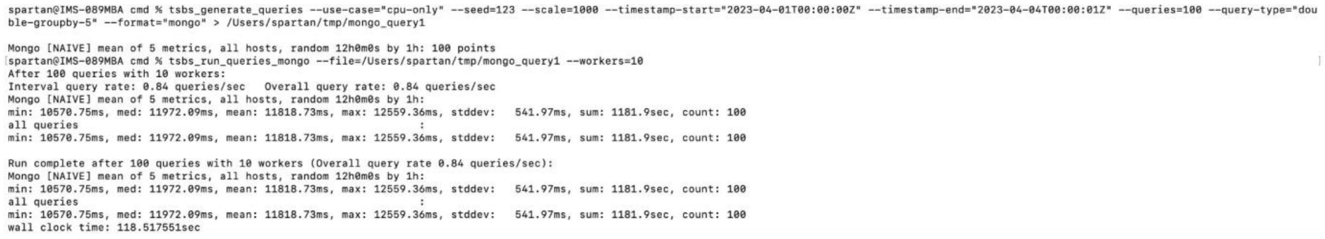
[8] “Introduction: Questdb,” QuestDB Blog RSS. [Online]. Available: https: https://questdb.io/docs/. [Accessed: 06-May-2023].

SCREENSHOT OF EXPERIMENTS

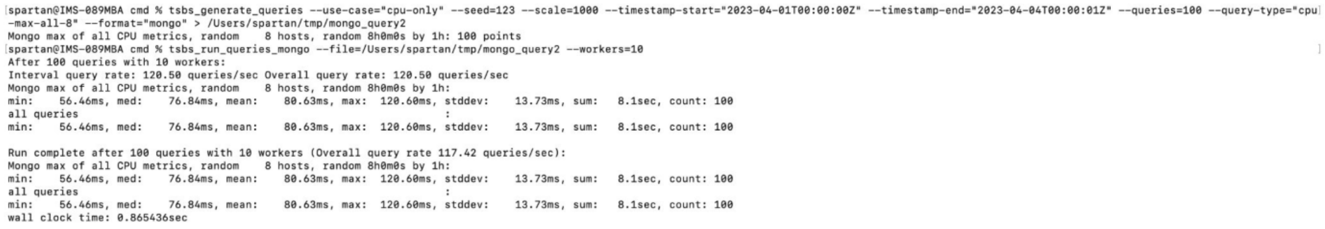
MongoDB Naive Data Loading and query execution Results:



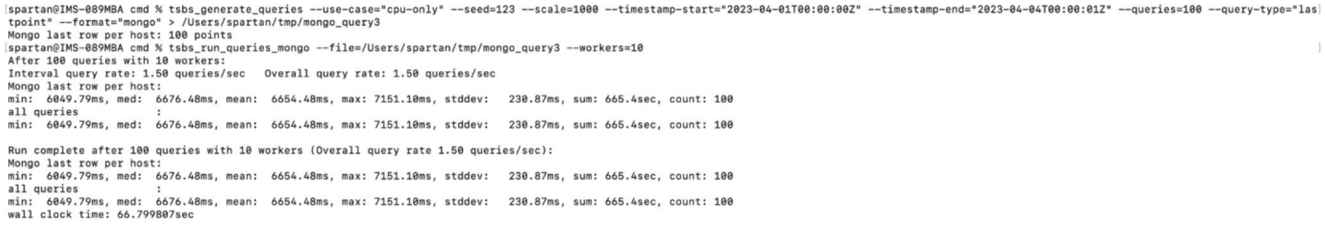
```
spartan@IMS-089MBA cmd % tsbs_load_mongo --file=/Users/spartan/tmp/mongo_data --document-per-event=true --meta-field-index="" --timeseries-collection=true --workers=10
time,per. metric/s,metric total,overall metric/s,per. row/s,row total,overall row/s
1682541455,1008856.33,1.010000E+07,1008856.33,-,-,-
1682541465,1141174.71,2.150000E+07,1074943.97,-,-,-
1682541475,1120001.31,3.270000E+07,1089962.55,-,-,-
1682541485,1080083.40,4.350000E+07,1087492.97,-,-,-
1682541495,1029832.62,5.380000E+07,1075959.46,-,-,-
1682541505,1050088.23,6.430000E+07,1071648.02,-,-,-
1682541515,1089998.52,7.520000E+07,1074269.49,-,-,-
1682541525,1040055.23,8.560000E+07,1069992.96,-,-,-
1682541535,1069941.01,9.630000E+07,1069987.19,-,-,-
1682541545,1080003.73,1.071000E+08,1070988.83,-,-,-
1682541555,1079998.83,1.179000E+08,1071807.91,-,-,-
1682541565,1060000.29,1.285000E+08,1070823.95,-,-,-
1682541575,1059998.87,1.391000E+08,1069991.26,-,-,-
1682541585,1060087.32,1.497000E+08,1069283.89,-,-,-
1682541595,1059914.89,1.603000E+08,1068659.25,-,-,-
1682541605,1069034.10,1.710000E+08,1068682.70,-,-,-
1682541615,1070965.06,1.817000E+08,1068816.83,-,-,-
1682541625,1030002.63,1.920000E+08,1066660.50,-,-,-
1682541635,1029999.70,2.023000E+08,1064731.00,-,-,-
1682541645,979999.67,2.121000E+08,1060494.45,-,-,-
1682541655,1020012.62,2.223000E+08,1058566.78,-,-,-
1682541665,1059984.55,2.329000E+08,1058631.22,-,-,-
1682541675,1050075.86,2.434000E+08,1058259.28,-,-,-
1682541685,1029927.12,2.537000E+08,1057078.69,-,-,-

Summary:
loaded 259200000 metrics in 246.463sec with 10 workers (mean rate 1051678.34 metrics/sec)
```



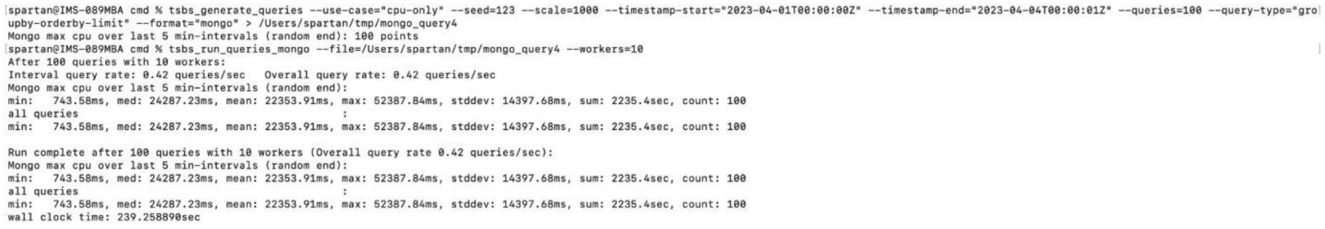
```
spartan@IMS-089MBA cmd % tsbs_generate_queries --use-case="cpu-only" --seed=123 --scale=1000 --timestamp-start="2023-04-01T00:00:00Z" --timestamp-end="2023-04-04T00:00:01Z" --queries=100 --query-type="dou
ble-groupby-5" --format="mongo" > /Users/spartan/tmp/mongo_query1

Mongo [NAIVE] mean of 5 metrics, all hosts, random 12h0m0s by 1h: 100 points
spartan@IMS-089MBA cmd % tsbs_run_queries_mongo --file=/Users/spartan/tmp/mongo_query1 --workers=10
After 100 queries with 10 workers:
Interval query rate: 0.84 queries/sec   Overall query rate: 0.84 queries/sec
Mongo [NAIVE] mean of 5 metrics, all hosts, random 12h0m0s by 1h:
min: 10570.75ms, med: 11972.09ms, mean: 11818.73ms, max: 12559.36ms, stddev:   541.97ms, sum: 1181.9sec, count: 100
all queries                                                      :
min: 10570.75ms, med: 11972.09ms, mean: 11818.73ms, max: 12559.36ms, stddev:   541.97ms, sum: 1181.9sec, count: 100

Run complete after 100 queries with 10 workers (Overall query rate 0.84 queries/sec):
Mongo [NAIVE] mean of 5 metrics, all hosts, random 12h0m0s by 1h:
min: 10570.75ms, med: 11972.09ms, mean: 11818.73ms, max: 12559.36ms, stddev:   541.97ms, sum: 1181.9sec, count: 100
all queries                                                      :
min: 10570.75ms, med: 11972.09ms, mean: 11818.73ms, max: 12559.36ms, stddev:   541.97ms, sum: 1181.9sec, count: 100
wall clock time: 118.517551sec
```



```
spartan@IMS-089MBA cmd % tsbs_generate_queries --use-case="cpu-only" --seed=123 --scale=1000 --timestamp-start="2023-04-01T00:00:00Z" --timestamp-end="2023-04-04T00:00:01Z" --queries=100 --query-type="cpu
-max-all-8" --format="mongo" > /Users/spartan/tmp/mongo_query2
Mongo max of all CPU metrics, random    8 hosts, random 8h0m0s by 1h: 100 points
spartan@IMS-089MBA cmd % tsbs_run_queries_mongo --file=/Users/spartan/tmp/mongo_query2 --workers=10
After 100 queries with 10 workers:
Interval query rate: 120.50 queries/sec Overall query rate: 120.50 queries/sec
Mongo max of all CPU metrics, random    8 hosts, random 8h0m0s by 1h:
min:    56.46ms, med:    76.84ms, mean:    80.63ms, max:  120.60ms, stddev:    13.73ms, sum:   8.1sec, count: 100
all queries                                                            :
min:    56.46ms, med:    76.84ms, mean:    80.63ms, max:  120.60ms, stddev:    13.73ms, sum:   8.1sec, count: 100

Run complete after 100 queries with 10 workers (Overall query rate 117.42 queries/sec):
Mongo max of all CPU metrics, random    8 hosts, random 8h0m0s by 1h:
min:    56.46ms, med:    76.84ms, mean:    80.63ms, max:  120.60ms, stddev:    13.73ms, sum:   8.1sec, count: 100
all queries                                                            :
min:    56.46ms, med:    76.84ms, mean:    80.63ms, max:  120.60ms, stddev:    13.73ms, sum:   8.1sec, count: 100
wall clock time: 0.865436sec
```

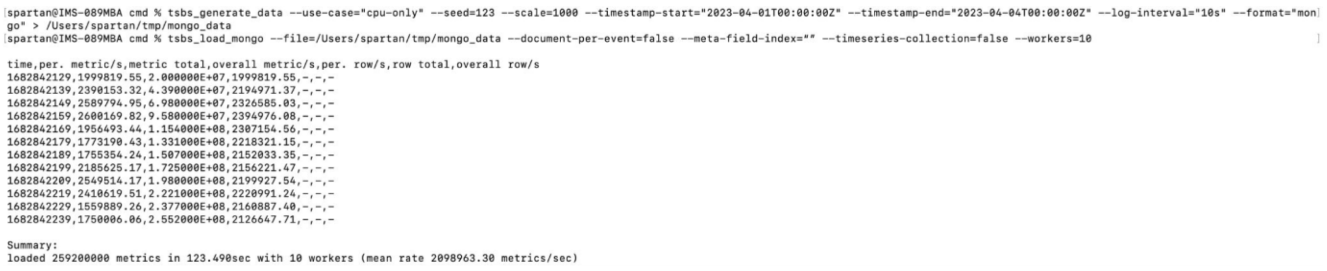


```
spartan@IMS-089MBA cmd % tsbs_generate_queries --use-case="cpu-only" --seed=123 --scale=1000 --timestamp-start="2023-04-01T00:00:00Z" --timestamp-end="2023-04-04T00:00:01Z" --queries=100 --query-type="las
tpoint" --format="mongo" > /Users/spartan/tmp/mongo_query3
Mongo last row per host: 100 points
spartan@IMS-089MBA cmd % tsbs_run_queries_mongo --file=/Users/spartan/tmp/mongo_query3 --workers=10
After 100 queries with 10 workers:
Interval query rate: 1.50 queries/sec   Overall query rate: 1.50 queries/sec
Mongo last row per host:
min:  6049.79ms, med:  6676.48ms, mean:  6654.48ms, max: 7151.10ms, stddev:   230.87ms, sum: 665.4sec, count: 100
all queries            :
min:  6049.79ms, med:  6676.48ms, mean:  6654.48ms, max: 7151.10ms, stddev:   230.87ms, sum: 665.4sec, count: 100

Run complete after 100 queries with 10 workers (Overall query rate 1.50 queries/sec):
Mongo last row per host:
min:  6049.79ms, med:  6676.48ms, mean:  6654.48ms, max: 7151.10ms, stddev:   230.87ms, sum: 665.4sec, count: 100
all queries            :
min:  6049.79ms, med:  6676.48ms, mean:  6654.48ms, max: 7151.10ms, stddev:   230.87ms, sum: 665.4sec, count: 100
wall clock time: 66.799807sec
```

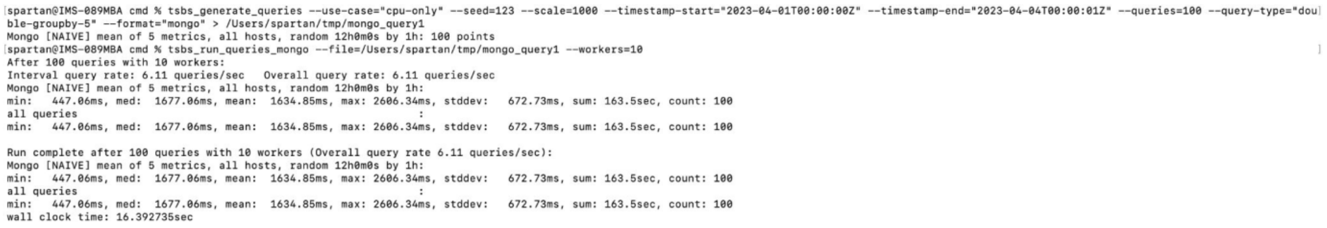


```
spartan@IMS-089MBA cmd % tsbs_generate_queries --use-case="cpu-only" --seed=123 --scale=1000 --timestamp-start="2023-04-01T00:00:00Z" --timestamp-end="2023-04-04T00:00:01Z" --queries=100 --query-type="gro
upby-orderby-limit" --format="mongo" > /Users/spartan/tmp/mongo_query4
Mongo max cpu over last 5 min-intervals (random end): 100 points
spartan@IMS-089MBA cmd % tsbs_run_queries_mongo --file=/Users/spartan/tmp/mongo_query4 --workers=10
After 100 queries with 10 workers:
Interval query rate: 0.42 queries/sec   Overall query rate: 0.42 queries/sec
Mongo max cpu over last 5 min-intervals (random end):
min:   743.58ms, med: 24287.23ms, mean: 22353.91ms, max: 52387.84ms, stddev: 14397.68ms, sum: 2235.4sec, count: 100
all queries                                          :
min:   743.58ms, med: 24287.23ms, mean: 22353.91ms, max: 52387.84ms, stddev: 14397.68ms, sum: 2235.4sec, count: 100

Run complete after 100 queries with 10 workers (Overall query rate 0.42 queries/sec):
Mongo max cpu over last 5 min-intervals (random end):
min:   743.58ms, med: 24287.23ms, mean: 22353.91ms, max: 52387.84ms, stddev: 14397.68ms, sum: 2235.4sec, count: 100
all queries                                          :
min:   743.58ms, med: 24287.23ms, mean: 22353.91ms, max: 52387.84ms, stddev: 14397.68ms, sum: 2235.4sec, count: 100
wall clock time: 239.258890sec
```

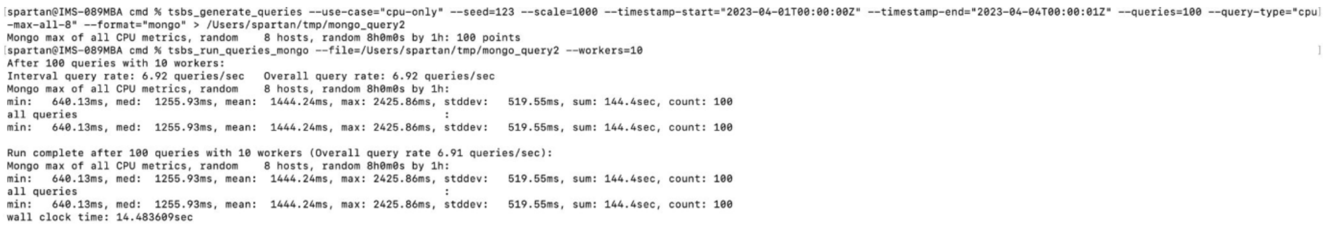
MongoDB Recommended Data Loading and query execution Results:



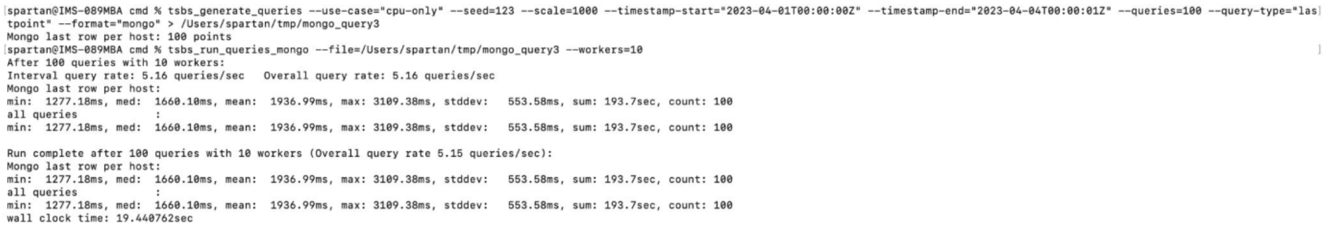
```
spartan@IMS-089MBA cmd % tsbs_generate_data --use-case="cpu-only" --seed=123 --scale=1000 --timestamp-start="2023-04-01T00:00:00Z" --timestamp-end="2023-04-04T00:00:00Z" --log-interval="10s" --format="mon
go" > /Users/spartan/tmp/mongo_data
spartan@IMS-089MBA cmd % tsbs_load_mongo --file=/Users/spartan/tmp/mongo_data --document-per-event=false --meta-field-index="" --timeseries-collection=false --workers=10

time,per. metric/s,metric total,overall metric/s,per. row/s,row total,overall row/s
1682842129,1999819.55,2.000000E+07,1999819.55,-,-,-
1682842139,2390153.32,4.390000E+07,2194971.37,-,-,-
1682842149,2589794.95,6.980000E+07,2326585.03,-,-,-
1682842159,2600169.82,9.580000E+07,2394976.08,-,-,-
1682842169,1956493.44,1.154000E+08,2307154.56,-,-,-
1682842179,1773190.43,1.331000E+08,2218321.15,-,-,-
1682842189,1755354.24,1.507000E+08,2152033.35,-,-,-
1682842199,2185625.17,1.725000E+08,2156221.47,-,-,-
1682842209,2549514.17,1.980000E+08,2199927.54,-,-,-
1682842219,2410619.51,2.221000E+08,2220991.24,-,-,-
1682842229,1559889.26,2.377000E+08,2160887.40,-,-,-
1682842239,1750006.06,2.552000E+08,2126647.71,-,-,-

Summary:
loaded 259200000 metrics in 123.490sec with 10 workers (mean rate 2098963.30 metrics/sec)
```



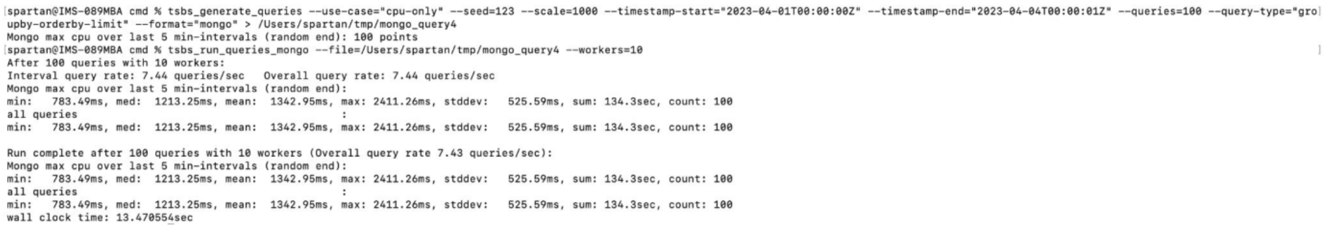
```
spartan@IMS-089MBA cmd % tsbs_generate_queries --use-case="cpu-only" --seed=123 --scale=1000 --timestamp-start="2023-04-01T00:00:00Z" --timestamp-end="2023-04-04T00:00:01Z" --queries=100 --query-type="dou
ble-groupby-5" --format="mongo" > /Users/spartan/tmp/mongo_query1
Mongo [NAIVE] mean of 5 metrics, all hosts, random 12h0m0s by 1h: 100 points
spartan@IMS-089MBA cmd % tsbs_run_queries_mongo --file=/Users/spartan/tmp/mongo_query1 --workers=10
After 100 queries with 10 workers:
Interval query rate: 6.11 queries/sec   Overall query rate: 6.11 queries/sec
Mongo [NAIVE] mean of 5 metrics, all hosts, random 12h0m0s by 1h:
min:   447.06ms, med:  1677.06ms, mean:  1634.85ms, max: 2606.34ms, stddev:   672.73ms, sum: 163.5sec, count: 100
all queries                                                  :
min:   447.06ms, med:  1677.06ms, mean:  1634.85ms, max: 2606.34ms, stddev:   672.73ms, sum: 163.5sec, count: 100

Run complete after 100 queries with 10 workers (Overall query rate 6.11 queries/sec):
Mongo [NAIVE] mean of 5 metrics, all hosts, random 12h0m0s by 1h:
min:   447.06ms, med:  1677.06ms, mean:  1634.85ms, max: 2606.34ms, stddev:   672.73ms, sum: 163.5sec, count: 100
all queries                                                  :
min:   447.06ms, med:  1677.06ms, mean:  1634.85ms, max: 2606.34ms, stddev:   672.73ms, sum: 163.5sec, count: 100
wall clock time: 16.392735sec
```



```
spartan@IMS-089MBA cmd % tsbs_generate_queries --use-case="cpu-only" --seed=123 --scale=1000 --timestamp-start="2023-04-01T00:00:00Z" --timestamp-end="2023-04-04T00:00:01Z" --queries=100 --query-type="cpu
-max-all-8" --format="mongo" > /Users/spartan/tmp/mongo_query2
Mongo max of all CPU metrics, random    8 hosts, random 8h0m0s by 1h: 100 points
spartan@IMS-089MBA cmd % tsbs_run_queries_mongo --file=/Users/spartan/tmp/mongo_query2 --workers=10
After 100 queries with 10 workers:
Interval query rate: 6.92 queries/sec   Overall query rate: 6.92 queries/sec
Mongo max of all CPU metrics, random    8 hosts, random 8h0m0s by 1h:
min:   640.13ms, med:  1255.93ms, mean:  1444.24ms, max: 2425.86ms, stddev:   519.55ms, sum: 144.4sec, count: 100
all queries                                                    :
min:   640.13ms, med:  1255.93ms, mean:  1444.24ms, max: 2425.86ms, stddev:   519.55ms, sum: 144.4sec, count: 100

Run complete after 100 queries with 10 workers (Overall query rate 6.91 queries/sec):
Mongo max of all CPU metrics, random    8 hosts, random 8h0m0s by 1h:
min:   640.13ms, med:  1255.93ms, mean:  1444.24ms, max: 2425.86ms, stddev:   519.55ms, sum: 144.4sec, count: 100
all queries                                                    :
min:   640.13ms, med:  1255.93ms, mean:  1444.24ms, max: 2425.86ms, stddev:   519.55ms, sum: 144.4sec, count: 100
wall clock time: 14.483609sec
```

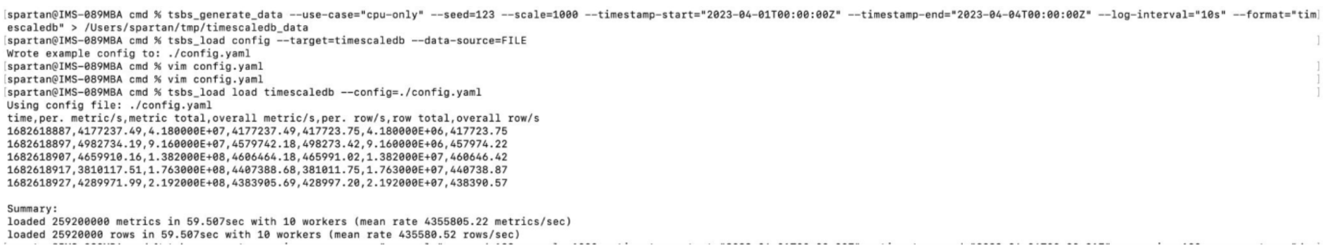


```
spartan@IMS-089MBA cmd % tsbs_generate_queries --use-case="cpu-only" --seed=123 --scale=1000 --timestamp-start="2023-04-01T00:00:00Z" --timestamp-end="2023-04-04T00:00:01Z" --queries=100 --query-type="las
tpoint" --format="mongo" > /Users/spartan/tmp/mongo_query3
Mongo last row per host: 100 points
spartan@IMS-089MBA cmd % tsbs_run_queries_mongo --file=/Users/spartan/tmp/mongo_query3 --workers=10
After 100 queries with 10 workers:
Interval query rate: 5.16 queries/sec   Overall query rate: 5.16 queries/sec
Mongo last row per host:
min:  1277.18ms, med:  1660.10ms, mean:  1936.99ms, max: 3109.38ms, stddev:   553.58ms, sum: 193.7sec, count: 100
all queries            :
min:  1277.18ms, med:  1660.10ms, mean:  1936.99ms, max: 3109.38ms, stddev:   553.58ms, sum: 193.7sec, count: 100

Run complete after 100 queries with 10 workers (Overall query rate 5.15 queries/sec):
Mongo last row per host:
min:  1277.18ms, med:  1660.10ms, mean:  1936.99ms, max: 3109.38ms, stddev:   553.58ms, sum: 193.7sec, count: 100
all queries            :
min:  1277.18ms, med:  1660.10ms, mean:  1936.99ms, max: 3109.38ms, stddev:   553.58ms, sum: 193.7sec, count: 100
wall clock time: 19.440762sec
```

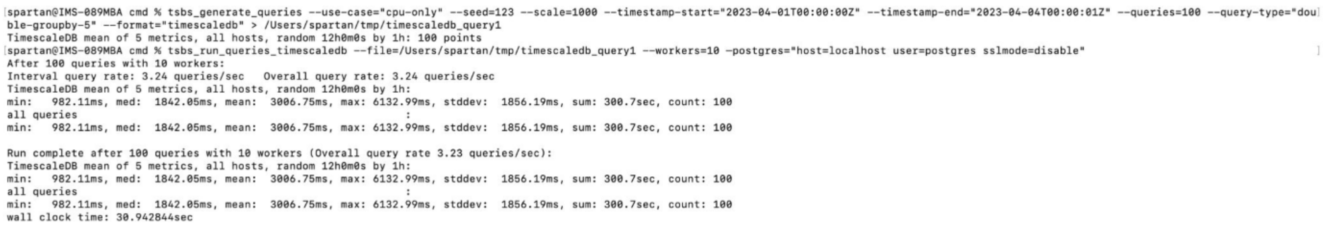


```
spartan@IMS-089MBA cmd % tsbs_generate_queries --use-case="cpu-only" --seed=123 --scale=1000 --timestamp-start="2023-04-01T00:00:00Z" --timestamp-end="2023-04-04T00:00:01Z" --queries=100 --query-type="gro
upby-orderby-limit" --format="mongo" > /Users/spartan/tmp/mongo_query4
Mongo max cpu over last 5 min-intervals (random end): 100 points
spartan@IMS-089MBA cmd % tsbs_run_queries_mongo --file=/Users/spartan/tmp/mongo_query4 --workers=10
After 100 queries with 10 workers:
Interval query rate: 7.44 queries/sec   Overall query rate: 7.44 queries/sec
Mongo max cpu over last 5 min-intervals (random end):
min:   783.49ms, med:  1213.25ms, mean:  1342.95ms, max: 2411.26ms, stddev:   525.59ms, sum: 134.3sec, count: 100
all queries                                          :
min:   783.49ms, med:  1213.25ms, mean:  1342.95ms, max: 2411.26ms, stddev:   525.59ms, sum: 134.3sec, count: 100

Run complete after 100 queries with 10 workers (Overall query rate 7.43 queries/sec):
Mongo max cpu over last 5 min-intervals (random end):
min:   783.49ms, med:  1213.25ms, mean:  1342.95ms, max: 2411.26ms, stddev:   525.59ms, sum: 134.3sec, count: 100
all queries                                          :
min:   783.49ms, med:  1213.25ms, mean:  1342.95ms, max: 2411.26ms, stddev:   525.59ms, sum: 134.3sec, count: 100
wall clock time: 13.470554sec
```

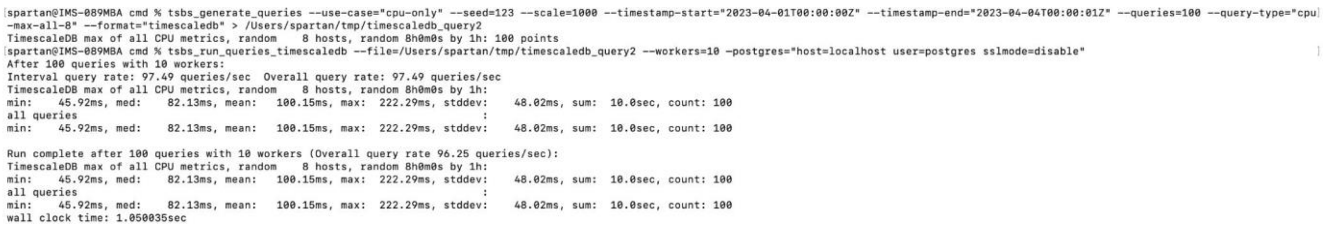
TimescaleDB Data Loading and query execution Results:



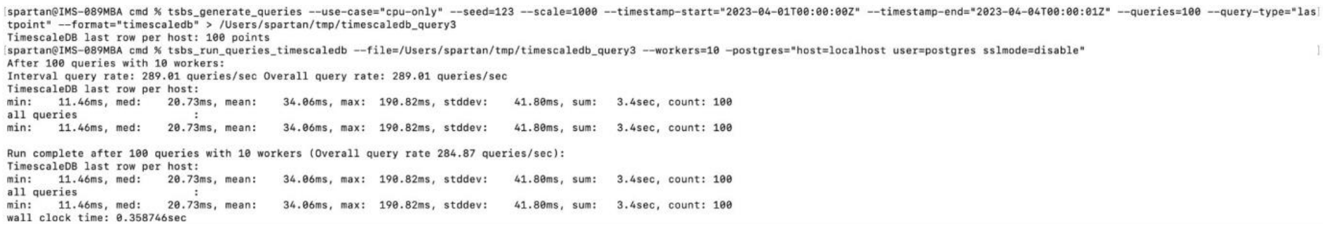
```
spartan@IMS-089MBA cmd % tsbs_generate_data --use-case="cpu-only" --seed=123 --scale=1000 --timestamp-start="2023-04-01T00:00:00Z" --timestamp-end="2023-04-04T00:00:00Z" --log-interval="10s" --format="tim
escaledb" > /Users/spartan/tmp/timescaledb_data
spartan@IMS-089MBA cmd % tsbs_load config --target=timescaledb --data-source=FILE
Wrote example config to: ./config.yaml
spartan@IMS-089MBA cmd % vim config.yaml
spartan@IMS-089MBA cmd % vim config.yaml
spartan@IMS-089MBA cmd % tsbs_load load timescaledb --config=./config.yaml
Using config file: ./config.yaml
time,per. metric/s,metric total,overall metric/s,per. row/s,row total,overall row/s
1682618887,4177237.49,4.180000E+07,4177237.49,417723.75,4.180000E+06,417723.75
1682618897,4982734.19,9.160000E+07,4579742.18,498273.42,9.160000E+06,457974.22
1682618907,4659910.16,1.382000E+08,4606464.18,465991.02,1.382000E+07,460646.42
1682618917,3810117.51,1.763000E+08,4407388.68,381011.75,1.763000E+07,440738.87
1682618927,4289971.99,2.192000E+08,4383905.69,428997.20,2.192000E+07,438390.57

Summary:
loaded 259200000 metrics in 59.507sec with 10 workers (mean rate 4355805.22 metrics/sec)
loaded 25920000 rows in 59.507sec with 10 workers (mean rate 435580.52 rows/sec)
```



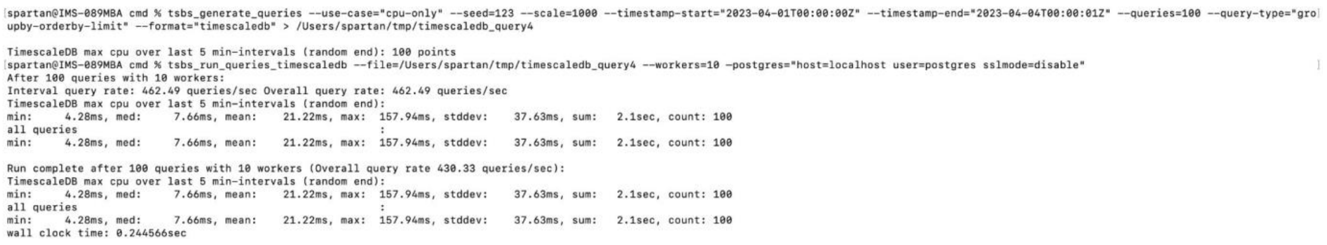
```
spartan@IMS-089MBA cmd % tsbs_generate_queries --use-case="cpu-only" --seed=123 --scale=1000 --timestamp-start="2023-04-01T00:00:00Z" --timestamp-end="2023-04-04T00:00:01Z" --queries=100 --query-type="dou
ble-groupby-5" --format="timescaledb" > /Users/spartan/tmp/timescaledb_query1
TimescaleDB mean of 5 metrics, all hosts, random 12h0m0s by 1h: 100 points
spartan@IMS-089MBA cmd % tsbs_run_queries_timescaledb --file=/Users/spartan/tmp/timescaledb_query1 --workers=10 -postgres="host=localhost user=postgres sslmode=disable"
After 100 queries with 10 workers:
Interval query rate: 3.24 queries/sec   Overall query rate: 3.24 queries/sec
TimescaleDB mean of 5 metrics, all hosts, random 12h0m0s by 1h:
min:   982.11ms, med:  1842.05ms, mean:  3006.75ms, max: 6132.99ms, stddev:  1856.19ms, sum: 300.7sec, count: 100
all queries                                                    :
min:   982.11ms, med:  1842.05ms, mean:  3006.75ms, max: 6132.99ms, stddev:  1856.19ms, sum: 300.7sec, count: 100

Run complete after 100 queries with 10 workers (Overall query rate 3.23 queries/sec):
TimescaleDB mean of 5 metrics, all hosts, random 12h0m0s by 1h:
min:   982.11ms, med:  1842.05ms, mean:  3006.75ms, max: 6132.99ms, stddev:  1856.19ms, sum: 300.7sec, count: 100
all queries                                                    :
min:   982.11ms, med:  1842.05ms, mean:  3006.75ms, max: 6132.99ms, stddev:  1856.19ms, sum: 300.7sec, count: 100
wall clock time: 30.942844sec
```



```
spartan@IMS-089MBA cmd % tsbs_generate_queries --use-case="cpu-only" --seed=123 --scale=1000 --timestamp-start="2023-04-01T00:00:00Z" --timestamp-end="2023-04-04T00:00:01Z" --queries=100 --query-type="cpu
-max-all-8" --format="timescaledb" > /Users/spartan/tmp/timescaledb_query2
TimescaleDB max of all CPU metrics, random    8 hosts, random 8h0m0s by 1h: 100 points
spartan@IMS-089MBA cmd % tsbs_run_queries_timescaledb --file=/Users/spartan/tmp/timescaledb_query2 --workers=10 -postgres="host=localhost user=postgres sslmode=disable"
After 100 queries with 10 workers:
Interval query rate: 97.49 queries/sec  Overall query rate: 97.49 queries/sec
TimescaleDB max of all CPU metrics, random    8 hosts, random 8h0m0s by 1h:
min:    45.92ms, med:    82.13ms, mean:   100.15ms, max:  222.29ms, stddev:    48.02ms, sum:  10.0sec, count: 100
all queries                                                                     :
min:    45.92ms, med:    82.13ms, mean:   100.15ms, max:  222.29ms, stddev:    48.02ms, sum:  10.0sec, count: 100

Run complete after 100 queries with 10 workers (Overall query rate 96.25 queries/sec):
TimescaleDB max of all CPU metrics, random    8 hosts, random 8h0m0s by 1h:
min:    45.92ms, med:    82.13ms, mean:   100.15ms, max:  222.29ms, stddev:    48.02ms, sum:  10.0sec, count: 100
all queries                                                                     :
min:    45.92ms, med:    82.13ms, mean:   100.15ms, max:  222.29ms, stddev:    48.02ms, sum:  10.0sec, count: 100
wall clock time: 1.050035sec
```

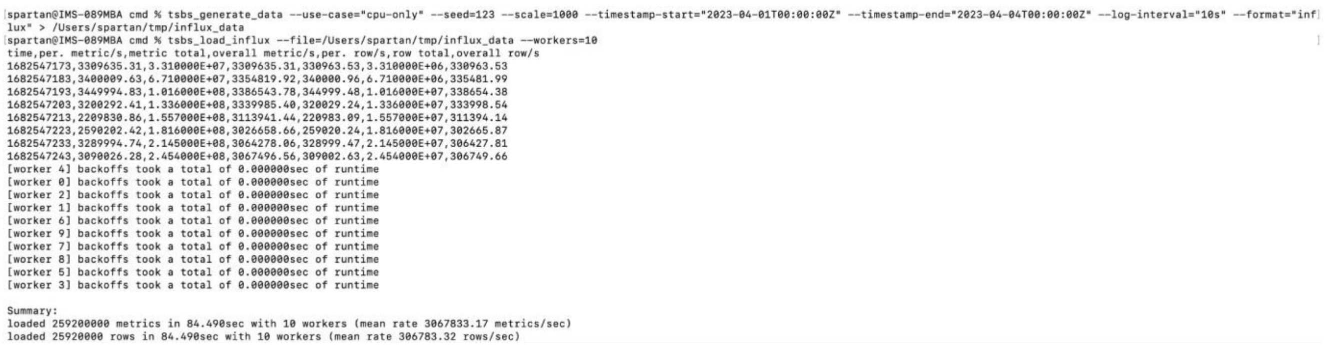


```
spartan@IMS-089MBA cmd % tsbs_generate_queries --use-case="cpu-only" --seed=123 --scale=1000 --timestamp-start="2023-04-01T00:00:00Z" --timestamp-end="2023-04-04T00:00:01Z" --queries=100 --query-type="las
tpoint" --format="timescaledb" > /Users/spartan/tmp/timescaledb_query3
TimescaleDB last row per host: 100 points
spartan@IMS-089MBA cmd % tsbs_run_queries_timescaledb --file=/Users/spartan/tmp/timescaledb_query3 --workers=10 -postgres="host=localhost user=postgres sslmode=disable"
After 100 queries with 10 workers:
Interval query rate: 289.01 queries/sec Overall query rate: 289.01 queries/sec
TimescaleDB last row per host:
min:    11.46ms, med:    20.73ms, mean:    34.06ms, max:  190.82ms, stddev:    41.80ms, sum:   3.4sec, count: 100
all queries                    :
min:    11.46ms, med:    20.73ms, mean:    34.06ms, max:  190.82ms, stddev:    41.80ms, sum:   3.4sec, count: 100

Run complete after 100 queries with 10 workers (Overall query rate 284.87 queries/sec):
TimescaleDB last row per host:
min:    11.46ms, med:    20.73ms, mean:    34.06ms, max:  190.82ms, stddev:    41.80ms, sum:   3.4sec, count: 100
all queries                    :
min:    11.46ms, med:    20.73ms, mean:    34.06ms, max:  190.82ms, stddev:    41.80ms, sum:   3.4sec, count: 100
wall clock time: 0.358746sec
```

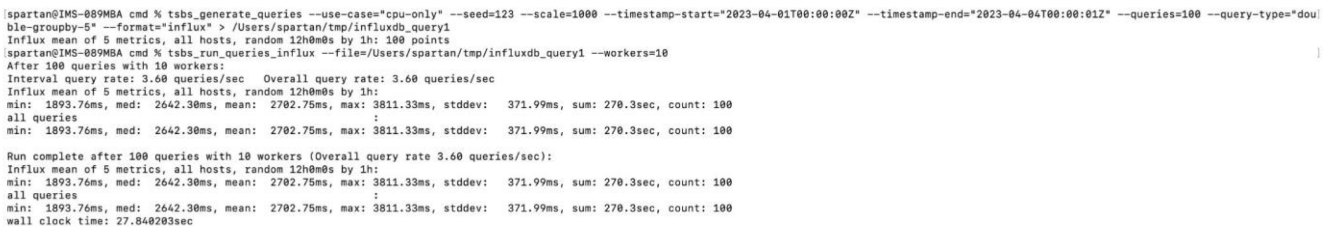


```
spartan@IMS-089MBA cmd % tsbs_generate_queries --use-case="cpu-only" --seed=123 --scale=1000 --timestamp-start="2023-04-01T00:00:00Z" --timestamp-end="2023-04-04T00:00:01Z" --queries=100 --query-type="gro
upby-orderby-limit" --format="timescaledb" > /Users/spartan/tmp/timescaledb_query4

TimescaleDB max cpu over last 5 min-intervals (random end): 100 points
spartan@IMS-089MBA cmd % tsbs_run_queries_timescaledb --file=/Users/spartan/tmp/timescaledb_query4 --workers=10 -postgres="host=localhost user=postgres sslmode=disable"
After 100 queries with 10 workers:
Interval query rate: 462.49 queries/sec Overall query rate: 462.49 queries/sec
TimescaleDB max cpu over last 5 min-intervals (random end):
min:     4.28ms, med:     7.66ms, mean:    21.22ms, max:  157.94ms, stddev:    37.63ms, sum:   2.1sec, count: 100
all queries                                                :
min:     4.28ms, med:     7.66ms, mean:    21.22ms, max:  157.94ms, stddev:    37.63ms, sum:   2.1sec, count: 100

Run complete after 100 queries with 10 workers (Overall query rate 430.33 queries/sec):
TimescaleDB max cpu over last 5 min-intervals (random end):
min:     4.28ms, med:     7.66ms, mean:    21.22ms, max:  157.94ms, stddev:    37.63ms, sum:   2.1sec, count: 100
all queries                                                :
min:     4.28ms, med:     7.66ms, mean:    21.22ms, max:  157.94ms, stddev:    37.63ms, sum:   2.1sec, count: 100
wall clock time: 0.244566sec
```

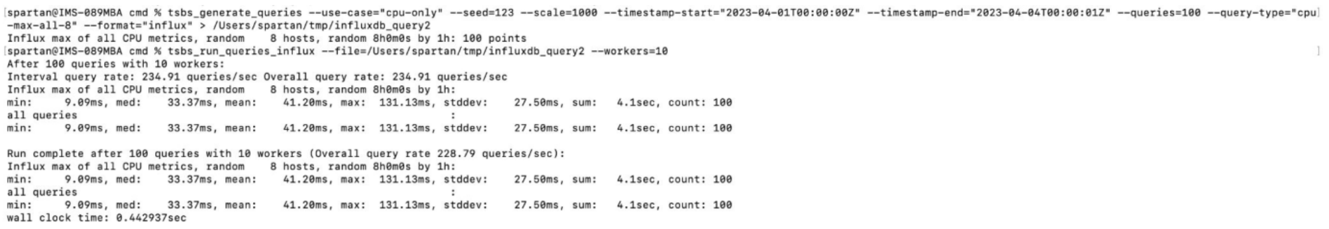
InfluxDB Data Loading and query execution Results:



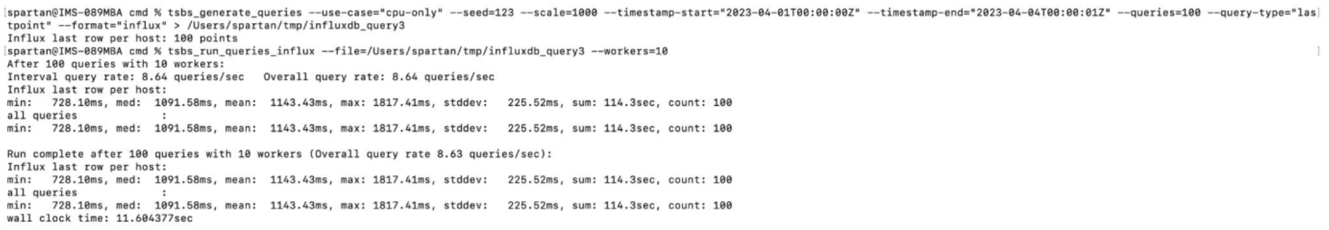
```
spartan@IMS-089MBA cmd % tsbs_generate_data --use-case="cpu-only" --seed=123 --scale=1000 --timestamp-start="2023-04-01T00:00:00Z" --timestamp-end="2023-04-04T00:00:00Z" --log-interval="10s" --format="inf
lux" > /Users/spartan/tmp/influx_data
spartan@IMS-089MBA cmd % tsbs_load_influx --file=/Users/spartan/tmp/influx_data --workers=10
time,per. metric/s,metric total,overall metric/s,per. row/s,row total,overall row/s
1682547173,3309635.31,3.310000E+07,3309635.31,330963.53,3.310000E+06,330963.53
1682547183,3400009.63,6.710000E+07,3354819.92,340000.96,6.710000E+06,335481.99
1682547193,3449994.83,1.016000E+08,3386543.78,344999.48,1.016000E+07,338654.38
1682547203,3200292.41,1.336000E+08,3339985.40,320029.24,1.336000E+07,333998.54
1682547213,2209830.86,1.557000E+08,3113941.44,220983.09,1.557000E+07,311394.14
1682547223,2590202.42,1.816000E+08,3026658.66,259020.24,1.816000E+07,302665.87
1682547233,3289994.74,2.145000E+08,3064278.06,328999.47,2.145000E+07,306427.81
1682547243,3090026.28,2.454000E+08,3067496.56,309002.63,2.454000E+07,306749.66
[worker 4] backoffs took a total of 0.000000sec of runtime
[worker 0] backoffs took a total of 0.000000sec of runtime
[worker 2] backoffs took a total of 0.000000sec of runtime
[worker 1] backoffs took a total of 0.000000sec of runtime
[worker 6] backoffs took a total of 0.000000sec of runtime
[worker 9] backoffs took a total of 0.000000sec of runtime
[worker 7] backoffs took a total of 0.000000sec of runtime
[worker 8] backoffs took a total of 0.000000sec of runtime
[worker 5] backoffs took a total of 0.000000sec of runtime
[worker 3] backoffs took a total of 0.000000sec of runtime

Summary:
loaded 259200000 metrics in 84.490sec with 10 workers (mean rate 3067833.17 metrics/sec)
loaded 25920000 rows in 84.490sec with 10 workers (mean rate 306783.32 rows/sec)
```



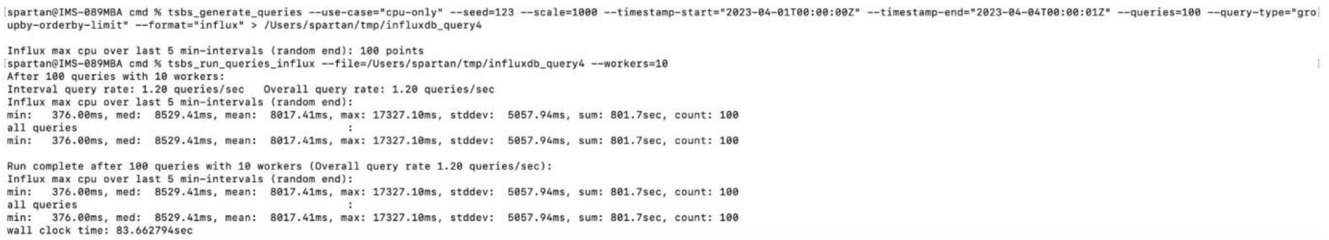
```
spartan@IMS-089MBA cmd % tsbs_generate_queries --use-case="cpu-only" --seed=123 --scale=1000 --timestamp-start="2023-04-01T00:00:00Z" --timestamp-end="2023-04-04T00:00:01Z" --queries=100 --query-type="dou
ble-groupby-5" --format="influx" > /Users/spartan/tmp/influxdb_query1
Influx mean of 5 metrics, all hosts, random 12h0m0s by 1h: 100 points
spartan@IMS-089MBA cmd % tsbs_run_queries_influx --file=/Users/spartan/tmp/influxdb_query1 --workers=10
After 100 queries with 10 workers:
Interval query rate: 3.60 queries/sec   Overall query rate: 3.60 queries/sec
Influx mean of 5 metrics, all hosts, random 12h0m0s by 1h:
min:  1893.76ms, med:  2642.30ms, mean:  2702.75ms, max: 3811.33ms, stddev:   371.99ms, sum: 270.3sec, count: 100
all queries                                              :
min:  1893.76ms, med:  2642.30ms, mean:  2702.75ms, max: 3811.33ms, stddev:   371.99ms, sum: 270.3sec, count: 100

Run complete after 100 queries with 10 workers (Overall query rate 3.60 queries/sec):
Influx mean of 5 metrics, all hosts, random 12h0m0s by 1h:
min:  1893.76ms, med:  2642.30ms, mean:  2702.75ms, max: 3811.33ms, stddev:   371.99ms, sum: 270.3sec, count: 100
all queries                                              :
min:  1893.76ms, med:  2642.30ms, mean:  2702.75ms, max: 3811.33ms, stddev:   371.99ms, sum: 270.3sec, count: 100
wall clock time: 27.840203sec
```



```
spartan@IMS-089MBA cmd % tsbs_generate_queries --use-case="cpu-only" --seed=123 --scale=1000 --timestamp-start="2023-04-01T00:00:00Z" --timestamp-end="2023-04-04T00:00:01Z" --queries=100 --query-type="cpu
-max-all-8" --format="influx" > /Users/spartan/tmp/influxdb_query2
Influx max of all CPU metrics, random    8 hosts, random 8h0m0s by 1h: 100 points
spartan@IMS-089MBA cmd % tsbs_run_queries_influx --file=/Users/spartan/tmp/influxdb_query2 --workers=10
After 100 queries with 10 workers:
Interval query rate: 234.91 queries/sec Overall query rate: 234.91 queries/sec
Influx max of all CPU metrics, random    8 hosts, random 8h0m0s by 1h:
min:     9.09ms, med:    33.37ms, mean:    41.20ms, max:  131.13ms, stddev:    27.50ms, sum:   4.1sec, count: 100
all queries                                                        :
min:     9.09ms, med:    33.37ms, mean:    41.20ms, max:  131.13ms, stddev:    27.50ms, sum:   4.1sec, count: 100

Run complete after 100 queries with 10 workers (Overall query rate 228.79 queries/sec):
Influx max of all CPU metrics, random    8 hosts, random 8h0m0s by 1h:
min:     9.09ms, med:    33.37ms, mean:    41.20ms, max:  131.13ms, stddev:    27.50ms, sum:   4.1sec, count: 100
all queries                                                        :
min:     9.09ms, med:    33.37ms, mean:    41.20ms, max:  131.13ms, stddev:    27.50ms, sum:   4.1sec, count: 100
wall clock time: 0.442937sec
```

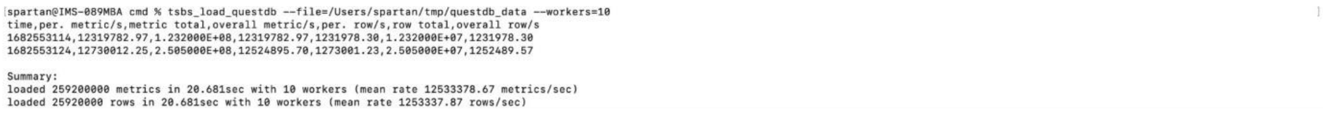


```
spartan@IMS-089MBA cmd % tsbs_generate_queries --use-case="cpu-only" --seed=123 --scale=1000 --timestamp-start="2023-04-01T00:00:00Z" --timestamp-end="2023-04-04T00:00:01Z" --queries=100 --query-type="las
tpoint" --format="influx" > /Users/spartan/tmp/influxdb_query3
Influx last row per host: 100 points
spartan@IMS-089MBA cmd % tsbs_run_queries_influx --file=/Users/spartan/tmp/influxdb_query3 --workers=10
After 100 queries with 10 workers:
Interval query rate: 8.64 queries/sec   Overall query rate: 8.64 queries/sec
Influx last row per host:
min:   728.10ms, med:  1091.58ms, mean:  1143.43ms, max: 1817.41ms, stddev:   225.52ms, sum: 114.3sec, count: 100
all queries             :
min:   728.10ms, med:  1091.58ms, mean:  1143.43ms, max: 1817.41ms, stddev:   225.52ms, sum: 114.3sec, count: 100

Run complete after 100 queries with 10 workers (Overall query rate 8.63 queries/sec):
Influx last row per host:
min:   728.10ms, med:  1091.58ms, mean:  1143.43ms, max: 1817.41ms, stddev:   225.52ms, sum: 114.3sec, count: 100
all queries             :
min:   728.10ms, med:  1091.58ms, mean:  1143.43ms, max: 1817.41ms, stddev:   225.52ms, sum: 114.3sec, count: 100
wall clock time: 11.604377sec
```

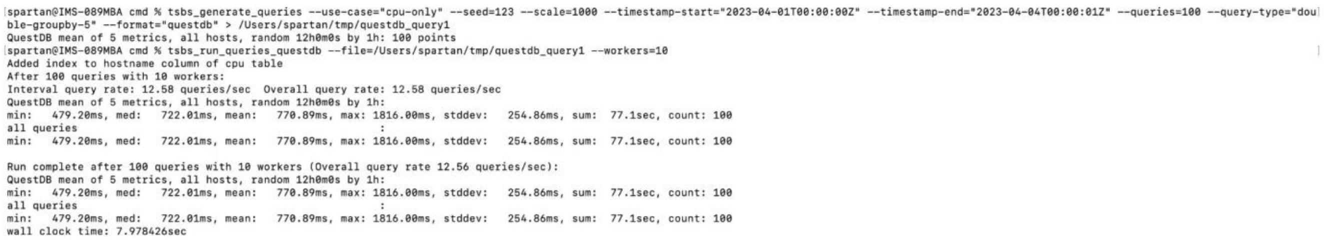


```
spartan@IMS-089MBA cmd % tsbs_generate_queries --use-case="cpu-only" --seed=123 --scale=1000 --timestamp-start="2023-04-01T00:00:00Z" --timestamp-end="2023-04-04T00:00:01Z" --queries=100 --query-type="gro
upby-orderby-limit" --format="influx" > /Users/spartan/tmp/influxdb_query4

Influx max cpu over last 5 min-intervals (random end): 100 points
spartan@IMS-089MBA cmd % tsbs_run_queries_influx --file=/Users/spartan/tmp/influxdb_query4 --workers=10
After 100 queries with 10 workers:
Interval query rate: 1.20 queries/sec   Overall query rate: 1.20 queries/sec
Influx max cpu over last 5 min-intervals (random end):
min:   376.00ms, med:  8529.41ms, mean:  8017.41ms, max: 17327.10ms, stddev:  5057.94ms, sum: 801.7sec, count: 100
all queries                                          :
min:   376.00ms, med:  8529.41ms, mean:  8017.41ms, max: 17327.10ms, stddev:  5057.94ms, sum: 801.7sec, count: 100

Run complete after 100 queries with 10 workers (Overall query rate 1.20 queries/sec):
Influx max cpu over last 5 min-intervals (random end):
min:   376.00ms, med:  8529.41ms, mean:  8017.41ms, max: 17327.10ms, stddev:  5057.94ms, sum: 801.7sec, count: 100
all queries                                          :
min:   376.00ms, med:  8529.41ms, mean:  8017.41ms, max: 17327.10ms, stddev:  5057.94ms, sum: 801.7sec, count: 100
wall clock time: 83.662794sec
```

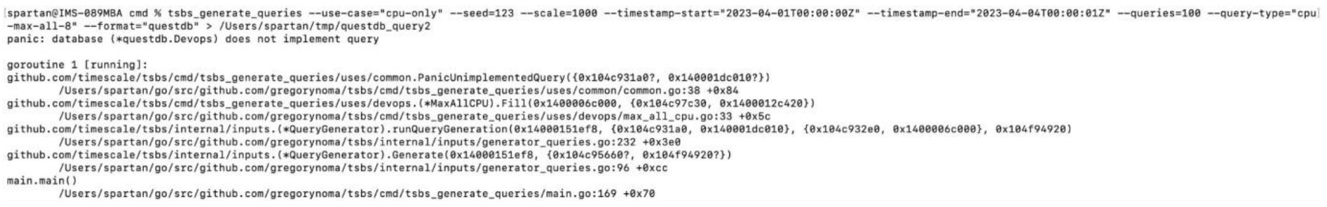
QuestDB Data Loading and query execution Results:



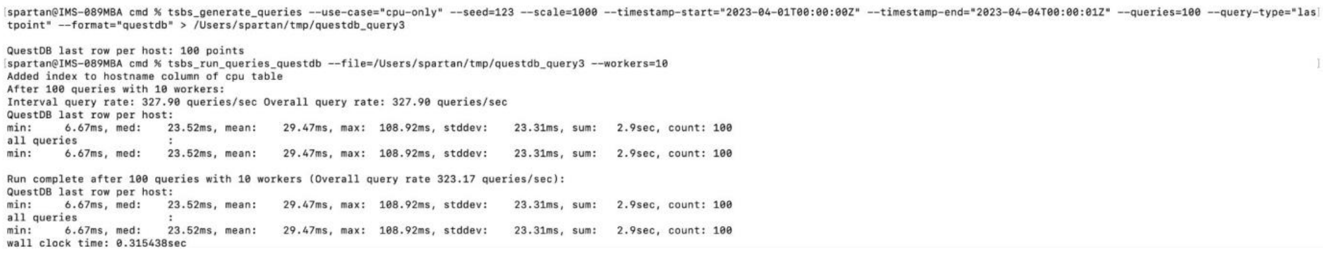
```
spartan@IMS-089MBA cmd % tsbs_load_questdb --file=/Users/spartan/tmp/questdb_data --workers=10
time,per. metric/s,metric total,overall metric/s,per. row/s,row total,overall row/s
1682553114,12319782.97,1.232000E+08,12319782.97,1231978.30,1.232000E+07,1231978.30
1682553124,12730012.25,2.505000E+08,12524895.70,1273001.23,2.505000E+07,1252489.57

Summary:
loaded 259200000 metrics in 20.681sec with 10 workers (mean rate 12533378.67 metrics/sec)
loaded 25920000 rows in 20.681sec with 10 workers (mean rate 1253337.87 rows/sec)
```



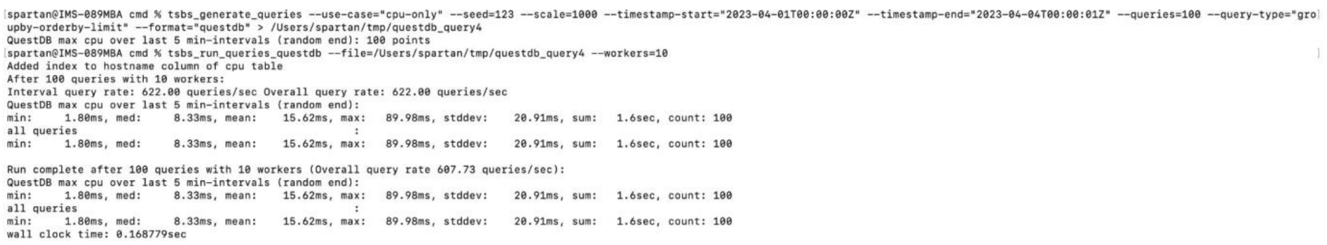
```
spartan@IMS-089MBA cmd % tsbs_generate_queries --use-case="cpu-only" --seed=123 --scale=1000 --timestamp-start="2023-04-01T00:00:00Z" --timestamp-end="2023-04-04T00:00:01Z" --queries=100 --query-type="dou
ble-groupby-5" --format="questdb" > /Users/spartan/tmp/questdb_query1
QuestDB mean of 5 metrics, all hosts, random 12h0m0s by 1h: 100 points
spartan@IMS-089MBA cmd % tsbs_run_queries_questdb --file=/Users/spartan/tmp/questdb_query1 --workers=10
Added index to hostname column of cpu table
After 100 queries with 10 workers:
Interval query rate: 12.58 queries/sec  Overall query rate: 12.58 queries/sec
QuestDB mean of 5 metrics, all hosts, random 12h0m0s by 1h:
min:   479.20ms, med:   722.01ms, mean:   770.89ms, max: 1816.00ms, stddev:   254.86ms, sum:  77.1sec, count: 100
all queries                                            :
min:   479.20ms, med:   722.01ms, mean:   770.89ms, max: 1816.00ms, stddev:   254.86ms, sum:  77.1sec, count: 100

Run complete after 100 queries with 10 workers (Overall query rate 12.56 queries/sec):
QuestDB mean of 5 metrics, all hosts, random 12h0m0s by 1h:
min:   479.20ms, med:   722.01ms, mean:   770.89ms, max: 1816.00ms, stddev:   254.86ms, sum:  77.1sec, count: 100
all queries                                            :
min:   479.20ms, med:   722.01ms, mean:   770.89ms, max: 1816.00ms, stddev:   254.86ms, sum:  77.1sec, count: 100
wall clock time: 7.978426sec
```



```
spartan@IMS-089MBA cmd % tsbs_generate_queries --use-case="cpu-only" --seed=123 --scale=1000 --timestamp-start="2023-04-01T00:00:00Z" --timestamp-end="2023-04-04T00:00:01Z" --queries=100 --query-type="cpu
-max-all-8" --format="questdb" > /Users/spartan/tmp/questdb_query2
panic: database (*questdb.Devops) does not implement query

goroutine 1 [running]:
github.com/timescale/tsbs/cmd/tsbs_generate_queries/uses/common.PanicUnimplementedQuery({0x104c931a0?, 0x140001dc010?})
        /Users/spartan/go/src/github.com/gregorynoma/tsbs/cmd/tsbs_generate_queries/uses/common/common.go:38 +0x84
github.com/timescale/tsbs/cmd/tsbs_generate_queries/uses/devops.(*MaxAllCPU).Fill(0x1400006c000, {0x104c97c30, 0x1400012c420})
        /Users/spartan/go/src/github.com/gregorynoma/tsbs/cmd/tsbs_generate_queries/uses/devops/max_all_cpu.go:33 +0x5c
github.com/timescale/tsbs/internal/inputs.(*QueryGenerator).runQueryGeneration(0x14000151ef8, {0x104c931a0, 0x140001dc010}, {0x104c932e0, 0x1400006c000}, 0x104f94920)
        /Users/spartan/go/src/github.com/gregorynoma/tsbs/internal/inputs/generator_queries.go:232 +0x3e0
github.com/timescale/tsbs/internal/inputs.(*QueryGenerator).Generate(0x14000151ef8, {0x104c95660?, 0x104f94920?})
        /Users/spartan/go/src/github.com/gregorynoma/tsbs/internal/inputs/generator_queries.go:96 +0xcc
main.main()
        /Users/spartan/go/src/github.com/gregorynoma/tsbs/cmd/tsbs_generate_queries/main.go:169 +0x70
```



```
spartan@IMS-089MBA cmd % tsbs_generate_queries --use-case="cpu-only" --seed=123 --scale=1000 --timestamp-start="2023-04-01T00:00:00Z" --timestamp-end="2023-04-04T00:00:01Z" --queries=100 --query-type="las
tpoint" --format="questdb" > /Users/spartan/tmp/questdb_query3

QuestDB last row per host: 100 points
spartan@IMS-089MBA cmd % tsbs_run_queries_questdb --file=/Users/spartan/tmp/questdb_query3 --workers=10
Added index to hostname column of cpu table
After 100 queries with 10 workers:
Interval query rate: 327.90 queries/sec Overall query rate: 327.90 queries/sec
QuestDB last row per host:
min:     6.67ms, med:    23.52ms, mean:    29.47ms, max:  108.92ms, stddev:    23.31ms, sum:   2.9sec, count: 100
all queries             :
min:     6.67ms, med:    23.52ms, mean:    29.47ms, max:  108.92ms, stddev:    23.31ms, sum:   2.9sec, count: 100

Run complete after 100 queries with 10 workers (Overall query rate 323.17 queries/sec):
QuestDB last row per host:
min:     6.67ms, med:    23.52ms, mean:    29.47ms, max:  108.92ms, stddev:    23.31ms, sum:   2.9sec, count: 100
all queries             :
min:     6.67ms, med:    23.52ms, mean:    29.47ms, max:  108.92ms, stddev:    23.31ms, sum:   2.9sec, count: 100
wall clock time: 0.315438sec
```



```
spartan@IMS-089MBA cmd % tsbs_generate_queries --use-case="cpu-only" --seed=123 --scale=1000 --timestamp-start="2023-04-01T00:00:00Z" --timestamp-end="2023-04-04T00:00:01Z" --queries=100 --query-type="gro
upby-orderby-limit" --format="questdb" > /Users/spartan/tmp/questdb_query4
QuestDB max cpu over last 5 min-intervals (random end): 100 points
spartan@IMS-089MBA cmd % tsbs_run_queries_questdb --file=/Users/spartan/tmp/questdb_query4 --workers=10
Added index to hostname column of cpu table
After 100 queries with 10 workers:
Interval query rate: 622.00 queries/sec Overall query rate: 622.00 queries/sec
QuestDB max cpu over last 5 min-intervals (random end):
min:     1.80ms, med:     8.33ms, mean:    15.62ms, max:   89.98ms, stddev:    20.91ms, sum:   1.6sec, count: 100
all queries                                         :
min:     1.80ms, med:     8.33ms, mean:    15.62ms, max:   89.98ms, stddev:    20.91ms, sum:   1.6sec, count: 100

Run complete after 100 queries with 10 workers (Overall query rate 607.73 queries/sec):
QuestDB max cpu over last 5 min-intervals (random end):
min:     1.80ms, med:     8.33ms, mean:    15.62ms, max:   89.98ms, stddev:    20.91ms, sum:   1.6sec, count: 100
all queries                                         :
min:     1.80ms, med:     8.33ms, mean:    15.62ms, max:   89.98ms, stddev:    20.91ms, sum:   1.6sec, count: 100
wall clock time: 0.168779sec
```