Comparison between Time Series Databases

Alexandros Tzanakakis
Department of *Electrical and Computer Engineering*
National-Technical University of Athens
Athens Greece
el18431@mail.ntua.gr

Fragiskos Thomas Kondilis
Department of *Electrical and Computer Engineering*
National-Technical University of Athens
Athens, Greece
el18828@mail.ntua.gr

Ioannis Palaios
Department of *Electrical and Computer Engineering*
National-Technical University of Athens
Athens Greece
el18184@mail.ntua.gr

***Abstract*—Nowadays, Time series data are ubiquitous; they are produced in abundant quantities by Internet of Things (IoT) applications, as well as the financial, medical and various other sectors that effectively utilize time stamped data. As all of these sectors see rapid growth, the generation of Time series data increases exponentially and as a result, the need for efficiently storing and querying this specific type of data emerges. In order to meet this need, Time Series Databases were created, which are optimized for handling Time series data.The need for efficient database systems is at a great need and many developers and companies are wondering which database system is the most efficient for their use case Correspondingly, effectively benchmarking and comparing databases might be essential to real world applications. In this paper, we will compare and benchmark two of the most popular and effective database systems in modern Time Series Databases, InfluxDB and TimescaleDB in terms of insert and write perfomance. Overall, our study highlights the strengths and weaknesses of each database and provides insights into the factors that influence their performance. These results can be used by developers and architects to make informed decisions about which database to use for their specific use case and requirements.**

I. Introduction

In the modern world, Time series data are ubiquitous; they are produced in large volumes by Internet of Things (IoT) applications, the financial, medical and various other sectors that effectively produce and utilize time stamped data. Typically, Time series data are data that collectively represents how a system, process, or behavior changes over time and some common specific examples include stock price performance research, tracking the performance of a server and capturing the output of a sensor.

While historically, the first main application of this kind of data was financial data, then, as the conditions of computing changed dramatically over the last decade, the number of devices with sensors, that need monitoring of some kind, is skyrocketing, emitting a relentless stream of metrics and events or time series data. Due to this sudden increase in the volume of data, a need for optimizing the way we utilize all these data is born. Traditional Database Management Systems might enable users to query and store these kind of data but, ultimately, it is not done with the optimal way. For this reason, Time Series Databases were created, which are optimized for handling and operating on this type of data. We have chosen to compare two popular Databases: InfluxDB and TimescaleDB, both specialized for handling Time Series data but with different implementations. It is critical to effectively compare those two Time Series Databases, choosing the right tool and benchmarking the right operations.

We chose the **Time Series Benchmark Suite**, a tool specialized for comparing and evaluating time series databases, in order to assist us with the benchmarking, executing the tool's implemented commands. In order to effectively benchmark these two Time Series Databases, we will compare their performance when loading, storing and querying data.

The goal is to showcase the strengths and weaknesses of each Time series Database comparatively and objectively through performing the benchmarks, visualizing the results with graphs and then justifying them in terms of each Database's design and structure.

II. Time Series Databases and their Benchmarking

A. The need for Time Series Databases

Time series data workloads are comprised primarily of high volume of high-frequency writes mostly from a recent time interval and in sequential order. Also, updates to modify or correct individual values already written are rare while data deletion is almost always done across large time ranges

like (days, months or years) [1], [2]. These characteristics make Time series data fundamentally different than other data workloads such as OLTP, which modify small amounts of data frequently and usually involve a balance of reads and writes and usually run queries involving just one or few records [3]. As traditional relational databases were built specifically for OLTP workloads, it is obvious that they will not be optimal for Time series data because of all the fundamental differences we mentioned above.

Traditional Non-Relational databases are also not the optimal way for handling Time series data. These databases store their data in Log-Structured Merge (LSM) trees and as a result, they don't perform "in-place" writes (where a small change to an existing page requires reading/writing that entire page from/to disk) and instead queue up several new updates to the database (like writes and deletes) into pages and write them as a single batch to disk, reducing the cost of making small writes. This modification of Non-Relational Databases have some significant trade-offs like higher-memory requirements (looking up a value for a key gets more complex - first check the memory table and then look to potentially many on-disk tables) and poor secondary index support because LSM trees do not naturally support secondary indexes (because they lack any global sorted order) [4].

As a result, storing and querying Time series data can be optimized with techniques different than those used in traditional relational and non relational databases. Time series databases must provide fast and easy range queries through paying attention to the data location (related data must be located together in physical storage) and optimizing the query language, have high write and read performance (with high availability and write/read speed), effective data compression and ability to handle rapid scalability [6]. Both TimescaleDB and InfluxDB have made different optimizations in order to specialize for handling Time Series Data.

B. InfluxDB

InfluxDB is an open-source schemaless time series database, written in the Go programming language and is optimized to handle time series data. It also provides an SQL-like query language called InfluxQL.

As the most important concept in InfluxDB is time, every InfluxDB database includes a Time column, storing discrete timestamps which are associated with specific data. Then, the rest of the columns of the table are called Fields (another term for attributes) and Tags. Each field consists of a field key (the name of the column) and a field value (the actual data). Similarly, Tags consist of a tag key and a tag value while, the difference with Fields is that Tags are indexed, which means that queries on Tags are faster compared to Fields. The primary key of each row consist of the timestamp and the tags.

Another characteristic of InfluxDB is its Retention Policy, which describes how long the data is stored in the database and how many copies are stored in the cluster. The default retention policy is called 'autogen' which has infinite duration and a replication factor on 1. One of the most important concepts in InfluxDB are 'series' which is a collection of data with common retention policy, measurement and tag set and a 'point' which is essentially all the fields and tag sets for this specific timestamp.

InfluxDB stores data in partitions called 'shard groups', organized by a specific retention policy and store data with timestamps that fall within a specific time interval. Internally, InfluxDB's storage engine uses its in-house built data structure, the Time Structured Merge Tree (TSM Tree). The TSM files store compressed series data in a columnar format, storing only differences between values in a series, improving efficiency and compression rate [6], [7].

C. TimescaleDB

TimescaleDB is an open-source Relational Database optimized for time series data, written in the C programming language. It is built on top of PostgreSQL, providing the full feature set of PostgreSQL, including full SQL support plus the features of a purpose-built time series database. TimescaleDB combines the best features of relational and NoSQL databases and provides a clustered architecture that scales horizontally, by transparently performing query optimizations and automatic space-time partitioning, to support high ingest rates.

TimescaleDB stores data in hypertables, which are defined with standard schemas specifying column names and types, with at least one column that specifies a time value. In deployments for clusters, data can be partitioned by a "partitioning key", which is specified in one column of a hypertable. Internally, the hypertable is splitted into chunks, automatically, across a specific time interval and the partitioning key over a primary index for the data.

Although the underlying storage might be sharded or partitioned between multiple servers, it exposes to users the abstraction of a single global table, which allows users to act as if they were interacting with one single, continuous table. Each hypertable chunk is implemented using a standard database table that can either be replicated between multiple nodes or placed on one specific database node. For flexibility, a single TimescaleDB deployment allows for storing multiple hypertables with unique schemas. [8]

D. Benchmarks

The comparison of InfluxDB and TimescaleDB will be made based on each database's performance in loading and querying the same time series data while also comparing how each database stores the same amount of data.

In order to ensure the validity and integrity of the results and to enable us to work more efficiently, we will use the Time Series Benchmark Suite (TSBS) tool in order to help us generate and load the data and then generating and executing the queries for each database.

Via TSBS, we will generate data for two use cases : IoT, simulating the data load in an IoT environment, and DevOps simulating the data load of a real world dev ops scenario (e.g. monitoring CPU, memory, disk, etc.). For each use case, we will also generate 3 sizes of datasets: small, medium and large

in order to test each database's performance in relation to the size of data loaded and queried. Then we will bulk load the data into the two databases, measuring the overall metric per second or row per second while also specifying the amount of workers. We will benchmark loading performance for 1 worker and an amount of workers equal to the CPU cores of our Virtual Machine.

After the loading of the same data in each database, we must check the size that each database takes up on the disk and compare the data compression that each database has implemented for each use case.

In order to efficiently benchmark the two databases, it is critical to test each database's performance in different types of queries. The results will be different due to different types of indexing and optimization for selecting data in each database. We will test different queries for each use case, targeting the time dimension in both point and range queries with multiple group-by, average and window queries.

III. Setting up our Databases and the Benchmarking tool

A. Our Resources

For our benchmarks, we used Oceanos-Knossos virtual machine resources. We set up two different virtual machines (VMs) one for InfluxDb and one for TimescaleDB. Each consisting of a 4 core Intel Xeon CPU @ 2.30GHz, 8GB RAM and 30GB HDD. We upgraded the operating system to Ubuntu 18.04 for the 16.04 version because the two Databases had unmet dependencies on Ubuntu 16.04 (the default Operating System of our VMs) causing various errors and bugs.

B. Installing TimescaleDB

On one of our two Virtual Machines, we set up As we previously said, TimescaleDB is an extension to PostgreSQL thus we first installed PostgreSQL 10.22 with this command :

```
$ sudo apt-get install postgresql
    postgresql-contrib -y
```

Then, following the official TimescaleDB documentation [10], we installed TimescaleDB version 1.7.5 with the following commands :

```
$ sudo apt-get install gnupg2
    software-properties-common
    curl git unzip -y
$ sudo apt-get install
    timescaledb-postgresql-10 -y
$ sudo apt-get install libpq-dev
$ sudo timescaledb-tune --quiet --yes
$ sudo service postgresql restart
```

C. Installing InfluxDB

The installation of InfluxDB was made in accordance with InfluxDB's official documentation for version 2.6 [11]. We executed the following commands :

```
$ wget -qO-
    https://repos.influxdata.com
        /influxdb.key
    | gpg --dearmor
    > /etc/apt/trusted.gpg.d
        /influxdb.gpg
$ sudo curl -sL
    https://repos.influxdata.com
        /influxdb.key
    | sudo apt-key add -
$ sudo echo "deb
    https://repos.influxdata.com/ubuntu
        bionic stable"
    | sudo tee /etc/apt
        /sources.list.d/influxdb.list
$ sudo apt update
$ sudo apt install influxdb
$ sudo systemctl status influxdb
```

Then we start the InfluxDB extension :

```
$ sudo service influxdb start
```

D. Setting up Time Series Benchmark Suite

In order to install TSBS, we executed following procedure in both we need to first install Go language:

```
$ sudo apt install golang-go
```

And then we use Golang to help us fetch TSBS and its dependencies :

```
$ go get github.com/timescale/tsbs
```

And then we proceed to build the TSBS Go programs executing the following command in TSBS's folder :

```
$ cd $GOPATH/src/github.com/timescale/tsbs
$ make
```

Then we add all the TSBS commands that were build in our VM's path adding the following line in the end of the global .bashrc file (in order to make them accessible globally) :

```
PATH=$PATH:/home/user/go/bin
```

IV. Data Generation and Loading

A. Data Generation

For the data generation we used TSBS's script :

```
tsbs_generate_data .
```

We generated data for two use cases that TSBS offered. First, the IoT use case, containing data which can contain out-of-order, missing, or empty entries to better simulate the data load in an IoT environment. Specifically, this use case simulates data streaming from a set of trucks, generating diagnostic data and metrics for each truck. And secondly, the Dev ops use case which generates data from system that could be monitored in real word Dev Ops scenario (e.g. CPU, memory, disk, etc.).

Using these commands, the size of the generated data can be specified through the scale option which specifies the number of trucks or devices (CPU, memory, etc.) and the time interval to generate data for, and how much time should be between each reading per device/truck (the log interval). In our case, we will define the size of the file changing the time interval (and/or the log interval, only in the devops use case) while keeping the number of trucks/devices unchanged.

For each use case, we generated 3 sizes of data : small , medium and large in order to check each database's performance in response to data size. For example the command for generating the large sized data, for IoT use case in the specific format for TimescaleDB is :

```
tsbs_generate_data --use-case="iot"
    --seed=123 --scale=4000
    --timestamp-start=
        "2016-01-01T00:00:00Z"
    --timestamp-end=
        "2016-01-10T00:00:00Z"
    --interleaved-generation-groups=10
    --log-interval="10s"
    --format="timescaledb"
    > ~/data/large-data.csv
```

We specify the database we want to generate data for via the "–format" option (we use either "timescaledb" or "influx"), the time interval via the time between the "–timestamp-start" and "–timestamp-end" options and the use case via the "–use-case" option (either "iot" or "devops").

Each day in the time interval will add an additional 33M rows, so for the IoT use case, the small will be 12 hours (700 MB), the medium 2 and a half days (2700 MB) and the large 9 days (12.5GB) (all for the same "–log-interval" = 10 seconds). Additionally, for the Dev Ops use case, the small will be one day, the medium will be 5 days and the large will be 14 days, while we increase the "–log-interval" to 100 seconds.

B. Loading the Data

In order to efficiently load the data while benchmarking each database's performance at the same time, we will use TSBS's script :

```
scripts/load/load_<database>.sh
```

where database = "timescaledb" or "influx". Firstly, we modify the script

```
scripts/load/load_common.sh
```

in order to change the directory that it can access the data from and other variables to ensure database connectivity. Then, we modify also the

```
scripts/load/load_<database>.sh
```

script to change the amount of workers (either 1 or 4 - equal to the CPU cores we have in each VM), other variables to ensure database connectivity, the different file to load (for different sizes and use cases) and the batch size which we eventually kept at default 10,000. Of course, we direct the result of the scripts to a local folder storing all the output files.

We expect TimescaleDB to be faster than InfluxDB in the data insertion, because of the data compression that InfluxDB performs on when data are loaded. But also expect to see some fluctuations, because the Storage Engines and Architectures of each database are completely different.

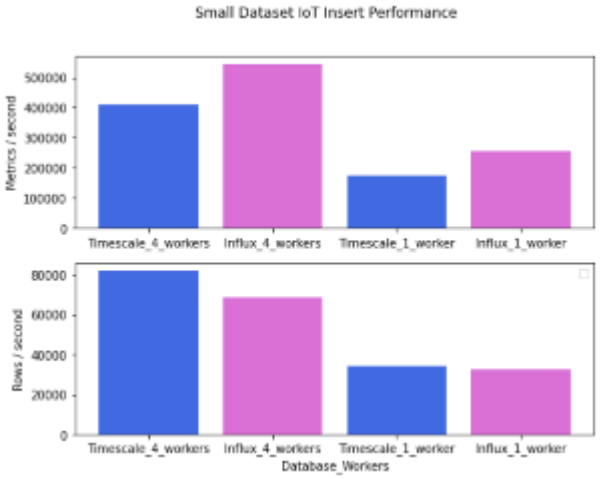

Fig. 1. Small Dataset IoT Load Performance Comparison

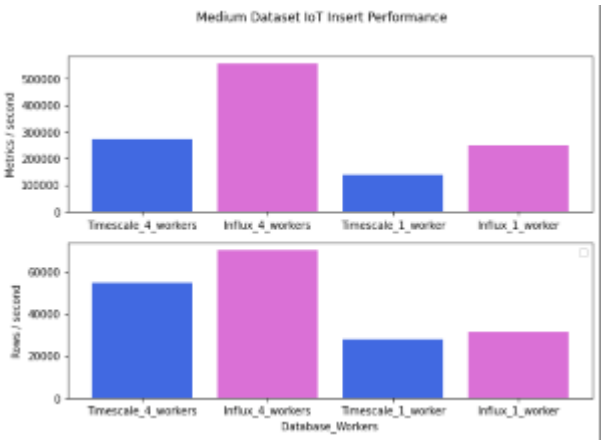

Fig. 2. Medium Dataset IoT Load Performance Comparison

Through the Fig. 1,2,3. we indeed see the fluctuation that we expected, that InfluxDB is faster at data ingestion than TimescaleDB, possibly due to their differences in implementation. Specifically, it may be due to the fact that TimescaleDB is a Relational database while InfluxDB is Non-Relational allowing InfluxDB to ingest the data faster.

However increasing the –scale option in the command

```
tsbs_generate_data
```

and therefore increasing the number of trucks/devices in the dataset we produced, it is expected that InfluxDB performance will drop due to its reliance on TSM trees which doesn't

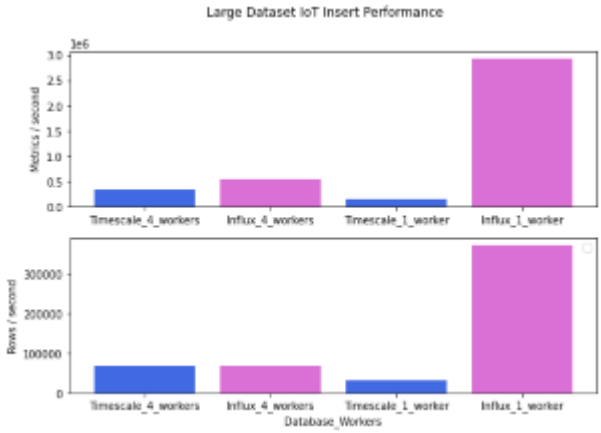

Fig. 3. Large Dataset IoT Load Performance Comparison

perform well in higher cardinality datasets (similarly to Log Structured Merge trees). [12]

V. Storage Space

InfluxDB, like most databases that use a column-oriented approach, is expected to offer significantly better on-disk compression than TimescaleDB (and PostgreSQL). We checked each database's size with these commands :

```
\l+
```

on PostgreSQL shell in order to check the size of the TimescaleDB database named 'benchmark' (which is the default database name TSBS creates when loading data - the same is true for InfluxDB), and

```
$ du -sh /var/lib/influxdb/data/benchmark
```

on the terminal in order to check the size of the InfluxDB database called 'benchmark'.

It is obvious in Fig. 4. that InfluxDB does a much better job in compressing the data than TimescaleDB. TimescaleDB roughly compresses the data, taking up disk space roughly as much as the raw data that are not loaded on the database (sometimes even more). On the other hand, InfluxDB achieves a compression ratio of roughly 10 per cent of the original file size (in pseudo csv type). This compression is achieved as each block in the InfluxDB storage engine is has all of its timestamps and values compressed and then stored separately using encodings dependent on the data type and its shape.

VI. Query Generation and Performance Measurement

A. Query Generation

We generated the queries with the help of TSBS. We run this script :

```
scripts/generate_queries.sh
```

while we changed the options "timestamp-start" and "timestamp-end" depending on the time interval the dataset that we have loaded at the database at the time has been created

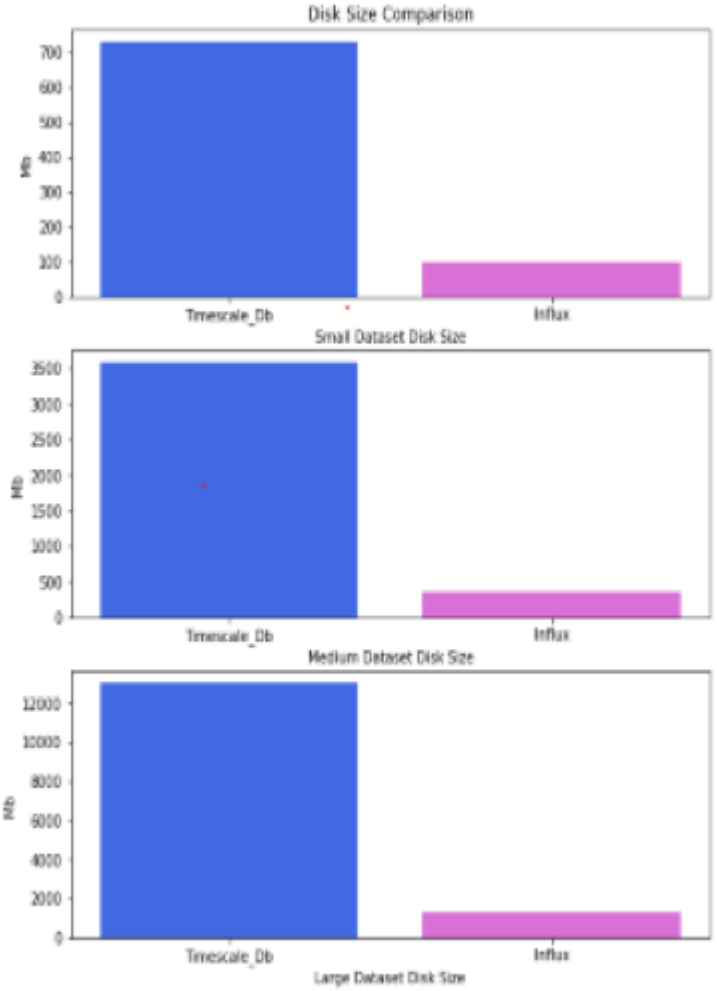

Fig. 4. Disk size comparison

in. Also, we changed the format depending on the database we are targeting ("influx" or "timescaledb") and the use case depending on the use case the currently loaded data have been created for. Finally, for each of the two use cases, we generated all the available queries that TSBS had available. For example, for the IoT use case, TSBS had various aggregate and group by queries.

B. Query Performance Measurement

We measured the queries again with the help of TSBS. In order to automate running the scripts, we made a script called :

```
~/scripts/run_queries.sh
```

for the IoT or

```
~/scripts/run_queries_devops.sh
```

for the Dev Ops use case. We want to direct the output of the commands "tsbs_run_queries_timescaledb" and "tsbs_run_queries_influx" onto a folder called output/queries/-query_name.out in order to have the benchmarks saved locally.

Each of those scripts run one of the above commands for all of the queries that we want to run :

```
cat /tmp/bulk_queries/<query_name>.gz | \
    gunzip | tsbs_run_queries_timescaledb
        --workers=4 \
        --postgres="host=localhost
        user=postgres password=password
        sslmode=disable "
        > ~/output/queries/
            <query_name>.out
```

in order to run the query with name = query_name for TimescaleDB. The equivalent for InfluxDB would be :

```
cat /tmp/bulk_queries/<query_name>.gz |
    gunzip | tsbs_run_queries_influx
    --workers=4 \
    ~/outputs/queries/<query_name>.out
```

As a result, running the above scripts will output all the benchmarks of the queries in the folder

```
~/outputs/queries
```

one data size (small, medium or large) at a time.

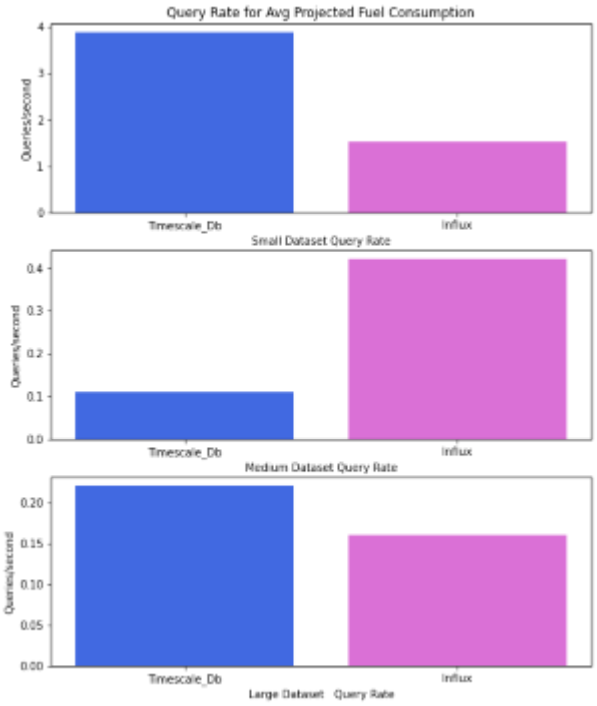

Fig. 5. The Query Rates for the Average versus Projected Fuel Consumption queries (for the IoT Use Case)

We observe in Figure 5 that for the query of Average versus Projected Fuel Consumption TimescaleDB has a bigger Queries/Second rate only for the small and large dataset but the same thing is not true for the medium dataset.

In Figure 6, we see the results for the Long Driving Sessions queries. These are simpler aggregate queries asking

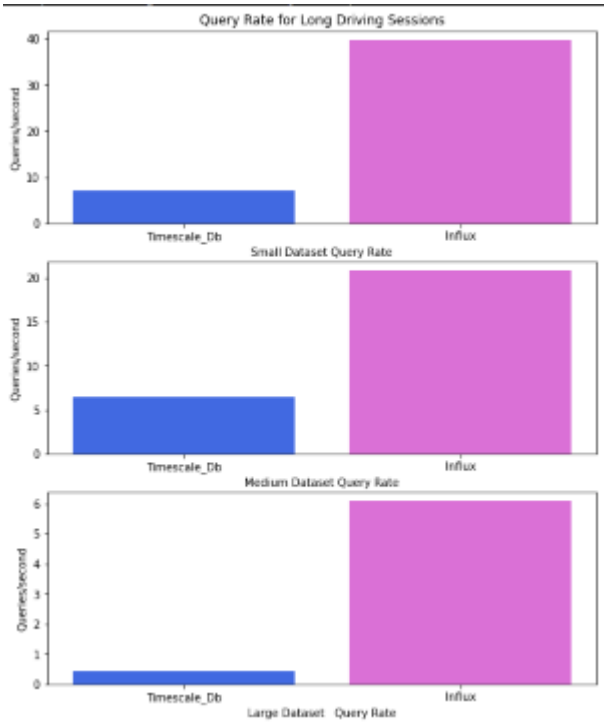

Fig. 6. The Query Rates for the Long Driving Sessions queries (for the IoT Use Case)

the databases to "Get trucks which haven't rested for at least 20 mins in the last 4 hours" [9]. We see that InfluxDB has almost double the query rate that TimescaleDB has.

It is observed in Fig. 8 that TimescaleDB's query rate is greater for queries that aggregate across both time and host in the DevOps use case, which could be explained by the fact that TimescaleDB is optimized for complex SQL queries and has advanced features for time-series data modeling. TimescaleDB's hypertables allow for efficient storage and retrieval of time-series data, which can improve performance for complex queries that involve aggregation across multiple dimensions. In contrast, InfluxDB is designed for high write throughput and real-time monitoring, which may make it less optimized for complex aggregation queries that involve multiple dimensions. InfluxDB's schemaless design may also make it more difficult to perform complex aggregation queries, as the data may not be structured in a way that allows for efficient aggregation. However, when it comes to simple aggregation queries on a single metric, InfluxDB's design and indexing techniques can provide faster query performance compared to TimescaleDB, as seen in Fig 8.

Overall, the difference in query rate between TimescaleDB and InfluxDB in the DevOps use case may be due to the differences in their architecture and design, with TimescaleDB optimized for complex aggregation queries and InfluxDB optimized for real-time monitoring and high write throughput. Developers and architects should consider the specific query workload and requirements of their application when selecting a time-series database, taking into account the strengths and

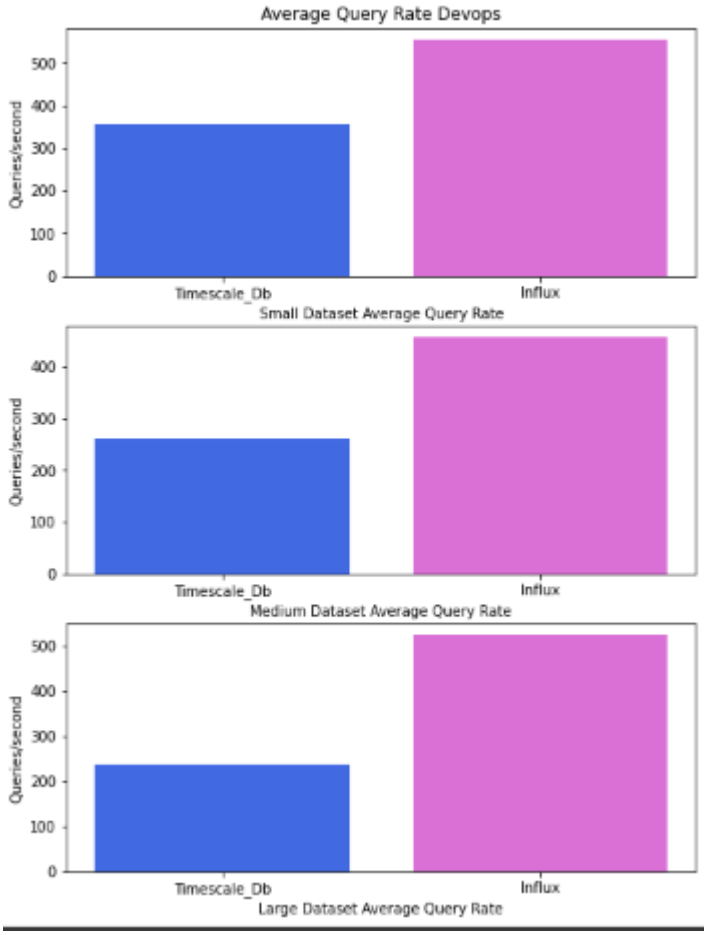

Fig. 7. The Average Query Rates for all the Dev Ops queries

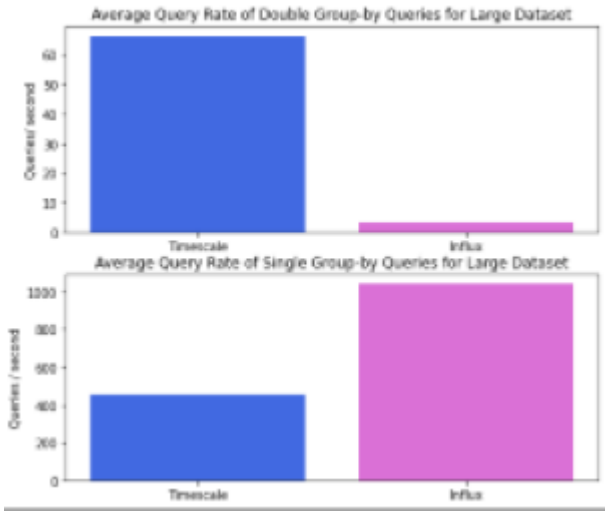

Fig. 8. The Average Query Rates for Double Group By Queries vs Single Group By Queries

weaknesses of each database.

VII. Conclusion

In conclusion, InfluxDB and TimescaleDB are two powerful time-series databases that provide different strengths and features. InfluxDB excels at high write throughput and real-time monitoring, making it an excellent choice for applications that require fast data ingestion and real-time analysis. In contrast, TimescaleDB is better suited for complex SQL queries and advanced analytics, making it more suitable for applications that require advanced data analysis and querying. Our comparison using the Time Series Benchmark Suite showed that InfluxDB performed better for simple aggregation queries or projections, while TimescaleDB had some performance advantages for certain queries that require more complex aggregations.

Moreover, InfluxDB has built-in compression algorithms that can reduce the size of time-series data by up to 90 percent, which makes it more efficient than TimescaleDB when it comes to storage and retrieval of time-series data, an attribute that can prove highly useful.

Ultimately, the choice between InfluxDB and TimescaleDB should be based on the specific requirements and query workloads of the application. Both databases provide powerful features and can be optimized for high performance, but their strengths and weaknesses depend on the specific use case. As such, developers and architects should carefully evaluate the requirements of their application and consider the features, performance, and scalability of each database before making a final decision.

A. GitHub Link

https://github.com/johnpalaios/time-series-db-benchmarks

References

[1] https://www.timescale.com/blog/time-series-data
[2] http://get.influxdata.com/ rs/972GDU533/images/InfluxDB/vs_Elasticsearch.pdf
[3] https://www.oracle.com/database/what-is-oltp
[4] https://www.timescale.com/blog/time-series-data-why-and-how-to-use-a-relational-database-instead-of-nosql-d0cd6975e87c
[5] http://jmoiron.net/blog/thoughts-on-timeseries-databases/
[6] https://cs.ulb.ac.be/public/_media/teaching/influxdb_2017.pdf
[7] https://docs.influxdata.com/influxdb/v2.6/reference/internals/storage-engine
[8] TimescaleDB : SQL made scalable for time-series data. https://www.semanticscholar.org/paper/TimescaleDB-
[9] Time-Series-Databases-Benchmarks https://github.com/timescale/tsbs
[10] TimescaleDB Install Documentation https://docs.timescale.com/install/latest
[11] InfluxDB v2.6 Install Documentation https://docs.influxdata.com/influxdb/v2.6/install/?t=Linux
[12] TimescaleDB vs. InfluxDB: Purpose Built Differently for Time-Series Datahttps://www.timescale.com/blog/timescaledb-vs-influxdb-for-time-series-data-timescale-influx-sql-nosql-36489299877
[13] The InfluxDB Storage Engine and the Time-Structured Merge Tree (TSM) https://archive.docs.influxdata.com/influxdb/v1.0/concepts/storage_engine