Project Report

Exploring Computational Capabilities of InfluxDB and TimescaleDB

CS550 – Advanced Operating Systems

May 1st 2021

Elliot Tan

A20382909

Zhenghao Zhao

A20429116

1. Introduction & Background Information

Time series data has increasingly attracted attention from both research areas and industry companies for the past few years. With the development of Artificial Intelligence, time series data became widely used in machine learning and data mining cases. Financial data is one of the most interesting and lucrative data, such as stock market prices and index prices. Also, time series data plays an important role in forecasting the trend of COVID19 pandemic. Time series databases, which are able to handle time series data efficiently, therefore also became essential. In this project, our team took two most representative time series databases, InfluxDB and TimescaleDB, as examples to learn about the capabilities, operations and architecture of such systems. In addition, we benchmarked the both of the time series databases using Time Series Benchmark Suite, analyzed their performances and compared the differences and similarities between them.

Some properties of time series databases are as follows[1]. Related data is co-located by a time series database and stored in the same physical location, allowing for better I/O performance. Because of this, queries over a range of data over time tend to also be very fast, and some databases also implement ways to easily make such queries. Time series databases also have to be able to handle a large number of write requests. Take for example continuously recording stock market prices, or measurements from a network in an Internet of Things. This project will take a closer look at how much faster and more efficiently time series databases handle time series information as supposed traditional databases.

There has been some research done on comparing different time series databases. Sanaboyana Tulasi Priyanka conducted a performance evaluation on time series databases and compared the energy consumed by both OpenTSDB and InfluxDB on the same set of machines with the same data. She compared them under 3 conditions: no load, synchronization, and during read write operations. She concluded that InfluxDB used less energy than OpenTSDB in the same scenarios[5]. Another research project by Bader, Kopp, and Falkenthal was a survey and comparison between different open source time series databases. They found 83 TSDBs from their systematic search and grouped and compared them by their scalability, load balancing, and others[3]. Piotr Grzesik and Dariusz Mrozek did a comparative analysis of the time taken to insert and read data for 5 different databases: TimscaleDB, InfluxDB, Riak TS, PostgreSQL, and SQLite. They concluded that while PostgreSQL performed the best on smaller datasets, InfluxDB and Timescale DB were more applicable as they provided[2]. An article by Rob Kiefer shows his comparison between PostgreSQL and TimescaleDB and showed that TimescaleDB far outperforms PostgreSQL when handling time-based data. TimescaleDB was able to perform 20 times better at inserts and 2000 times better at deletes than PostgreSQL.

2. Problem Statement

The goal of this project is to compare two time series databases: InfluxDB and TimescaleDB. The comparison problem of two databases can be divided into three sections. First, the architecture of the databases need to be explored. The architecture of databases helps us to understand how data is stored, and how queries are executed in different databases. Second, it is necessary to get to know the unique operational capabilities of the databases. This is also an important section. As we know both InfluxDB and TimescaleDB are popular time series databases on the market. There must be some functional differences between them, and our task is to figure out their unique operation capabilities. Third, the computational performance should be evaluated. The performance of a database is very essential. One database may perform variously on different types of queries. Therefore, we need to figure out for InfluxDB and TimescaleDB, which database performs better on what kind of queries. Also, the data insertion performance is an important feature, which needs to be considered in this project.

3. Solution

In this project, we research InfluxDb and TimescaleDB to get a deep understanding of these two databases. The contributions of this work are:

1) We researched architecture of both databases
2) We compared the operational differences of both databases
3) With Time Series Benchmarking Suite (TSBS), we evaluated and compared the computational performance of both databases

3.1 Introduction of selected databases

3.1.1 InfluxDB Introduction

InfluxDB is an open source time series database written in the Go programming language. It was developed by InfluxData and is the most popular time series database according to db-engines [7]. It offers high performance in data storage and retrieval, and is useful for data monitoring and analysis.

InfluxDB uses an SQL-like language called Flux. This language is both a query language and a scripting language, and thus has more flexibility that traditional SQL, and offers more functionality than SQL. This will be shown more in depth in section 3.4

Part of InfluxDB's popularity is attributed to the fact that users are able to not only access InfluxDB through their terminal, but also through its user interface system. This allows users to have a visualization of their data, and to draw insights from it better. The data shown in the UI can be explored using Flux and it's various packages and functions.

3.1.2 TimescaleDB Introduction

TimescaleDB is an open source time-series database, which is implemented as an extension on PostgreSQL[8]. It offers high performance for fast ingest and complex queries.

TimescaleDB leverages many advantages of PostgreSQL. First, TimescaleDB is fully SQL supported. Second, since It is built on PostgreSQL, remaining the lowest levels of PostgreSQL storage, TimescaleDB is rock-solid on its security and reliability. Third, TimescaleDB can be easily connected to a wide range of third-party tools supported by PostgreSQL.

3.2 Architecture of selected databases

3.2.1 InfluxDB Architecture

In InfluxDB, time is used to index and identify the data that is being written, and it writes in ascending order. InfluxDB also strongly controls updates and deletes on existing data. Time series data generally doesn't require changing previous writes, and this also reduces the potential for read write conflicts.

In InfluxDB, traditional tables can be thought of as measurements, but a more formal definition will be given later. Although users can display measurements, the data is not actually stored as such, but is only a view of the data that is kept in their buckets. InfluxDB is schemaless, but it uses buckets to hold all of the data. The way in which data is organized will be as follows.
A point in influxDB is what would normally be called a row in traditional databases, and it consists of tags, fields, and a timestamp. A timestamp is both the time in which the data is written and the index to the point. Time stamps are recorded in RFC3339 UTC format.

Fields and tags can be thought of as columns of the points. Field key and tag keys are the column headers, while field values and tag values are the entries in the points themselves. An important thing to note is that field values can be recorded as any data type (string, int, float, bool, etc) and

use aggregate functions on fields, but the tag values are recorded only as strings and cannot be used in aggregate functions. As such, users will want to define fields for the values that they want to measure, and tags for the metadata of the fields. Tag values are indexed, while field values are not. Below is an example of a measurement.

stock_prices_and_bids

time_stamp	stock	price	num_of_bids
2019-08-18T00:00:00Z	AAPL	100	55
2019-08-18T00:01:00Z	AAPL	101	76
2019-08-18T00:02:00Z	AAPL	103	100
2019-08-18T00:00:00Z	SPF	110	104
2019-08-18T00:01:00Z	SPF	104	98

In this example, the measurement name would be “stock_prices_and_bids”. The field keys would be “price” and “num_of_bids”, and their corresponding values are the data that is recorded in each point. The “stock” column is a tag because it helps describe the fields. As said previously, tags are indexed. If a user wanted to get all the points where the stock was “AAPL”, it could easily and quickly do so. However, if it was a field, and was not indexed, it would have to first search through all the data in the bucket and find the data where the stock is AAPL. Remember that InfluxDB is schemaless. A measurement is more formally a container for tags, fields, and timestamps. It is a way to view the data. When a user writes points, they also specify which measurement the point belongs to.

After looking at tags, fields, and measurements, the next piece of InfluxDB’s architecture is a series. A series is a collection of points that share a tag set (combination of all the tag keys), a measurement, and a single field key. InfluxDB maintains an in-memory index of all the series so that it can quickly run queries and provide the measurements to the user. The series cardinality is the number of unique combinations of measurements, tag sets and field keys.

In time series, data often gets less important, the less recently it was written. Older data is generally used less frequently than newer data. To improve data storage and querying, InfluxDB implements a retention policy. A retention policy dictates how long the data is stored, and how many copies of the data is made. This is used for when data is stored into shards by InfluxDB. InfluxDB creates shards according to blocks of time, and each of these shards is mapped to an underlying storage engine database.Data is stored using InfluxDB’s custom time structured merge tree (TSM tree). It is similar to a log structured merge tree. The TSM tree stores series data that is sorted and compressed.

There are multiple parts to the storage engine. There is the In-memory index, which allows for fast access to the series data. The Write ahead log (WAL) stores writes and allows for later. TSM files store compressed series data. The cache is an in-memory representation of the data stored in WAL, and it is queried and merged with TSM files at runtime. Filestore handles access to TSM files on disk, and it makes sure TSM files are installed automatically when existing ones are replaced, and removes unused TSM files. The Compactor makes Cache and TSM data more

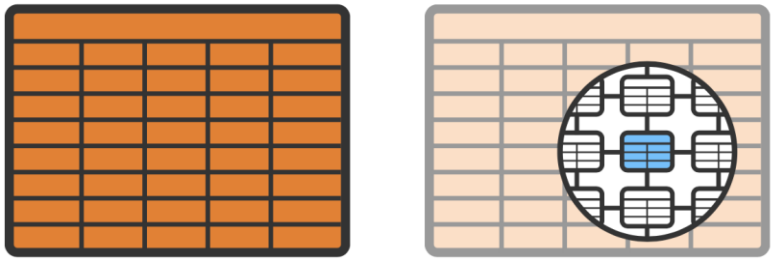
read-optimized by compressing series, removing deleted data, optimizing indices and merging smaller files into larger ones. The Compaction planner tells the Compactor which files to compact and prevents potential interference from concurrent compactions. There is also compression that change their compression strategy based on the data type, and writers and readers which help read and write the different data formats.

3.2.2 TimescaleDB Architecture

As discussed above, TimescaleDB is an extension of PostgreSQL, so its architecture is similar to PostgreSQL. In this section we focus on the architecture which helps TimescaleDB handle time-series data.

From a user point of view, TimescaleDB implements **Hypertables** architecture for time-series data. Hypertable is an abstraction of many small individual tables, which are **chunks**. We will introduce hypertables and chunks below.

Formally, a hypertable is the abstraction of a single continuous table across all space and time intervals, such that users can query it via standard SQL. The relationship of a hypertable and chunks is shown in the figure below.



Chunks are splitted from a hypertable. Each chunk is an individual standard database table. Each chunk stores data of a time period, and chunks do not cover each other.

To improve the reliability, there are some strategies when chunks are implemented. For example, to avoid thrashing issues when modifying location in B-tree, chunks are right-sized B-trees, which ensures all table's indexes can reside in memory during inserts. Besides, there are some strategies of chunks to improve the performance of the database. For instance, by avoiding overly large chunks, the database can avoid expensive "vacuuming" operations when deleting data, because it is often easier to drop chunks than deleting rows in chunks.

In addition, TimescaleDB supports distributing hypertables over multiple nodes. For distributed hypertables, some chunks are not stored locally, so there are some limitations for distributed hypertables. For example, with distributed hypertables, continuous aggregates are not supported, and reordering chunks is not supported.

3.3 Operational Capabilities

3.3.1 InfluxDB Operational Capabilities

InfluxDB boasts a large selection of packages and functions users can choose from. It consists of functions created by InfluxData and contributions from InfluxDB users.

The built-in Flux language supports functions for inputting and outputting data (either from the UI or from the terminal), as well as transformation functions such as aggregate functions (max, min, avg etc) and conversion functions (toBool, toInt, toString, bytes etc). It also has selector functions such as “first”, where users select the most recent point, “lowest min”, where users take the lowest n values from an input stream over a given amount of time, and “sample”, a function that returns a random sample of the input. Users can also check the current time and have their system sleep for a given time.

Within the UI, users are able to explore their data using the InfluxDB Data Explorer. For newer users who do not have a good grasp on Flux, InfluxDB provides a dropdown menu style option for creating queries. Users can use this to query their data and also select the time frame in which they would like to see their data. The UI has an option for visualizing data as different graphs, heatmaps, histograms, and more.

Next we will take a look at some of the multiple packages Flux has that users are provided. The array package allows users to create tables/measurements using an array as input. The CSV package allows users to import their data as csv. The data package gives users a larger set of constants and functions to use when handling the dates. The dictionary package allows users to create dictionaries and store/handle data. The Flux HTTP package gives functions for transferring data using the HTTP protocols (such as authentication and specifying the end point for data). The Flux InfluxDB monitor package allows the user to monitor incoming data and changes to the data. This includes functions such as “monitor.check”, which checks incoming data and assigns a level to it (warning, ok, info, crit) based on a function. (if water level is >= 80, assign level of crit etc). “Monitor.deadman” is a function that detects when data has stopped being reported. It also tells the user how long it has been since the last input from a datastream. “Monitor.notify” logs down any notification the user wanted note of and then sends a notification to some receiving endpoint.

Flux JSON package gives users the ability to convert values into JSON files. The math package gives users many mathematical constants (ln2, pi, phi etc) and functions (arctan, mod, floor, ceiling, sin, cos, sinh etc). The profiler package gives users tools to time their queries and operations down to the nanosecond. The regular expressions package gives users the tools for handling strings in regular expressions such as replacing and identifying specific strings. Flux packages even include an SQL package that translates SQL queries into Flux queries and vice versa.

The InfluxDB packages provided for Flux give users various analytics about their data, such as determining the series cardinality. More can be found on their website regarding those packages. Flux also has an experimental package.This package contains functions and other tools that work but are subject to change at any time.

3.3.2 TimescaleDB Operational Capabilities

Besides basic SQL commands, TimescaleDB also supports several advanced analytic queries, this includes queries supported by PostgreSQL and TimescaleDB unique SQL functions. In this report we focus on TimescaleDB unique SQL queries.

Time Bucket: TimescaleDB’s time_bucket query accepts time intervals and an optional offset, returns the bucket start time.

First, Last: TimescaleDB’s first and last queries return the first or last value of a column.

Histogram: TimescaleDB's histogram query accepts a lower bound, an upper bound and number of bins, returns the number of elements for each bin in histogram.

Gap Filling: "gap filling" refers to that filling data to a time period (usually record a "0" for missing period) when no data was recorded during that period. TimescaleDB's time_bucket_gapfill accepts some interval, and generates a series of time buckets across a specific period.

Last Observation Carried Forward (locf): This function is for the situation when the data collection only records rows when the actual value changes, but still all data have to be displayed. In this case TimescaleDB's locf query returns the last observed value.

3.4 Computational Capabilities

3.4.1 Introduction of Time Series Benchmark Suite (TSBS)

Time Series Benchmark Suite (TSBS) is a well known collection of Go programs, which are used to benchmark the read/write performance of time series databases. It was initially developed by TimescaleDB developers for database benchmarking. Since it is open source, many time series databases have evolved. Now it is a strong benchmarking suite supporting 10 various time series databases including Akumuli, Cassandra, ClickHouse, CreateDB, InfluxDB, MongoDB, SiriDB, TimescaleDB, Timestream, VictoriaMetrics. It has also been used as a reference in other InfluxDB and TimescaleDB research papers to compare their IoT performance [9].

TSBS supports two use cases for InfluxDB and TimescaleDB: "Dev ops" case and "Internet of Things (IoT)" case. "Dev ops" is meant to simulate a real world dev ops scenario. "IoT" is meant to simulate the data load in an IoT environment. There are 15 types of "dev ops" queries and 12 types of "IoT" queries which can be generated using TSBS. For example, "single-groupby-5-1-12" is a query type of "dev ops" use case, which is a simple aggregate function on 5 metrics for 1 host, every 5 mins for 12 hours. "Long-driving-sessions" is a query type of "IoT" use case, which is to find trucks that haven't rested for at least 20 mins in the last 4 hours.

There are three phases to benchmark time series databases using TSBS: data and queries generation, data insertion performance benchmarking and queries performance execution.

For data generation, TSBS provides a method to generate data. We can customize the arguments to generate unique datasets. For example, "format" is the name of the time series database which the data will be used on. "Scale" is the number of devices which is simulated in "dev ops" use cases. "Timestamp-start" and "timestamp-end" are used to define a period of time, and "log-interval" is used to define the duration between data points. Similar to data generation, TSBS provides a method and a script to generate queries. Since the script can generate multiple types of queries at once, in this experiment we used the script to generate queries.

For data insertion performance benchmarking, TSBS also provides a method, and a script for local tests. We can still customize the arguments to simulate different testing scenarios.

For query execution performance benchmarking, TSBS provides a method and a script to generate scripts to test multiple query types at once.

The detailed benchmarking steps using TSBS are introduced in the technique report attached.

3.4.2 Experient Deployment Setup

In this experiment we used Amazon Web Services (AWS) EC2 instance for our testing. The hardware information of the EC2 instance is shown in the table below.

CPU	Cpu: Intel(R) Xeon(R) CPU E5-2676 v3 @ 2.40GHz
Memory SIze	1 GiB
Disk	30 GiB SSD
Operating System	Ubuntu 20.04

And the table below is the version of softwares we used in this experiment. Since we use TimescaleDb, which is an extension of PostgreSQL, and TSBS, which is a Go program, we also give the version of PostgreSQL and Go here.

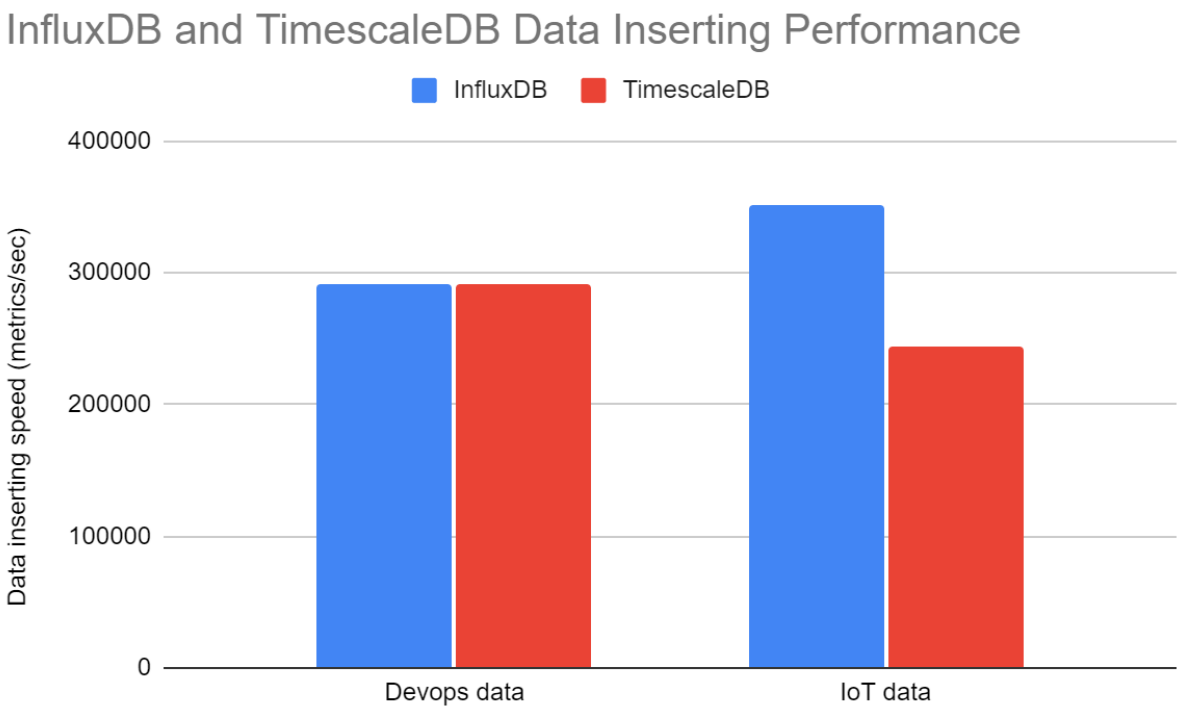
InfluxDB	v1.8
PostgreSQL	v13
TimescaleDB	v2.2.0
Go	v1.16.3 linux/amd64

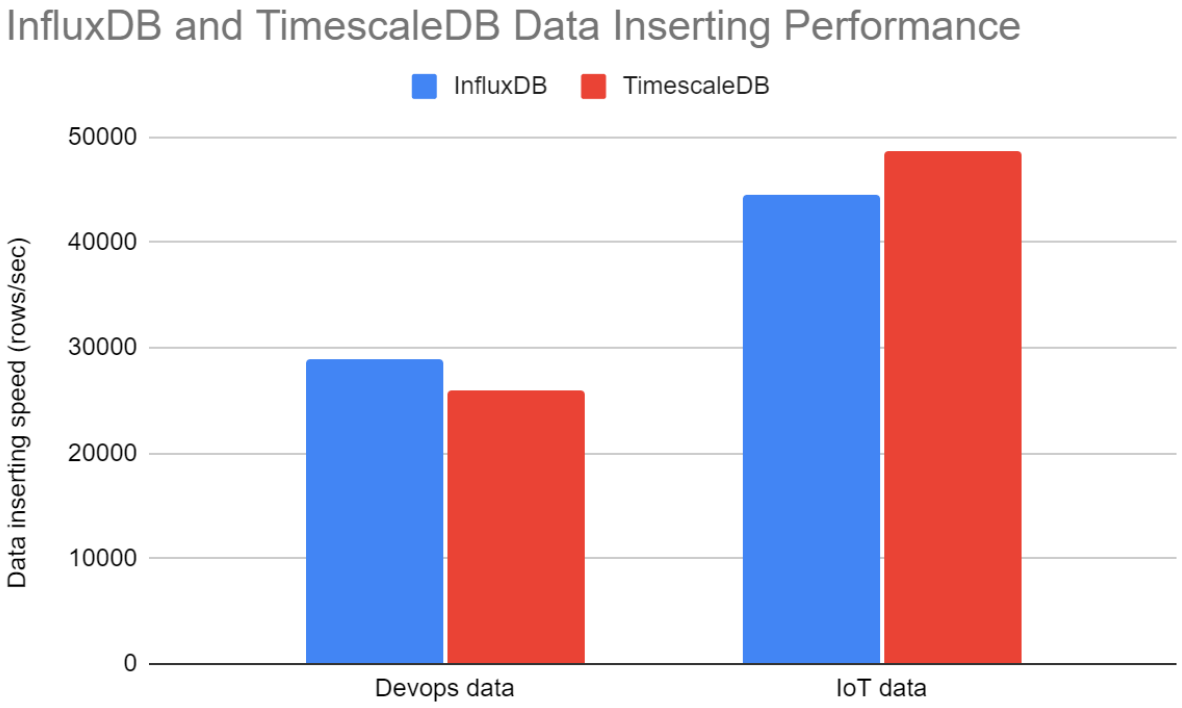
3.4.2 Results Evaluation

1) Data Insertion Performance

In this project we tested the data insertion performance on InfluxDB and TimescaleDB for both “dev ops” and “IoT” use cases, and the results are shown as the table and figures below.

Use case	InfluxDB	TimescaleDB
Dev ops data	291807.43 metrics/sec 29004.54 rows/sec	292049.35 metrics/sec 26024.20 rows/sec
IoT data	351518.15 metrics/sec 44462.95 rows/sec	243367.49 metrics/sec 48675.63 rows/sec



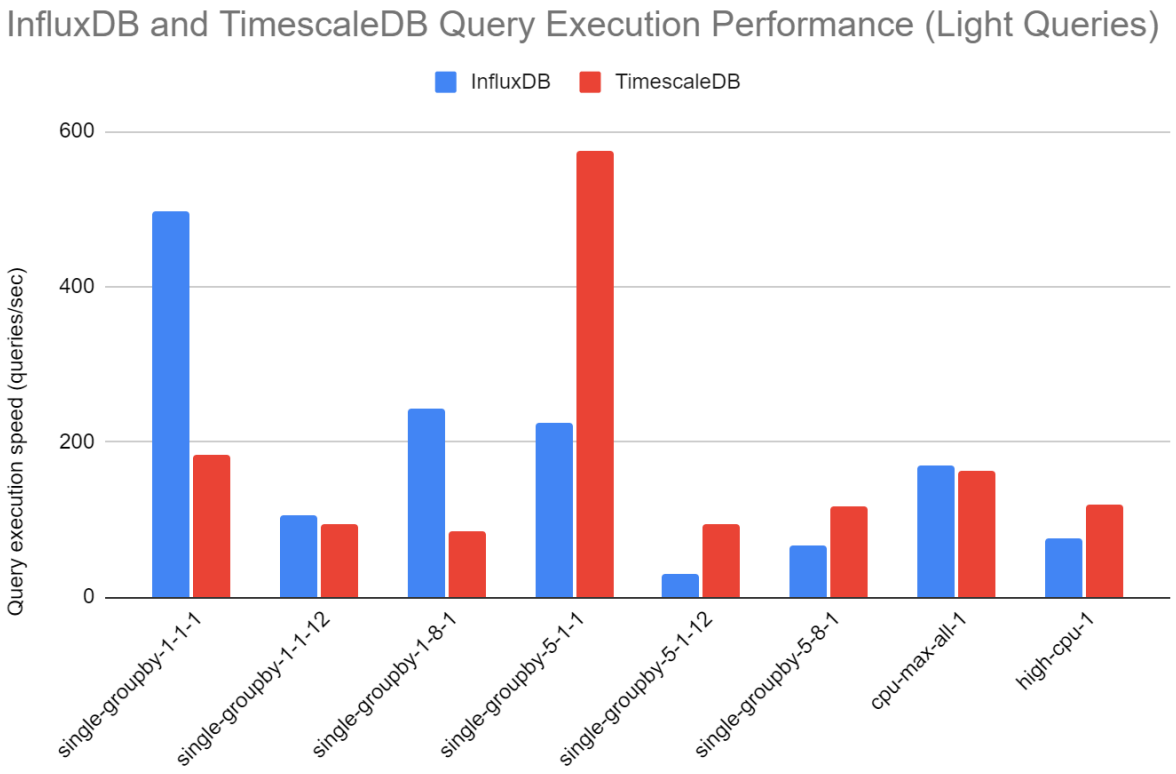


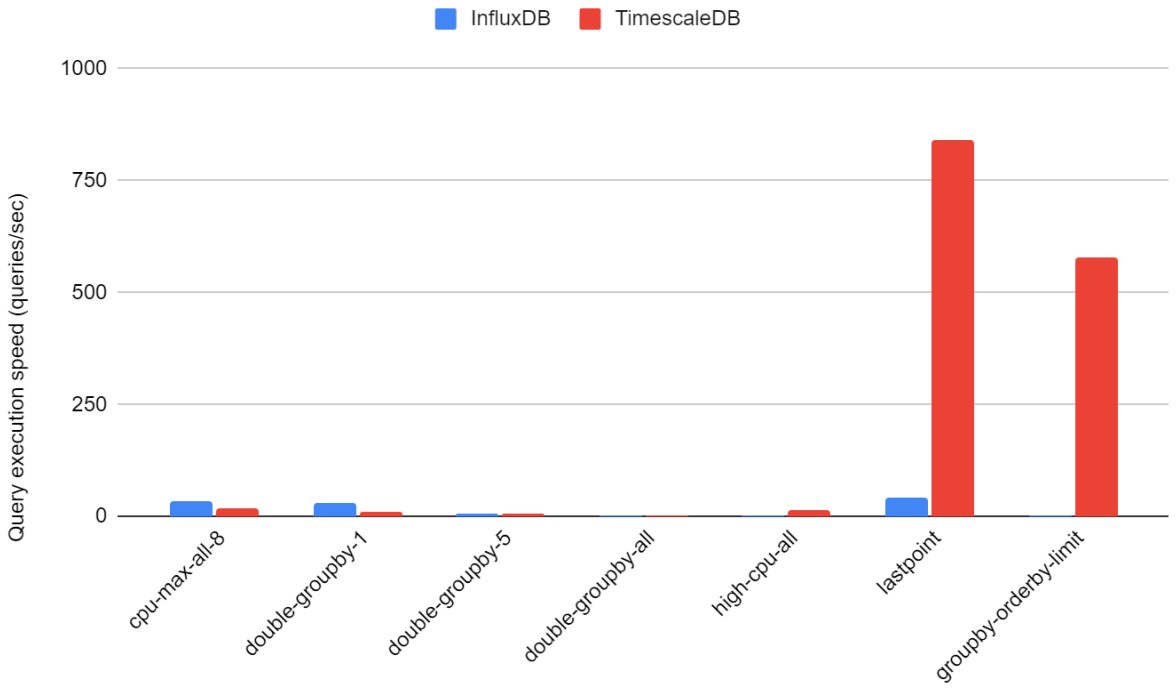
As we can see from the results above, for “dev ops” use cases, two databases have similar data insertion performance with about 290k metrics inserted per second and about 27k rows inserted per second. However, for “IoT” data inserting, InfluxDB performs better than TimescaleDB, with about 351k metrics inserted per second but TimescaleDB can only insert about 243k metrics per

second. We think that InfluxDB has better performance maybe because that the TSM engine of InfluxDB allows for high igest speed and data compression.

2) Query Execution Performance for “Dev ops” Use Case
For query execution performance benchmarking, we divided into two parts: on “dev ops” use case and on “IoT” use case. This section will discuss the evaluation of “dev ops” use cases. We tested all 15 query types for both databases under “dev ops” use case. The results are shown in the table and figures below.

Query type	InfluxDB performance (number of queries executed/sec)	TimescaleDB performance (number of queries executed/sec)
single-groupby-1-1-1	498.12	184.62
single-groupby-1-1-12	106.31	93.68
single-groupby-1-8-1	243.41	85.66
single-groupby-5-1-1	224.98	574.37
single-groupby-5-1-12	29.82	95.1
single-groupby-5-8-1	67.76	117.22
cpu-max-all-1	170.69	161.88
cpu-max-all-8	32.65	16.66
double-groupby-1	28.7	9.21
double-groupby-5	6.32	5.88
double-groupby-all	3.21	2.76
high-cpu-all	1.62	13.51
high-cpu-1	76.41	120.25
lastpoint	41.13	841.94
groupby-orderby-limit	2.06	578.67


InfluxDB and TimescaleDB Query Execution Performance (Light Queries)
InfluxDB
TimescaleDB
Query execution speed (queries/sec)
600
400
200
0
single-groupby-1-1-1
single-groupby-1-1-12
single-groupby-1-8-1
single-groupby-5-1-1
single-groupby-5-1-12
single-groupby-5-8-1
cpu-max-all-1
high-cpu-1


InfluxDB and TimescaleDB Query Execution Performance (Heavy Queries)
InfluxDB
TimescaleDB
Query execution speed (queries/sec)
1000
750
500
250
0
cpu-max-all-8
double-groupby-1
double-groupby-5
double-groupby-all
high-cpu-all
lastpoint
groupby-orderby-limit

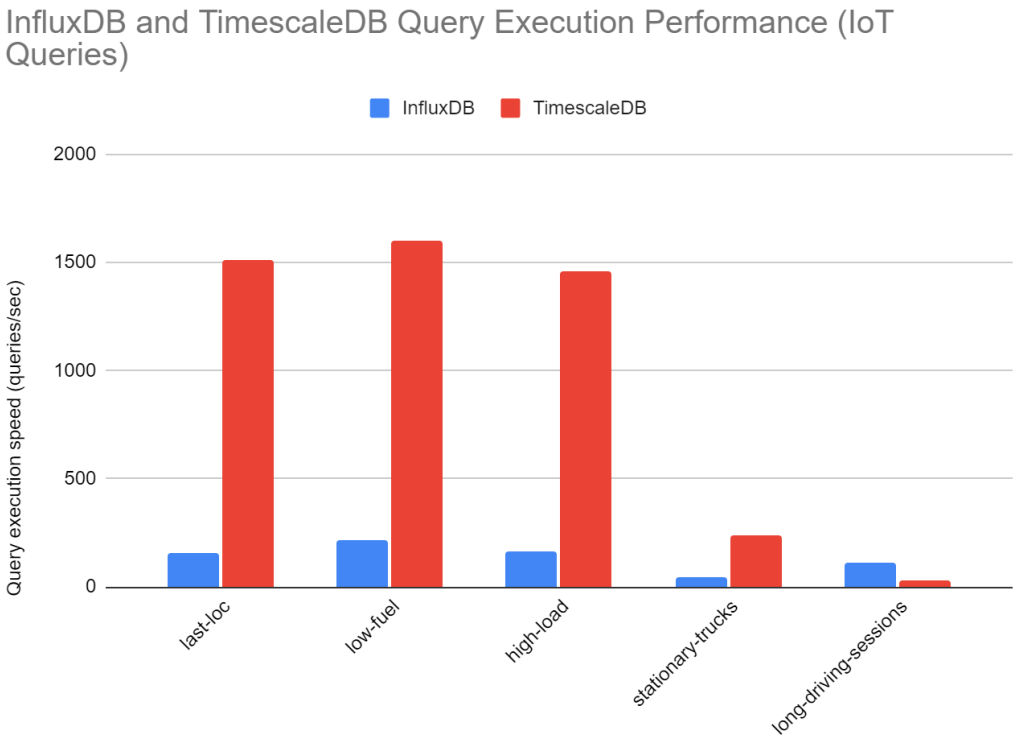
As shown in the figures, we divide 15 types of queries into “light” queries and “heavy” queries for easier to evaluate. “Light” queries typically take milliseconds to be executed while “heavy” queries may take a few seconds to be executed.

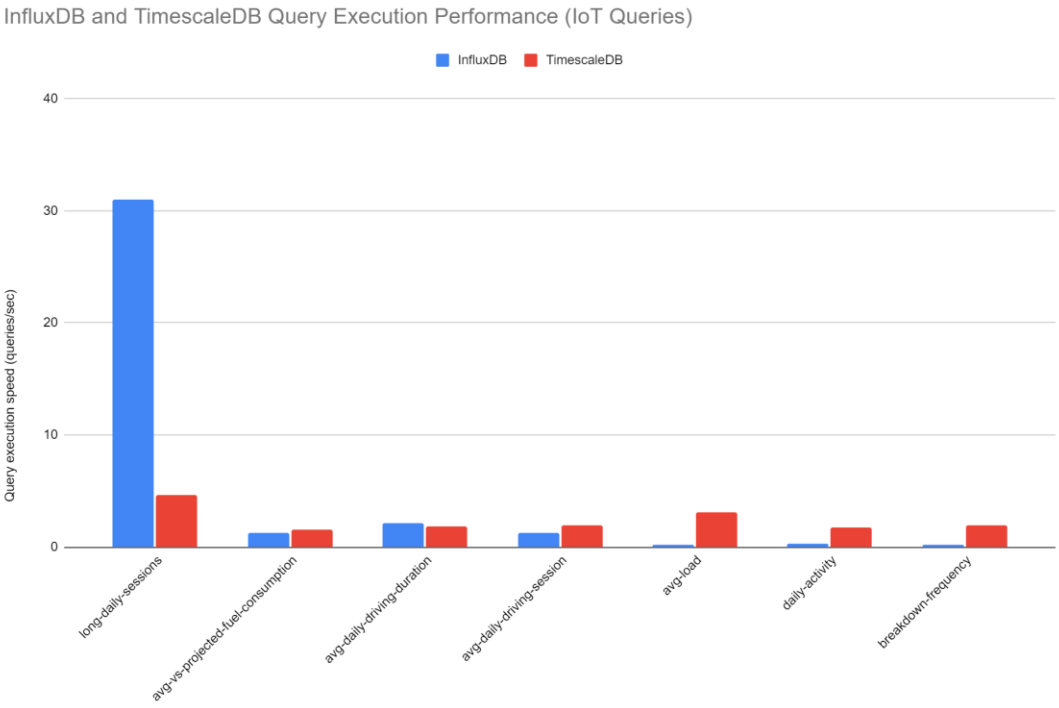
For “light” queries we can observe that InfluxDb performs better when the aggregate function on 1 metric (single-groupby-1-1-1, single-groupby-1-1-12, single-groupby-1-8 -1), and TimescaleDB is more efficient when the number of metrics increases (single-groupby-5-1-1, single-groupby-5-1-12, single-groupby-5-8-1). Our guess of this results is that since InfluxDB uses expressive queries developed by themselves, it might be easy to query aggregated data on a single metric.

For “heavy” queries, we can observe that InfluxDB execute queries inefficiently on all “heavy” queries. Besides, TimescaleDB performs similarly with InfluxDB except “last point” queries and “groupby-orderby-limit” queries. We think it might be because both queries use the “last” row in time series, and TimescaleDB is able to return the last row very quickly using the index.

3) Query Execution Performance for “IoT” Use Case
The result of query execution performance of “IoT” use case is shown as table and figures below.

Query type	InfluxDB performance (number of queries executed/sec)	TimescaleDB performance (number of queries executed/sec)
last-loc	158.36	1510.09
low-fuel	216.33	1603.95
high-load	163.54	1457.69
stationary-trucks	44.02	235.29
long-driving-sessions	112.86	28.15
long-daily-sessions	30.98	4.63
avg-vs-projected-fuel-consumption	1.28	1.53
avg-daily-driving-duration	2.12	1.84
avg-daily-driving-session	1.26	1.89
avg-load	0.19	3.07
daily-activity	0.3	1.7
breakdown-frequency	0.17	1.96





As shown in the results above, we can observe that TimescaleDB performs better than InfluxDB on most IoT query types. IoT is more efficient on “long-driving-sessions” and “long-daily-sessions” than TimescaleDB. From our perspective, many type of queries of “IoT” use cases uses lastpoint

operations (“last-loc”, “low-fuel”, ‘high-load’, “stationary-trucks”), which is what TimescaleDB’s strength, as we discussed in the “dev ops” queries execution performance evaluation.

4. Conclusion and Future Work

In this project, we looked at 2 different Time Series Databases, explored their architecture, functionality, and computational capabilities. Through our research and testing, we determine that InfluxDB has a much larger breadth of functionality due to its numerous packages and custom SQL-like scripting language. It also has many other uses for data analytics and data exploration. From our benchmarking, InfluxDB shows to have much faster single host query execution than TimescaleDB. In the cases where users would want to perform data analysis and draw information from the data, we would highly recommend InfluxDB over TimescaleDB for these reasons. However, TimescaleDB performs better with multiple host queries and has faster last point retrieval (getting the most recent input of data). Because of this we would recommend TimescaleDB to users and organizations who need fast and reliable database servers for multiple users and do not need much analytical functionality.

Due to time constraints, we were unable to perform tests on multiple nodes. We also would have like to look more deeply into the analytical tools InfluxDB provides to maybe compare them with other TimeSeries databases that had similar tools. During our research, we also learned that the TSBS queries could be altered, and we will be including the links to those sections from the TSBS github in our technical report.

5. References

[1] Naqvi, S. N. Z., Yfantidou, S., & Zimányi, E. (2017). Time series databases and influxdb. Studienarbeit, Université Libre de Bruxelles, 12.

[2] Grzesik, P., & Mrozek, D. (2020, June). Comparative Analysis of Time Series Databases in the Context of Edge Computing for Low Power Sensor Networks. In International Conference on Computational Science (pp. 371-383). Springer, Cham.

[3]Bader, A., Kopp, O., & Falkenthal, M. (2017). Survey and comparison of open source time series databases. Datenbanksysteme für Business, Technologie und Web (BTW 2017)-Workshopband.

[4] Ganz, J., Beyer, M., & Plotzky, C. (2017). Time-series based solution using InfluxDB. no. i.

[5] Sanaboyina, T. P. (2016). Performance evaluation of time series databases based on energy consumption.

[6] Kiefer, R. (2017). TimescaleDB vs. PostgreSQL for time-series: 20x higher inserts, 2000x faster deletes, 1.2 x-14,000 x faster queries. Timescale Blog.

[7] https://db-engines.com/en/ranking/time+series+dbms

[8] https://docs.timescale.com/latest/introduction

[9] Rui Liu, Jun Yan. (2019). Benchmarking Time Series Databases with IoTDB-Benchmark for IoT scenarios