

**ΟΙΚΟΝΟΜΙΚΟ  
ΠΑΝΕΠΙΣΤΗΜΙΟ  
ΑΘΗΝΩΝ**



**ATHENS UNIVERSITY  
OF ECONOMICS  
AND BUSINESS**

**Bachelor Thesis**

**in**

**Computer Science**

**Time series databases: Study & Benchmarking**

Vasileios-Ilias Drouzas

Supervisor: Prof. Yannis Kotidis

Athens, June 2022

# Contents

|   |          |
|---|----------|
| <b>1 Abstract</b>   | <b>4</b> |
| <b>2 Acknowledgements</b>                                     | <b>5</b> |
| <b>3 Introduction to time series</b>                          | <b>6</b> |
| 3.1 What is a time series? . . . . .                          | 6        |
| 3.2 Where is time series data used? . . . . .                 | 7        |
| 3.3 Why time series data? . . . . .                           | 7        |
| 3.4 Why do i need a time series database? . . . . .           | 7        |
| <b>4 Studying some popular time series databases</b>          | <b>8</b> |
| 4.1 Ranking . . . . .   | 8        |
| 4.2 Choosing databases . . . . .                              | 9        |
| 4.3 InfluxDB . . . . .  | 10       |
| 4.3.1 Data elements . . . . .                                 | 10       |
| 4.3.2 InfluxQL vs Flux . . . . .                              | 12       |
| 4.3.3 Influx Line Protocol . . . . .                          | 13       |
| 4.3.4 How is data stored? . . . . .                           | 14       |
| 4.3.5 Downsampling . . . . .                                  | 14       |
| 4.3.6 Storage engine . . . . .                                | 14       |
| 4.3.7 Data replication . . . . .                              | 16       |
| 4.3.8 Pros-cons of InfluxDB . . . . .                         | 18       |
| 4.4 TimescaleDB . . . . .                                     | 19       |
| 4.4.1 Partitioning with chunks . . . . .                      | 19       |
| 4.4.2 Scaling . . . . .                                       | 20       |
| 4.4.3 Continuous aggregates . . . . .                         | 21       |
| 4.4.4 Data retention . . . . .                                | 21       |
| 4.4.5 Data model . . . . .                                    | 22       |
| 4.5 QuestDB . . . . .   | 23       |
| 4.5.1 Data model . . . . .                                    | 23       |
| 4.5.2 Partitioning . . . . .                                  | 23       |
| 4.5.3 Indexing . . . . .                                      | 24       |
| 4.5.4 InfluxLineProtocol and comparison to InfluxDB . . . . . | 24       |

|  |           |
|--|-----------|
| 4.5.5 Schemaless ingestion using the ILP . . . . .   | 26        |
| 4.6 Prometheus . . . . .   | 29        |
| 4.6.1 Data model . . . . .   | 29        |
| 4.6.2 Storage . . . . .  | 31        |
| 4.6.3 Alerting . . . . .   | 31        |
| 4.7 Recap: InfluxDB vs Prometheus vs TimescaleDB vs QuestDB . . . .                                | 33        |
| 4.8 A comparison between some popular time series databases . . . . .                              | 34        |
| 4.9 Comparing relational and time series databases . . . . .                                       | 36        |
| <b>5 Benchmarking some popular time series databases</b>   | <b>39</b> |
| 5.1 Introduction to TSBS . . . . .   | 39        |
| 5.2 Ingestion performance: InfluxDB vs TimescaleDB vs QuestDB . . . .                              | 41        |
| 5.3 Query execution performance . . . . .  | 42        |
| 5.4 How cardinality impacts ingestion/query performance rates . . . . .                            | 47        |
| 5.4.1 Impacting ingestion rates . . . . .  | 48        |
| 5.4.2 Impacting query execution rates . . . . .  | 49        |
| 5.5 How multi-threading affects ingestion . . . . .  | 49        |
| 5.6 How multi-threading affects query execution . . . . .  | 50        |
| 5.7 Materialized views . . . . .   | 51        |
| 5.7.1 Materialized views in TSBS . . . . .   | 51        |
| 5.7.2 Exploring TimescaleDB's continuous aggregates and InfluxDB's<br>continuous queries . . . . . | 52        |
| 5.8 Benchmarking - conclusion . . . . .  | 57        |
| <b>6 Conclusion</b>  | <b>59</b> |
| <b>7 Appendix</b>  | <b>60</b> |
| 7.1 Python script to handle message formatting in QuestDB . . . . .                                | 60        |
| 7.2 Execution instructions for TSBS . . . . .  | 61        |
| <b>8 References</b>  | <b>64</b> |

# **1 Abstract**

In our days, the amount of data is increasing, which brings the need for a storage system (known as a database). Time series data refer to data changing according to time. There are specific databases that are distinct to the traditional relational databases designed for this type of data, time series databases. In this thesis, the most popular time series databases are examined in depth and they are compared with practical experiments in order to conduct performance tests in data ingestion and query execution.

## **2 Acknowledgements**

First of all, I would like to thank my supervisor Prof. Yannis Kotidis for his advice and support throughout this thesis work. His detailed comments helped me a lot throughout this thesis.

Furthermore, I would like to thank my family and friends for their support at every step of my education at Athens University of Economics and Business.

### 3 Introduction to time series

Before moving on to the main section of this thesis, regarding the time series databases, an introduction to time series is needed. What is a time series, where is it used and why there is need specifically for time series data? Finally, why are time series databases needed and we don't just use a typical database? These questions are answered in this chapter.

#### 3.1 What is a time series?

A time series is a collection of observations obtained through ordered measurements over time. For example, weather records, economic indicators and health metrics are all time series data. Below, we can see a graph depicting the range of temperature in a certain period of time (from 12 am to 12 pm).

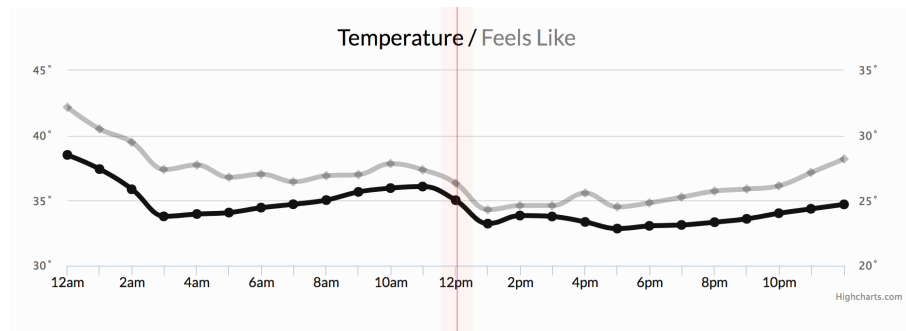


Figure 1: Weather changes.

Some other examples include monthly sales of a business, number of births each year, rate of heartbeat, network monitoring data etc.

How to distinguish time series data? Time series data express measurements over time. Time is the main comparative tool to monitor and analyze the data. A simple way to determine if a dataset contains time series is to check if one of the axes is time.

Time series data can be distinguished in two categories: univariate and multivariate. A univariate time series consists of single observations recorded sequentially over time (univariate data depends only on one variable). A multivari-

ate time series, on the other hand, extends the capabilities of a univariate time series and involves two or more variables.

### **3.2 Where is time series data used?**

Time series can be applied in forecasting (found in statistics, econometrics, meteorology, seismology and geophysics), in clustering/classification & anomaly detection (found in data mining, machine learning) and in signal detection/estimation (found in communications).

### **3.3 Why time series data?**

An example: Imagine one transferring \$10 in another account, and receiving \$10 in reverse, the balances look the same as before this process. The traditional bank database may think “nothing has changed this month”. But if when having a time-series database, the bank could spot this transaction and if it happens often, they can conclude “these 2 accounts have some kind of relationship”.

One second example is about average temperature per day. It may only vary slightly from day to day in any location, but what if the environmental factors change dramatically in this period? Knowing how temperature has changed throughout the day, combined with cloud cover, wind, and humidity levels can improve the ability to model and also make weather forecasts.

These two examples show how modern time series data is different from traditional relational data. This data doesn't just include time as a metric, but as a primary component that is used to assist in data analysis and derivation of meaningful insights.

### **3.4 Why do i need a time series database?**

One might ask: Why can't I just use a “normal” (i.e., non-time-series) database? Well.. in fact it can technically happen, some people do. But time series databases (TSDBs) offer you something more than traditional databases: scale and usability.

**Scale:** Time series data accumulates so quickly that normal databases cannot handle the weight. Traditional databases have poor performance when it

comes to large datasets. While TSDBs offer performance metrics that can be achieved only when time is treated as the first class citizen (traditional databases would have time as an extra field in a table -and nothing more- so processing efficiency would decrease dramatically). With scalability , storage can be increased.

**Usability:** TSDBs include operations such as retention policies, continuous queries, queries that involve time aggregations etc. These features cannot be present at a traditional database model and make the analysis of data significantly easier.

Due to these reasons, developers created specific purpose databases in order to maximize performance and query capabilities, the TSDBs.

## **4 Studying some popular time series databases**

After introducing time series data, a detailed study of the most popular time series databases follows. In this chapter, a thorough analysis on TSDBs will be conducted. Three TSDBs will be examined and some more will be presented briefly to discuss the main differences between them.

### **4.1 Ranking**

In the DB engines [website](#) there is a current ranking of time series databases. The ranking is updated monthly and the comparison criteria is described on the website. In June 2022, the scores were as follows:



| Rank     |          |          | DBMS                   | Database Model           | Score    |          |          |
|----------|----------|----------|------------------------|--------------------------|----------|----------|----------|
| Jun 2022 | May 2022 | Jun 2021 |                        |                          | Jun 2022 | May 2022 | Jun 2021 |
| 1.       | 1.       | 1.       | InfluxDB               | Time Series, Multi-model | 29.86    | +0.30    | +1.22    |
| 2.       | 2.       | 2.       | Kdb+                   | Time Series, Multi-model | 9.12     | +0.13    | +0.80    |
| 3.       | 3.       | 3.       | Prometheus             | Time Series              | 6.32     | +0.19    | +0.40    |
| 4.       | 4.       | 4.       | Graphite               | Time Series              | 5.35     | -0.10    | +0.71    |
| 5.       | 5.       | 5.       | TimescaleDB            | Time Series, Multi-model | 4.56     | -0.14    | +1.32    |
| 6.       | 6.       | 6.       | Apache Druid           | Multi-model              | 2.94     | -0.06    | +0.13    |
| 7.       | 7.       | 7.       | RRDtool                | Time Series              | 2.43     | -0.07    | -0.26    |
| 8.       | 8.       | 8.       | OpenTSDB               | Time Series              | 1.86     | +0.02    | +0.02    |
| 9.       | 9.       | 11.      | DolphinDB              | Time Series, Multi-model | 1.65     | 0.00     | +0.68    |
| 10.      | 10.      | 9.       | Fauna                  | Multi-model              | 1.33     | -0.04    | -0.34    |
| 11.      | 11.      | 10.      | GridDB                 | Time Series, Multi-model | 1.28     | +0.05    | +0.13    |
| 12.      | 12.      | 16.      | QuestDB                | Time Series, Multi-model | 1.24     | +0.05    | +0.69    |
| 13.      | 13.      | 14.      | Amazon Timestream      | Time Series              | 1.02     | +0.05    | +0.39    |
| 14.      | 14.      |          | TDengine               | Time Series, Multi-model | 0.96     | +0.05    |          |
| 15.      | 16.      | 12.      | KairosDB               | Time Series              | 0.65     | +0.00    | -0.13    |
| 16.      | 15.      | 13.      | eXtremeDB              | Multi-model              | 0.64     | -0.04    | -0.10    |
| 17.      | 17.      | 17.      | VictoriaMetrics        | Time Series              | 0.56     | -0.02    | +0.09    |
| 18.      | 18.      | 15.      | Raima Database Manager | Multi-model              | 0.48     | -0.02    | -0.11    |
| 19.      | 19.      | 20.      | IBM Db2 Event Store    | Multi-model              | 0.47     | -0.02    | +0.13    |
| 20.      | 20.      | 25.      | Apache IoTDB           | Time Series              | 0.43     | +0.01    | +0.27    |
| 21.      | 21.      | 24.      | M3DB                   | Time Series              | 0.41     | +0.05    | +0.21    |
| 22.      | 25.      | 22.      | Heroic                 | Time Series              | 0.25     | +0.03    | +0.01    |
| 23.      | 23.      | 19.      | Axibase                | Time Series              | 0.24     | -0.01    | -0.11    |
| 24.      | 24.      | 18.      | Riak TS                | Time Series              | 0.23     | -0.01    | -0.12    |
| 25.      | 22.      | 21.      | Alibaba Cloud TSDB     | Time Series              | 0.23     | -0.03    | -0.02    |
| 26.      | 26.      | 28.      | Warp 10                | Time Series              | 0.19     | +0.01    | +0.08    |
| 27.      | 27.      |          | ArcadeDB               | Multi-model              | 0.13     | -0.03    |          |
| 28.      | 28.      | 23.      | Quasardb               | Time Series              | 0.09     | -0.02    | -0.13    |
| 29.      | 29.      | 29.      | Bangdb                 | Multi-model              | 0.04     | +0.00    | -0.06    |
| 30.      | 30.      | 32.      | Hawkular Metrics       | Time Series              | 0.02     | +0.00    | +0.01    |
| 31.      | 33.      | 30.      | SiriDB                 | Time Series              | 0.01     | +0.01    | -0.07    |
| 32.      | 32.      | 26.      | Machbase               | Time Series              | 0.01     | 0.00     | -0.14    |
| 33.      | 31.      | 27.      | Blueflood              | Time Series              | 0.00     | -0.01    | -0.11    |
| 34.      | 33.      | 33.      | Hyprcubd               | Time Series              | 0.00     | ±0.00    | ±0.00    |
| 34.      | 33.      | 33.      | IRONdb                 | Time Series              | 0.00     | ±0.00    | ±0.00    |
| 34.      | 33.      | 33.      | Newts                  | Time Series              | 0.00     | ±0.00    | ±0.00    |
| 34.      | 33.      | 33.      | NSDb                   | Time Series              | 0.00     | ±0.00    | ±0.00    |
| 34.      | 33.      | 31.      | SiteWhere              | Time Series              | 0.00     | ±0.00    | -0.03    |
| 34.      | 33.      | 33.      | Yanza                  | Time Series              | 0.00     | ±0.00    | ±0.00    |

Figure 2: Ranking of TSDBs (June 2022).

## 4.2 Choosing databases

For our analysis, we will pick four of the most important time series databases: InfluxDB, which is the most popular time series database and widely used in applications, TimescaleDB which is also a popular choice and has the benefit of relational format (familiar for most database users), QuestDB which is reported to be the fastest time series database and a monitoring system, different from these previous three databases, called Prometheus. Later in the section we will discuss briefly about more time series databases, such as OpenTSDB, GridDB

and relational databases like PostgreSQL, MySQL Community Server in order to compare them for a variety of characteristics.

### 4.3 InfluxDB

InfluxDB is an open source time series database developed by InfluxData. It is written in Go programming language and is especially designed to handle time series data. InfluxDB contains an SQL-like query language, called InfluxQL and an alternative to InfluxQL, called Flux, which can overcome certain InfluxQL limitations. There are two versions of InfluxDB:

- a. The open source version (a.k.a. the TICK Stack). It consists of four components: Telegraf, InfluxDB, Chronograf, Kapacitor and can run on cloud and locally.
- b. Closed versions, such as InfluxCloud which offer extra functionalities (scalability, backup, restore etc.)

InfluxDB is distributed by design. This provides reliability, as data is located in multiple places, and scalability for both write and query load.

#### 4.3.1 Data elements

The most common data elements in InfluxQL are:

- **Timestamp.** Exact time snapshot, stored in epoch nanosecond format.
- **Measurement.** Measurement names are strings. A measurement can act as a container for tags, fields and timestamps. A measurement is used to describe the type of data and is similar to an SQL table. An explicit schema constrains the shape of data that can be written to that measurement.
- **Fields.** They consist of:
  - a) field keys (string that represents the name of the field) which store meta-data,
  - b) field values (the value of the associated field. Can be string, float, integer or boolean) which are the actual data,

c) field sets (a collection of field key-value pairs associated with a timestamp).

Fields, while they are required to exist, they are not indexed. Queries which filter field values have to scan all field values in order to match query conditions. They are similar to unindexed columns in an SQL table. Thus, one would rather store commonly queried data in tags (see next).

- **Tags.** They consist of tag keys (record metadata), tag values (also record metadata) and tag sets.

In contrast to fields, tags are optional but indexed. So queries on tags run faster than queries on fields. They are similar to indexed columns in an SQL table.

- **Series.** A collection of data with common retention policy<sup>1</sup>, measurement and tag set.
- **Point.** The field set in the same series with a specific timestamp. Similar to an SQL row.

Measurement (SQL table)      Fields (not indexed)      Tags (indexed)

name: **census**

-----

time      **butterflies**      **honeybees**      **location**      **scientist**

2015-08-18T00:00:00Z      12      23      1      langstroth

2015-08-18T00:00:00Z      1      30      1      perpetua

2015-08-18T00:06:00Z      11      28      1      langstroth

**2015-08-18T00:06:00Z**      **3**      **28**      **1**      **perpetua**

2015-08-18T05:54:00Z      2      11      2      langstroth

2015-08-18T06:00:00Z      1      10      2      langstroth

2015-08-18T06:06:00Z      8      23      2      perpetua

2015-08-18T06:12:00Z      7      22      2      perpetua

Point (SQL record)

Figure 3: InfluxDB data model.

<sup>1</sup>A retention policy (RP) describes how long InfluxDB keeps data, how many copies to store in a cluster, and the time range covered by shard groups. RPs are unique per database. The default retention policy with infinite duration and no replication is called autogen.

### 4.3.2 InfluxQL vs Flux

InfluxDB supports two query languages, InfluxQL and Flux. InfluxQL is an SQL-like query language for interacting with data in InfluxDB. It provides statements for data and schema exploration, database management, continuous queries and more. InfluxQL supports basic SELECT statement, and clauses such as WHERE, GROUP BY, INTO, ORDER BY, LIMIT and OFFSET. It also provides sub-queries capability similar to SQL's HAVING clause.

Below there are some examples of InfluxQL queries, using the NOAA<sup>2</sup> sample dataset.

1. *Find big water levels in Santa Monica.*

```
SELECT "water_level"  
FROM "h2o_feet"  
WHERE "location" = 'santa_monica'  
AND water_level > 9.95
```

2. *Find average water levels in each location.*

```
SELECT MEAN("water_level")  
FROM "h2o_feet"  
GROUP BY "location"
```

3. *What is the number of non-null field values for every field key that contains the word 'water' in the h2o\_feet measurement?*

```
SELECT COUNT(/water/)  
FROM "h2o_feet"
```

4. *Return a list of the unique field values in the level description field key. Cover the time range between 2017-08-17T23:48:00Z and 2017-08-18T00:54:00Z*

---

<sup>2</sup>[NOAA](#) sample dataset

*and group results into 10-minute time intervals and per tag.*

```
SELECT DISTINCT("level description")
FROM "h2o_feet"
WHERE time >= '2017-08-17T23:48:00Z'
AND time <= '2017-08-18T00:54:00Z'
GROUP BY time(10m)
```

b) Flux.

Using flux, use `filter()` to query data based on fields, tags, or any other column value. `Filter()` performs similar operations to the `SELECT` statement and the `WHERE` clause in InfluxQL.

```
from(bucket: "example-bucket")
  |> range(start: -1h)
  |> filter(fn: (r) =>
    r._measurement == "example-measurement" and
    r._field == "example-field" and
    r.tag == "example-tag")
```

#### **4.3.3 Influx Line Protocol**

The InfluxLineProtocol (ILP) is used by InfluxDB to write data points, sending data via sockets. The format is simple:

*measurement, tag\_set<whitespace>, field\_set<whitespace>, timestamp*

For example, using the above dataset, a record could seem like this:

*weather,location=us-midwest temperature=82 1465839830100400200*

The ILP will be examined in practice in section 4.5.5 .

#### 4.3.4 How is data stored?

All InfluxDB data is stored in a **bucket**, which combines the concept of a database and a retention policy (the duration of time that data can be stored). A bucket belongs to an **organization**. An organization is a workspace for a group of users. Every dashboard<sup>3</sup>, bucket and all users belong to an organization.

#### 4.3.5 Downsampling

Downsampling is a process where high resolution time series is aggregated within windows of time and then the lower resolution aggregation is stored in a new bucket. It helps speed up queries because it reduces the total amount of data used.

Downsampling allows us to diminish overall disk usage and improve query performance. In InfluxDB, *continuous aggregates* are used to implement downsampling.

Continuous aggregates are very similar to materialized views in a RDBMS. Expensive query results are pre-computed and stored instead of being computed on-the-fly. Continuous aggregates can transform and isolate data from one series into another for more efficiency. Examples comparing the performance of queries when using continuous queries and when not are included in Section 6.3.

#### 4.3.6 Storage engine

The storage engine currently uses the TSM Tree model (Figure 4). It ensures safety of data, data integrity (meaning that queried data is returned complete and correct). The following components are included:

---

<sup>3</sup>Dashboards are a visual way to represent data.

a) Write Ahead Log (**WAL**).

Stores compressed blocks of writes and deletes. Retains influxdb data when the storage engine restarts. Ensures data is durable in case of failure.

b) Cache. It is an in-memory copy of data points currently stored in the WAL. It organizes points by key, stores uncompressed data, and gets updates from the WAL each time the storage engine restarts. Cache is queried at runtime and merged with the data stored in the TSM files.

c) Time - Structured Merge Tree (**TSM**).

To efficiently compact and store data, the storage engine groups field values by series key, then orders these field values by time. The storage engine uses a TSM data format. TSM files store compressed series data in columnar format. After fields are stored safely in TSM files, WAL is truncated and cache is cleared.

d) Time Series Index (**TSI**).

Tries to keep queries fast as data cardinality grows. TSI stores keys grouped by measurement, tag and field. TSI moves the index to files on disk, files are mapped to memory.

In conclusion: When writing time data, first wal is added, then it is written to the cache, and finally it is flushed to the disk TSM file regularly or when it is full, as shown below.

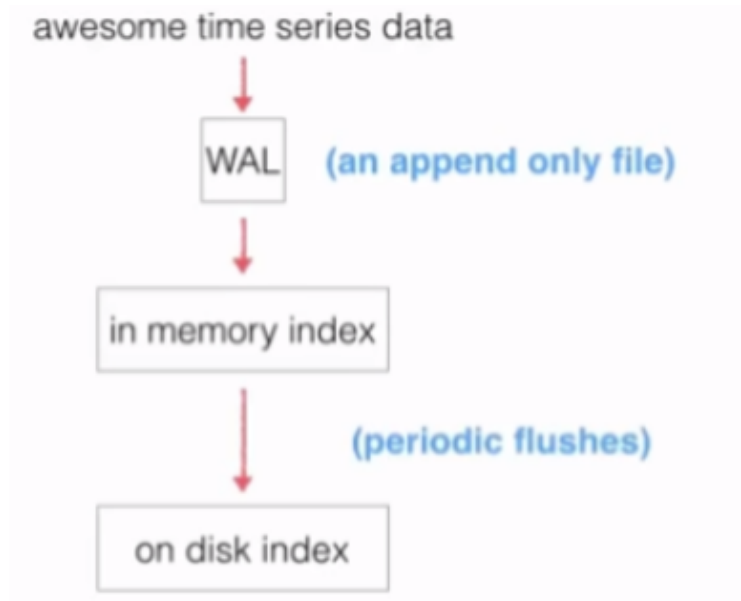


Figure 4: TSM data model.

#### 4.3.7 Data replication

##### a. WAL

As we mentioned earlier, InfluxDB makes use of WAL files. WAL files provide durability (ensuring our actions will be executed even if the database crashes) and atomicity (a system property that is guaranteed to occur completely). However, WAL files have some disadvantages. They aren't scalable because they can only be as big as ram, not big enough for a multi-tenant cloud solution. Also, maybe WAL is fast, but Kafka (see below) is faster. Finally, WAL is as durable as the filesystem it's using. So, there is no trace of replication, which means that a possible death of the disk will bring a potential loss of all the data.

InfluxDB cloud makes use of Kafka as a WAL to overcome these problems.

##### b. Kafka

According to [Wikipedia](#), "Apache Kafka is an open-source stream processing software platform, written in Scala and Java. The project aims to provide a unified, high-throughput, low-latency platform for handling real-time data feeds". Kafka supports 3 operations:



- Message Queue (reading and writing data streams)
- Message Broker (message processing)
- Data Store (message publishing and consuming)

Kafka is horizontally scalable and fault-tolerant. Also, it is really fast because it writes everything to disk through Sequential I/O (instead of memory). Also, Kafka avoids redundant data copy and batches messages.

### c. From WALs to Kafka

Since WAL cannot support multi-tenancy, Kafka handles the weight by using data partitioning. Kafka partitions are used as WALs and are used with the broker nodes. Kafka acts as a large and distributed WAL. The InfluxDB storage tier architecture is shown below.

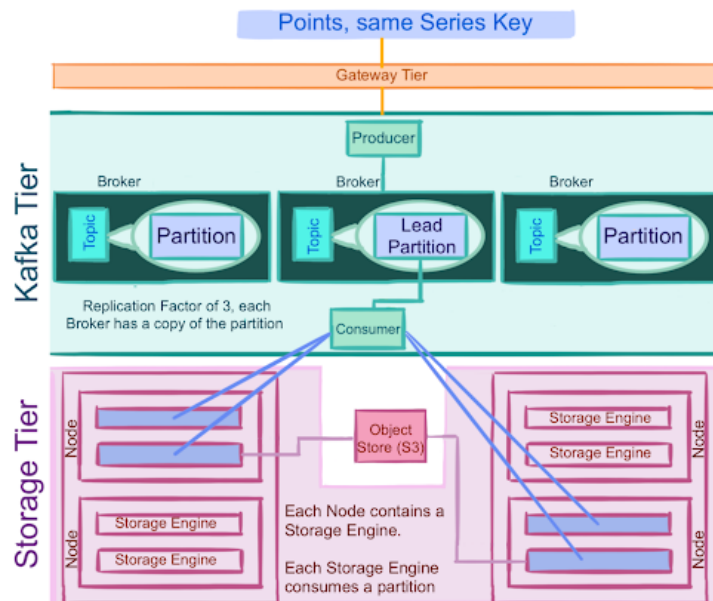


Figure 5: InfluxDB storage tier architecture.

Point data is evenly distributed among partitions by series key. Each partition is replicated 3 times. Producers write to a single leader to load balance production.

Kafka manages replication across nodes. The client has to write into one

node, which is the leader. If this node is unavailable, a node with replicas will become the new leader.

The storage tier's write API (not shown here) makes sure an optimal number of points is written to the Kafka tier, to ensure efficiency.

Multiple storage engines consume the same partition. A storage node has multiple storage engines, each storage engine consumes one Kafka partition.

The consumption of partitions can happen at different rates because there are multiple replica data sources to read from. The query tier uses the storage tier's Kafka consumption progress to determine which replica is most up to date.

Using Kafka as a sophisticated WAL provides InfluxDB cloud scalability and multi-tenancy.

#### **4.3.8 Pros-cons of InfluxDB**

In this section, some advantages and disadvantages of using InfluxDB will be discussed.

##### Pros(+)

InfluxDB is especially made for Time Series Data: It is designed to handle time series data with great efficiency and has strong write and read throughputs.

It provides an SQL-like query language: InfluxQL.

Also, Influxdb supports connection with Grafana. Grafana has a plugin for InfluxDB as a data source for their dashboards.

InfluxDB offers support for multiple programming languages including Python, R, Ruby, Scala, PHP, Java and more.

Finally, it uses a user-friendly web administrator interface, for users who are not comfortable with command line interfaces. and provides a variety of services, such as the TICK stack, free.

Also it offers InfluxCloud ,which provides high availability, scalability and advanced backup/restore functionalities.

##### Cons(-)

In InfluxDB, scalability is offered only as a close-source feature: Currently, InfluxDB high availability and scalability features are close-source.

Furthermore, InfluxDB is not a full CRUD (Create-Read-Update-Delete) database but more like a CR-ud, giving more priority to the performance of data creation and read over update and delete. If a lot of updates and deletions are required, InfluxDB is not recommended.

## **4.4 TimescaleDB**

TimescaleDB is also an open source time-series database. It is fully constructed on PostgreSQL and supports SQL. For PostgreSQL users, TimescaleDB will be easy to use because it seems like a traditional relational database, but it also features scalability.

The data model in TimescaleDB is wide-column based (the traditional relational database type). It consists of hypertables which are actually sets of smaller tables (chunks). User interactions with TimescaleDB are made possible with hypertables. However, the actual data is stored in the chunks<sup>4</sup>.

### **4.4.1 Partitioning with chunks**

When partitioning a hypertable's data into one or multiple dimensions, chunks are created. The partition is made by the values belonging to a time column (which may be in timestamp, date or integer forms).

For example, if the partitioning interval is one hour, then rows with timestamps belonging to the same hour will be placed on the same chunk, while rows belonging to different hours will be placed on different chunks.

The chunks are created automatically as rows are inserted in the database. If the timestamp of a newly-inserted row does not belong to an hour that exists

---

<sup>4</sup>A chunk is a small set of data.

in the database, a new chunk is created. Otherwise, TimescaleDB assigns to an existing chunk the new row.

Additional columns can be used to partition a hypertable (e.g. id, location, identifier etc.) Using both time and additional columns for partitioning is primarily used for distributed hypertables. In general, a chunk includes constraints that specify the partitioning ranges.

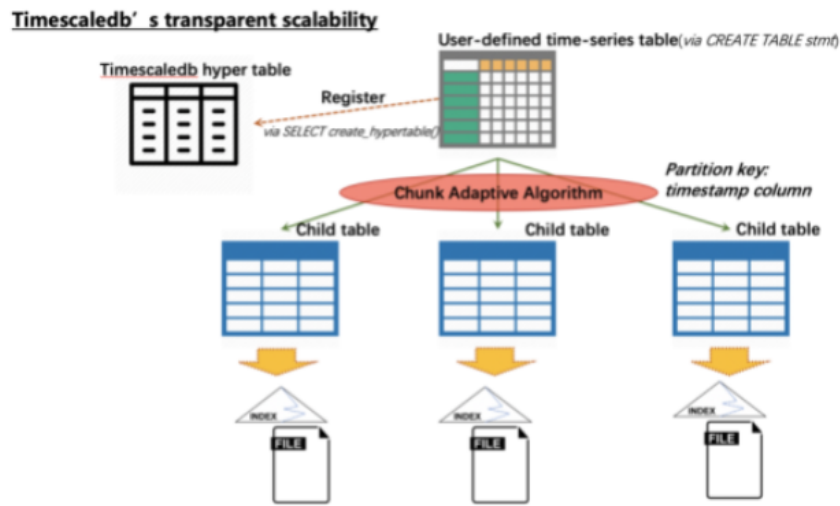


Figure 6: TimescaleDB partitioning.

#### 4.4.2 Scaling

There are three ways to perform scaling in TimescaleDB:

##### 1) **Single instance (node)**

Each of the chunks that are created by TimescaleDB is stored as a separate database table, all of its indexes are built across these smaller tables (rather than a single table that represents the entire dataset, in PostgreSQL). Giving the appropriate sizes to the chunks can help us fit the latest tables completely in memory.

## 2) **Primary-backup replication**

TimescaleDB supports streaming replication from a primary database to separate replica servers by using the PostgreSQL physical replication protocol. Records of database changes are streamed from the primary server to one or more replicas. WAL files are used to capture every database change. Replication is done by continuously shipping segments from the WAL to the connected replicas. Then each replica applies the changes and makes them available for querying capabilities. Replication can occur both synchronously and asynchronously.

## 3) **Multi-node TimescaleDB**

Instead of a primary node, here a distributed hypertable can be spread across multiple nodes, so that each node only stores a part of the distributed hypertable. This gives TimescaleDB the opportunity to scale to more than one instances.

### **4.4.3 Continuous aggregates**

Continuous aggregates look similar to PostgreSQL's materialized views, though they are continuously refreshed. They are supported for most aggregate functions that can be parallelized by PostgreSQL, which includes the SUM and AVG functions. However, aggregates with ORDER BY and DISTINCT keywords cannot be used with continuous aggregates because they are not possible to parallelize by PostgreSQL. A query on a continuous aggregate by default uses real-time aggregation to combine materialized aggregates with recent data from the source hypertable.

We will see a benchmarking example with continuous aggregates later in section 5.7 .

### **4.4.4 Data retention**

Just like InfluxDB's retention policies, we have retention policies for Timescale too. Timescale includes a background job scheduling framework for automating data management tasks. One of them is enabling data retention policies with which you can set data retention standards on each hypertable and allow Timescale to drop data.

#### 4.4.5 Data model

TimescaleDB supports two kinds of data model, the *wide-table* and *narrow-table* ones. The *wide-table* data model looks like this:

| timestamp              | device_id | cpu_1m_avg | free_mem | temperature | location_id |
|------------------------|-----------|------------|----------|-------------|-------------|
| 2017-01-01 01:02:00    | abc123    | 80         | 500MB    | 72          | 42          |
| 2017-01-01 01:02:23    | def456    | 90         | 400MB    | 64          | 42          |
| 2017-01-01<br>01:02:30 | ghi789    | 120        | 0MB      | 56          | 77          |
| 2017-01-01 01:03:12    | abc123    | 80         | 500MB    | 72          | 42          |
| 2017-01-01<br>01:03:35 | def456    | 95         | 350MB    | 64          | 42          |
| 2017-01-01<br>01:03:42 | ghi789    | 100        | 100MB    | 56          | 77          |

Figure 7: TimescaleDB's wide table data model.

This is the format that one would commonly find within a relational database. TimescaleDB's data model also supports joins, just like in relational databases. You can store additional metadata in a secondary table and then use that data at query time.

In contrast, in the *Narrow-table* model each metric combination is considered an individual time series, containing time/value pairs. A narrow model looks like this:

1. *name: cpu\_1m\_avg, device\_id: abc123, location\_id: 335*
2. *name: cpu\_1m\_avg, device\_id: def456, location\_id: 335f*
3. *name: cpu\_1m\_avg, device\_id: ghi789, location\_id: 77*
4. *name: free\_mem, device\_id: abc123, location\_id: 335*
5. *name: free\_mem, device\_id: def456, location\_id: 335*
6. *name: free\_mem, device\_id: ghi789, location\_id: 77*

A model like this is used in many time series databases and makes sense if you collect each metric independently. You can add new metrics by adding a

new tag without altering the schema. However, a narrow model demands higher storage requirements. This is because, if many metrics with the same timestamp are collected, a timestamp is required for each metric while the wide-table model will use less.

## **4.5 QuestDB**

QuestDB is also an open-source time series database. It uses SQL, just like TimescaleDB and makes use of the InfluxLineProtocol for faster ingestion.

### **4.5.1 Data model**

QuestDB uses a column-based storage model. Tables contain the data with each column stored in its own file. The model is FIFO-like, meaning that ingestion order is preserved until storing the data.

QuestDB appends one column at a time. There is a mapping between the column file and the memory page in RAM. For reading, table columns are randomly accessible.

### **4.5.2 Partitioning**

In QuestDB, a column may be chosen to be ejected as a designated timestamp. This makes the column indexed, which ultimately boosts performance. Only one column of type timestamp can be ejected as a designated timestamp.

QuestDB may partition tables by time intervals. Data for each interval is stored in different sets of files. Some available intervals are NONE, YEAR, MONTH, DAY, HOUR, while the default is NONE. After partitioning, the performance is improved because of less seek times and reduced DISK I/O.

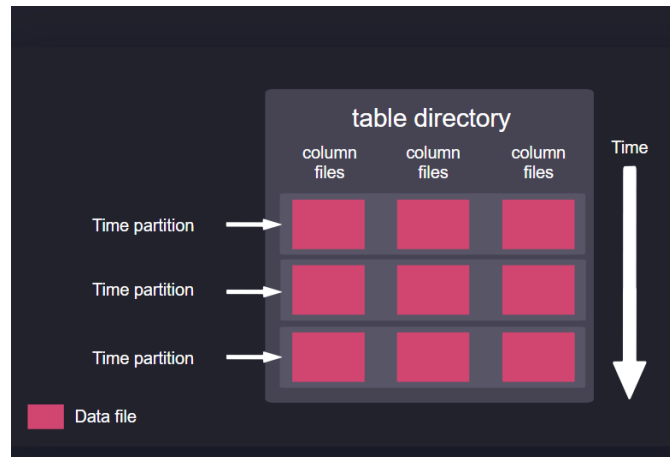


Figure 8: QuestDB partitioning.

Partitioning though is only possible on tables that support a designated timestamp.

### 4.5.3 Indexing

Indexing assists significantly in performance improvement, reducing the complexity of queries scanned. A new index table is created containing the row locations for each distinct value of the target symbol. After creating an index, the table will be updated with each set of new data. Lookups are performed in the index table directly which is significantly smaller, saving time. The only drawback is that the index table will trigger a small cost of storage.

### 4.5.4 InfluxLineProtocol and comparison to InfluxDB

QuestDB can ingest data through the InfluxLineProtocol (ILP) to take advantage of SQL to query Influx data but keeping at the same time the flexibility of ILP.



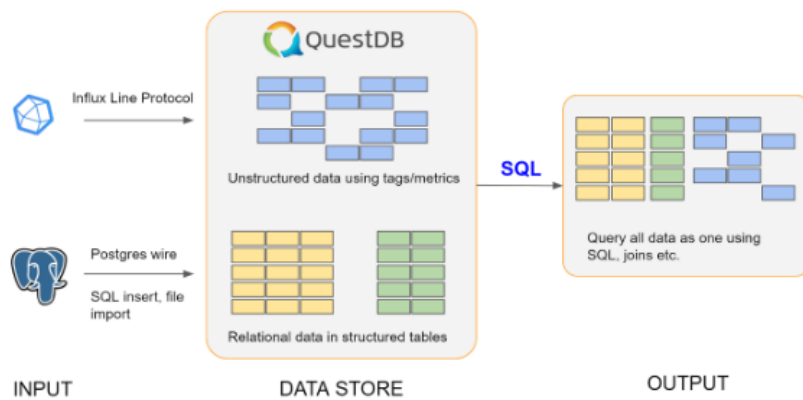


Figure 9: QuestDB ingestion using the InfluxLineProtocol.

In the background, InfluxDB with big cardinality rates starts facing problems and eventually cannibalizing the CPU. That is something we personally noticed while having benchmarking tests (see Chapter 5), as when we tested InfluxDB, the computer started getting out of its way and slowing down all the other programs at that time.

QuestDB maximizes the utilization of the CPU, while it does not stay idle. It can work in parallel, utilizing many cores while InfluxDB is limited to single receiver throughput.

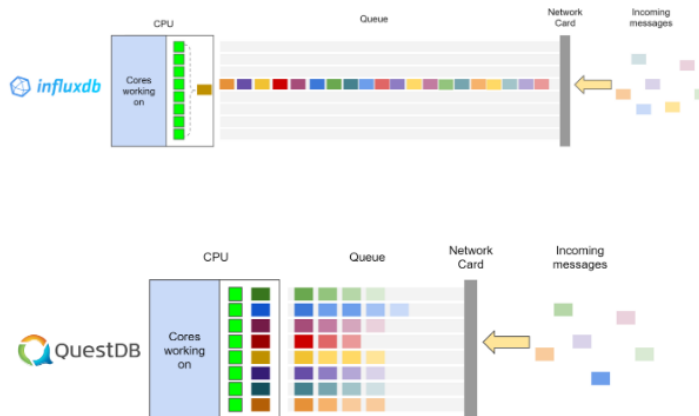


Figure 10: Differences on data ingestion between InfluxDB (above) and QuestDB (below).

QuestDB uses each core separately, which means it allows many users to query or write with no delays. In contrast, in InfluxDB, each user takes advantage of all the CPU, while the other users must wait for their turn.

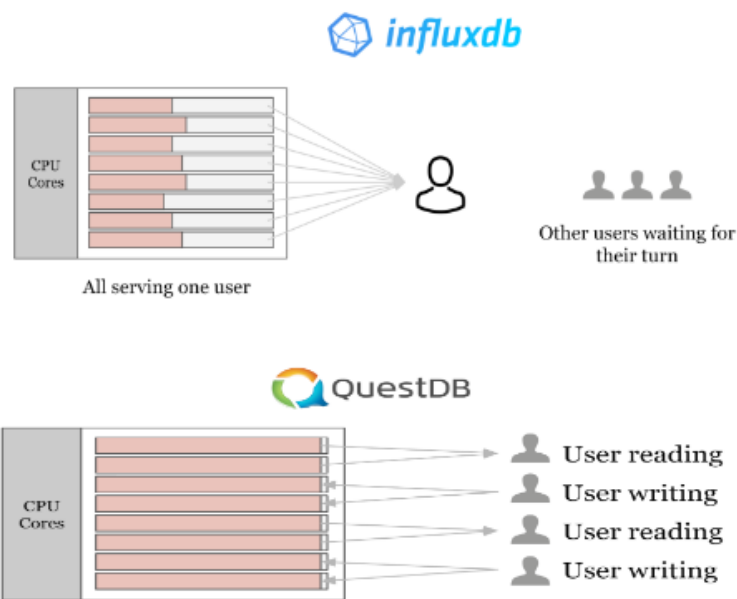


Figure 11: Using CPU on InfluxDB (above) and QuestDB (below).

#### 4.5.5 Schemaless ingestion using the ILP

As we have already mentioned, QuestDB uses the ILP to send data over sockets. The format is the following:

*measurement, tag\_set<whitespace>, field\_set<whitespace>, timestamp*

We will take a use case of a dataset containing the diagnostics of a car engine. An example of the ILP in QuestDB would be the following:

```
engine_diagnostics, Make=Volkswagen, Model=Polo engine_temperature=80.2, oil_gauge=99.19 16655
```

QuestDB listens by default to port 9009 and gets the data as a byte stream. It supports data ingestion using the ILP over TCP and UDP.

Using a python script (see Appendix I) we can generate 10000 data points for the measurement engine\_diagnostics.

**First ingestion:** Once running the script, when heading to the QuestDB web console, we verify that the table has been created with 10000 records.

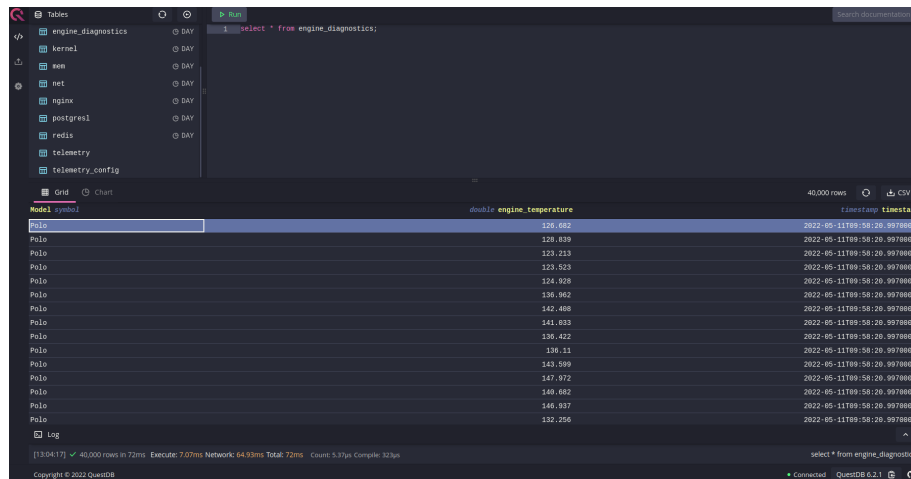
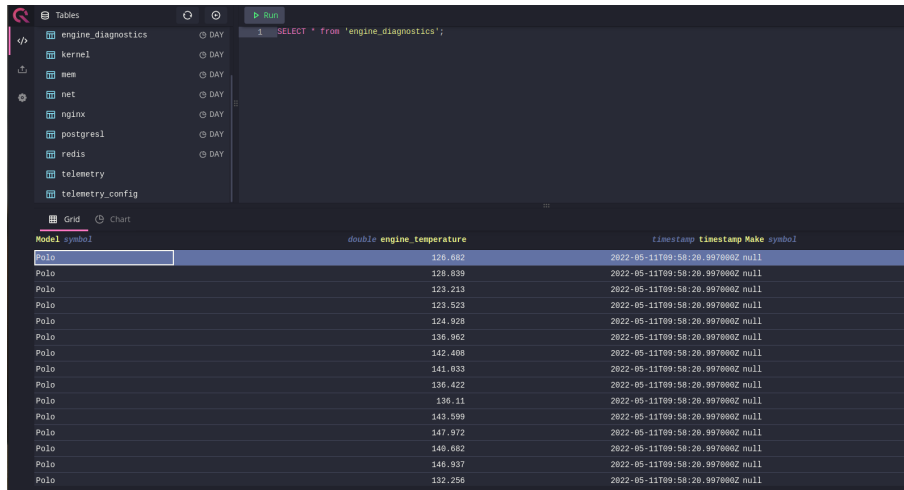


Figure 12: QuestDB console: Validating the data is inserted correctly.

In QuestDB, a separate file is used to store every column and the data is partitioned by time. This results in faster queries, especially when you have to fetch a) specific columns of a table or b) data for a specific time interval. QuestDB's storage model makes it quicker to create new columns on-the-fly, so QuestDB supports schemaless data.

**Second ingestion:** Let's add a tag to the measurement. Using the line `metric.add_tag('Make','Volkswagen')` the script creates the Make column and inserts

10000 new data points to the engine\_diagnostics table.



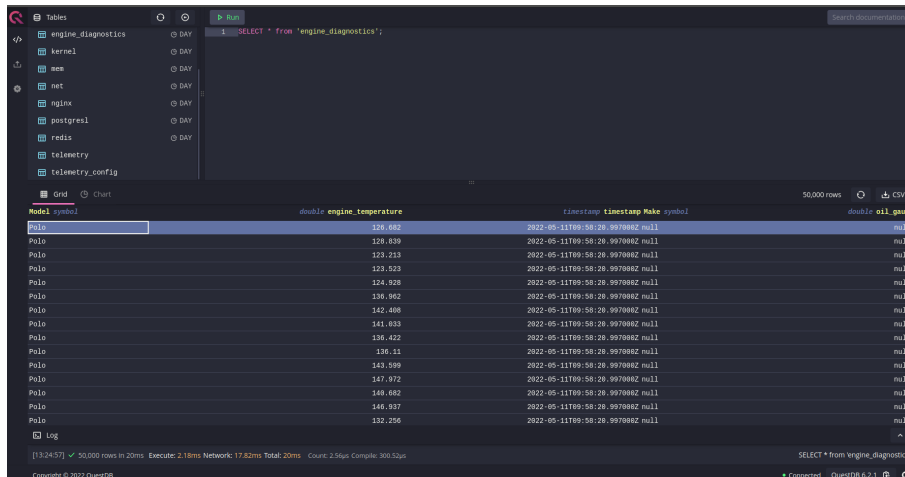
The screenshot shows the QuestDB console interface. On the left, a sidebar lists various database tables: engine\_diagnostics, kernel, mem, net, nginx, postgresql, redis, telemetry, and telemetry\_config. The main area displays a SQL query: `SELECT * FROM 'engine_diagnostics';`. Below the query, a table view shows the first 10 rows of data. The table has columns: Model, symbol, double engine\_temperature, timestamp, timestamp Make, and symbol. The data shows multiple entries for 'Polo' with varying engine temperatures and timestamps.

| Model | symbol | double engine_temperature | timestamp                   | timestamp Make | symbol |
|-------|--------|---------------------------|-----------------------------|----------------|--------|
| Polo  |        | 126.682                   | 2022-05-11T09:58:20.997000Z | null           |        |
| Polo  |        | 126.839                   | 2022-05-11T09:58:20.997000Z | null           |        |
| Polo  |        | 123.213                   | 2022-05-11T09:58:20.997000Z | null           |        |
| Polo  |        | 123.523                   | 2022-05-11T09:58:20.997000Z | null           |        |
| Polo  |        | 124.928                   | 2022-05-11T09:58:20.997000Z | null           |        |
| Polo  |        | 136.962                   | 2022-05-11T09:58:20.997000Z | null           |        |
| Polo  |        | 142.408                   | 2022-05-11T09:58:20.997000Z | null           |        |
| Polo  |        | 141.833                   | 2022-05-11T09:58:20.997000Z | null           |        |
| Polo  |        | 136.422                   | 2022-05-11T09:58:20.997000Z | null           |        |
| Polo  |        | 136.11                    | 2022-05-11T09:58:20.997000Z | null           |        |

Figure 13: QuestDB console: adding a tag to the data (changing schema dynamically).

Next we verify that both the batches - one without and one with the Make column have the same number of records. This means that QuestDB supports on-the-fly schema changes with no charge to the user. Also, we don't need to fill with NULL values, as QuestDB takes care of that using the append-only and columnar storage model. This feature lets QuestDB handle irregularly structured data, which may come closer to the real world.

**Third ingestion:** Now we can insert a new value. Using the line `metric.add_value('oil_gauge', random.uniform(91.0,99.5))`, a new column is created on-the-fly. You can see below how we were able to ingest new tags and values without changing the table's structure.



The screenshot shows the QuestDB console interface after inserting a new column. The SQL query remains the same: `SELECT * FROM 'engine_diagnostics';`. The table view now shows an additional column, 'double oil\_gauge', which contains NULL values for all rows. The status bar at the bottom indicates that the table now contains 50,000 rows.

| Model | symbol | double engine_temperature | timestamp                   | timestamp Make | symbol | double oil_gauge |
|-------|--------|---------------------------|-----------------------------|----------------|--------|------------------|
| Polo  |        | 126.682                   | 2022-05-11T09:58:20.997000Z | null           |        | null             |
| Polo  |        | 126.839                   | 2022-05-11T09:58:20.997000Z | null           |        | null             |
| Polo  |        | 123.213                   | 2022-05-11T09:58:20.997000Z | null           |        | null             |
| Polo  |        | 123.523                   | 2022-05-11T09:58:20.997000Z | null           |        | null             |
| Polo  |        | 124.928                   | 2022-05-11T09:58:20.997000Z | null           |        | null             |
| Polo  |        | 136.962                   | 2022-05-11T09:58:20.997000Z | null           |        | null             |
| Polo  |        | 142.408                   | 2022-05-11T09:58:20.997000Z | null           |        | null             |
| Polo  |        | 141.833                   | 2022-05-11T09:58:20.997000Z | null           |        | null             |
| Polo  |        | 136.422                   | 2022-05-11T09:58:20.997000Z | null           |        | null             |
| Polo  |        | 136.11                    | 2022-05-11T09:58:20.997000Z | null           |        | null             |

Figure 14: QuestDB console: adding a value to the data (changing schema dynamically).

We have successfully ingested batches of 10000 records. Our conclusion is that QuestDB uses the InfluxLineProtocol to ingest data without having to worry about updating the schema when needing to insert new tags and values from the measurements. InfluxDB uses the ILP in the same way to benefit from schemaless ingestion. However, this feature is not present in TimescaleDB and most other time series databases.

## 4.6 Prometheus

Prometheus is an open-source systems monitoring and alerting system. It collects and stores metrics as time series data.

Some basic features of Prometheus include a multi-dimensional data model with time series data identified by metric name and key/value pairs, PromQL, which is Prometheus' query language. Also Prometheus does not rely on distributed storage and time series collection can be achieved via a pull model over HTTP.

### 4.6.1 Data model

Prometheus stores all data as time series, which means streams of time-stamped values belong to the same metric and the same set of labeled dimensions. Every time series is identified by its metric name and optionally labels, which are key-value pairs.

Metric name: specifies the general features of a system that is measured

Label: Any given combination of labels for the same metric name gives a particular dimensional instantiation of that metric.

The notation used is the following:

*<metric name> {<label name>=<label value>, ...}*

An example:

*api\_http\_requests\_total method="POST"*

Here *api\_http\_requests\_total* is the metric name and *method="POST"* is the label. This command returns the number of HTTP requests received using POST method.

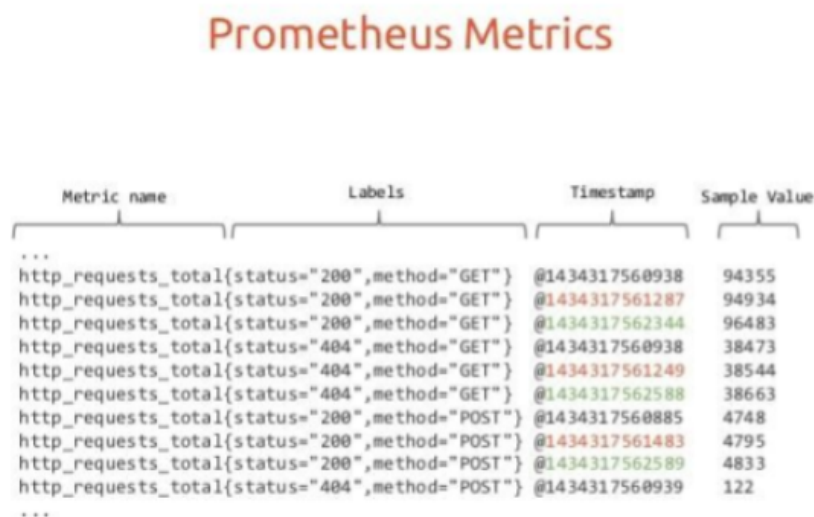


Figure 15: Notation of queries and results in Prometheus.

Some examples of PromQL:

1) Return all time series with the metric *http\_requests\_total*:

*http\_requests\_total*

2) Return all time series with the metric *http\_requests\_total* and the given job and handler labels.

```
http_requests_total{job="apiserver", handler="/api/comments"}
```

3) Return the 5-minute rate of the `http_requests_total` metric for the past 30 minutes with a resolution of 1 minute.

```
rate(http_requests_total[5m])[30m:1m]
```

#### 4.6.2 Storage

Prometheus includes a local time series database, but also it can integrate with remote storage systems.

##### Local storage layout:

Ingested samples are grouped into blocks of 2 hours. The 2-hour blocks are compacted into longer blocks in the background.

Each block consists of a `chunks` subdirectory which contains all the time series samples for that window of time, a metadata file and an index file.

The samples in the `chunks` directory are grouped together into one or more segment files of up to 512MB each.

The current block for incoming samples is secured against crashes by a WAL file that can be replayed once the Prometheus server starts again. WAL files are kept in the `wal` directory, in 128MB segments. They contain raw data that has not been compacted yet (so they are pretty large). Prometheus retains a minimum of 3 WALs.

The local storage is not clustered, which means that it is not distributed; it should be treated like any other single node database. For storage availability, RAID is used and snapshots for backups.

#### 4.6.3 Alerting

Imagine if one has deployed Prometheus to monitor their systems to keep track if everything is going as expected. But at some moment, some of the metrics start to divert from their expected values. To keep a check on this, they need to put

someone to monitor Prometheus values day and night (not feasible) or they need to automate the things! If Prometheus can alert e.g. by mail when something is behaving unexpectedly then many problems could be solved. This is the concept of alerting and is a very useful tool in Prometheus.

Alerting rules in Prometheus send alerts to Alertmanager. Alertmanager then manages those alerts in various ways (e.g. silencing, aggregation, sending notifications via methods such as email, chat platforms etc.)

Core concepts of AlertManager:

1) **Grouping**: Groups alerts of similar nature into a single notification. Example: Hundreds of instances of a service are running in a cluster when a network partition occurs. Some of these instances cannot reach the database. Alerting rules are configured to send an alert for each case if it cannot communicate with the database.

2) **Inhibition**: Suppresses notifications for certain alerts if certain other alerts are already firing.

*Example*: An alert is firing that informs that a cluster is not reachable. Alertmanager can then be configured to mute the other alerts concerning this cluster if that particular alert is firing.

3) **Silences**: Mute alerts for a given time.

Figure 7 shows a brief overview of Prometheus monitoring system.



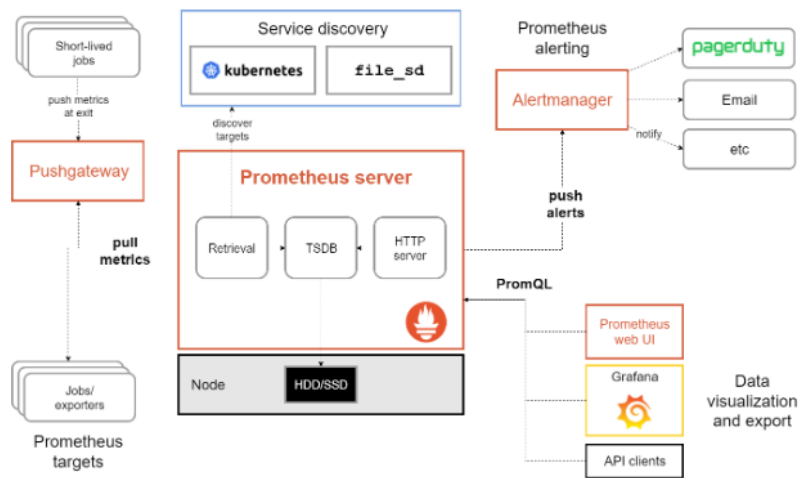


Figure 16: Prometheus Architecture.

Warning: Prometheus is not(!) a time series database. It is a monitoring system that happens to use a time series database under the hood.

#### 4.7 Recap: InfluxDB vs Prometheus vs TimescaleDB vs QuestDB

Time to make a head to head comparison between the 4 players:

| Name                                   | InfluxDB   | Prometheus  | TimescaleDB  | QuestDB  |
|--|--|---|--|--|
| <i>Description</i>                     | DBMS for storing time series, events and metrics       | Open-source TimeSeries DBMS and monitoring system | Time series DBMS, optimized for fast ingest and complex queries, based on PostgreSQL | Fast open-source time series DBMS, supporting SQL    |
| <i>Initial release</i>                 | 2013   | 2015  | 2017   | 2014   |
| <i>Implementation language</i>         | Go   | Go  | C  | C, Java  |
| <i>Data scheme</i>                     | Schema-free  | ✓   | ✓  | ✓<br>(can be schema-free via the InfluxLineProtocol) |
| <i>Types</i>                           | Numeric data, strings                                  | Numeric data                                      | Numerics, strings, arrays, currencies, binary data, JSON blobs etc.                  | Numeric data, strings                                |
| <i>XML Support</i>                     | ✗  | ✗   | ✓  | ✗  |
| <i>Secondary indexes</i>               | ✗  | ✗   | ✓  | ✗  |
| <i>Query language</i>                  | InfluxQL(SQL-like), Flux                               | PromQL  | SQL  | SQL  |
| <i>Supported programming languages</i> | Java, Javascript, Go, PHP, Python, R, Ruby, Scala etc. | C++, Go, Java, Node.js, Python, Ruby etc.         | C, C++, Java, Javascript, Python, R, PHP etc.  | C, C++, Go, Java, Javascript, Python, Rust etc.      |
| <i>Triggers</i>                        | ✗  | ✗   | ✓  | ✗  |
| <i>Server operating systems</i>        | Linux, OS X  | Linux, Windows                                    | Linux, OS X, Windows   | Linux, Mac OS, Windows                               |

Table 2: InfluxDB vs Prometheus vs TimescaleDB vs QuestDB.

## 4.8 A comparison between some popular time series databases

Before proceeding to the table, we need to mention some things about two databases we haven't talked about: OpenTSDB, GridDB.

**OpenTSDB** is flexible and with high ingest performance. There is no schema and it can ingest millions of data per second. It is scalable<sup>5</sup> and has a strong metrics system. But on the other hand, it is difficult to set up and get used to due to a completely different interface (telnet).

**GridDB** uses a key-container data model which looks like the traditional relational data model with a fixed schema. GridDB is also scalable (performance will scale when adding more nodes) and offers functionality such as data aggregation, data retention policies.

| #           | InfluxDB  | TimescaleDB   | OpenTSDB                            | GridDB                                   | QuestDB  |
|-------------|---|---|-------------------------------------|--|--|
| <u>Pros</u> | Easy to install, Good performance for one node  | SQL-based (familiar), fast ingest, fixed schema             | very scalable, very fast ingest     | Good performance, Fixed schema, scalable | SQL-based (familiar), fast ingestion, uses InfluxDB line protocol (fast), exposes HTTP API |
| <u>Cons</u> | Lack of strict schema (maybe we can see it as a decrement). Only one node in open source. | Fixed schema (in a way we could see it too was a decrement) | Difficult to learn, no fixed schema | Fixed schema                             | Smaller community, little available integrations, lack of some features                    |

Table 3: Comparing time series databases.

---

<sup>5</sup>scalability is the ability to increase storage/performance by adding more nodes.

## 4.9 Comparing relational and time series databases

We will be based on a research article<sup>6</sup> for this presentation.

The following combination of relational and time series databases will be used:

- 1) InfluxDB
- 2) PostgreSQL
- 3) OpenTSDB
- 4) MySQL community server

The comparison criteria are as follows:

- Distribution/Clusterability
- Functions and Long-term storage
- Granularity
- Interfaces, Extensibility
- Support and license

(In the tables below, ✕ means the feature it indicates to is not available and ✓ means the feature is available.)

### 1) Distribution/Clusterability.

CF gives the possibility to face unexpected node failures and network partitioning. Load balancing is the possibility to equally distribute queries across nodes in a TSDB, in order to balance the workload on each node.

| TSDB                      | CF | Scalability | Load Balancing |
|---------------------------|----|-------------|----------------|
| OpenTSDB                  | ✓  | ✓           | ✕              |
| InfluxDB                  | ✓  | ✓           | ✓              |
| MySQL<br>Community Server | ✕  | ✕           | ✕              |
| PostgreSQL                | ✕  | ✕           | ✕              |

Table 4: Comparison criteria #1: Distribution/Clusterability.

---

<sup>6</sup>[https://www.researchgate.net/publication/315838456\\_Survey\\_and\\_Comparison\\_of\\_Open\\_Source\\_Time\\_Series](https://www.researchgate.net/publication/315838456_Survey_and_Comparison_of_Open_Source_Time_Series)

## 2) Functions and long-term storage

Availability of AVG,SUM,COUNT functions and long-term storage are compared here. Also we check if a TSDB can calculate functions continuously based on the input data (called Continuous calculation), e.g. calculating an average for a minute. Finally, we check if the database can store huge amounts of data in the long term.

| TSDB                   | Continuous calculation | AVG | SUM | COUNT | Long-term storage |
|------------------------|------------------------|-----|-----|-------|-------------------|
| OpenTSDB               | ×                      | ✓   | ✓   | ✓     | ×                 |
| InfluxDB               | ✓                      | ✓   | ✓   | ✓     | ✓                 |
| MySQL Community Server | ✓                      | ✓   | ✓   | ✓     | ×                 |
| PostgreSQL             | ✓                      | ✓   | ✓   | ✓     | ×                 |

Table 5: Comparison criteria #2: Functions and long-term storage.

## 3) Granularity

Granularity is the level of detail at which data is stored in a database.

TSDBs offer downsampling functionality to customize time granularity. This means that one can query with a specified time granularity and zoom out. Downsampling is very helpful when events happen at a specified time interval but not to the exact (micro) second.

Here we compare the smallest possible granularities that can be used for storage and functions.

| TSDB                   | Downsampling | Smallest sample interval | Smallest granularity for storage |
|------------------------|--------------|--------------------------|----------------------------------|
| OpenTSDB               | ✓            | 1 ms                     | 1 ms                             |
| InfluxDB               | ✓            | 1 ms                     | 1 ms                             |
| MySQL Community Server | ✓            | 1 ms                     | 1 ms                             |
| PostgreSQL             | ✓            | 1 ms                     | 1 ms                             |

Table 6: Comparison criteria #3: Granularity.

#### 4) Interfaces and Extensibility

Here we compare APIs, interfaces, client libraries and plugins.

| <b>TSDB</b>            | <b>APIs and interfaces</b>                | <b>Client libraries</b>   | <b>Plugins</b> |
|------------------------|---|---------------------------|----------------|
| OpenTSDB               | CLI, HTTP (REST+JSON, GUI), Kafka, Azure  | Java                      | ✓              |
| InfluxDB               | CLI, HTTP (InfluxQL, Flux, GUI), OpenTSDB | ✗                         | ✓              |
| MySQL Community Server | CLI                                       | Java, Python, C, C++ etc. | ✓              |
| PostgreSQL             | CLI                                       | Java, Python, C, C++ etc. | ✓              |

Table 7: Comparison criteria #4: Interfaces and Extensibility.

#### 5) Support and license

This group compares the availability of a stable version and commercial support. What we want is a branch that receives updates that do not massively change the program (this is a stable version). Also, we check the license, which says how and by whom a product can be used and what modifications are allowed to it.

| <b>TSDB</b>            | <b>Stable version</b> | <b>Commercial support</b> | <b>License</b>         |
|------------------------|-----------------------|---------------------------|------------------------|
| OpenTSDB               | ✗                     | ✗                         | LGPIv2.1+, GPL.v3+     |
| InfluxDB               | ✗                     | ✓                         | MIT                    |
| MySQL Community Server | ✓                     | ✓                         | GPLv2                  |
| PostgreSQL             | ✓                     | ✗                         | The PostgreSQL License |

Table 8: Comparison criteria #5: Support and license.

## 5 Benchmarking some popular time series databases

After an extensive study of the most popular time series databases, now it's time to compare them in action. This section aims to provide answers to questions regarding data ingestion and query performance in order to assist in discovering the suitability of each time series databases for the above operations. In this run, the insert/write and query execution performance of InfluxDB, TimescaleDB and QuestDB will be compared. These three databases are all popular and each one is designed in a different way. InfluxDB is a powerhouse in the Time Series Databases, while TimescaleDB relies on PostgreSQL (comfortable for most database users) and QuestDB is reported to be the fastest time series database. Prometheus will not be used because it is not a time series database (though it uses one under the hood) and is not supported by TSBS.

### 5.1 Introduction to TSBS

In our evaluation we have used an open-source benchmarking tool, called Time Series Benchmark Suite ([TSBS](#)). InfluxDB originally developed it, and currently it is maintained by TimescaleDB.

For traditional databases, like PostgreSQL and MySQL, simple tools like HammerDB and sysbench are popular options to measure read and write performance of databases. We need a tool that simulates real-life occasions for this purpose that is created especially for time series databases, and a good one is TSBS.

TSBS supports two kinds of loads, **DevOps** and **IoT**. DevOps imitates data usage generated by servers tracking CPU usage, memory/disk usage etc. IoT imitates the IoT data generated from sensors of a trucking company.

The data is randomly generated and to ensure the contrast is fair, TSBS supplies with the same PRNG(pseudo-random number generator) seed to the generation programs, so each database is loaded using identical data and queried

sing the identical queries.

Note: This benchmark run was completed on an 64-bit Intel Core i5-6200U CPU @2.30 GHz 2.40 GHz, with 8 GB of installed RAM.

TSBS will generate DevOps/IoT data for 24 hours. Nine different metrics are collected every 10 seconds for 200 devices. Then the generated data will be loaded, which will help in discovering the write and ingestion speeds of each system. After that, queries to run the loaded data will be generated and finally they will be executed to measure query execution performance.

In order to have a fair comparison, the same parameters and same number of data for all databases will be used. Also, for every benchmark run, the cache is made sure to be "cold" (ensured by restarting TSBS every time).

#### DevOps / IoT use cases

*DevOps use case:* It is used to generate, insert and measure data from 9 systems. These systems generate 100 metrics per reading interval. Also, tags are generated for each host with readings in the dataset. Using the scale flag, the number of hosts can be adjusted.

*IoT use case:* Simulates the data streaming from a set of trucks belonging to a trucking company. In this case, the scale flag can be used to find the number of trucks tracked.

#### Terms

- *Metrics* : The number of elements in the database (a row contains a certain number of metrics).
- *Ingestion rate* : The number of rows loaded per second (rate). Ingestion can



be measured using the tsbs\_load executable. It uses YAML files to specify the configuration for running the load benchmark.

## 5.2 Ingestion performance: InfluxDB vs TimescaleDB vs QuestDB

|                     | DevOps              | IoT                |
|---------------------|---------------------|--------------------|
| # of metrics loaded | 174.528.000         | 24.592.735         |
| # of workers        | 1                   | 1                  |
| # of rows loaded    | 15.552.000          | 3.110.314          |
| InfluxDB time       | <b>1011.743 sec</b> | <b>228.138 sec</b> |
| TimescaleDB time    | <b>2144.979 sec</b> | <b>126.931 sec</b> |
| QuestDB time        | <b>1067.183 sec</b> | <b>200.779 sec</b> |

Table 9: Ingestion performance.

Write performance of InfluxDB is **1.05x** faster than QuestDB and **2.12x** faster than TimescaleDB for DevOps data. InfluxDB is a marginal winner with QuestDB closely behind and TimescaleDB being more than 2 times slower.

For IoT data, TimescaleDB is the winner and this is because it uses PostgreSQL in the background.

The main difference between DevOps and IoT use case is that the IoT generates data which may contain out-of-order, missing or empty entries to represent a real life scenario.

InfluxDB is optimized for handling large volumes of data. And there is an explanation for that: InfluxDB uses its fast line protocol and achieves an efficient way of handling the data.

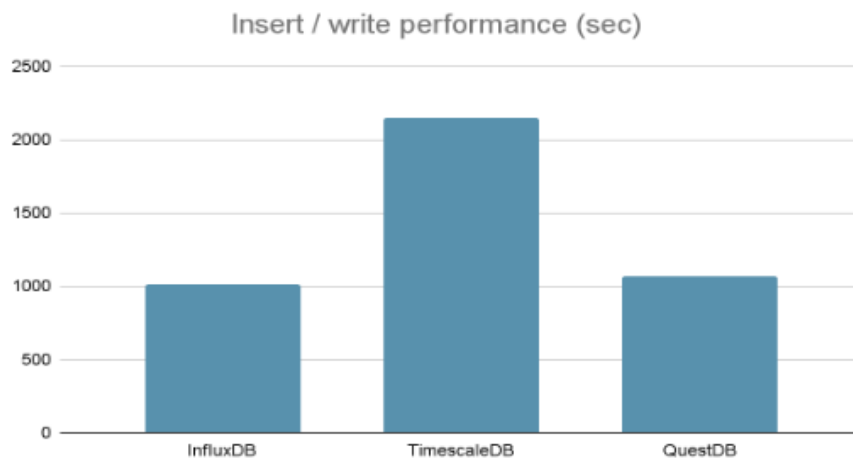


Figure 17: Comparing time series databases with TSBS, insert/write performance (DevOps data).



Figure 18: Comparing time series databases with TSBS, insert/write performance (IoT data).

### 5.3 Query execution performance

In order to test execution performance, several query types will be used:

1) Query type “high-cpu-all” , which returns all the readings where one metric is above a threshold across all hosts.

After 1000 queries with 1 worker, we get:

| DB          | Query rate                  | Wall Clock time        |
|-------------|-----------------------------|------------------------|
| InfluxDB    | <b>0.18<br/>queries/sec</b> | <b>5468.780001 sec</b> |
| TimescaleDB | <b>0.81<br/>queries/sec</b> | <b>1229.735591 sec</b> |
| QuestDB     | <b>4.04<br/>queries/sec</b> | <b>247.396037 sec</b>  |

Table 10: Query execution performance (“high-cpu-all”).

QuestDB is a triumphant winner here, being 4.97x faster than TimescaleDB and 22.1x faster than InfluxDB.

2) Query type “lastpoint”, which returns the last reading for each host.

After 1000 queries with 1 worker, we get:

| DB          | Query rate                    | Wall Clock time      |
|-------------|-------------------------------|----------------------|
| InfluxDB    | <b>13.32<br/>queries/sec</b>  | <b>75.172207 sec</b> |
| TimescaleDB | <b>120.81<br/>queries/sec</b> | <b>8.240822 sec</b>  |
| QuestDB     | <b>154.18<br/>queries/sec</b> | <b>6.637054 sec</b>  |

Table 11: Query execution performance ("lastpoint").

3) Query type "Single-groupby-1-1-1" , which is a simple aggregation (MAX) on one metric for one host every five minutes for an hour.

After 1000 queries with one worker, we get:

| DB          | Query rate                   | Wall Clock time      |
|-------------|------------------------------|----------------------|
| InfluxDB    | <b>30.12<br/>queries/sec</b> | <b>45.218232 sec</b> |
| TimescaleDB | <b>32.97<br/>queries/sec</b> | <b>30.453123 sec</b> |
| QuestDB     | <b>72.45<br/>queries/sec</b> | <b>14.003394 sec</b> |

Table 12: Query execution performance ("Single-groupby-1-1-1").

### Commenting on the results

#### **single-groupby-1-1-1:**

For this query, we observe that QuestDB is the most efficient, achieving at least 2x bigger query rate than the other two databases. For aggregations, QuestDB uses SIMD instructions (specific CPU instruction sets using synthetic parallelization). SIMD performs operations on a variety of items using only one CPU instruction. For example, if we wanted to add 5 numbers together, SIMD can do that in one operation instead of five. QuestDB claims to have achieved 100x faster queries using the SIMD operations. The SIMD operations are available for aggregation queries (e.g. `SELECT AVG(value) FROM table`) and as so our example too.

#### **lastpoint:**

Here is how the query 'lastpoint' looks like on vanilla SQL:

```
SELECT DISTINCT ON (<hostname>)
FROM table1 t1
INNER JOIN LATERAL(
SELECT *
FROM table2 t2
WHERE t1.id=t2.id
ORDER BY time DESC
LIMIT 1) AS b
ORDER BY t1.hostname, b.time DESC
```

('lateral' keyword is needed in order to join the output of the outer query with the output of the lateral subquery.)

QuestDB again proves to be the fastest database with a small difference to TimescaleDB and a vast difference of both to InfluxDB. Here, the first two databases

can use a dependent join. You cannot join series in InfluxQL using arbitrary columns and this is why it performs so slow. InfluxQL only supports joining time series based on the time column, unlike most relational databases. The query has to be translated into Flux in order to support the join clause.

Overall, QuestDB verifies the title of the fastest time series database, defeating InfluxDB and TimescaleDB. With its support for the InfluxDB line protocol, ingestion of data is a cakewalk and using the SQL capabilities, it seems that QuestDB has managed to combine the advantages from both InfluxDB and TimescaleDB in a single database.



Figure 19: Comparing time series databases with TSBS, query type "high-cpu-all".

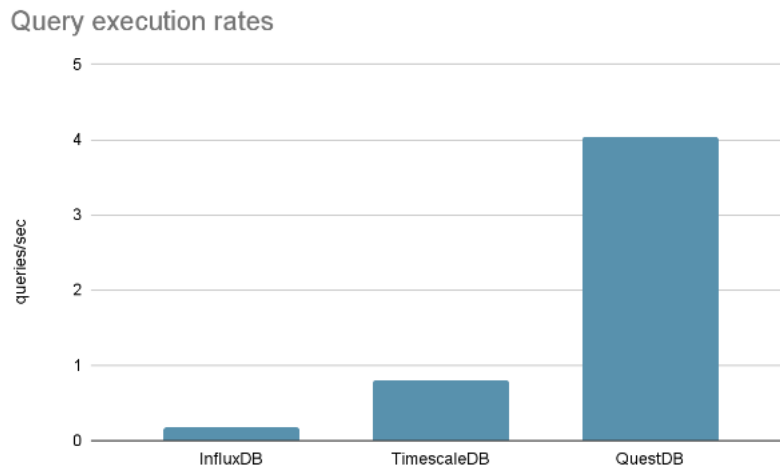


Figure 20: Query execution rates(queries/sec) for the query type "high-cpu-all".

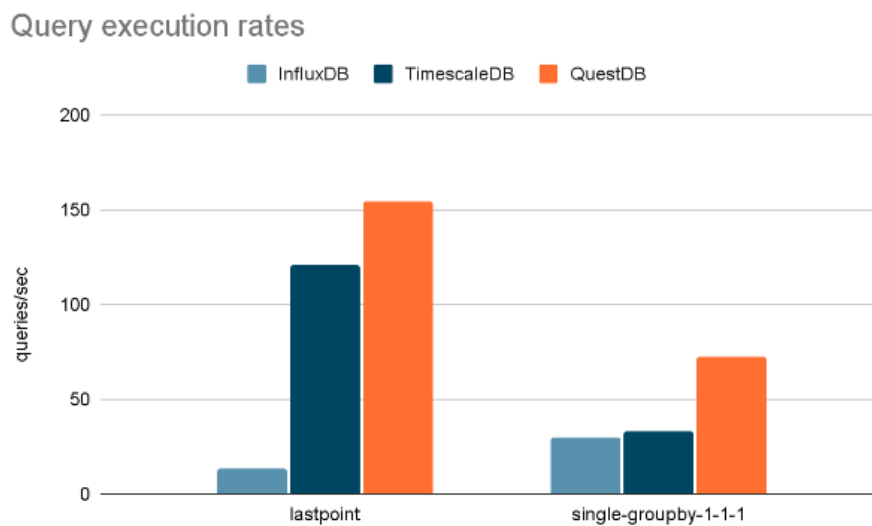


Figure 21: Query execution rates(queries/sec) for "lastpoint", "single-groupby-1-1-1".

#### 5.4 How cardinality impacts ingestion/query performance rates

In a database, high cardinality typically refers to two scenarios, either a table has many indexed columns or each indexed column has many unique values. In this

case the second one is examined. Also, here the DevOps use case is examined using the query type "high-cpu-all".

#### 5.4.1 Impacting ingestion rates

In TSDBs, cardinality refers to the number of unique values that are contained in a column. High cardinality means we have a large percentage of unique values, while low cardinality refers to a lot of repeated data. First, the databases will be tested adjusting the cardinality rates (scale flag).

Below, we can see the number of metrics captured per second for each database.

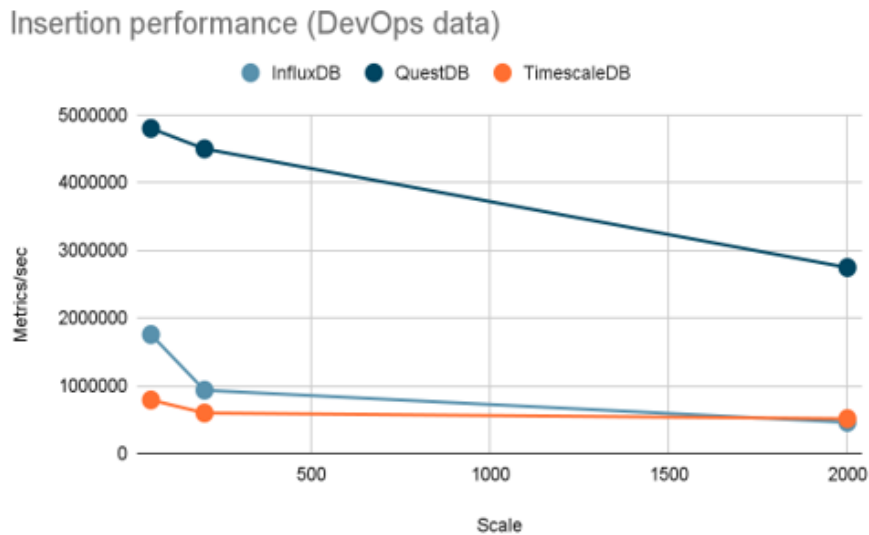


Figure 22: Insertion performance.

For workloads with small cardinality (up to 200 devices), InfluxDB defeats TimescaleDB. InfluxDB is not optimized for handling high cardinality data. Performance of InfluxDB declines faster than TimescaleDB (for the above reason). When cardinality reaches 2000+ devices, Timescale outperforms InfluxDB.

QuestDB takes advantage of the “Time based arrays” data model, which processes all the data in parallel. Data is ordered in memory and sorted by time before reaching the database, so QuestDB does not depend on computationally expensive indexes to reorder data.



### 5.4.2 Impacting query execution rates

Below, we can see the number of queries executed per second for each database.

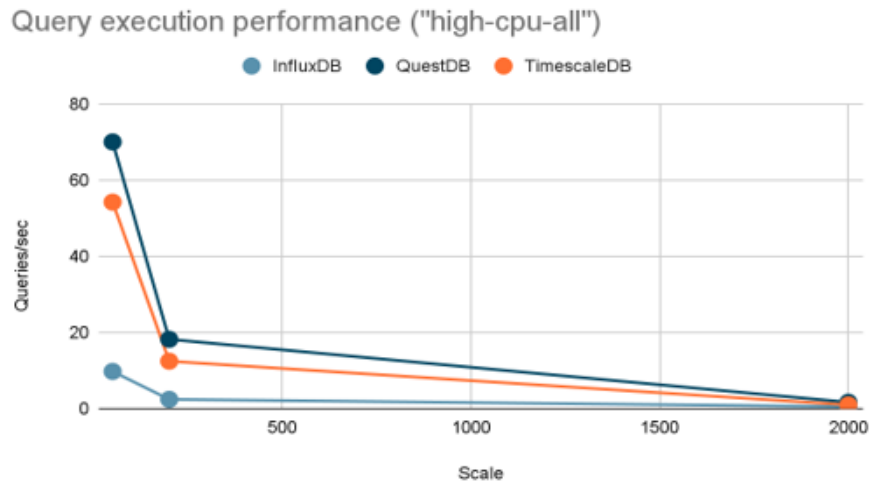


Figure 23: Execution performance.

With a small number of devices, QuestDB outperforms both InfluxDB and TimescaleDB. As the number of devices increases, the performance rate diminishes for all databases tending to 0. The reduction in performance probably occurs because of the allocation of data in multiple devices, where each one gets less data as scale increases.

## 5.5 How multi-threading affects ingestion

Here we will test how a change in the number of threads impacts ingestion, using the query type "high-cpu-all".

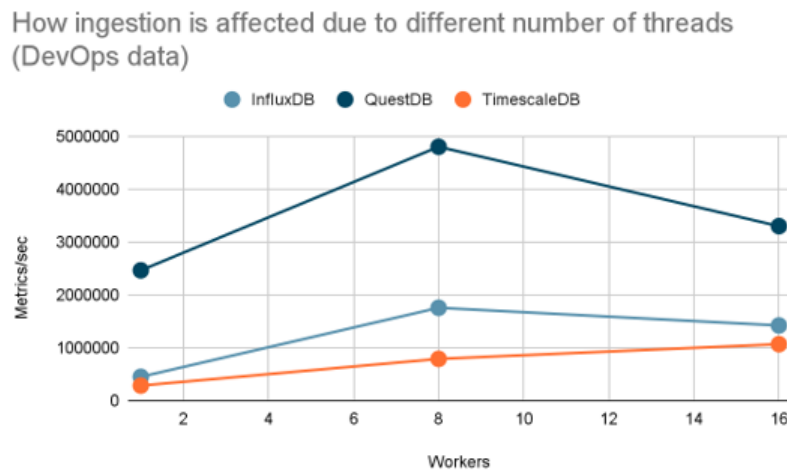


Figure 24: Testing ingestion rates with different number of workers.

QuestDB and InfluxDB reach maximum ingestion performance (metrics) using 8 threads (sensible because of the specs of the running machine), while TimescaleDB requires more workers to achieve maximum throughput.

## 5.6 How multi-threading affects query execution

Here we will test how a change in the number of threads impacts query execution, using again the query type "high-cpu-all".

How query performance is affected due to different number of threads ("high-cpu-all" query, DevOps data)

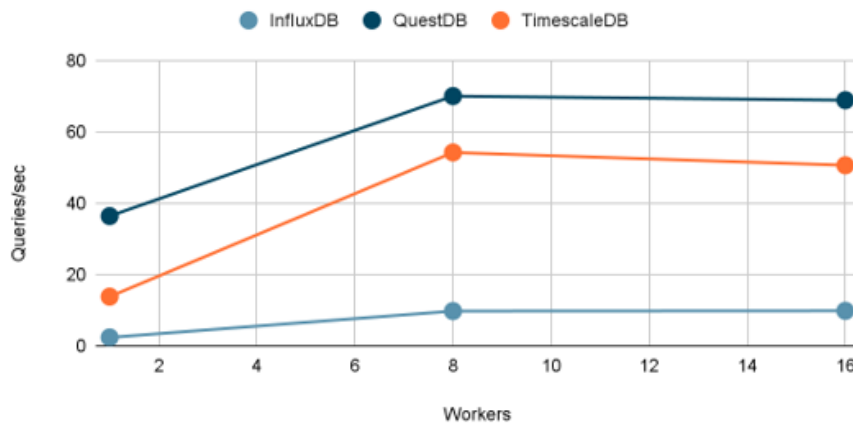


Figure 25: Testing query performance rates with different number of workers (query type "high-cpu-all").

All databases reach the maximum throughput with 8 workers, for the same reason as above.

## 5.7 Materialized views

A materialized view is a database object, used to store query results. Materialized views may help boost the performance of queries, especially when having to use computationally expensive indexes.

### 5.7.1 Materialized views in TSBS

Unfortunately, TSBS currently does not support continuous queries. Upon asking the developers, they reported that this is because in a challenge like this, while they could show on most systems that an aggregate query does improve significantly with pre-aggregated data, the frequency at which the aggregate would

be updated in real life usage differs between each system. That would make it difficult to benchmark during the time of a test.

### 5.7.2 Exploring TimescaleDB's continuous aggregates and InfluxDB's continuous queries

#### A. InfluxDB

By using continuous queries in InfluxDB we can automatically downsample data from a single field and write the results in another measurement in the same database.

For example, suppose we have a database named `noaa_water_database` (the dataset can be found [here](#)). The measurements in the database are:

*h2o\_feet, average\_temperature, average\_degree, h2o\_temperature.*

#### 1. Aggregates

```
CREATE CONTINUOUS QUERY "cq_basic"
ON "noaa_water_database"
BEGIN
SELECT mean("degrees") INTO "average_degrees"
FROM "average_temperature"
GROUP BY time(1h)
END
```

```
SELECT * FROM "cq_basic"
```

The original query is the following:

```
SELECT MEAN("degrees")
FROM "average_temperature"
```

```
GROUP BY time(1h)
LIMIT 5;
```

Both queries return the same output:

| time                | mean  |
|---------------------|-------|
| ----                | ----  |
| 1566000000000000000 | 79.6  |
| 1566003600000000000 | 81.5  |
| 1566007200000000000 | 81.8  |
| 1566010800000000000 | 80.05 |
| 1566014400000000000 | 79.85 |

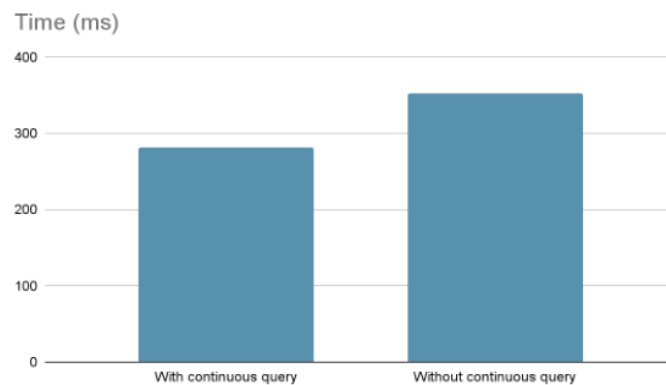


Figure 26: InfluxDB: Comparing execution time for the aggregate query using and without using the continuous query.

The original query processed 5 MB of data while the continuous one reduced the data size to just below 4 MB. The execution time using the continuous query

correspondingly saved 20% of the total time.

## 2. Joins

As we mentioned earlier, InfluxQL does not support joins.

### B. TimescaleDB

Like InfluxDB's continuous queries, continuous aggregates in TimescaleDB are designed to fasten queries. They use PostgreSQL's materialized views to refresh queries in order to change the whole dataset once you run a query.

We will use the same dataset here (noaa water data).

The same examples from above can be translated in continuous aggregates like this:

#### 1. Aggregates

```
CREATE MATERIALIZED VIEW cq_basic
WITH (timescaledb.continuous) AS
SELECT mean("degrees"), time_bucket(INTERVAL '1 hour', time) AS bucket,
FROM average_temperature
GROUP BY bucket;
```

After that we can create a policy to refresh the view every hour:

```
SELECT add_continuous_aggregate_policy('cq_basic',
start_offset => INTERVAL '1 month',
end_offset => INTERVAL '1 day',
schedule_interval => INTERVAL '1 hour');
```

Then, we can query the view like this:

```
SELECT * FROM cq_basic ;
```

We get the same results as before:

| time                | mean  |
|---------------------|-------|
| ----                | ----  |
| 1566000000000000000 | 79.6  |
| 1566003600000000000 | 81.5  |
| 1566007200000000000 | 81.8  |
| 1566010800000000000 | 80.05 |
| 1566014400000000000 | 79.85 |

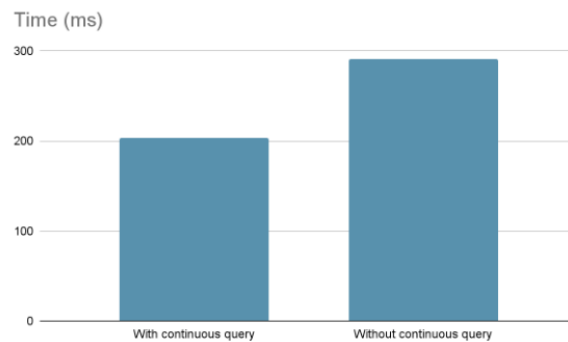


Figure 27: TimescaleDB: Comparing execution time for the aggregate query using and without using the continuous aggregate.

The original query processed 5 MB of data while the continuous one reduced the data size to approximately 3.5 MB. The execution time using the continuous query correspondingly saved 30% of the total time.

## 2. Joins

```

CREATE MATERIALIZED VIEW cq_basic
WITH (timescaledb.continuous) AS
SELECT degrees, time_bucket(INTERVAL '1 hour', time) AS bucket,
FROM average_temperature
INNER JOIN h2o_temperature
WHERE h2o_temperature.time=average_temperature.time
GROUP BY bucket
LIMIT 5;

```

Then the output we get is the following:

| time                | degrees |
|---------------------|---------|
| ----                | ----    |
| 1566000000000000000 | 82.5    |
| 1566003600000000000 | 84.15   |
| 1566007200000000000 | 78.9    |
| 1566010800000000000 | 80.5    |
| 1566014400000000000 | 79.4    |

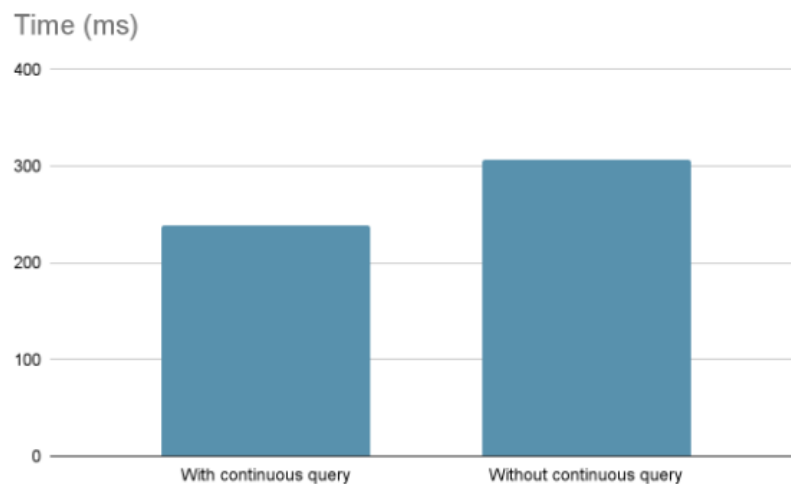


Figure 28: TimescaleDB: Comparing execution time for the join query using



and without using the continuous aggregate.

The original query processed 5 MB of data while the continuous one reduced the data size to 3.77 MB. The execution time using the continuous query correspondingly saved 24.5% of the total time.

So we see that in both databases, InfluxDB and TimescaleDB the use of continuous features helps speed up query performance, especially in the cases we covered: **join** and **aggregation** operations.

As new data is inserted (or old data is modified), continuous aggregates are refreshed automatically. Pre-aggregated data from the continuous aggregate adds to the new data that is aggregated and this gives up-to-date results for each query.

As stated before, continuous aggregates require a `time_bucket`, which allows us to determine a time interval, instead of using specific timestamps.

Continuous aggregates support most aggregate functions, like SUM, AVG, MEAN. However, aggregates using ORDER BY and DISTINCT cannot be used with continuous aggregates because they are not possible for PostgreSQL parallelization. Also, TimescaleDB currently does not support the FILTER clause and windows functions in continuous aggregates.

Continuous queries can also prove useful when for example, there is a data stream or when writing historical data. The execution service executes again the continuous query for periods that have passed the maximum interval setting (feature “recompute-no-older-than”).

## 5.8 Benchmarking - conclusion

In conclusion, it can be verified that QuestDB is the fastest of these three databases, especially in ingestion rates and in some types of queries. For applications that are mainly concerned to query performance, QuestDB is the ideal

solution with the only drawback that it has not become so popular yet, so feedback and sources may be limited.

InfluxDB is a powerhouse in the TSDBs domain, it has the powerful InfluxDB line protocol, which ensures fast ingestion as we observed earlier in action and a big community behind.

TimescaleDB has the flexibility of SQL as a query language, which is a big facilitation for most users having a background in relational databases and currently has a bigger audience than QuestDB.

So the choice of a time series database depends on the user's real needs.

Finally, the use of materialized views in InfluxDB and TimescaleDB can help boost the query execution performance and is a feature that may diminish the distance between these two and QuestDB in speed rates.

## 6 Conclusion

Time series are very common in daily life. As the number of data in the 21st century is on the rise, the need for related databases optimizing scalability and usability is more apparent than ever.

In this survey, a variety of time series databases were mentioned and their capabilities were tested. Based on specific uses cases, their strengths and weaknesses were found so that the user may choose the best one, according to their real needs. InfluxDB was found to be strong on data ingestion, while QuestDB outperformed both InfluxDB and TimescaleDB on query execution. TimescaleDB though seemed to perform exceptionally on DevOps data (out-of-order data, with missing or empty entries, closer to real life). Materialized views are confirmed to increase the query performance in both InfluxDB and TimescaleDB, while QuestDB, even though not supporting this feature, may use the InfluxLineProtocol to gain a rise in query execution performance.

## 7 Appendix

### 7.1 Python script to handle message formatting in QuestDB

```
from influx_line_protocol import Metric

import datetime
import socket
import random

# QuestDB installed on local listening for TCP/UDP packets on
port 9009

host = "localhost"
port = 9009

# Current time in nanoseconds
def current_timestamp():
    return int((datetime.datetime.utcnow() -
datetime.datetime(1970, 1, 1)).total_seconds() * 1000) * 1000000

# Create a batch of 10000 random metrics to ingest.
metric = Metric("engine_diagnostics")
str_metric = ""
metrics = ""
for i in range(10000):
    metric.with_timestamp(current_timestamp())
    metric.add_tag('Make', 'Volkswagen')
    metric.add_tag('Model', 'Polo')
    metric.add_value('engine_temperature', random.uniform(120.0,
150.0))
    metric.add_value('oil_gauge', random.uniform(91.0,99.5))
    str_metric = str(metric)
    str_metric += "\n"
    metrics += str_metric
```

```

# Convert string to bytes
bytes_metric = bytes(metrics, "utf-8")

# Open a socket & connect to 9009
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((host, port))
s.sendall(bytes_metric)
s.close()

```

## 7.2 Execution instructions for TSBS

As described earlier, TSBS<sup>7</sup> is an open-source benchmarking tool used for benchmarking in time series databases. It can be used to measure both data ingestion and query execution.

### Instructions

1. Installation (recommended in docker format) of the TSDBs systems you are interested in and supported by TSBS. Indicatively, suppose we choose InfluxDB, TimescaleDB, QuestDB. For this tutorial, we are using a Linux OS. The versions of the databases used in Docker for the needs of this thesis <sup>8</sup>:

a) **QuestDB.**

```
docker run -p 9000:9000 -p 9009:9009 -p 8812:8812 -p 9003:9003 questdb/questdb
```

b) **InfluxDB.**

```
docker run --name=influxdb -d -p 8086:8086 influxdb:1.8
```

c) **TimescaleDB.**

```
docker run -d --name timescaledb -p 5432:5432
-e POSTGRES_PASSWORD=password timescale/timescaledb:latest-pg14
```

---

<sup>7</sup>[TSBS Detailed manual](#)

<sup>8</sup>These versions were the latest ones at the writing time of the thesis. On benchmarking, it is recommended to obtain the latest versions of the databases for more up-to-date results.

2. Then, we produce test data. For example, at QuestDB:

```
./tsbs_generate_data  
-use-case="devops" -seed=123 -scale=200  
-timestamp-start="2016-01-01T00:00:00Z"  
-timestamp-end="2016-01-02T00:00:00Z"  
-log-interval="10s" -format="influx" > /tmp/questdb-data
```

3. We load the data into the database (*tsbs\_load command*) and get benchmarking results for the data ingestion.

```
./tsbs_load_questdb -file /tmp/questdb-data -workers 8
```

4. We produce queries about the benchmarking process.

```
/tsbs_generate_queries -use-case="devops" -seed=123 -scale=200 -timestamp-  
start="2016-01-01T00:00:00Z" -timestamp-end="2016-01-02T00:00:01Z" -  
queries=1000 -query-type="high-cpu-all" -format="questdb" > /tmp/queries_questdb-  
high-cpu-all
```

5. We execute the queries.

```
/tsbs_run_queries_questdb -file /tmp/queries_questdb-high-cpu-all - work-  
ers=8 -print-interval 500
```

Note: Special parameters, such as the number of workers (threads) or the value of scale, which determines cardinality, can be determined based on your preferences.

### Specific comments

- If you want to see the data generated by the TSBS (suppose we have named it influx-data) you may do so with the command:

```
$cat /tmp/influx-data
```

- If you want to see how large your files are in the central folder of TSBS:

```
$ ls -lah /tmp/
```

- When you double-check the QuestDB load, the second time you may receive the following error: *CPU Table already exists*. (shouldn't happen if you make sure the cache is cold). To solve the problem, you will need to delete the QuestDB container:

*docker ps -a (take the id of the QuestDB container)*

*docker stop <container\_id>*

*docker rm <container\_id>*

And then reinstall the container as we showed earlier. This error does not appear for the other two databases (InfluxDB, TimescaleDB).

## 8 References

### I. Text sources

- [1] InfluxData, “What is time series data” ([URL](#))
- [2] Vasavi Ayalasomayajula, “Visualizing Time Series Data: 7 types of Temporal Visualizations” ([URL](#))
- [3] Steve Ranger, “What is the IoT? Everything you need to know about the Internet of Things right now” ([URL](#))
- [4] Ajay Kulkarni, Ryan Booz on Timescale, “What the heck is time-series data (and why I need a time series database?)” ([URL](#))
- [5] InfluxData, “InfluxDB data elements” ([URL](#))
- [6] Syeda Naqvi, Sofia Yfantidou, “Time Series Databases and InfluxDB” ([URL](#))
- [7] InfluxData, “InfluxDB internals” ([URL](#))
- [8] Gregory Trubetskoy, “How InfluxDB stores data” ([URL](#))
- [9] InfluxData, “Query data in InfluxDB” ([URL](#))
- [10] InfluxData, “InfluxDB and Kafka: How InfluxData Uses Kafka in Production” ([URL](#))
- [11] Prometheus, “Overview” ([URL](#))
- [12] Mike Elsmore, “Prometheus vs. InfluxDB: A Monitoring Comparison” ([URL](#))
- [13] Ivan Celichko, “Prometheus is not a tsdb” ([URL](#))
- [14] Timescale Docs, “Overview” ([URL](#))
- [15] Israel Imru, “Time Series Database Comparison” ([URL](#))
- [16] Andreas Bader, Oliver Kopp, Michael Falkenthal, “Comparison of Time Series Databases” ([URL](#))
- [17] Ted Dunning & Ellen Friedman “Time Series Databases, New Ways to Store and Access Data”, Chapter 7
- [18] Alexandra Mazak, Sabine Wolny, Abel Gomez, Jordi Cabot, Manuel Wimmer, Gerti Kappel, “Temporal Models on Time Series Databases” ([URL](#))
- [19] Sunil Kumar, Saravanan C, “A comprehensive study of data visualization



tool-Grafana” ([URL](#))

[20] Time Series Benchmark Suite (TSBS) ([URL](#))

[21] Kovid Rathee, “QuestDB vs TimescaleDB” ([URL](#))

[22] Sebastian Insausti, “Which time series is better: InfluxDB vs TimescaleDB” ([URL](#))

[23] Tancrede Colard, “Speeding up Influx Line Protocol” ([URL](#))

[24] Kovid Rathee, “Schemaless Ingestion in QuestDB using InfluxDB Line Protocol” ([URL](#))

[25] InfluxData, “Under the hood with continuous queries-part ii” ([URL](#))

[26] QuestDB, “Aggregating billions of rows per second with SIMD” ([URL](#))

[27] NOAA sample data ([URL](#))

[28] QuestDB documentation ([URL](#))

## II. Image sources

Figure 1: ([URL](#))

Figure 2: ([URL](#))

Figure 4: ([URL](#))

Figure 5: ([URL](#))

Figure 6: ([URL](#))

Figure 7: ([URL](#))

Figure 9: ([URL](#))

Figure 10: ([URL](#))

Figure 11: ([URL](#))

Figure 15: ([URL](#))

Figure 16: ([URL](#))