Time Series Databases: Core Principles and a Comparative Benchmarking Exploration

December 2023

Zbyněk Juřica
juriczby@fit.cvut.cz
Faculty of Information Technology
Czech Technical University in Prague

Abstract

This paper provides an introductory overview of time series databases, focusing on fundamental principles and features. It covers the basics of managing time series data, with an exploration of prominent databases - InfluxDB, TimescaleDB, QuestDB, PostgreSQL, and MongoDB. A key component of the study is a benchmarking analysis to evaluate the performance of these databases in typical use cases in the IoT domain. This benchmarking is designed to offer practical insights for selecting an appropriate database for time series data management, considering ease of use and efficiency in typical real-world scenarios.

1 Preface

This paper assumes a basic understanding of PostgreSQL (or similar) and MongoDB. I will skip over the general concepts underpinning databases in general and will focus solely on features and issues specific to handling time series data.

2 Introduction

Time series data is no longer limited to finance; it's everywhere. Businesses now use it from various sources like IoT devices, performance metrics, weather data, sales numbers, and more. To make the most of this data and support large-scale operations, efficient solutions for storing, processing, analyzing, and querying are essential.

Throughout this paper, we will work with some fictional data to demonstrate the usage of these technologies in practice. Consider that we have deployed some IoT devices and wish to monitor them. We maintain n such devices, each sending snapshots every 10 minutes; these snapshots contain:

1. **`stationId` and `timestamp`** - the id of the station (IoT device) and the timestamp of the snapshot.
2. **`voltage`** - Current voltage measurement.
3. **`outsideTemperature`** - Current outside temperature.
4. **`heatingTemperature`** - Current temperature of some heating component.
5. **`coolingTemperature`** - Current temperature of some cooling component.

2.1 Definition of Time Series Data

Time series data consist of measurements or events that are tracked, monitored, down-sampled, and aggregated over time. The key difference between normal data and time series data is that the latter has a time component that is crucial in understanding the data itself. Another way of looking at it is that the data can be indexed or listed in time order. [1]–[3]

Examples of events that can be tracked:

- Server metrics
- Application performance monitoring

- Network data
- Sensor data
- Events
- Clicks

In a way, this definition suggests that time series data can be treated like any other data, allowing for storage in a general-purpose database. This approach is entirely valid; however, knowing you will work with time series data allows for certain assumptions and more specific approaches in some areas. This gives us opportunities for optimizations, performance benefits, compression and also developer specific tools and improved developer experience and productivity.

2.2 Challenges when dealing with Time Series Data

There are some common challenges when dealing with time series data. I will outline these challenges and examine how databases designed for such data address and combat these issues.

- **Data volume** - Commonly, time series data will come in with high intensity.
    - Data ingestion - Rapidly and continuously process data coming in from various sources
    - Query speed - Ensuring timely retrieval of information from big datasets.
    - Storage efficiency - Efficient storing of time series data on disk utilizing compression techniques
    - Aggregations - Creating summaries and aggregations efficiently and possibly supporting real-time aggregation of incoming data stream
- **Scalability** - The system needs to be ready to scale depending on the demand
    - Horizontal scalability - Scaling the system horizontally while maintaining some level consistency within the system
    - Distributed data - Maintaining data consistency and integrity cross distributed system
    - Load balancing - Distributing work among many servers or clusters
- **Data variability** - Time series data might come in with irregular timestamps, missing values, noisy data, varying levels of data quality
- **Data retention** - Defining when and how to remove old or irrelevant data

Another factor is also the ease of working with the time component of the data due to the database' features, syntax and approach.

3 Time Series Database

I have selected three time series databases based on the latest trends, popularity, and interest. I've also chosen 2 multi-purpose databases to better understand the difference and advantages the time series databases actually bring and if it actually does make sense to use them over a conventional solution. [4]

I will go over each database and describe their approach to time series data and what are the specific and most important features they offer.

3.1 InfluxDB

InfluxDB is a prominent open-source time series database. InfluxDB, along with its associated tooling, is specifically designed for time series data and its use cases. InfluxDB's combination of performance, scalability and ease of use has made it the number one go-to solution in the space of time series databases.

In this section I will go over the basics of working with InfluxDB and also point out some of the specific and more advanced features of InfluxDB and also some quality of life features it offers. [5], [6]

3.1.1 Distributions and Ecosystem

InfluxDB comes in different forms. The distributions can be categorized as either managed or self-managed.

If you are looking for a managed solution, InfluxDB offers 3 options:

- **InfluxDB Cloud (TSM)** - Standard cloud-based solution.
- **InfluxDB Cloud Serverless** - Automatically scales resources based on demand.
- **InfluxDB Cloud Dedicated** - Dedicated environment to a single tenant with more customizations, offering more control for security and compliance requirements.

InfluxDB also offers self-managed solutions:

- **InfluxDB OSS** - The most basic, open-source version for individual use or smaller applications.
- **InfluxDB Clustered** - Designed for making a cluster on your own infrastructure.
- **InfluxDB Enterprise** - A more advanced version than OSS, suitable for big businesses looking for robust and scalable solutions.

The InfluxDB world is, as of time of the writing, disrupted by the partial release (for only selected distributions) of InfluxDB 3.0 and their transition from Flux (the query language they use) to SQL. These new changes are supposed to roll out to all distribution, but currently 3.0 is only available on Cloud Serverless, Cloud Dedicated and Clustered distributions. As such this paper will still focus on the previous version with no SQL capabilities using the InfluxDB OSS 2.7 distribution.

3.1.2 Writing data using Line Protocol

InfluxDB uses Line protocol, a simple format for providing the necessary information to store data. It simplifies sending data from any device - simply send an HTTP/HTTPS POST request to InfluxDB with the data in the correct format.

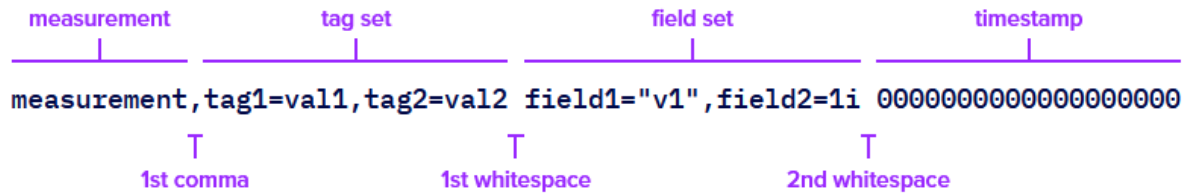

Figure 1: Line Protocol format

- ***Measurement** - name of the measurement (analogous to table name).
- ***Field set** - key-value pairs containing the data.
- **Tag set** - key-value pairs containing metadata (analogous to indexed columns for querying data).
- **Timestamp** - the actual time in nanoseconds (defaults to current time).

Measurements are then stored in a bucket - a set of measurements.

Even though InfluxDB is very simple in this regard, you should still keep in mind some basic rules when desigining your schema (the way you represent data using Line Protocol).

You should not use measurement, field keys or tag keys as a way of storing data, they should be static and general. In our example, stations can have regions (north, south, west, east). We should use a tag to store this information (e.g. `region=north`), definitely not storing it in the measurement name (e.g. `iot_snapshot.north`) - such representation would make querying more difficult and costly.

In general, fields are for storing unique and numeric data, tags are for metadata to improve query performance. Tags should be kept simple, split complex data into multiple tags (e.g. don't combine city and country into one tag, but split them into two).

3.1.3 Querying data using Flux

Flux is a functional scripting language for querying data from InfluxDB and other sources. The simplified idea of querying using Flux is to get the data from a source and pipe it through functions that alter the data in a desired way to finally obtain the output we want.

- **`from()`** - Queries data from a bucket.
- **`range()`** - Filters data based on time range.
- **`filter()`** - Filters data based on measurement name, tags, fields.
- **`sort()`** - Sort based on columns.
- **`limit()`** - Get only n first results.
- **Pipe forward operator `|>`** - Pipes output of one function as an input to the next function.

Flux offers many more functionalities for windows, aggregates, moving average, histograms, or even joins and many more.

Personally, as an enthusiast of functional programming, I think it is a very elegant, but yet powerful way of querying data, even though it requires you to learn some of the concepts, which I think are pretty intuitive.

3.1.4 Querying data using InfluxQL

InfluxDB offers another way of querying data - InfluxQL, which is an SQL-like query language.

Queries follow an SQL-like pattern of:

- **`SELECT`** - Which fields and tags to query.
- **`FROM`** - Which measurement to query.
- **`WHERE`** - Filter data based on fields, tags and time.
- **`ORDER BY`** - Sort the data.
- **`LIMIT`** - Get first n results.

InfluxQL also supports functions that can be applied to the fields and tags specified in the SELECT part of the query. These functions can be:

- **Aggregate Functions** - count, mean, median, sum, ...
- **Selector Functions** - min, max, percentile, ...
- **Transformation Functions** - abs, log, moving average, sqrt, round, ...
- **Advanced Technical Functions** - exponential moving average, ...

3.1.5 InfluxDB Tasks

Tasks are a very powerful feature of InfluxDB. They allow you to create a script that takes a stream of input data, processes it and then writes it back to InfluxDB or does some other action with it.

Each task contains options - the name and scheduling (how often/when it runs), and the script - detailing the data it processes, how it is processed, and where the results are stored.

Tasks can be used for continuous aggregations, data transformations, data analysis, data retention policies, downsampling of data and so much more.

3.1.6 TICK Stack

InfluxDB is actually a part of a bigger ecosystem of tools to provide a platform that can capture, monitor, store and visualize time series data.

Telegraf

A plugin-driven server agent collects data from various sources and stores it in InfluxDB.

It has capabilities for working with specific systems, but also with other services and datastores such as Kafka or MQTT.

It can also create aggregates as it is collecting the data or do other analysis. [7]

InfluxDB

The database that stores the data for later analysis and queries.

On its own, InfluxDB also offers a basic UI for simple visualizations and querying of data.

Chronograf

User interface for visualizations and alerting. It makes it easy to build dashboards for monitoring and real-time visualizations. [8]

Kapacitor

A tool for advanced data analysis using specific functions, machine learning and statistical approaches for detecting anomalies. [9]

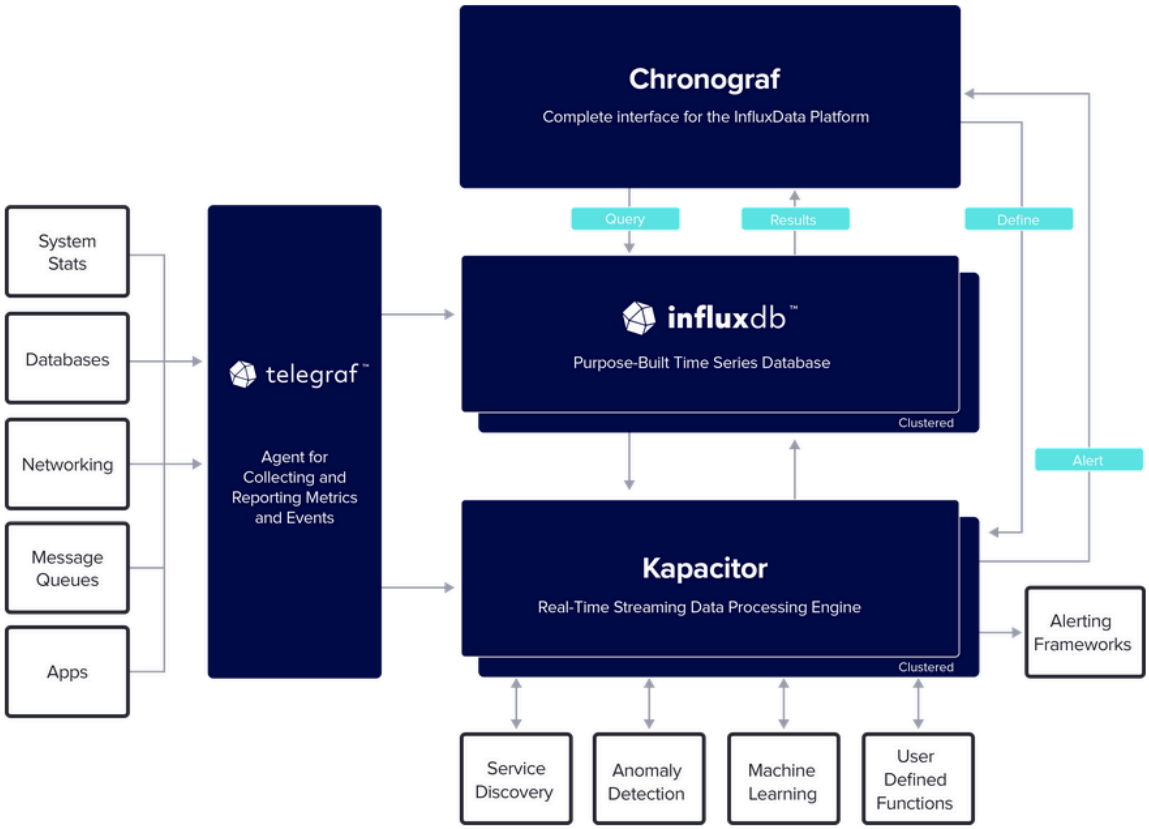

Figure 2: TICK stack - platform for capturing, monitoring, storing, and visualizing time series data

3.1.7 Examples

After we create a bucket to hold our measurements (let's call it `station_data`). Then we can start sending data using the line protocol on `http://{IP}:8086/api/v2/write?org={org}`

`&bucket={bucket_name}&precision=s`. We can specify the organization and bucket for data insertion, as well as the precision (maximum precision is nanoseconds). We don't need to create a measurement schema or anything like that, we can just start inserting data:

```
iot_snapshots,

stationId=1

voltage=3.3,
outside_temperature=22.0,
heating_temperature=25.0,
cooling_temperature=18.0
```

Listing 1: Inserting data using Line Protocol

I formatted the code for the purposes of this paper, but this would all be on one line. Remember that there is a space between tags and fields (for Influx to know where tags end and fields start).

To get all data from the last day from station with id 1, we can run the following query in Flux:

```
from(bucket: "stations")
  |> range(start: -1d)
  |> filter(fn: (r) => r._measurement == "iot_snapshots" and r.stationId == "1")
```

Listing 2: Querying last day's data from station 1 using Flux

or similarly in InfluxQL:

```
SELECT * FROM "iot_snapshots"
WHERE "stationId" = '1' AND time > now() - 1d
```

Listing 3: Querying last day's data from station 1 using InfluxQL

We can also create a continuous aggregate using an Influx Task:

```
option task = {name: "CalculateHourlyAvgTemp", every: 1h}

from(bucket: "stations")
  |> range(start: -task.every)
  |> filter(fn: (r) => r._measurement == "iot_snapshots" && r._field ==
"outside_temperature")
  |> aggregateWindow(every: 1h, fn: mean)
  |> to(bucket: "stations")
```

Listing 4: Creating a task for continuous aggregates

3.2 TimescaleDB

TimescaleDB is an open-source extension for PostgreSQL. It combines the best of both worlds by integrating the popular SQL database with added capabilities for handling time series data. Consequently, you can continue using PostgreSQL as usual and connect existing tables to Timescale's Hypertables, which store the time series data, therefore getting the best of both worlds.

Working with TimescaleDB is pretty much identical as to working with a normal PostgreSQL databases and as such I will focus on pointing out the new features and differences TimescaleDB brings to the table. [10]

3.2.1 Hyptertables

Hypertables, as the name suggests, function like normal tables but with enhanced capabilities for time series data. Existing alongside regular tables, they allow interaction much like standard tables.

You use hypertables for time series data, which will give you improved insert and query performance.

Hypertables automatically partition data by time, and optionally, by space. Data is partitioned into chunks, with each chunk holding data from a specific time range (the default is 7 days). The chunk size should be small enough to fit into memory (including indexes) for improved insert and query times, but not so small as to hinder query planning and compression. All hypertables come with an index on the time component (descending).

3.2.2 Time buckets

The `time_bucket` function enables aggregation of data into time-based buckets. You can specify the size of the bucket (= time span - number of minutes/hours/days/...).

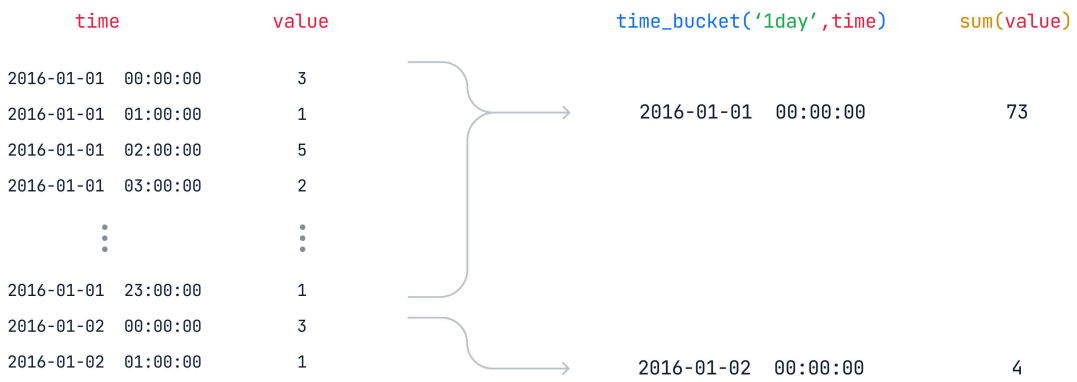

Figure 3: Visualization of how time buckets group data together

Time buckets group data starting from a specific origin, preventing scenarios where weekly time buckets might begin midweek, such as on a Wednesday, if that's when the first data point occurs. This would shift all the buckets to start on Wednesday, which could be confusing. The default origin is January 3, 2000, at 00:00:00 - chosen because it's a Monday. For monthly, yearly, or larger time buckets, the origin is January 1.

3.2.3 Compression

Compression can drastically reduce chunk size (by more than 90%), which also speeds up queries. Enabling compression means that multiple records in one chunk gets grouped into a single row. Basically the columns act as an array to hold the data.

There is also a trade-off with compression. Queries that target a lot of columns run better on uncompressed data. Queries that target only a specific column run better on compressed data.

It is also likely that newer data will get queried with more (if not all) columns. Queries like show me the current state of the station are more likely for newer data. As data gets older it becomes more likely that the query will focus on just one column instead. These might be aggregate queries getting the average temperature in the past month.

Another problem with compression is modifying the data. Updating the queries that have been compressed is possible but is inefficient. Deleting compressed data is not possible.

Therefore, compression is not done immediately and setting the correct policy on when data should get compressed is crucial for query speeds and modifying. It depends on the use case, but it should be set to when queries go from shallow and wide to deep and narrow and are unlikely to be deleted.

3.2.4 Continuous aggregates

Continuous aggregates are a special kind of hypertable - they get automatically refreshed and updated in the background as new data come in, or old ones are modified. These updates are done incrementally, further saving resources. Since they are essentially hypertables, you can interact with them as with any other hypertable - querying, enabling compression, etc

There is a distinction between continuous aggregates and real-time aggregates. Real-time aggregates are a special type of continuous aggregates that include even the latest data points. Normally the aggregates are periodically updated; real-time aggregates are updated with each incoming data point. Changes to historical data are still just eventually consistent.

You can also create continuous aggregates on top of each other. For example, we can aggregate by second and then use this aggregation to build an hourly aggregation, which can be used for daily aggregation.

PostgreSQL offers materialized views, which cache the results of complex queries. They are not update regularly and need to be manually refreshed. They are also not updated incrementally. Therefore they require more resources to maintain.

Continuous aggregates can also be used for data retention policies. Data gets continually aggregated and when it reaches certain age it can get dropped.

3.2.5 Hyperfunctions

Hyperfunctions are specialized functions for analyzing time series data. The number of these functions is high, they save time in simple queries, but especially in deep analysis of underlying data.

Examples of hyperfunctions:

- **General purpose and utility functions**:
  - time_bucket - see timebuckets.
  - first/last - getting the first and last value within a time range.
  - interpolate - fill in missing values using linear interpolation.
- **Aggregation functions**:
  - `min`/`max` - getting the n greatest/smallest.
  - `average`, `sum` and many more general aggregations.
- **Statistical functions**:
  - `stddev`/`variance` - for getting the standard deviation/variance.
  - `histogram` - partitions the data into n buckets.
- **Financial functions**:
  - `exponential_moving_average` - for identifying trends over time.
  - `candlestick` - for getting candlestick data.
  - `close`/`high`/`low` - getting the closing/high/low price from a candlestick aggregate.

3.2.6 Example usage

You can convert an existing table to a TimescaleDB hypertable by running the following code, with the name of your table and the name of the column that holds the time data - this column needs to have the data type of `timestamptz`.

```
CREATE TABLE iot_snapshots (
    time TIMESTAMPTZ NOT NULL,
    stationId INT NOT NULL,
    voltage REAL,
    outside_temperature REAL,
    heating_temperature REAL,
    cooling_temperature REAL
);
SELECT create_hypertable('iot_snapshots', by_range('time'));
```

Listing 5: Creating a hypertable to store snapshot data

To query data from such a table - for instance, to retrieve all data from the past day:

```
SELECT * FROM iot_snapshots
WHERE stationId = 1 AND time > NOW() - INTERVAL '1 day';
```

Listing 6: Querying last day's data from station 1

A time bucket can be created from this table by specifying the data it should contain and the size:

```
SELECT time_bucket('1 day', time) AS bucket,
  avg(outside_temperature) AS avg_outside_temperature
FROM iot_snapshots
GROUP BY bucket
ORDER BY bucket ASC;
```

Listing 7: Creating a time bucket with average outside temperature for each day

A continuous aggregate can also be created to calculate the average temperature for each day at each station:

```
CREATE MATERIALIZED VIEW average_daily_outside_temperature
WITH (timescaledb.continuous) AS
SELECT stationId,
   time_bucket(INTERVAL '1 day', time) AS bucket,
   AVG(outside_temperature),
FROM iot_snapshots
GROUP BY stationId, bucket;
```

Listing 8: Creating a continuous aggregate with average oustide temperature per day

The key difference between the time bucket and continuous aggregate approach is that the time bucket calculation occurs each time the query is executed.

3.3 QuestDB

QuestDB is another database that specializes in time series data. It prides itself on being the fastest time series database with the biggest ingestion throughput. QuestDB combines support for many different protocols and features from different databases. It supports InfluxDB's Line Protocol, PostgreSQL Wire Protocol, REST API, Telegraf and many more. [11], [12]

3.3.1 Inserting data

When it comes to inserting, QuestDB offers several different ways:

- **InfluxDB's Line Protocol** - you can read more about this method in the InfluxDB chapter.
- **PostgreSQL Wire Protocol** - standard SQL insert.
- **REST API** with many capabilities:
  - GET Request to `/exec?query=...` to run ad-hoc SQL queries.
  - POST Request to `/imp` to insert csv data.
  - Using the Web Console's editor to run queries or upload CSV files.

When it comes to inserting data it is recommended to pretty much default to Line Protocol for both Periodic batch ingest and Real-time ingest. If you want to find out more about the line protocol you can check it out in the InfluxDB chapter.

3.3.2 Querying data

For querying, several options are available:

- **PostgreSQL Wire Protocol** - standard SQL queries
  - You can even connect to QuestDB using psql
- **REST API**
  - GET Request to `/exp?query=...` to get data as CSV based on SQL query.
  - GET Request to `/exec?query=...` to get data as JSON based on SQL query.
  - Using the Web Console's editor run to an SQL query.

3.3.3 High Cardinality

Cardinality is a measure of how many distinct values make up a dataset. High cardinality means that there is a large number of unique values for the dataset. We can consider a cardinality of a column or an entire dataset/table. [13]

There are low cardinality columns such as `region` (with possible values `north`, `south`, `west`, `east`) - there can ever just be 4 possible distinct values. On the other hand, 'stationId' would be a high cardinality column as it has unlimited number of values (depending on how many stations we have, it might not be that bad).

The problem becomes worse, when we try to calculate the cardinality for a dataset - there the cardinalities multiply with each row, making it grow exponentially with each new row.

Time series databases often store many high cardinality columns, leading to datasets with high cardinality that can become slow.

QuestDB claims that, compared to its competitors - namely InfluxDB, it can deal with high cardinality data much better due to its column based storage model and other optimizations.

3.3.4 Functions

QuestDB offers a variety of utility, predicate, and aggregate functions:

- **Utility**
  - `coalesce` - takes n arguments and returns the first non-null argument.
  - Time functions - utility functions for converting time data to different formats, getting the month, day, etc.
- **Predicate**
  - Pattern matching - you can use `(string) ~ (regex)` to check if some string matches a regex; similarly `(string) LIKE (pattern)` to check if string matches a pattern with wildcards _ and `%` (match single char and 0 to n chars).
- **Aggregate**

- `avg`, `count`, `max`, `min`, `sum` - the common functions that you would expect.
- `stddev`, `variance` - stastical aggregates.
- `approx_percentile` - for calculating percentiles.

3.3.5 Integrations

QuestDB supports common third-party tools like Kafka, Prometheus, Spark, and Grafana.

Notably, it also supports connection to Telegraf (developed by InfluxDB), a client for collecting data from different inputs, processing them and sending them to various outputs.

3.3.6 Maturity

When I was going through the documentation of these databases it became very apparent that QuestDB is not as mature as the other databases on this list. They do not currently support as many features as their competitors.

Great example is indexes. I think of indexes as one of the most fundamental features of any database. QuestDB only supports indexing `SYMBOL` columns and the time component. I think this is a pretty drastic limitation.

I think that QuestDB is more focused on solving problems that might arise when using InfluxDB or similar databases - mainly performance issues with high cardinality data and high ingestion.

Their benchmarks, which should be considered cautiously, indicate significant outperformance in some categories compared to InfluxDB and TimescaleDB. On the other hand they are lacking some features that might make your life easier, such as continuous aggregates or other integrations.

3.3.7 Examples

Examples for this section are going to be a repeat of previous sections as ingestion is done using Line Protocol and querying is done using SQL.

```
iot_snapshots,

stationId=1

voltage=3.3,
outside_temperature=22.0,
heating_temperature=25.0,
cooling_temperature=18.0
```

Listing 9: Inserting data using Line Protocol

```
SELECT * FROM iot_snapshots
WHERE "stationId" = '1' AND timestamp > NOW() - INTERVAL '1 day';
```

Listing 10: Querying last day's data from station 1

3.4 MongoDB

MongoDB introduced dedicated support for time series data with their time series collections in version 5.0, furthermore in version 6.3 it automatically creates a compound index on the time and metadata fields. [3]

3.4.1 Time series collections

Time series collections use columnar storage format and store data in time order, which:

- Reduces complexity when working with time series data

- Improves query speeds
- Reduces disk usage
- Reduces I/O for read operations
- Increases cache usage

You can use time series collections as you would any other collection because they create a non-materialized view on top of the internal collection.

When creating the time series collection we can specify:

- Name of the collection
- Name of the field that holds the time data
- Name of the field that hold metadata
- Optionally granularity of time or bucket span
- Optionally expiration time in seconds

3.4.2 Views

MongoDB offers views which are read-only queryable objects defined by an aggregation pipeline. Views are not stored to disk; they are create on-demand and forgotten after.

We can use views for creating aggregates or metrics, later we can easily query this data instead of using the complicated aggregation pipeline construct.

3.4.3 On-demand Materialized Views

On-demand materialized views are a pre-computed aggregation pipeline whose results are then stored to the disk to be retrieved later.

You can then work with them as you would with any other collection - including creating indexes.

3.4.4 Examples

Firstly, we need to create time series collection:

```
db.createCollection("iot_snapshots", {
  timeseries: {
      timeField: "time",
      metaField: "stationId",
      granularity: "seconds"
  }
});
```

Listing 11: Creating a time series collection

Data can be inserted using either `insertOne` or `insertMany`, as follows:

```
db.iot_snapshots.insertOne({
  stationId: 1,
  time: new Date("2023-12-10T10:00:00Z"),
  voltage: 3.3,
  outside_temperature: 22.0,
  heating_temperature: 25.0,
  cooling_temperature: 18.0
});
```

Listing 12: Adding a datapoint

To query the data, we use `find`, where we specify the object to match against. We want to query data from station 1 (`stationdId: 1`) and limit ourselves to only last day's data - we can use `$gte` (greater

than or equal to). Note that if there are data points in the collection with future timestamps, they would also be included in the query result.

```
db.iot_snapshots.find({
    stationId: 1,
    time: {
        $gte: new Date(new Date().setDate(new Date().getDate() - 1))
    }
});
```

Listing 13: Querying last day's data from station 1

Similar to the previous databases, an aggregate query can be made to obtain the overall average temperature of station 1.

```
db.iot_snapshots.aggregate([
    { $match: { stationId: 1 } },
    { $group: {
        _id: "$stationId",
        averageOutsideTemperature: { $avg: "$outside_temperature" }
    }}
]);
```

Listing 14: Aggregate query - calculates the average outside temperature of station 1

We can even simulate the time bucketing idea from specialized time series databases using an aggregate query. This query will give us the average temperature on each day.

```
db.iot_snapshots.aggregate([
    { $match: { stationId: 1 } },
    { $group: {
        _id: {
            $dateTrunc: {
                date: "$time",
                unit: "day",
            }
        },
        averageOutsideTemperature: { $avg: "$outside_temperature" }
    }},
]);
```

Listing 15: Aggregate query - calculates daily averages of station 1

3.5 PostgreSQL

Last but not least I am going to quickly go over PostgreSQL. It is the most versatile database on this list, suitable for almost any use case with commendable performance. It offers fundamental building blocks that are general enough to build powerful and specific solutions.

I will not delve into extensive detail, as PostgreSQL does not have many features specifically tailored for time-series data. However, I think it is important for me to make it clear that PostgreSQL can absolutely be a great choice for time series data even if there are not features specifically designed for this use-case. [14]

3.5.1 Views
Views are virtual tables that represent a result of another query. They are used to simplify complex queries. They can be queried like normal tables but hide underlying complexities - each time we run a query on a view, the view needs to be recalculated, which can be costly.

They can be useful for advanced aggregations and transformations of time series data.

3.5.2 Materialized Views
Materialized views are views, where the query result gets stored as a physical table to be queried later again without recomputing the view.

They are useful for improving the performance of frequently run expensive operations, aggregations, and transformations.

3.5.3 Triggers
Triggers allow for running procedures on some event on a table (INSERT, UPDATE, DELETE).

In the context of time series data, triggers can automate the normalization or transformation of incoming data and fill missing data points. They can also be paired with a materialized view to create a continuous aggregate or time buckets (see Time buckets and Continuous Aggregates in TimescaleDB) - update a materialized view when the underlying data changes.

3.5.4 Window Function
The window function allows us to perform a calculation over multiple rows related to the current row. It is useful for calculating running totals, moving averages, and cumulative statistics, common requirements in time series analysis

3.5.5 Range types
PostgreSQL has range types that take two values of some specific type to make up a range that can then be manipulated more easily than when those values would be split.

Range types offered:

- `int4range` — Range of integer
- `int8range` — Range of bigint
- `numrange` — Range of numeric
- **`tsrange`** — Range of timestamp without time zone
- **`tstzrange`** — Range of timestamp with time zone
- **`daterange`** — Range of date

These types are particularly useful when working with certain types of time series data or aggregates - they can describe over what time period the aggregate was calculated.

4 Benchmarks
In this section, we delve into the benchmarking of five prominent databases: InfluxDB, TimescaleDB, QuestDB, PostgreSQL, and MongoDB. The primary aim is to evaluate the performance of these databases when handling time-series data. This performance evaluation extends beyond just the databases themselves to include the libraries used for connecting to them through Go, providing a holistic view of real-world database performance.

4.1 Methodology of benchmarks
In this benchmarking study, I evaluated the performance of five databases using a personal desktop computer equipped with an AMD Ryzen 7 5800X processor and 16GB of RAM, running Ubuntu 22.04.3

and Go version 1.19. The databases are accessed through their respective Go libraries, selected based on the official recommendations in their documentation. The entire setup is containerized using Docker Compose for consistency and isolation, with the setup and code available at https://github.com/zbyju/timeseries-benchmark.

For the benchmarks, I generate a dataset that mimics data from an IoT station, a common source of time-series data. This dataset includes various metrics indicative of real-world IoT scenarios:

- `Outside Temperature`: Exhibits natural fluctuations, representing typical environmental temperature changes
- `Voltage`: Primarily stable but occasionally shows spikes to simulate system failures or anomalies
- `Heating` and `Cooling Temperatures`: These values correlate with the outside temperature but will have significant variations when heating or cooling mechanisms are activated or deactivated

I used the InfluxDB UI that automatically comes with the InfluxDB database to visualize how the data can look like:

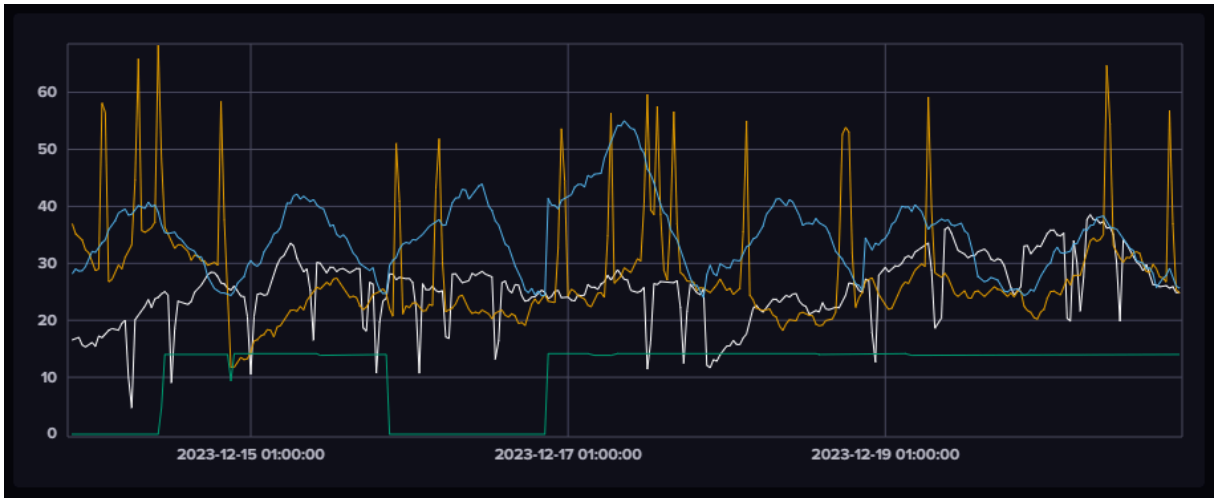

Figure 4: Visualization of generated data (blue = outside temperature, yellow = heating temperature, white = cooling temperature, green = voltage)

The benchmark focuses on three key operations: adding a large volume of data, querying data from the past day for a specific station, and computing the average of all data points of a specific station. Notably, the time spent on data generation, database connection, and initial setup is not included in the benchmark measurements. This ensures that the benchmarks accurately reflect the databases' efficiency in performing specific, relevant tasks with time-series data, mirroring real-world usage in IoT applications. [4], [6]

4.2 Benchmark results

The benchmark will show how databases deal with increasing amount of data. For each benchmark I generated 144 snapshots (simulating 1 snapshot each 10 minutes) for `D` amount of days and `S` amount of stations, to get the overall number of data inserted: $144 \times D \times S$.

I ran all benchmarks for inserting, querying and aggregating three times and took the minimum out of these runs.

4.3 Ingestion

Firstly, I looked at how long it takes to insert the data into the database.

Insert [s]	7,200	28,800	144,000	720,000	2,880,000
InfluxDB	0.08	0.18	0.61	2.79	10.50
PostgreSQL	0.08	0.30	1.22	5.80	23.88
TimescaleDB	0.11	0.35	1.46	6.73	27.48
QuestDB	0.01	0.03	0.22	0.78	3.21
MongoDB	0.06	0.22	0.87	4.16	19.41

Figure 5: Table of run times in seconds for inserting `N` data-points for each database

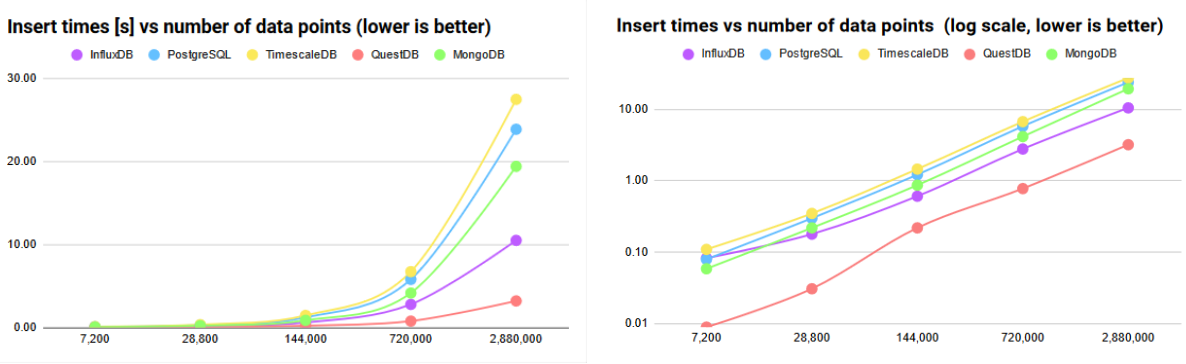

Figure 6: Visualization of run times in seconds for inserting `N` data-points for each database (chart on the right has log scale)

You can see that databases using Line Protocol perform significantly better than the competition. QuestDB is overall the best performing database, living up to the authors' claims. Both SQL databases perform the worst, while TimescaleDB is slightly slower, probably due to extra overhead when processing the data.

4.4 Querying

After the data is inserted, I queried it. A query that I thought would be useful in a real scenario is getting the last day's data of a particular station.

Get [ms]	7,200	28,800	144,000	720,000	2,880,000
InfluxDB	6.85	6.27	7.29	8.73	9.26
PostgreSQL	0.87	1.24	0.94	1.04	1.54
TimescaleDB	1.39	1.47	1.59	1.31	1.52
QuestDB	5.31	10.55	6.64	7.39	7.36
MongoDB	2.77	3.79	6.36	11.54	55.85

Figure 7: Table of run times in seconds for inserting `N` data-points for each database

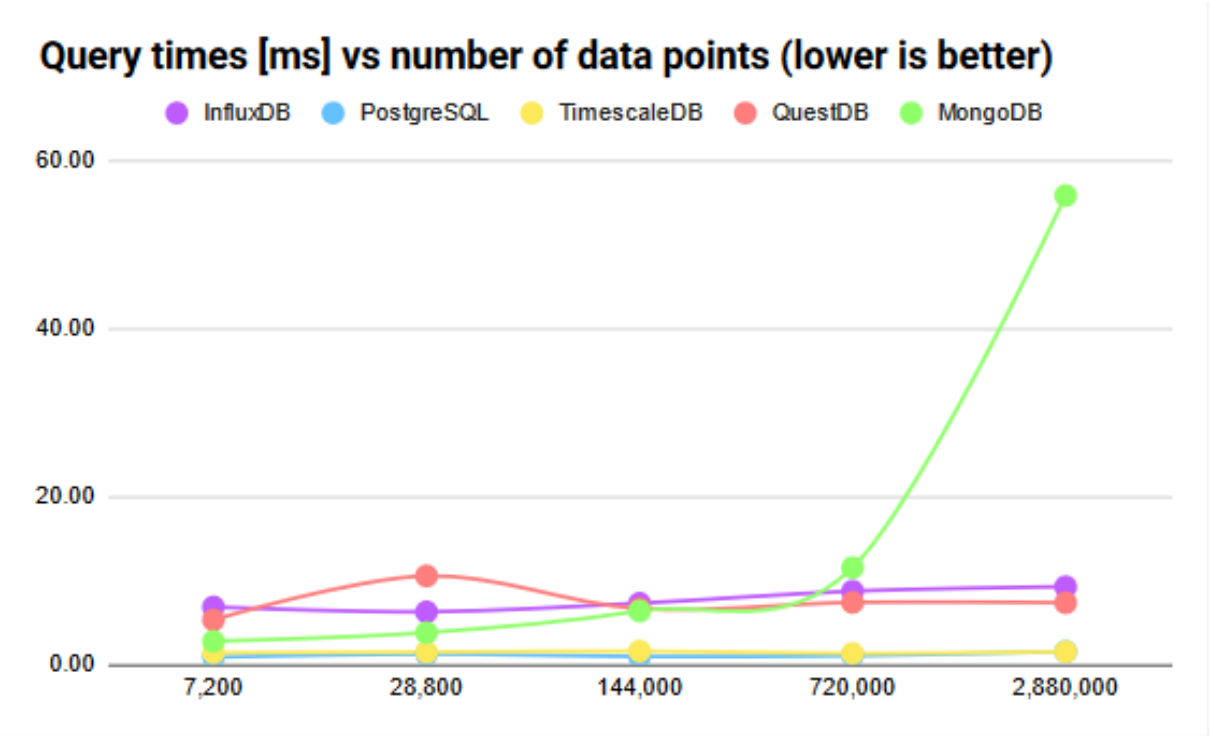

Figure 8: Visualization of run times in seconds for querying last day's data for each database

You can see that the situation is suddenly the complete opposite - SQL databases are suddenly dominating, while some are struggling. Intersting case is MongoDB, which performs well for low amount of data, but then becomes very slow for high amount of data.

4.5 Aggregation

Finally, I also ran an aggregation benchmark. It takes a particular station and finds the averages of all the variables of all time.

Avg [ms]	7,200	28,800	144,000	720,000	2,880,000
InfluxDB	4.30	4.92	6.96	12.20	19.58
PostgreSQL	1.05	1.29	2.13	3.42	6.06
TimescaleDB	1.32	1.24	1.19	1.12	1.28
QuestDB	6.98	9.83	7.51	6.13	5.93
MongoDB	2.31	4.10	8.73	16.60	50.45

Figure 9: Table of run times in seconds for inserting `N` data-points for each database

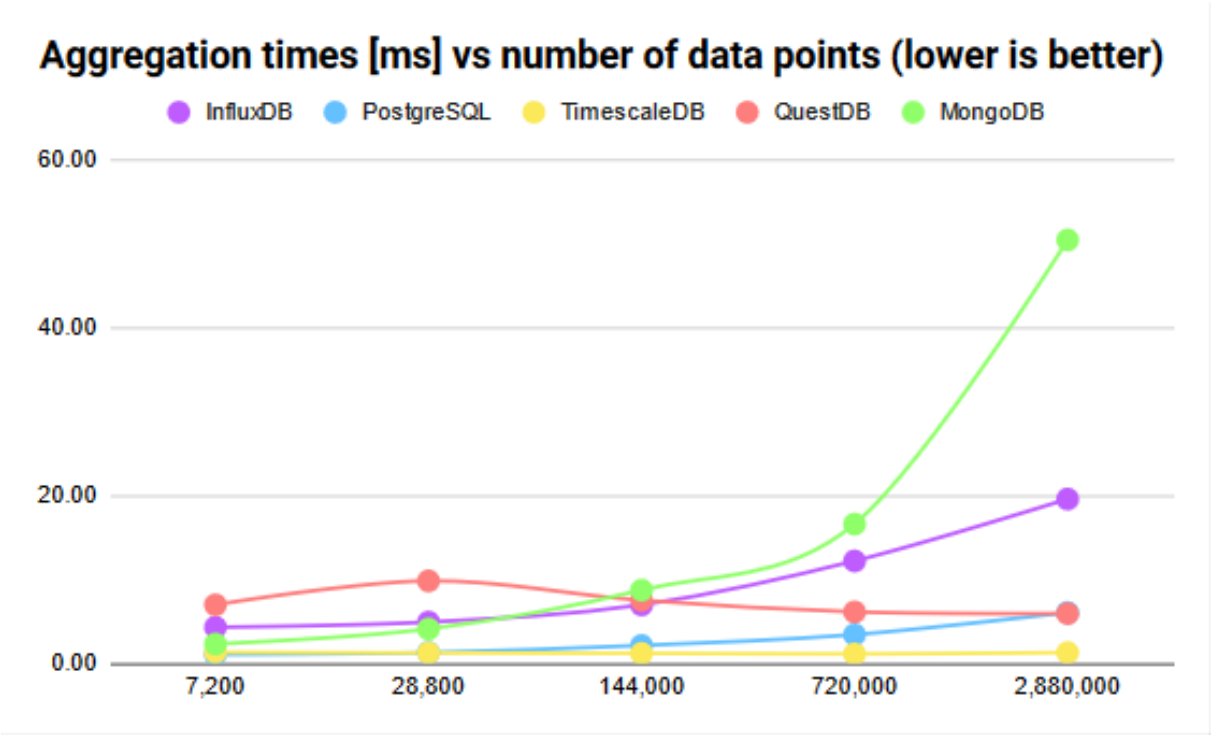

Figure 10: Visualization of run times in seconds for calculating averages for each database

We can see a similar trend - SQL databases are dominating, but with enough data PostgreSQL is not able to keep up with TimescaleDB, which scales really well. MongoDB is struggling again with high amounts of data, even if it started off strong.

4.6 Comments

I believe the results shown here are a good starting point, though they may not be 100% accurate. I tried optimizing the data storage for all 3 categories, but I am not an expert in this field. These results represent what you can expect if you dedicate *some*, non-zero, amount of effort using the recommended libraries in Go.

The effort put into optimizing the performance and building the queries for these databases was not divided equally.

I had to spend a lot of time setting up both of the SQL databases (creating a database, creating a table, etc.), while other databases take care of this for you (namely InfluxDB and QuestDB).

Queries in SQL (for PostgreSQL, TimescaleDB, QuestDB) could generally be reused, but there were some nuances and differences, that required changes.

InfluxDB required setup from the browser (initial setup for username, password, creating an organization and a bucket) to get the token for connecting to the database. There are some solutions for doing this automatically, but they would take some time to implement.

Otherwise, overall implementation was by far fastest for InfluxDB, no real pain points and everything worked out of the box.

The most significant challenge was with MongoDB, for which I had to analyze and fix some performance issues. At first the ingestion time for 720,000 data points was ~7 minutes, now it is ~4 seconds, but querying and aggregation times are still not perfect. The final solution for this benchmark is *not using a time series collection* as that was slowing it down by a factor of 100 (while not improving any

other category). I am sure this was some kind of misconfiguration on my part, but by simply making it a normal collection it fixed the performance issues (even if I created the same index the time series collection creates automatically; the performance boost was drastic).

I could have spent more time optimizing each solution, but that was not primary goal of this benchmarking exploration. The point was to see, how these databases compare with minimal setup out of the box using the recommended approches.

5 Conclusion

The realm of time series databases and related technologies is fascinating. There are a lot of things to consider and it is exciting to explore these options. Deciding between these solutions is not an easy task and it definitely comes down to specific needs and requirements.

InfluxDB stands out as the most well rounded solution for time series data. It uses Line Protocol for high ingestion, Flux for intuitive querying and InfluxQL for SQL-like experience. It is part of a robust ecosystem, including the TICK stack, that allow it to be even more powerful. It worked great out of the box offering good performance without much effort, while also having a good documentation and resources online to debug potential issues.

TimescaleDB, an extension of PostgreSQL, offers enhanced capabilities over PostgreSQL and further enhancing its performance. It is an incredibely effective solution, if you are already working with PostgreSQL and want to make even better for the time series use-case. It uses SQL, therefore there will never be a lack of resources to debug issues or learn more; moreover TimescaleDB also offers great documentation. The overall performance for querying and aggregation was outstanding.

QuestDB excelled in the parts it promised on - namely ingestion speed - it went unmatched in this category. On the other hand, it clearly showed it is less mature both when looking up some resources and also in features and other limitations.

PostgreSQL proved its versatility and robustness. It is clearly a great solution even in the space of time series data. I see little point in using pure PostgreSQL when there are solutions such as TimescaleDB that enhance the experience with virtually no down-sides.

MongoDB showed some promising results, despite some initial performance challenges. Its flexibility and widespread adoption make it a viable in the space of time series data as well, especially if you are already using MongoDB. It has great documentation and resources, however, I still had trouble optimizing the performance. I believe the benchmark results showed that with enough care, MongoDB can definitely compete even with the biggest players in this space.

The benchmark results indicate an inescapable trade-off between ingestion and query speed. Databases that use Line Protocol for ingestion excelled in that category, SQL databases, on the other hand, excelled in query speeds, while MongoDB was flowing in the middle.

In conclusion, the choice of a time series database should be driven by the specific demands of the application. All of these solutions focus on a different use-case and offer different features. Some solutions might be more convenient, some more exciting due to an alternative approach. This analysis can be taken as a starting point for such decision-making process.

Bibliography

[1] Paul Dix, “Why Time Series Matter For Metrics, Real-Time Analytics And Sensor Data”. Dec. 2023. Accessed: Dec. 20, 2023. [Online]. Available: https://www.influxdata.com/time-series-database/#download

[2] Ajay Kulkarni, Ryan Booz, and Attila Toth, “What Is Time-Series Data? Definitions & Examples”. Accessed: Dec. 20, 2023. [Online]. Available: https://www.timescale.com/blog/time-series-data/

[3] MongoDB, “Time Series”. Dec. 2023. Accessed: Dec. 20, 2023. [Online]. Available: https://www.mongodb.com/docs/manual/core/timeseries-collections/

[4] DB-Engines, “DB-Engines Ranking - Trend of Time Series DBMS Popularity”. Accessed: Dec. 20, 2023. [Online]. Available: https://db-engines.com/en/ranking_trend/time+series+dbms

[5] InfluxData, “Get started with InfluxDB v2”. Dec. 2023. Accessed: Dec. 20, 2023. [Online]. Available: https://docs.influxdata.com/influxdb/v2/

[6] InfluxData, “Get started with InfluxDB”. Dec. 2023. Accessed: Dec. 20, 2023. [Online]. Available: https://docs.influxdata.com/

[7] InfluxData, “Telegraf documentation”. Dec. 2023. Accessed: Dec. 20, 2023. [Online]. Available: https://docs.influxdata.com/telegraf/v1/

[8] InfluxData, “Chronograf documentation”. Dec. 2023. Accessed: Dec. 20, 2023. [Online]. Available: https://docs.influxdata.com/chronograf/v1/

[9] InfluxData, “Kapacitor documentation”. Dec. 2023. Accessed: Dec. 20, 2023. [Online]. Available: https://docs.influxdata.com/kapacitor/v1/

[10] TimescaleDB, “Timescale Documentation”. Dec. 2023. Accessed: Dec. 20, 2023. [Online]. Available: https://docs.timescale.com/

[11] QuestDB, “QuestDB Documentation - Introduction”. Dec. 2023. Accessed: Dec. 20, 2023. [Online]. Available: https://questdb.io/docs/

[12] Yitaek Hwang, “Comparing InfluxDB, TimescaleDB, and QuestDB Time-Series Databases”. Accessed: Dec. 20, 2023. [Online]. Available: https://questdb.io/blog/comparing-influxdb-timescaledb-questdb-time-series-databases/

[13] QuestDB, “What Is High Cardinality?”. Dec. 2023. Accessed: Dec. 20, 2023. [Online]. Available: https://questdb.io/glossary/high-cardinality/

[14] PostgreSQL, “PostgreSQL 16.1 Documentation”. Dec. 2023. Accessed: Dec. 20, 2023. [Online]. Available: https://www.postgresql.org/docs/current/index.html