AMERICAN UNIVERSITY OF BEIRUT

Evaluation of Distributed Time Series Databases for KATRIN Intelligent Control System

by
Sara Wehbe

Advisor(s)
Dr. Suren Chilingaryan (KIT)
Jalal Mostapha (KIT)
Wissam Sammouri (AUB)

A Capstone Project
submitted in partial fulfillment of the requirements
for the degree of Master's in Business Analytics
to the Suliman S. Olayan School of Business
at the American University of Beirut

Beirut, Lebanon
August 2021

An Abstract of the Capstone Project of

Sara Wehbe for Master's in Business Analytics (MSBA)

Title: Evaluation of Distributed Time Series Databases for KATRIN Intelligent Control System

The slow control system of Karlsruhe TRItium Neutrino (KATRIN) experiment at the Campus North of the Karlsruhe Institute for Technology includes hundred thousand sensors monitoring all the critical operation parameters of the ongoing experiment. Sampled with high rates, the readings from these sensors are used as secondary data source in the physical analysis of detector data and are crucial for understanding the experimental results and detecting early problems in the setup.

The purpose of this project is to find a state-of-the-art backend database system that can handle the big amount of data generated by the SCS while providing real-time monitoring of recent data and analytical capabilities for historical data analysis.

In our analysis, we use KATRIN as a use case representative of a slow control system for large scientific experiment to understand the usage patterns of the system and the functional and performance requirements of the back-end database.

Currently KATRIN uses a relational back-end database to store around 12 TB of compacted raw monitoring data, this database presents several limitations in terms of performance, scalability, and ease of maintenance.

Since the initial setup of KATRIN experiment, several promising database technologies for high ingestion and fast access to time-series data have emerged, and Time Series Database (TSDB) became the fastest-growing type of database in 2021 according to db-engine classification[1].

In our analysis, we survey several commercial-off-the-shelf TSDB technologies by technical features and performance to understand how they respond to the most pressing needs of a slow control system in terms of high ingestion rate and fast query execution. We propose an extensible novel benchmarking framework to evaluate their data ingestion rates and query latencies per one node, to propose a solution that best fits the performance requirements of a large slow control system. We conclude that ClickHouse with its ~600K records per second ingestion rate best-fits a large slow control system.

Table of Contents

1. Introduction

The outstanding advance in most of the scientific fields that we are witnessing today, like cosmology, medicine and physics, is the result of decades of research and work on large scientific experiments with teams of researchers from around the world.

A slow control system (SCS) plays an important role in monitoring the critical properties of a large scientific experiment to ensure proper operation and provide necessary data for analysis and processing. Generally, a SCS uses a set of sensors to constantly collect data about the experiment subsystems and send it to a centralized data repository for further analysis.

The KArlsruhe TRItium Neutrino (KATRIN) performed at the Campus North of the Karlsruhe Institute for Technology, is an example of large-scale scientific experiment aimed at determining the *absolute mass scale of neutrinos.* Neutrinos are the lightest particles in the Universe. Their tiny mass is a clear indication for physics beyond the standard model of elementary particle physics. On the largest scales, neutrinos act as "cosmic architects" and take part in shaping the visible structures in the Universe, as they influence the formation and the distribution of galaxies[2].

The slow control system of KATRIN consists of around 50 subsystems. Each subsystem contains up to 100 device groups with multiple sensors each[3]. The total number of sensors is expected to grow up to around 100K in total. Groups sample data at the same time at an average frequency of 1Hz with few groups sampling at 10 Hz and very few at 100 Hz, making it required to store timestamps with high precision. The infrastructure is maintained by a small team, and it is crucial to minimize the maintenance and operations time.

Sensors produce huge amounts of operational data that are used for monitoring the system, analyzing the experiment results, and detecting any anomaly in the setup.

Currently MySQL database is used to archive and retrieve operational data for the applications and analytical tools requesting it. The database is expected to ingest around 100K of metrics per second. MySQL is an open-source relational database used by a wide range of organizations around the world[4]. It is a general-purpose database designed for managing relational data and it is not optimized for time-series data. Therefore, it presents some limitations in terms of:

1. High availability for write operations as the current Master/Slave replication provides high availability for Read operations only. Although there is a Master-Master replication for MySQL developed by Percona, however this was not used in production as it was found to be very sensitive to cluster reboots.
2. Horizontal scalability for reads and writes to split the load among multiple nodes and ensure high ingestion rate and fast query execution.
3. Real-time ingestion with high rate : To handle the high ingestion rate of around 100K metrics per second, currently multiple Microsoft SQL Server (MSSQL) databases are used to collect data. A dedicated database is assigned to each subsystem. Then data is collected from these MSSQL databases offline using extensive batching to MySQL

database. This adds around 1 to 2 minutes latency between data generation and its availability in the system.

4. Custom development is required to pre-generate aggregated data. Pre-aggregation is needed to speed up the workloads that need to process large amounts of summarized data repeatedly.
5. High compression is needed to reduce the storage footprint and improve query efficiency. Although MySQL provides data compression, however, it is not optimal as with most row-oriented data models with many data types and ranges per row.

Hence the need for a database that overcomes these shortcomings. This novel database should be optimized for ingesting the huge amount of time-series data in real-time at a minimum rate of 100K data points per second with the ability to scale out to handle any increase in the amount of data generated in the future.

Given that the measured parameters of slow control systems are for physical analysis, it is critical to avoid any data loss during ingestion or later due to database or hardware failures. The novel database should be able to process large amounts of raw and aggregated data fast enough for the applications and tools that request it.

A lot of heavy queries are currently used to analyze sensors data e.g. retrieve time intervals within a year when the readings of a specific sensor are outside a certain threshold range. This can't be generated fast enough if all the raw data must be processed on the fly. Currently, a lot of manual work is done to create materialized views to store aggregated data and cron jobs are created to pull new data periodically and update the views. It is important that the novel database provides a way to configure continuous aggregations in an easy way to speed up and facilitate the creation and execution of such queries.

We started our project by conducting a usage pattern analysis on one month of KATRIN logs data to understand how KATRIN data is being used. We found a very interesting insight: 96% of the requests are real-time and they target the last values. Our recommendation was to offload these requests from the database and build a real-time interface to provide latest data to the applications that need it. The remainder of the requests were aligned with time-series data usage patterns in terms of the need to query recent data frequently along with the need to analyze data over long time periods.

Based on this, we decided to check some of the TSDBs that are available in the market and perform a feature analysis to determine the databases that have the critical requirements. Three databases were selected: InfluxDB, TimescaleDB and ClickHouse.

We then conducted a performance benchmark on the selected databases to compare the performance of data ingestion and query execution based on the usage patterns that we concluded in KATRIN usage analysis.

There are a few benchmarking tools in the market designed for TSDB (refer to Table 9), however we created our own tool to have the flexibility to customize the benchmarking test cases based on KATRIN data usage. We then presented and compared the benchmarking results of the selected databases based on synthetic data.

2. Background and Related Work

The evolution of time-series applications in the big data field like Internet of Things (IoT), operational monitoring, forecasting and financial trends led to a substantial rise of specialized databases for managing time-series data.
Many research studies have been carried out about TSDB technologies, most of them focus on a specific area of TSDB – some focus on finding the best TSDB system that suits a specific business need, others focus on studying the architecture and technologies behind existing TSDBs and some research focus on finding a standard approach to evaluate the performance of TSDB systems.

A research done by Matei-Eugen Vasile, Giuseppe Avolio and Igor Soloviev [5] is motivated by finding the best TSDB for managing and archiving the operational data generated by the Slow control system of the ATLAS experiment at the Large Hadron Collider at CERN. The research compares the ingestion performance of two TSDBs ClickHouse and InfluxDB with a reference write speed of 100k metrics per second. The testing involves two different data models with single or multiple tables. They concluded that adopting single table approach has better performance and that in most cases, ClickHouse has better ingestion performance than InfluxDB.

A research done by Alexey Struckova, Semen Yufaa, Alexander A. Visheratina and Denis Nasonov [6] performs a thorough theoretical and experimental investigation of modern databases that are used for storing time-series data. They perform a feature comparison on four TSDBs – ClickHouse, TimescaleDB, InfluxDB and OpenTSDB. They perform an experimental benchmarking on the selected databases based on three real use cases – DevOps, Internet of Things, and meteorological analysis. The results of this research allow formulating a set of recommendations for selecting a database depending on planned data volumes and request rate.

A research done by Rui Liu and Jun Yuan[7] is motivated by standardizing the benchmarking of time series databases to help businesses choose the most appropriate database that suits their business needs. They present the IoTDB-Benchmark framework, specifically designed for TSDB and IoT application scenarios. They benchmark predefined ingestion and basic query types with system resource consumption monitoring. They use this benchmark framework to compare four TSDB systems: InfluxDB, OpenTSDB, KairosDB and TimescaleDB.
Other TSDB benchmarking tools are available, refer to this List **of time-series databases** *for details about each tool.*

3. Modern database technologies - Theoretical background

Database technologies evolved over time to respond to the emerging business needs and technology advances. In this section, we go through the main concepts and usage of three main database types:

Relational database: is the traditional and widely used type of database. It organizes data in tables with predefined columns. The tables can be linked based on common data; this helps to maintain the integrity of the database. The main advantage of relational database is that it provides consistency of data by applying normalization and ensuring ACID properties. It also allows to retrieve meaningful insights about the data through its easy-to-use SQL scripting language.

Relational databases are well suited for OLTP (Online Transaction Processing) applications that support the ability to insert, update or delete relational data e.g. retails and financial transaction systems.

Oracle and MSSQL (Microsoft SQL server) are among the most popular commercial relational databases. MySQL and PostgreSQL are popular open-source relational databases.

NoSQL database: The evolution of social media and customer-centric applications resulted in huge amount of data being collected, edited, accessed, and processed concurrently. Therefore, scalability, performance and availability became a big challenge for relational databases that are hard to scale out due to their need to keep data consistent. These challenges resulted in the rise of NoSQL databases. There are different types of NoSQL databases that are built for different purposes, such as column-oriented databases like Cassandra, document databases like MongoDB etc.
Generally, NoSQL databases provide a flexible data model and are designed to scale horizontally across many commodity servers to accommodate large data volumes and application loads that exceed the capacity of a single server. NoSQL provides horizontal scalability by weakening data relationships and data consistency [8]
According to the CAP theorem that was conjectured by Eric Brewer in 2000, “It is impossible for a distributed system to provide all three of the following guarantees in case of any type of failure”:

- Consistency: All nodes see the same data at the same time
- Availability: The system is always on
- Partition Tolerance: The system continues to function even if the communication among the servers is unreliable

Knowing that distributed systems are partition tolerant by default, NoSQL databases generally do a trade-off between availability and consistency in case of a network failure. We can define two main groups of NoSQL databases:

- **AP** systems sacrifice consistency for availability, but are not inconsistent (eventually consistent [1]) e.g. Cassandra and RIAK
- **CP** systems sacrifice availability for consistency, but are not unavailable like MongoDB, HBASE

Time-series database:
A specialized database model to manage time-series data that is increasingly growing in popularity. **Time-series data** is a succession of data points that are continuously collected providing the ability to track data changes and patterns over time, example: sensor measurement, CPU usage, stock price etc. Time series data do not involve regular update and delete operations, however, data that arrive is added as a new data entry and it usually arrives in chronological order.
The continuous accumulation of data at granular time resolutions results in huge amount of data which brings a set of challenges to store this data and query it in a fast way. This was the motivation behind creating a new type of database that takes advantage of the specific characteristics of time-series data to solve these challenges. To achieve this, most TSDBs split data into separate partitions with a predefined time duration. This allows to create data indexes on the partition level and avoid having a huge index for the entire data. Knowing that the partitions are arranged by time, this allows the database to quickly locate the partition(s) to which an operation belongs and reduce the operation execution time.
Some TSDBs built their solution on top of existing relational databases like TimescaleDB that is built as an extension to PostgreSQL, or on top of NoSQL databases like KairosDB and Scylla that are built on top of Cassandra. Other TSDBs are built from scratch like Prometheus, Druid and InfluxDB.

3.1 Data Models

Traditionally databases are either row or column oriented [2]. Row oriented databases store all the values in a row together on the disk, whereas column-oriented databases store all the values in a column together on the disk.
As a result, queries that request many columns and few rows are faster on a row-oriented databases while queries that request few columns and many rows, including aggregation queries, are faster on column-oriented databases. Therefore row-oriented databases are best used with OLTP (Online Transaction Processing) workloads and column-oriented databases are best used with OLAP (online analytical processing) workloads.

Additionally, column-oriented data is more suitable for compression as all values of a column have the same data type. Inserts are slower in column-oriented databases as the system needs

[1] System will become consistent over time

[2] Some NoSQL databases have different data representations like key-value pair, document oriented (used to store json, blobs etc.) and graph (used to store elements that share connections)

to split each record into different columns and then store them separately on the disk.

Relational databases like Oracle are row-oriented, on the other hand, column-oriented databases like Cassandra and HBase are NoSQL and data warehouses like Redshift are usually column-oriented .

Most time-series databases are column oriented to take advantage of the optimized data compression and fast data aggregations. Other TSDBs like TimescaleDB combine both columnar and row data representations to achieve better performance. Being an extension of PostgreSQL, TimescaleDB natively inherits its row-oriented data representation, and then is uses data compression by keeping recent data as standard uncompressed database rows, which allows them to be written into the database at high rates. Then, after some amount of time, the set of rows are compressed into TimescaleDB's columnar format according to a compression policy [9].

3.2 Indexing structures

The B-tree and the Log-Structured Merge-tree (LSM-tree) are the two most widely used data structures to speed up access to data.

In most relational databases, a table is saved on the disk in fixed-size pages on top of which the system creates a data structure like B-tree to index the data. each B-tree node acts as an array with references to a page range. The leaf of the tree points to a specific page.
With this structure, queries can quickly find data without the need to scan the entire table.
When data is large that the B-tree index pages can't fit in memory, then updating a part of the tree can involve several disk I/O to load the needed data pages in memory, update them and write them back to disk, which affects the performance of the database [10].

Most NoSQL databases address the disk I/O performance issue by using Log structured merge tree (LSM). LSM do not perform in place writes, however it queues several data changes in memory in a sorted table called memtable, then it flushes them as a single batch to disk and stores them in a sorted table called SSTable (Sorted String Tables). This solution is suitable for fast writing; however, it is not beneficial for fast reading as it involves additional readings from memtable and sstables to find recent data.

4. Methodology

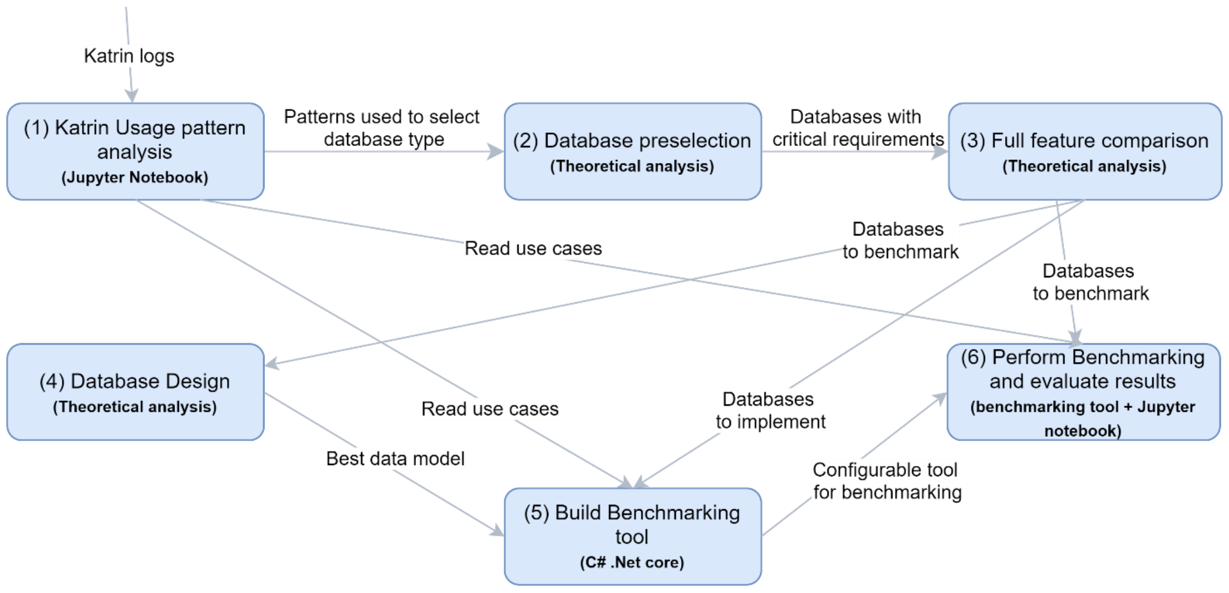
Katrin logs
(1) Katrin Usage pattern analysis (Jupyter Notebook)
Patterns used to select database type
(2) Database preselection (Theoretical analysis)
Databases with critical requirements
(3) Full feature comparison (Theoretical analysis)
Read use cases
Databases to benchmark
Databases to benchmark
(4) Database Design (Theoretical analysis)
(6) Perform Benchmarking and evaluate results (benchmarking tool + Jupyter notebook)
Read use cases
Databases to implement
Best data model
Configurable tool for benchmarking
(5) Build Benchmarking tool (C# .Net core)

Figure 1 - Evaluation methodology flowchart

4.1. KATRIN Usage Pattern Analysis

The first phase in this project is to understand the functional requirements of an SCS backend database in terms of usage patterns. For this we performed an analysis on KATRIN query logs.
Katrin data is used by multiple independent teams of researchers to analyze the experiment results, detect patterns and correlations, and monitor the overall setup.
The interactions with the database to pull, aggregate and filter data is done primarily through web API. Calls to the web API are logged in json format along with the query type, timestamp, query parameters and server response message.
To understand how KATRIN data is queried, we conducted an analysis on one month of web API query logs. The log files are generated daily with an average size of 10 GB per file. Therefore, our analysis is based on around 30 log files with around 300 GB of total size. The purpose of this analysis is to understand how KATRIN data is currently queried and to answer the following main questions:

- The proportion of queries targeting recent data vs older data
- The proportion of queries targeting real-time data
- If data is mostly retrieved over short or long timespans
- If queries tend to use limit on the number of requested records
- The number of columns returned per query
- The proportion of queries aimed at a single group (table)

4.1.1. Data

Each log file consists of nested json strings. Below is a list of the main json tags that we used in our analysis:

Parameter	Description
Timestamp	Timestamp when the message was logged
Source	Web API microservice being called. Our analysis is based on three main APIs: **SERVICE(getdata)**: Return data in the requested format synchronously **SERVICE(update):** Perform internal data processing ad generate plots **SERVICE(download):** Return data in the requested format asynchronously
SessionID	ID of the client session
Request	Unique identifier of a specific HTTP request
Latency	Execution time of a request in microsecond
db_server	Data source representing the subsystem
db_name	Database name
db_group	Table name representing the device group
db_mask	comma separated list of columns. If null, all the columns within the device group are retrieved
Srctree	Allows to query data from multiple groups which is impossible with original API parameters: db_server/db_name. It consists of a concatenated value containing the server(s), database(s) and columns being queried.
Width	Used with **SERVICE(update)** requests to specify the width of the plot
Height	Used with **SERVICE(update)** requests to specify the height of the plot
Resample	Desired sampling rate
Window	Time span in the following format: **0**: return all data **-1**: return last value **<from>-<to>**: specifies the window start and the end (as unix timestamps) **<number>**: specifies window width in seconds, the window is positioned in the end of experiment. In case there is a limit of the number of returned records, this is indicated by appending **,<limit>** at the end of the **window** value
RT (real-time)	Flag indicating that the request is coming from the status monitor. These kinds of requests should be excluded from the analysis as they will bypass the database in the future.

Table 1 – Main tags of KATRIN logs json files

4.1.2. Techniques and Approaches

4.1.2.1. Data loading and preparation

The purpose of this phase is to load the json log files, filter them based on the type of request, extract the needed json tags and generate a consolidated csv file with all the needed records and columns. We used Jupyter Notebook with Pandas library to perform our data analysis. However, given the big size of the log files, it was not possible to load and process them in memory especially that we are using a single node to perform the data analysis.

As a workaround, we used the logic of chunking by dividing each log file into smaller parts, we then performed the data filtering and tags loading on each chunk individually. At the end, we consolidated the extracted data in one csv file that is ready for data processing and analysis.

4.1.1.1. Data processing

In this phase we used the consolidated CSV dataset generated from the data preparation phase. The purpose of this phase is to clean the dataset, remove duplicate records and compute new columns that are needed to perform the analysis. Below are some of the computed columns:

Tag	Description
Time Span categorical column	Computed by generating the time span in seconds based on the window column and then transforming it into a categorical column with values like: Last value, <= 5mn, 1 – 2 days, 1 – 3 months, 1 week, > 2 years etc.
Request Recency categorical column	Computed based on the **window** and **Timespan** columns and then transformed into a categorical column with values like 1 hour, 1 day, 1 week, etc.
Limit column that indicates the limit on the number of returned records	Generated from the **window** column
Columns Number: the number of columns requested in a query	Retrieved from **db_mask** in case the query is aimed at one group, otherwise, it is calculated from **srctree** column
Groups Number: the number of groups requested in a query	**1** in case value is different than 'srctree', otherwise, data is retrieved from different groups, the number of groups is equal to the count of distinct group names in the **srctree** column value
Status: status of the request	Generated based on the value of the **Message** column. Status of a request can be finished, SQLError, TimeOutError etc.

Table 2 - Main computed columns used in KATRIN usage pattern analysis

4.1.2. Results

Once the dataset is ready, we performed the analysis on the data from different perspectives to understand how the queries are used.

4.1.2.1. Data limit

We investigated the proportion of real-time calls that retrieve the most recent values. We found that they constitute **96%** of the queries. We then excluded the real-time queries from the remainder of our analysis.
We analyzed the proportion of queries that specify a limit on the number of returned records, and we found that they constitute only **0.000234%** of the requests, therefore it is not a common pattern to have a limit of the number of returned records.

4.1.2.2. Requests Recency and Time Span Analysis

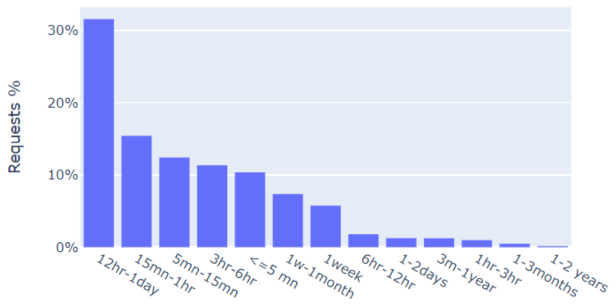

Figure 2 - Katrin requests duration

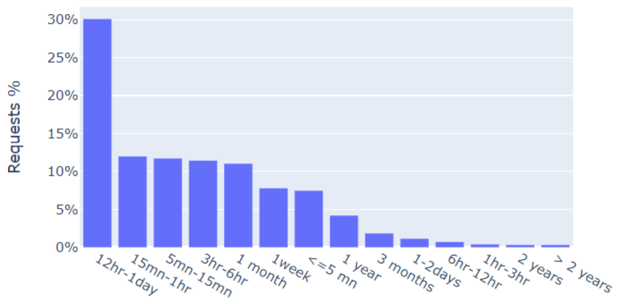

Figure 3 - Katrin requests recency

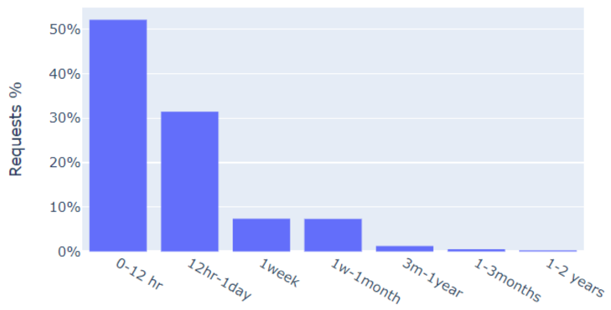

Figure 4 - Katrin requests duration - collapsed

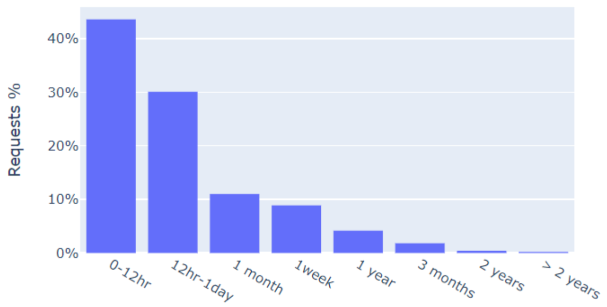

Figure 5 - Katrin requests recency - collapsed

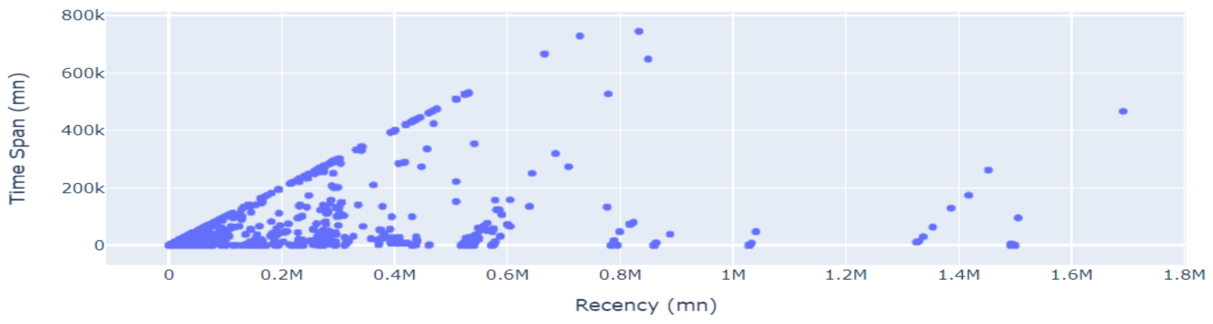

Figure 6 - Requests recency vs duration

The purpose of this analysis is to understand if users are interested in recent or historical data, and if they tend to work on data over short or long-time spans.

Note that data recency is different than time span, for example a query can request 2 days of data (time span) from 1 year (data recency).

We notice that queries are targeting recent and historical data, however most of the queries (74%) target recent data that is less than 1 day old.

Some of the queries aim to retrieve data over long-time spans like retrieving 1 year of data, 3 months of data etc. however 84% of the queries request data over short time spans (1 day).

We notice also that queries targeting old data tend to have long time spans.

4.1.2.3. Requests by number of subsystems, groups and sensors

The purpose of this analysis is to check which subsystems are mostly used and to understand if queries are more likely to target one or multiple sensor/group/subsystem.

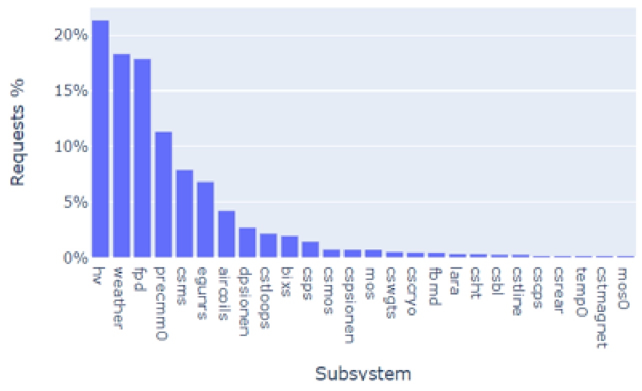

Figure 7 - Requests per subsystem

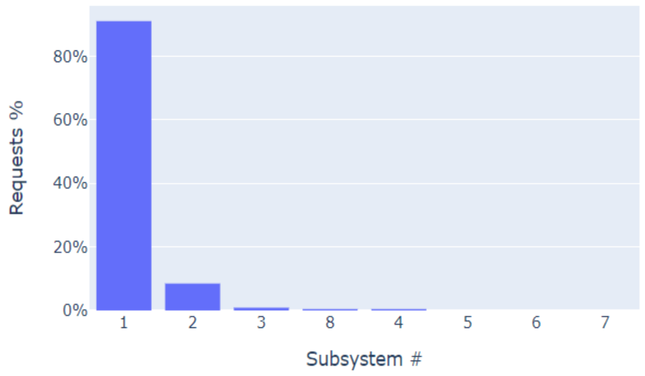

Figure 8 - Requests by number of subsystems

We notice that the queries are not evenly distributed between subsystems; some subsystems like hv, weather and fpd receive most of the load while other subsystems are not heavily loaded.

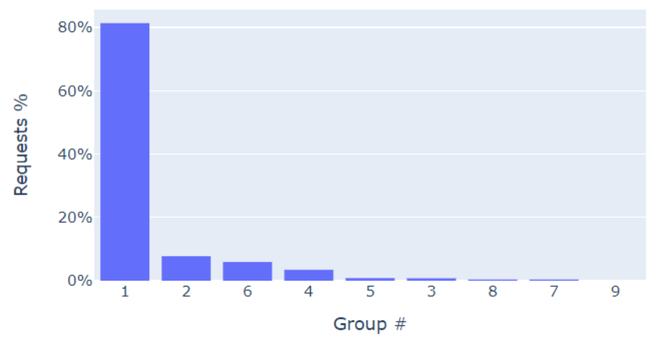

Figure 9 - Requests by number of groups

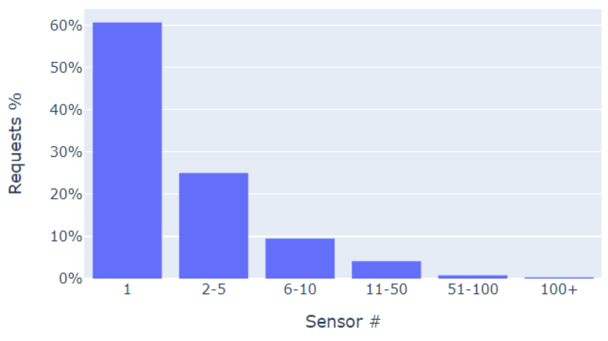

Figure 10 - Requests by number of sensors

We notice that most of the queries retrieve data from a single subsystem (90%), a single group (80%) and a single sensor (60%).

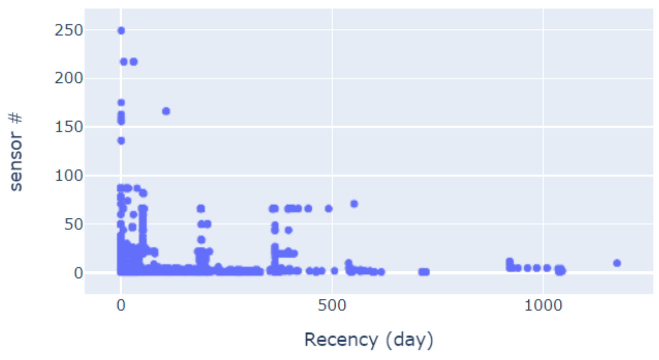

Figure 11 - Requests recency vs number of sensors

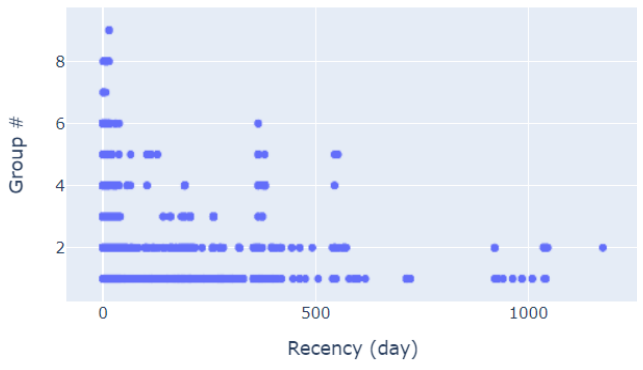

Figure 12 - Requests recency vs number of groups

We also notice that requests targeting very large number of sensors have relatively low recency, thus, the number of queried sensors decreases for historical data.

4.1.2.4. Latency Analysis

The purpose of this analysis is to find any correlation or pattern related to high query execution latency.

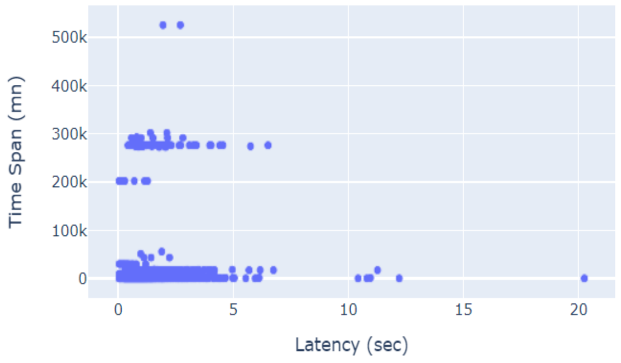

Figure 13 - Requests latency vs Duration

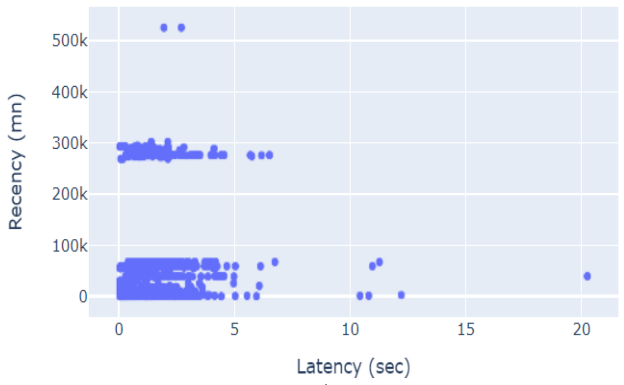

Figure 14 - Requests latency vs recency

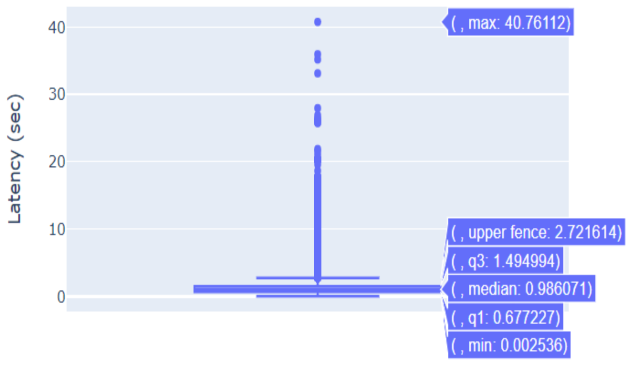

Figure 15 - Requests latency box plot

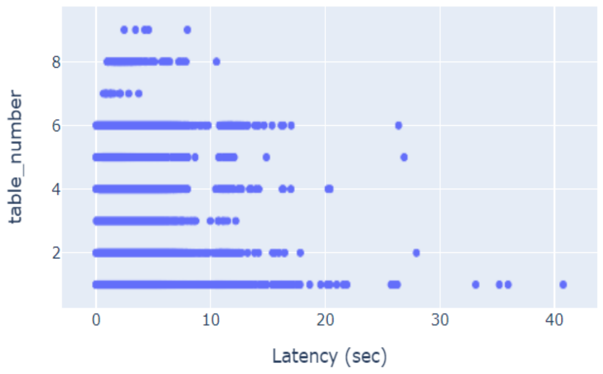

Figure 16 - Requests latency vs number of groups

-

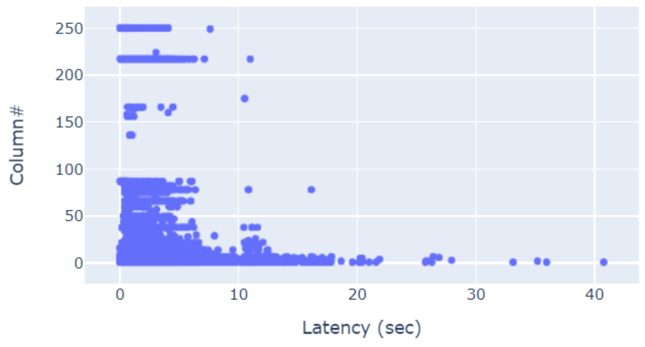

Figure 17 - Requests latency vs number of sensors

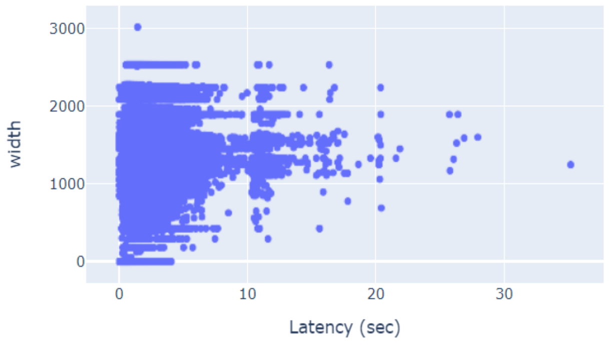

Figure 18 - Requests latency vs plot width

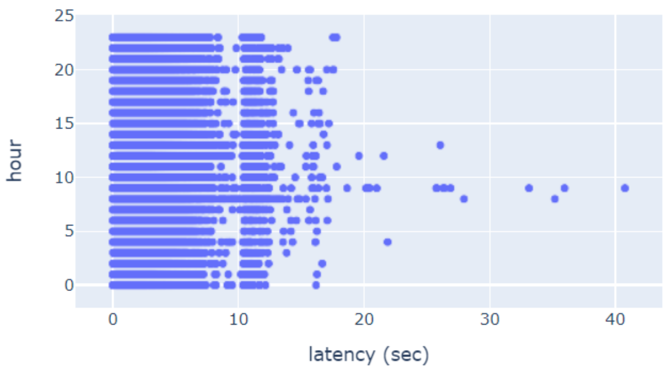

Figure 19 - Requests latency per hour of the day

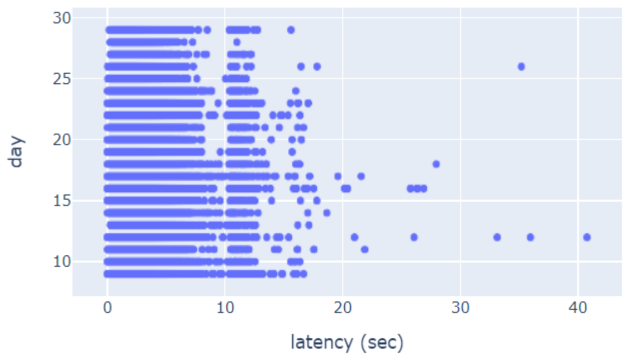

Figure 20 - Requests latency per day

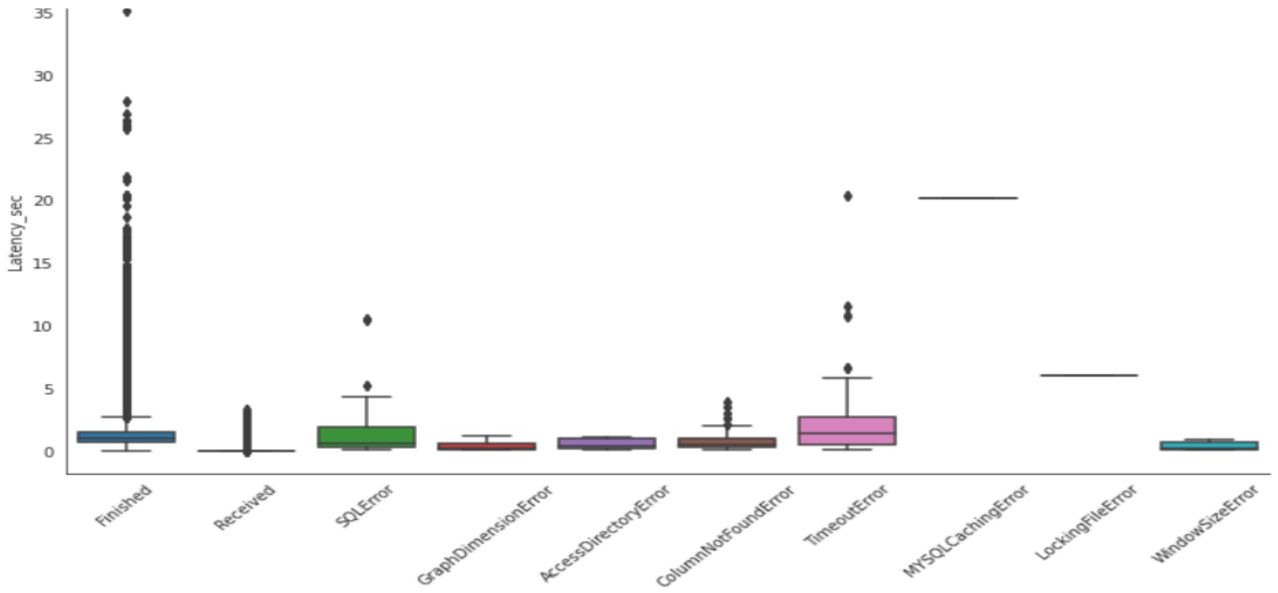

Figure 21 - Requests latency by query status

During our analysis, we found a big number of outliers with high latency. We tried to check if there is any relation between high latency and queries targeting historical data or data over long durations, or queries targeting multiple groups or sensors, or queries that plot data on charts with large width. However, we did not find any clear correlation with any of these parameters.

We also checked the relation between high latency and the day and hour of occurrence to see if high latency is happening in a specific day or at a specific period of the day. Figure 19 and Figure 20 show that few data points with very high latency happened at the same day and same hour, this might be related to an external factor like server overload, network issues… however no clear correlation was detected between latency and day or hour of occurrence.

We then tried to classify the requests based on the server response message to split between queries that completed, and queries that returned error messages, we also differentiated between the different error message like Time out error, SQL error etc. We noticed that most of the queries with high latency are queries that completed without error, and thus no definitive correlation was detected between latency and query status.

After discussing the results with the IT team, it turned out that the connection between the web API and the database is going over 1 Gbit of public network with uneven loads and capacity issues that happened in the past. Consequently, as this is irrelevant to database design, we decided to stop the investigation.

4.1.2.5. *Internal processing vs data access requests*

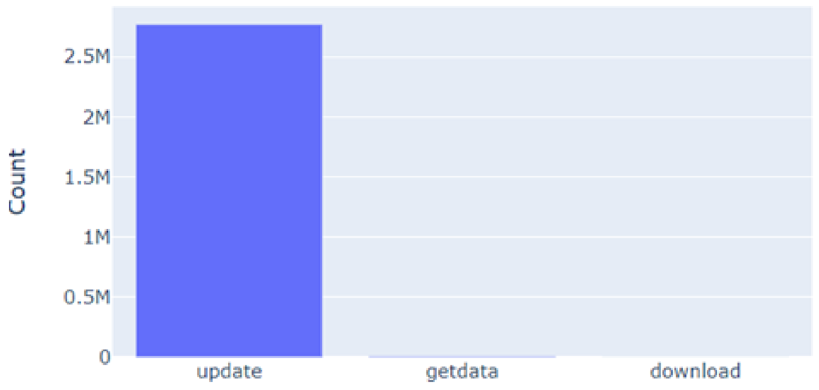

Figure 22 - Requests per query type

We notice that most of the queries are of type 'update' that consist of internal processing and plotting of data as opposed to requests of type 'getdata' and 'download' that return data in a specific format without internal processing.

4.1.3. *Discussion*

Based on KATRIN usage analysis, we conclude the following:

- Current load is aimed largely on the latest value of sensors; hence, a significant reduction of load can be achieved by using modern Control System to serve these requests without the need to pass through the database.
- It is common to request data over large time spans, this kind of requests use aggregated data over predefined intervals like second, minute, day etc. hence the need for a database that allows to easily generate and maintain pre-aggregated data with different aggregation intervals
- Most of the queries target recently collected data. To speed up the queries targeting recent data, in-memory[3] databases like Redis can be used. As an alternative, and to

[3] A purpose-built database that relies primarily on memory for data storage.

keep the system architecture simple, a TSDB that provides a good caching mechanism can be an alternative solution.

- Most of the queries target individual subsystems, groups, and sensors. The target database should provide a mechanism to index data by sensor and time.
- Most of the queries require internal processing and data plots, therefore, the target database should provide statistical capabilities and integration with common visualization tools and platforms.

4.2. Preselection

Given that the data collected from sensors is time-series data characterized by its high volume and chronological order and given the usage pattern of KATRIN data that mainly consists of (1) summarizing data over long time spans and (2) providing access to recent data, in addition to the high performance requirement for ingestion and query requests, we conclude that managing SCS data is a typical time-series management use case and therefore requires a database that is specialized for managing time-series data.

The next step is to compile a list of functional and non-functional requirements for SCS backend database and perform a feature analysis on a set of state-of-the-art TSDB systems to determine the databases that fulfill the requirements of a SCS data management.

With the wide adoption of TSDB, and the evolution of applications that require time-series data management, a wide range of TSDBs have emerged, some of these TSDBs are built as extensions to existing well-known relational, NoSQL or even other TSDB databases, and some of them are built from scratch. We have identified a set of TSDBs for our analysis – refer to Table 8.

Given the big number of the TSDBs to analyze, and to make the feature analysis process more efficient, we have split our requirements into critical and important requirements, we then performed a preselection process based on the critical requirements to eliminate the databases that lack one or more of these requirements.

Below is a list of the critical requirements:

- Stable and Mature Product: We measure product stability and maturity based on the following criteria:
    - Customer base: the database should have major customers
    - Stable Release: the database should have stable releases
    - Well established: This is measured by the funding stage, key investors, if it is backed by a grown community (based on the number of GIT stars, forks and number of contributors).
- Available for on-premises hosting – as it is an important feature for KATRIN
- Interfaces: Provides APIs and libraries to interact with
- Horizontal scalability and high availability for Reads and Writes
- Continuous aggregation: The database should provide a way to refresh pre-defined aggregated queries continuously and incrementally.

4.2.1. Evaluation

Database	Mature	On-prem	Integration	Scalability and HA	Continuous aggregation
TSDB built from scratch					
ClickHouse	Yes	Yes	HTTP, ODBC, JDBC, MySQL, HDFS, CSV, JSON, XML, Protobuf, CapnProto	Available	Available
InfluxDB	Yes	Yes	HTTP API, JSON over UDP, python, C#	Available[4]	Available
Druid	Yes	Yes	JDBC, RESTful HTTP/JSON API	Available	Not available
Prometheus	Yes	Yes	RESTful HTTP/JSON API, python	Not Available	Available
CrateDB	Yes	Yes	ADO.NET, JDBC MQTT, PostgreSQL wire protocol, Prometheus Remote Read/Write, RESTful HTTP API, python	Available	Not available
SiriDB	No	Yes	HTTP API, python	Available	Not available
SingleStore	Yes	Yes	JDBC, ODBC	Available	Not available
QuestDB	Yes	Yes	HTTP REST, InfluxDB Line Protocol, JDBC	Not available	Not available
VictoriaMetrics	No	Yes	PromQL	Available	Available
TDengine	No	Yes	JDBC connectors: C/C++, JAVA, Python, RESTful, Go, Node.JS	Available	Available
Akumuli	No	Yes	Http Rest API	Not available	Not available
TSDB built on top of relational database					
TimescaleDB	Yes	Yes	C, *, ADO.NET, JDBC, ODBC, python, streaming API for large objects	Available[5]	Available
Citus	No[6]	Yes	ADO.NET, JDBC, native C library, ODBC, streaming API for large objects, python	Available[7]	Available

[4] InfluxDB scalability and HA features are available with the enterprise edition.
[5] TimescaleDB multi-node supports horizontal scaling for data nodes, the access node can also be physically replicated for high availability using load balancer
[6] Complicated process to acquire Citus enterprise edition
[7] HA and streaming replication are available in Citus enterprise

TSDB built on top of NoSQL					
OpenTSDB	Yes	Yes	HTTP API, Telnet API, python	Available	Not available[8]
ScyllaDB	Yes	Yes	Proprietary protocol (CQL), RESTful HTTP API, Thrift[9]	Available [10]	Not available[11]
warp10	Yes	Yes	HTTP API Jupyter WebSocket	Available	Not available
Axibase	No	Yes	JDBC, Proprietary protocol (Network API), RESTful HTTP API, python	Available [12]	Not available
Metrictank	Yes	Yes	Graphite API, python	Available	Not available
Arctic	No	Yes	Python [13]	Available	Not available
hawkular-metrics	No	Yes	REST/HTTP with json output, python	Available	Not available
TSDB built on top of other TSDBs					
IoTDB	No	Yes	JDBC	Not Available [14]	Not available[15]
M3DB	No	Yes	M3QL, PromQL, Graphite	Available	Not available[16]
KairosDB	No	Yes	Graphite protocol, HTTP REST, Telnet API	Available	Not available
Thanos	Yes	Yes	RESTful HTTP/JSON API, PromQL	Available	Not available
FiloDB	No	Yes	Prometheus-compatible HTTP API	Available	Not available
Timely	No	Yes	UDP, TCP, HTTPS, WebSocket		Not available
Cortex	No	Yes	HTTP API	Available	Not available

Table 3 - TSDB preselection matrix

[8] OpenTSDB supports rollups at ingestion time through batch processing or stream processing using spark, flink etc.
[9] Scylla supports the same set of drivers that are compatible with CQL/Cassandra (like DataStax Python Driver)
[10] ScyllaDB scalability and HA features are available with the enterprise edition.
[11] ScyllaDB supports materialized views and native aggregates, however it does not support continuous aggregation
[12] Axibase scalability and HA are available with the enterprise edition
[13] Native compatibility with Python as the database is developed using python
[14] Currently only IoTDB standalone version is available
[15] Currently only the standalone version is available
[16] M3DB supports down-sampling at ingestion time. This requires defining all the aggregations before the ingestion

4.2.2. Results

Based on the preselection results, we notice that InfluxDB, ClickHouse and TimescaleDB provide all the must-have features. Therefore, we base the remainder of our analysis on these databases.

TimescaleDB is built on top of the popular open-source relational database PostgreSQL. It adds an abstraction layer called hypertable that chunks data into multiple underlying standard tables corresponding to a specific time interval each, while abstracting it as a single, large table for interacting with the data.

This mechanism allows to avoid slow disk operations by storing the most recent chunk in memory. TimescaleDB inherits the advantages of mature relational databases from PostgreSQL like high reliability and security, full support of the SQL features, compatibility with the tools supported by PostgreSQL etc. With the 2.0 release, TimescaleDB added support for horizontal scalability with multi-node setup [11]

InfluxDB is an open-source time series database written in the Go programming language for storage and retrieval of time series data. InfluxDB uses its own functional query language called Flux, it also provides a SQL like query language called InfluxQL. The community version of InfluxDB supports only single node installations. Scalability and replication are supported by the enterprise version.

ClickHouse is a popular open-source columnar database for online analytical processing (OLAP). It is optimized for large-batch ingestion and big query workload that can reach hundreds of queries per second per node. ClickHouse supports horizontal scalability for write and read operations, data replication, a SQL like query language that supports most of SQL queries like GROUP BY, ORDER BY, JOIN etc.

4.3. Feature comparison

After determining the databases that have all the critical requirements, we performed a more in-depth analysis on these databases based on the important features. The below table summarizes the results of this analysis:

Feature	TimescaleDB	InfluxDB	ClickHouse
Initial Release	2017	2013	2016
Commercial support	Yes	With enterprise edition	Through external service providers
Open Source	Yes	Only single-node version is available as open-source. The enterprise version is commercial	Yes
Supported OS	Linux, OS X, Windows	Linux, OS X	FreeBSD, Linux, macOS
Cloud version available	Fully managed cloud version available on AWS, AZURE, Google Cloud	Fully managed cloud version available on AWS, Azure, and Google Cloud	Fully managed cloud version available on Yandex Cloud, AWS, Alibaba Cloud, Tencent Cloud
SQL/ODBC Compatibility	Full PostgreSQL support	ODBC, InfluxQL (SQL like), Flux (functional language more powerful than InfluxQL)	Close to ANSI SQL - Compatible with ODBC
Data Ingestion frameworks	Prometheus through Promscale, Telegraf, Kafka	Telegraf	Kafka
Visualization frameworks	Compatible with visualization tools that work with PostgreSQL such as: Grafana, Tableau, PowerBI, Looker, Periscope, Mode, Chartio.	Influxdb UI, Grafana	Tabix (open source), HouseOps, LightHouse, Grafana, Looker…
Jupiter notebook integration	Yes	Yes	Yes

Replication	Available through master-slave (pgsql streaming replication only) [17]	Available with the enterprise edition	Available
Cluster rebalancing	New chunks are distributed among the available nodes. After adding a new node, existing chunks are not rebalanced between nodes (this is under development).	New data is distributed among the available nodes. After adding a new node, existing data is not automatically rebalanced between nodes.	New data is distributed among the available nodes. After adding a new node, existing data is not automatically rebalanced between nodes.
Architecture	Postgres Extension Hypertable is an abstraction or virtual view of many tables called chunks. Hypertables can be queried via standard SQL	Time-Structured Merge Tree	MergeTree / OLAP data is physically sorted by primary key (single or tuple of columns)
Data Model	row-oriented	Tagset – data is saved in columnar mode	Column-oriented
Indexing	Supports multiple indexes	The Time Series Index TSI stores series keys grouped by measurement, tag, and field	Single sparse index: primary key . Data can be logically grouped by partition key
In-memory cache	Yes	Yes	Yes
Compression	Yes	Yes	Yes
Time Resolution	Micro	nano	nano[18]
Statistical Query	Yes	Yes	yes
Aggregation intervals (m/h/d or configurable)	Configurable	Configurable	Live
User defined functions	Yes[19]	Yes	No
SQL Triggers	Yes	No	No
Retention Policy	Yes	Yes	Yes

Table 4 - TSDB feature comparison

[17] Cluster-wide replication across data nodes, built natively for TimescaleDB, is in development

[18] Through Datetime64

[19] Provided that the user defined functions are parallel safe

To better visualize the results side by side, we created the below table that summarizes some of the main features:

- Customer base: 0: Weak, 1: Moderate, 2: Strong
- Product maturity: 0: No stable release, 1: 1-3 years, 2: 4-9 years, 3: 10+ years
- Open source: 1: No (Single node), 2: Yes
- Scalability and high availability: 0: No, 1: Paid or partial, 2: Yes free
- Continuous aggregation: 0: No aggregation , 1: Not continuous , 2: Continuous
- SQL and ODBC support: 0: No, 1: Partial or SQL like, 2: Full
- Extensible Aggregation: 0: No, 1: Yes

	Customer base	**maturity**	**Open Source**	**HA & scalability**	**Continuous Aggregation**	**SQL / ODBC**	**Extensible Agg**	**Score**
ClickHouse	2[12]	2	2	2	2	1	1	12
InfluxDB	2[13]	2	2	1	2	1	2	12
TimescaleDB	2[14]	2	1	1	2	2	2	11

Table 5 - TSDB scoring

We notice that the three databases have a close score and therefore, there is a need to compare their performance to decide which one is the best for our case.

4.4. Database Design

At KATRIN, sensors are clustered by physical groups with each group belonging to a subsystem. Groups sample data at the same time at an average frequency of 1Hz with some of the groups sampling data at higher frequencies of 10 or 100 Hz.

Currently wide tables are used to store sensors data. Each table corresponds to a physical group with the following data structure: Timestamp, sensor1, sensor2…, sensor 100, …

Although using wide data format is typical when data is collected and stored together, however this is not a scalable solution in the case of KATRIN, as sensors are continuously added or removed from the system or continuously moved between groups. Adopting a wide data representation in this case requires changing the structure of the current tables to add or remove sensors, Consequently, the tables must be periodically archived to preserve the old structure. This complicates the queries that tend to get historical data for a specific sensor.

The novel database should solve the scalability, performance, and maintenance issues of the current data structure while providing a high ingestion rate of 100K metrics per second.

To design the perfect data structure and indexes, it is essential to understand how each one of the selected databases performs indexing and data chunking over time.

4.4.1. TimescaleDB data architecture

TimescaleDB is built as an extension to PostgreSQL, it natively supports relational data models and B-tree indexes. With B-tree index, a query can quickly find a row without scanning the entire table. If the B-tree index is small, it can be kept in memory, however, if the dataset is large enough that the entire B-tree can't be stored in memory, then accessing or updating data can involve significant disk I/O to read pages from disk into memory and write them back to disk. The problem exacerbates as more indexes are added.

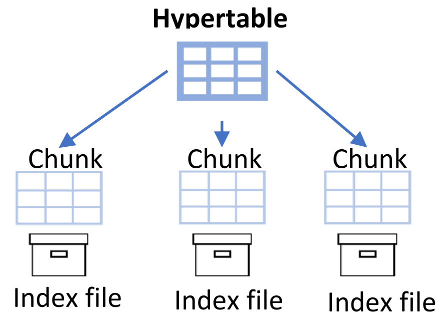

Figure 23 - TimescaleDB hypertable chunking

Given that time-series write operations are mostly inserts to a recent time interval, and they are primarily associated with a timestamp and a separate primary key like sensor Id etc.

TimescaleDB organizes data using an adaptive time/space chunking by splitting data into distinct *tables* called *chunks* according to two dimensions: the time interval and a space dimension like server ID. Because each of these chunks are stored as a database table itself, and the query planner is aware of the chunk's ranges (in time and space), the query planner can immediately tell to which chunk(s) an operation's data belongs.

The key benefit of this approach is that all the indexes are built only across these much smaller chunks (tables), rather than a single table representing the entire dataset. So, if the chunks are properly sized, the latest tables (and their B-trees) can completely fit in memory, and avoid this swap-to-disk problem, while maintaining support for multiple indexes.

Based on TimescaleDB recommendation, the size of data in a chunk should not exceed 25% of the available memory.

4.4.2. InfluxDB data architecture

InfluxDB structures data into elements like measurement, tag set, field set, and timestamp. InfluxDB uses line protocol to write data points. It is a text-based format that provides the measurement, tag set, field set, and timestamp of a data point. Below is a representation of a data point in InfluxDB:

Measurement	Tag Set	Field Set	Timestamp
Temperature,	machine=unit42,type=assembly	external=25,internal=37	1434067467000000000

Table 6 - InfluxDB data representation sample

Data Elements:

- Measurement: represents the name of the measurement. It acts as a container for tags, fields, and timestamps. A measurement maps to a table in standard database terms.
- Tags: indexed key-value pairs associated with a data point. Tags map to indexed columns.
- Fields: non-indexed field-value pairs for the data points. Fields map to non-indexed columns.
- Timestamp: unix timestamp for the data point.
- Series: logical grouping of measurement, tag set and field keys.
- Series cardinality: number of unique series.
- Series key: combination of unique measurement, tag set, and field key.
- Bucket: All InfluxDB data is stored in a bucket. A bucket maps to a database.

Storage Engine [15]:

Shard: InfluxDB separates data into shards according to pre-defined duration. Shard duration should be calculated based on the typical queries time range.
Write Ahead Log (WAL): retains data when the storage engine restarts. The WAL ensures data is durable in case of an unexpected failure. Write requests are written to the WAL file, then to the cache and then they are flushed to disk.
Cache: The cache is an in-memory copy of data points currently stored in the WAL. Queries to the storage engine merge data from the cache with data from the TSM files
TSM (Time-structured Merge Tree): To efficiently compact and store data, the storage engine groups field values by series key, and then orders those field values by time. TSM files store compressed series data in a columnar format. Each TSM file corresponds to a shard. A TSM file is split into

1. Data partition containing field values and timestamps stored in columnar format. This storage mode allows using different compression algorithms based on the data type to reduce the size of timestamps and field values.

2. Index partition containing series keys and field keys to easily locate the data partitions of specific series keys.

TSI (Time Series Index): used to ensure that queries remain fast as series cardinality grows. TSI allows to map between tag key, tag value and series keys. This allows to quickly locate the list of series keys involved in a query to retrieve the corresponding data from the TSM. Following is the data structure of a TSI file:

- map<SeriesID, SeriesKey>
- map<tagkey, map<tagvalue, List<SeriesID>>>

4.4.3. ClickHouse data architecture

ClickHouse is a column-based database that is not specialized for time-series data only. However, its scalability and high performance for analytics workloads makes it a good fit for time-series data. ClickHouse uses **MergeTree** to store data. MergeTree is a storage engine that supports indexing by primary key, it is designed for inserting a very large amount of data into a table. When a bunch of data is inserted into MergeTree, that bunch is sorted by primary key order and forms a new part that is directly written to the filesystem, then rules are applied for merging the parts in the background, therefore it is called MergeTree. .

MergeTree is not an LSM tree because it does not contain "memtable" and "log": inserted data is written directly to the filesystem. This makes it suitable only to INSERT data in batches, not by individual row and not very frequently – ClickHouse supports few write requests per second [16].

For further partitioning of data, ClickHouse uses **Partitioning key** to partition data by month, week, day etc. Data belonging to different partitions are separated into different parts. In the background, ClickHouse merges data parts for more efficient storage, however, parts belonging to different partitions are not merged.

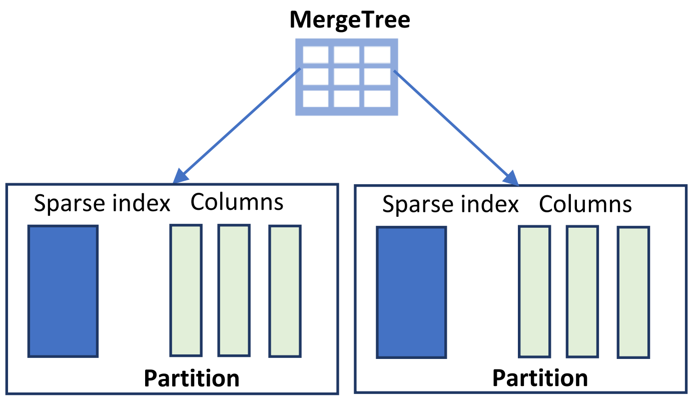

Figure 24 - ClickHouse Partitioning

4.4.4. Discussion

Currently KATRIN uses the wide-table model, where a table is created for each group, with the corresponding sensors created as columns. Using this model, it is faster to execute queries across multiple sensors in the same group, the ingestion is also faster as only one timestamp is written for all the sensors within the group. However, querying sensors in multiple groups is slower as it requires joins. The main problem with this data representation is that it lacks scalability in terms of moving sensors between groups.

To solve the scalability issue, we can decouple sensors from groups and record them individually in a narrow model where each data point will have a timestamp, sensor id and value.

The narrow model solves the scalability issue, and adds a performance overhead on the ingestion as the timestamp will be recorded with each sensor, and on the queries that perform an operation on different sensors from the same group e.g. difference between 2 sensors.

To keep the relationship between sensors, groups, and subsystems flexible, we can store the structural information about the sensors in a separate metadata database.

Based on the architecture of the different TSDBs, we recommend the following:

TimescaleDB: We recommend storing KATRIN sensors data into one single hypertable with the following data structure: *sensor_id, value, timestamp.* As per Figure 5, 74% of the queries target data that is less than 1 day (44 % last 12 hours and 30% 12-24 hours), based on this, we recommend setting the chunk size to 1 day.

The theoretical size of data generated in one day is : 100K points/sec * 86400 sec/day * (8byte (int) + 8byte (timestamp) + 8byte(double)) = around 207 GB of raw data excluding the index size. As per TimescaleDB recommendation, this should be 25% of the available memory, thus the total memory should be more than: 828 GB.

A multi-node installation is needed in this case with space partitioning to split the load among multiple nodes.

Given that data is often filtered by sensor, we recommend adding an index on sensor_id ordered by time in descending order.

InfluxDB: Given that data is often filtered by sensor, we recommend adding sensor_id as tag key or as a measurement. So, we have two options to represent Katrin data with InfluxDB: (1) Having one measurement for all data and defining the sensor_id as a tag key and sensor value as field set. (2) Having one measurement per sensor with no tag sets. After consulting with InfluxDB team, they recommended to use a single measurement with sensor_id as tag key. We recommend also to set the shard size to 1 day.

ClickHouse: We recommend using a single MergeTree to store all sensors data with the following structure: sensor_id, value, time. We recommend defining [sensor_id, time] as the primary key. We recommend also adding partition key by week as ClickHouse do not recommend having more than about a thousand partitions [17]. .

4.5. Benchmarking

The purpose of this phase is to benchmark the selected databases: TimescaleDB, InfluxDB and ClickHouse to compare and evaluate their performance for data ingestion and for specific read scenarios.

4.5.1. Performance Metrics

For read scenarios, the performance is evaluated based on the query execution time or latency. The latency is the time needed to execute a query and return the results.

For ingestion test cases, the performance is evaluated based on the ingestion rate which is equal to the total number of ingested data points divided by the execution time.

4.5.2. Test scenarios

Based on the usage analysis that we performed on KATRIN logs data, we came up with the below list of test scenarios to be used in our benchmarking:

4.5.2.1. Read scenarios

We executed the below read benchmarks on Cold data. We define cold data as data included in different time partition(s) (or chunks), we performed several executions of the same query on data from different time partitions.

- Aggregated Data: Get two days of aggregated data where sensor_id in?
- Raw Data: Get one hour of raw data for a specific sensor
- Out of range query: Find all time intervals within 3 months where the value of a specific sensor is out of the allowed range [min-max]
- Mathematical operations:
    - Calculate the difference between the aggregated values of 2 sensors over two days
    - Calculate the standard deviation of a specific sensor over two days

4.5.2.2. Ingestion scenarios

We evaluated the ingestion rate of the TSDBs while varying one of the following parameters: batch size, number of series and the number of parallel clients.

4.5.2.3. setup

We performed the benchmarking on two machines

- Server machine used to run a single-node TSDB installation of ClickHouse, InfluxDB and TimescaleDB with the following specs: Intel(R) Xeon(R) CPU E5-1620 v2 @ 3.70GHz, 8 CPU cores, 32 GB of memory, one RAID-5 disk drive of 60 TB, operating system: Centos7.

- Client machine used to run the benchmarking tool with the following specs: Intel(R) Xeon(R) CPU E5540 @ 2.53GHz, 8 CPU cores, 96 GB of memory, operating system: Centos8.

When performing a test on a specific database, we stop the other databases to make sure that they do not consume any resources like memory and CPU and therefore do not affect the test results.

4.5.3. Benchmarking tool

Our tool is implemented using C# and .Net Core and it is based on existing open-source .net client libraries. For TimescaleDB, we used **PostgreSQLCopyHelper** [18] which is a library for bulk inserts to PostgreSQL. It wraps the COPY method for bulk import from Npgsql [20].
For ClickHouse, we used **ClickHouse.Ado** [19], a .net provider that implements native ClickHouse protocol. For InfluxDB, we used **influxdb-client-csharp** [20], a C# client for InfluxDB 2.0 that supports querying using the Flux[21] language and supports bulk data insert using HTTP Line Protocol [22].
To ensure that the client libraries that we used do not add considerable performance overhead and that our results are as accurate as possible, we downloaded the libraries and checked the data processing logic used before executing the batch write, we found that **influxdb-client-csharp** is doing heavy data processing by looping through all the characters of the measurement, tags and field keys to make sure that they abide by InfluxDB Line protocol naming rules [21] . After excluding this logic from the code as it does not add any value to our test, we noticed a big increase in the ingestion rate results for InfluxDB, this is shown in Figure 25 where we can see the big difference between the ingestion rate of InfluxDB before and after optimizing the client library code.

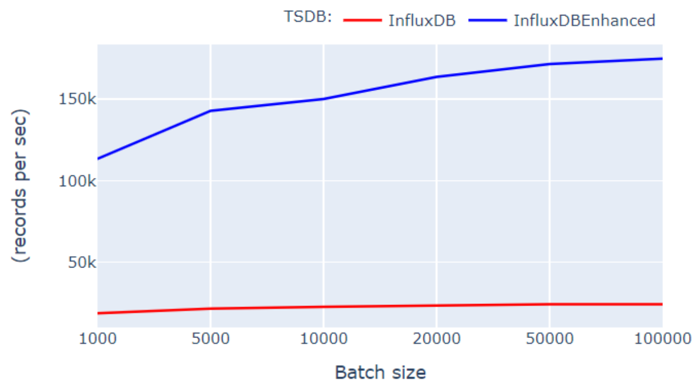

Figure 25 - Ingestion rate with InfluxDB client lib optimization

[20] A well-known ADO.NET library for PostgreSQL
[21] Standalone data scripting and query language from the makers of InfluxDB.
[22] Text-based format for writing points to the InfluxDB.

4.5.3.1. *Architecture*

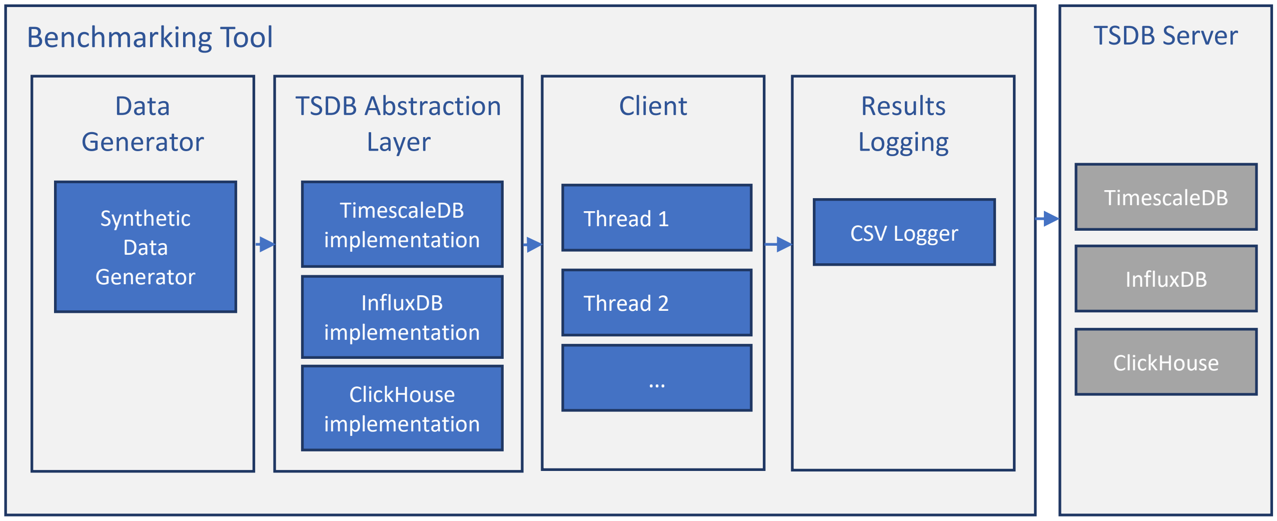

Figure 26 - Benchmarking tool architecture

The tool is built in an extensible way to allow to add support for more TSDBs. This is done through an abstraction layer by creating an interface to define the signature of the methods needed to interact with a TSDB. By implementing this interface, a user can add support to a new TSDB. For instance, the tool provides concrete implementations for TimescaleDB, InfluxDB and ClickHouse.

To be able to perform all the benchmark tests with the different queries, TSDBs and parameter combinations, we created a configuration layer that allows the user to change all the benchmark parameters through a configuration file at runtime without the need to do any changes to the code. For example, the user can indicate the TSDB to run the test on, the operation to perform (e.g. ingestion, raw query, out of range query), how many times to repeat the operation… In case of ingestion test, the user can configure the number of parallel client number(s) – s/he can enter one or multiple values to run the same test with different numbers of clients, the batch size(s), the series number… and in case of query operations, user can configure the aggregation interval, the time span, the aggregation method, the start date for pulling data, the sensor(s) to retrieve...

We used parallel threading to support concurrent client tests; based on the client number parameter, the tool initiates parallel threads and waits for all the threads to complete to return the results.

A synthetic data generator is used to automate the generation of data based on the narrow data schema: time (Timestamp), sensor_id (int) and value (decimal). The data generator creates batches of data points based on the predefined configuration params: batch size, number of parallel clients and number of sensors. Each batch is assigned a timestamp with a configurable offset between subsequent batches.

To ensure that we can analyze the benchmarking results, the tool appends the result of each benchmarking iteration to a csv file along with the main test details like the number of ingested data points, the number of data points that were not ingested successfully, the latency, the number of clients used, the batch size, the series number, the TSDB and the operation performed. This allows to analyze the results and compare the performance of the different databases.

4.5.4. Results

4.5.4.1. Ingestion Scenarios

The purpose of this test is to evaluate the ingestion performance of the database and to understand the relationship between performance and system parameters like batch size, number of parallel clients and number of sensors.
In all the ingestion tests, we made sure that all data points were successfully ingested, and we did not have any data loss.
We executed several test scenarios while varying one of these parameters and fixing the others:

Varying the batch size

In this test we compare the performance of the TSDBs with different batch sizes and with 100K sensors. We performed this test twice, with a single client and with 12 concurrent clients.

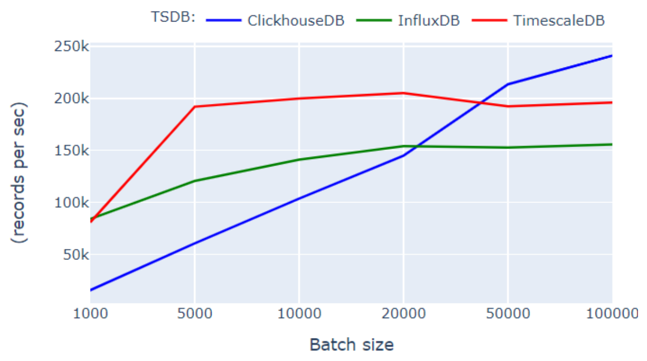

Figure 27 - Ingestion rate mean per batch size with single client

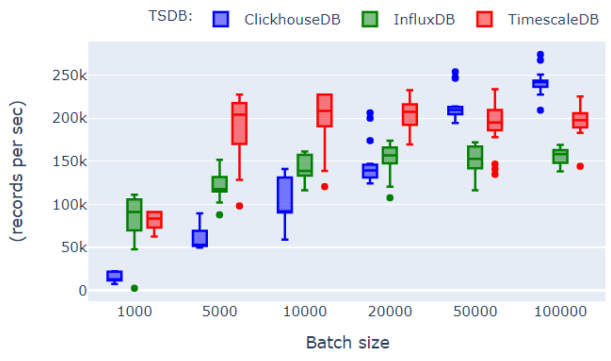

Figure 28 - Ingestion rate boxplot per batch size with single client

The result shows that with a single client, with the batch size increasing, the ingestion rate of ClickHouse is constantly growing while it is not changing considerable for InfluxDB. The Ingestion rate of TimescaleDB reaches its optimal value with a batch size of 5K.

TimescaleDB has better performance with batch sizes less than 50K, however, with larger batch sizes, the performance of ClickHouse outperforms Timescale and InfluxDB.

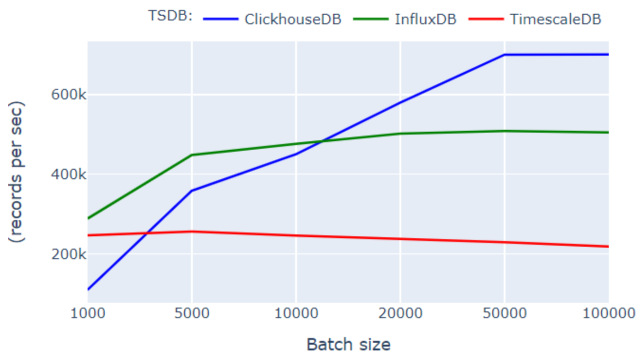

Figure 29 - Ingestion rate mean per batch size with 12 clients

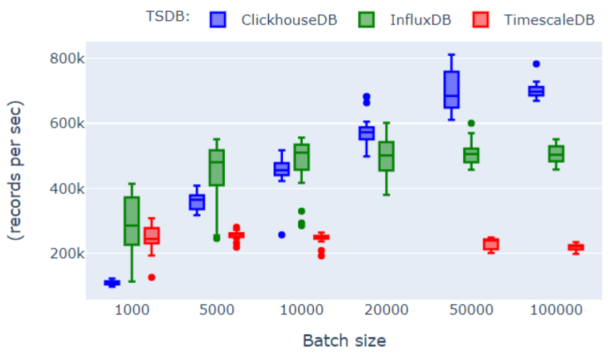

Figure 30 - Ingestion rate per batch size with 12 clients

With 12 clients, we notice that the performance of TimescaleDB is almost the same as with one client, it reaches its limit of around 200K records/sec and becomes insensitive to the batch size.

Whereas the performance of InfluxDB increases 3x and it reaches its optimal value of around 420K records/sec with 5K batch size.

ClickHouse performance also increases by 3x and continues increasing with the batch size to reach 700k data points/sec with 100K batch size.

Varying the number of sensors

In this test we compare the performance of the TSDBs with different sensors numbers to see how the cardinality affects the performance. We perform this test using a single client and a batch size of 20K. The choice of the batch size came from the fact that InfluxDB optimal batch size is 5K[22] , however ClickHouse performs better with high batch sizes (Figure 27), on the other hand, TimescaleDB is very sensitive to the available memory, so increasing the batch size will increase the memory usage and might affect the performance. Therefore, we chose a value in between which is 20K.

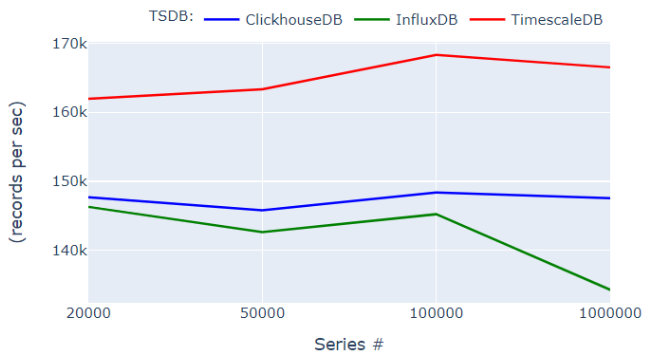

Figure 31 - Ingestion rate mean per series number

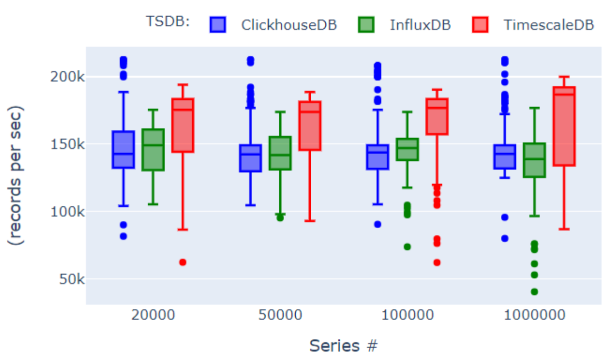

Figure 32 - Ingestion rate boxplot per series number

The result shows that with the series number increasing, the ingestion rate of ClickHouse and TimescaleDB do not change considerably, for InfluxDB, it drops slightly with 1M series number – refer to the boxplot in Figure 32.

Varying the number of parallel clients:

In this test scenario, we compare the ingestion rate of the TSDBs with different concurrent client thread numbers while fixing the batch size to 20K and the number of sensors to 20K.

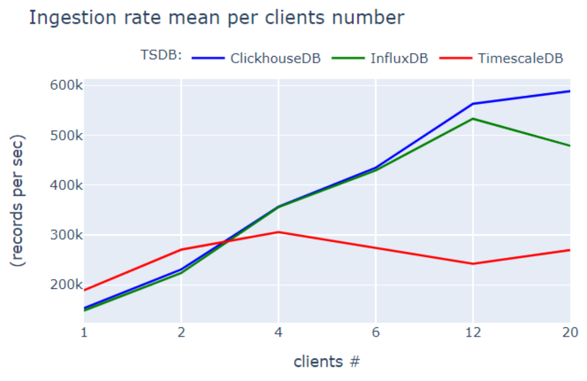

Figure 33 - Ingestion rate mean per concurrent clients #

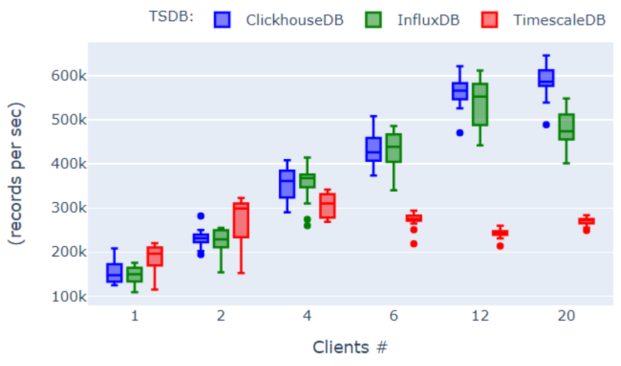

Figure 34 - Ingestion rate boxplot per concurrent clients #

We notice that with the number of concurrent clients increasing, the ingestion rate of ClickHouse and InfluxDB is growing with the same pace until 12 concurrent clients, where the ingestion rate of InfluxDB stopped to increase and the ingestion rate of ClickHouse increased at a slower pace. This is not the case with TimescaleDB that reached its optimal performance of 300K record/sec with 4 clients.

4.5.4.2. Read Scenarios

Data population

Before starting the benchmarking on Read queries, we reset the TSDBs, and we populate each one with synthetic data generated by 10K sensors over a period of two years. The data population script performs iterations with 15 minutes offset, at each iteration, each sensor generates 3 data points with an offset of 1 second. This makes 2 billion of generated data points which is theoretically equivalent to around 48 GB of uncompressed data.

Data Compression

When we first started comparing the latency of the TSDBs, we found that TimescaleDB is constantly giving high latency. We checked the storage size of the TSDBs and found that the storage occupied by TimescaleDB is much higher than ClickHouse and InfluxDB – refer to Table 7 This is due to that both ClickHouse and InfluxDB have the compression enabled by default, which is not the case for TimescaleDB. After enabling the compression on TimescaleDB, we found that the performance improved considerably – refer to Figure 35.
Table 7 shows the high compression rate of around 90% performed by TimescaleDB. It also shows that the compressed storage size of both ClickHouse and InfluxDB is smaller than TimescaleDB.

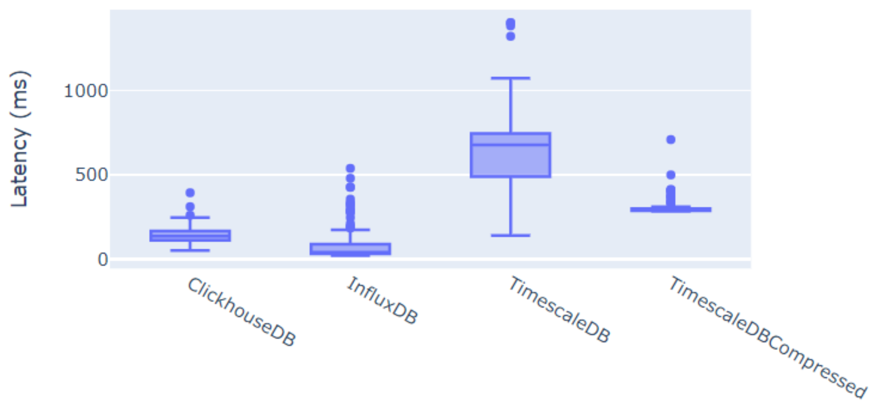

Figure 35 - Latency changes after performing compression on TimescaleDB

Database	Uncompressed size	Total compressed size
TimescaleDB	Data size: 114.3 GB – index size: 116.5 GB	21 GB
ClickHouse		16 GB
InfluxDB		Around 16 GB

Table 7 - TSDB storage size

Read scenarios benchmarking results

We repeated each test scenario 100 times on different data chunks that are chosen randomly over a period of 2 years.

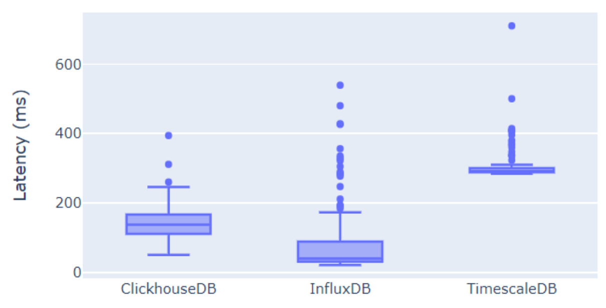

Figure 36 - Range query with aggregated data latency

Query: Return the average value for a set of sensors over a period of 2 days with 30 minutes aggregation

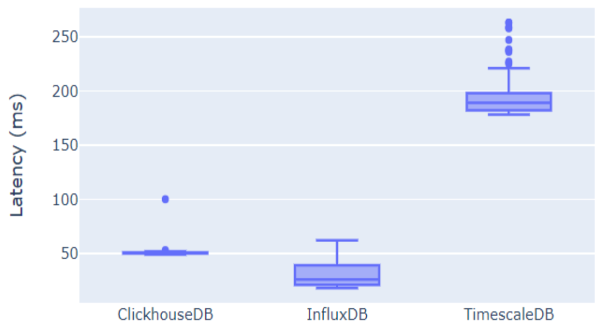

Figure 37 - Range query with raw data latency

Query: Return the values of a specific sensor over a period of one hour

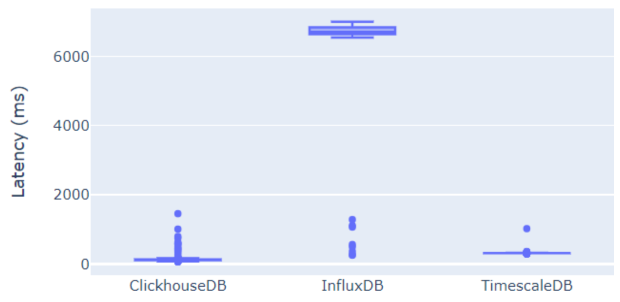

Figure 38 - Out of range query latency

Query: Return the time intervals over 3 months where the value of a specific sensor was out of [min-max] range

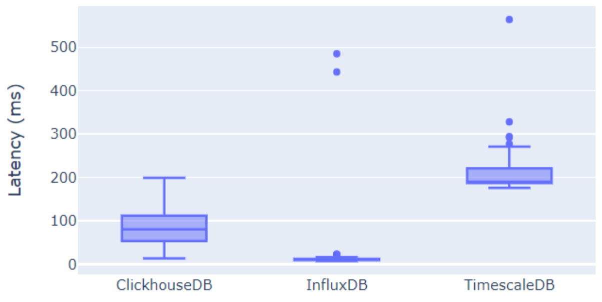

Figure 39 - Standard deviation query latency

Query: Return the standard deviation of a specific sensor over 2 days

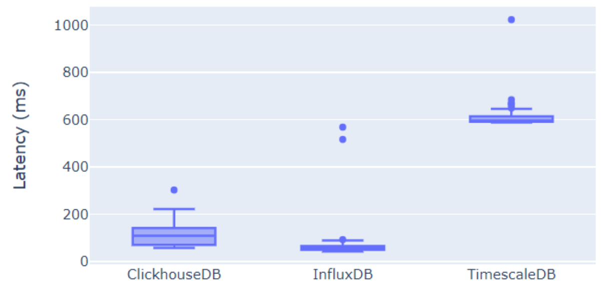

Figure 40 - Difference between sensor values query latency

Query: Return the difference of values between 2 sensors over 2 days with 30 minutes aggregation

The result shows that InfluxDB and then ClickHouse presents the lowest latency for all the test scenarios except for the Out-of-Range query where InfluxDB has the highest latency and ClickHouse and TimescaleDB have very close latency.

4.5.5. Discussion

In most cases, the ingestion performance of ClickHouse has been better than InfluxDB and TimescaleDB. ClickHouse performance was constantly increasing with the batch size and the number of concurrent clients; it reached an optimal performance of 700K records/sec with 12 clients and 100K batch size.

InfluxDB ingestion rate was increasing with the number of concurrent clients; however, it was less sensitive to the batch size, which is aligned with InfluxDB's recommendation of using an optimal batch size of 5K. InfluxDB reached 500K records/sec with 12 clients.

TimescaleDB reached its performance limit of around 300K records/sec with 4 concurrent clients and did not show further improvement with more clients or bigger batch sizes.

For the read scenarios, InfluxDB and ClickHouse showed comparable performance that outperformed TimescaleDB.

Due to its high ingestion rate, ClickHouse proved to be more suitable for managing SCS time-series data despite not being a purpose-built time-series database.

Given that KATRIN SCS includes 50 subsystems with 100 groups each, which makes a total of 5000 groups sampling data together and given that ClickHouse is not suitable for frequent inserts per second, we recommend implementing an intermediate layer to first process and transform data into narrow representation, and then buffer the readings from all the sensors in one batch of 100K data points and send it to ClickHouse once per second. To perform this, a streaming service like KAFKA can be used.

5. Conclusion

In this work, we provided KATRIN with an evaluation methodology to select the best TSDB for managing the SCS operational data based on its specific usage and requirements.

We helped in defining and discovering the functional requirements by analyzing KATRIN logs. Our analysis showed that 96% of queries are targeting the last values, so we recommended to provide a real-time interface to serve these requests and reduce the load of the back-end database. We proved that most of the queries target 1-day old data, whereas other queries target historical data. To speed up the query performance, we recommended partitioning the database based on time and use continuous aggregations for historical queries.

Several time-series databases have been studied, three of which proved to have all the requirements of the SCS: ClickHouse, TimescaleDB and InfluxDB. These databases have been further analyzed to find the best data model, indexing, and partitioning that best represent KATRIN data. We recommended to use 1-level data model with indexing on sensor id, to simplify the data hierarchy especially in a dynamic system with sensors being added and removed continuously. For the partition size, our recommendation was to use 1 day for TimescaleDB and InfluxDB and 1 week for ClickHouse as it does not recommend having more than about a thousand partitions per database.

To evaluate the performance of the selected databases, we created an extensible tool designed to test KATRIN usage scenarios. Based on the benchmarking results, the three databases were able to ingest more than 100K data points per sec – which is the requirement of KATRIN[23], however, ClickHouse showed the best performance and was therefore selected to manage the SCS data. ClickHouse proved to work well with big batch sizes, therefore our recommendation was to ingest 1 batch of 100K points per second. To perform this, KAFKA can be used to transform and buffer data points from the different sensor groups and send them to ClickHouse.

[23] The ingestion tests were performed on empty databases and therefore we cannot expect the same performance on production with real data size and concurrent queries. However, the tests give an intuition about the capabilities of the databases.

Appendix

List of time-series databases:

TSDB	Definition and notes
TimescaleDB	An open-source time-series SQL database optimized for fast ingest and complex queries. Packaged as a PostgreSQL extension. Developed by Timescale - New York. Series-B funding.
ClickHouse	A column-oriented database management system for online analytical processing of queries. Developed by Yandex LLC - Russia. Has been listed on the NASDAQ since 2011 under the ticker YNDX.
VictoriaMetrics	A fast, cost-effective monitoring solution and time series database. Developed by Victoria Metrics Inc.
Prometheus	Open-source TSDB and monitoring system. Originally developed at SoundCloud before being released as an open-source project. Cloud Native Computing Foundation graduated project.
InfluxDB	Scalable datastore for metrics, events, and real-time analytics. Developed by Influxdata – California. Series-D funding.
ScyllaDB	Cassandra and DynamoDB compatible wide column store. Developed by Scylla – California. Series C funding.
OpenTSDB	Scalable TSDB based on Hbase. Currently maintained by Yahoo and other contributors.
Akumuli	TSDB for modern hardware. It can be used to capture, store and process time-series data in real-time.
CrateDB	A distributed SQL database that makes it simple to store and analyze massive amounts of machine data in real-time. Developed by Crate – California. Series-A funding.
SiriDB	Highly scalable, robust, and super-fast time series database built from the ground up. Developed by Transceptor Technology – Netherlands.
M3DB	Prometheus compatible, easy to adopt metrics engine that provides visibility for some of the world's largest brands. Created by Uber and open sourced in 2018.
Druid	Open-source analytics data store designed for sub-second OLAP queries on high dimensionality and high cardinality data. Developed by Apache Software Foundation and contributors.
IoTDB	A lightweight IoT native database management system deployable on the device and syncing the collected data with the cloud and integrated with Hadoop and Spark. Developed by Apache Software Foundation. Entered the Apache Incubator in November 2018. Announced as a Top-Level Project (TLP) in 2020.

Axibase	A non-relational database optimized for collecting, storing, and analyzing temporal data from IT infrastructure, industrial equipment, smart meters, and IoT devices. Developed by Axibase Corporation – California.
Timely	A TSDB application that provides secure access to time series data. Timely is written in Java and designed to work with Apache Accumulo and Grafana. It started as a clone of OpenTSDB, which uses Accumulo instead of HBase for its underlying storage. Developed by NSA (National Security Agency).
hawkular-metrics	A scalable, asynchronous, multi-tenant, long term metrics storage engine that uses Cassandra as the data store and REST as the primary interface.
Metrictank	Multi-tenant timeseries platform for Graphite developed by Grafana Labs.
QuestDB	A high-performance open-source SQL database for time series data.
TDengine	A highly efficient platform to store, query, and analyze time-series data. Developed by taosdata - Headquartered in Beijing.
Thanos	Open source, highly available Prometheus setup with long term storage capabilities. Developed by IMPROBABLE – UK. Cloud Native Computing Foundation Incubating project.
warp10	TimeSeries DBMS specialized on timestamped geo data based on LevelDB or Hbase. Developed by SenX – France.
Arctic	Timeseries / dataframe database that sits atop MongoDB. Arctic supports serialization of several datatypes for storage in the mongo document model. Developed by Man AHL.
FiloDB	Distributed, Prometheus-compatible, real-time, in-memory, massively scalable, multi-schema time series / event / operational database. Developed by TupleJump and acquired by Apple in 2016.
Cortex	Open source timeseries database and monitoring system for applications and microservices. Based on Prometheus, Cortex adds horizontal scaling and virtually indefinite data retention. Cloud native computing foundation project with "incubating" status. Created by Weaveworks – UK.
Citus	Scalable hybrid operational and analytics RDBMS for big data use cases based on PostgreSQL. Acquired by Microsoft in 2019.
SingleStore	MySQL wire-compliant distributed RDBMS that combines an in-memory row-oriented and a disc-based column-oriented storage.
KairosDB	Time Series Database on Cassandra - Initially a rewrite of the original OpenTSDB project.

Table 8 - TSDB List

Existing TSDB benchmarking tools:

	TSBS by TimescaleDB	**IoTDB-Benchmark**	**TSDBBench**	**ts-benchmark**
Language	GO	Java	Based on YCSB-TS (Java) with python modules	Java in addition to a python module for data generation
Description	based on a fork of work initially made public by InfluxDB benchmark bulk load performance and query execution performance	• Developed in 2019 Tsinghua University Beijing, China • support of out-of-order • measurement of system resources	Developed in 2016 benchmarking framework to measure the performance of TSDB based on YCSB-TS - a fork of YCSB (Yahoo) - Designed for NoSQL DB benchmarking Complex	TSDB benchmark developed at the Renmin University of China in December 2018
Support concurrent queries	No	Yes	No	Yes
Supported databases	IoT use case implemented for Timescale and InfluxDB Devops use case implemented for: • Akumuli • Cassandra • ClickHouse • CrateDB • InfluxDB • MongoDB • SiriDB • TimescaleDB • Timestream • VictoriaMetrics	• IoTDB • InfluxDB • KairosDB • TimescaleDB • OpenTSDB	• OpenTSDB • Druid • InfluxDB	• InfluxDB • IotDB • TimescaleDB • Druid • OpenTSDB

Table 9 - TSDB benchmarking tools

References

[1] db-engines.com, DBMS popularity broken down by database model , Trend of the last 24 months. Retrieved from , https://db-engines.com/en/ranking_categories

[2] Joseph A.Formaggio et al. (2021). Direct measurements of neutrino mass. Retrieved from https://www.sciencedirect.com/science/article/abs/pii/S0370157321000636?via%3Dihub

[3] M. Aker et al. (2021). The Design, Construction, and Commissioning of the KATRIN Experiment. Retrieved from: https://arxiv.org/pdf/2103.04755.pdf

[4] MySQL, Why MySQL? Retrieved from https://www.mysql.com/why-mysql/?main=1&topic=46&type=5&lang=en

[5] Matei-Eugen Vasile et al. (2020). Evaluating InfluxDB and ClickHouse database technologies for improvements of the ATLAS operational monitoring data archiving. Retrieved from https://www.researchgate.net/publication/342775077_Evaluating_InfluxDB_and_ClickHouse_database_technologies_for_improvements_of_the_ATLAS_operational_monitoring_data_archiving

[6] Alexey Struckov et al. (2019). Evaluation of modern tools and techniques for storing time-series data. Retrieved from: https://www.sciencedirect.com/science/article/pii/S1877050919310439/pdf?md5=ecc390cfc9d0be432ca0c218985c94d5&pid=1-s2.0-S1877050919310439-main.pdf

[7] Rui Liu, Jun Yuan (2019). Benchmarking Time Series Databases with IoTDB-Benchmark for IoT Scenarios. Retrieved from: https://arxiv.org/abs/1901.08304

[8] Grolinger, K et al. (2013). Data management in cloud environments: NoSQL and NewSQL data stores. Retrieved from: https://journalofcloudcomputing.springeropen.com/articles/10.1186/2192-113X-2-22

[9] Gayathri Ayyappan et al. (2021). TimescaleDB 2.3: Improving columnar compression for time-series on PostgreSQL. Retrieved from: https://blog.timescale.com/blog/timescaledb-2-3-improving-columnar-compression-for-time-series-on-postgresql

[10] Mike Freedman (2017). Time-series data: Why (and how) to use a relational database instead of NoSQL. Retrieved from: https://blog.timescale.com/blog/time-series-data-why-and-how-to-use-a-relational-database-instead-of-nosql-d0cd6975e87c/

[11] Ajay Kulkarni and Mike Freedman (2020). TimescaleDB 2.0: A multi-node, petabyte-scale, completely free relational database for time-series. Retrieved from: https://blog.timescale.com/blog/timescaledb-2-0-a-multi-node-petabyte-scale-completely-free-relational-database-for-time-series/

[12] Yandex LLC, ClickHouse.tech. ClickHouse Adopters. Retrieved from: https://clickhouse.tech/docs/en/introduction/adopters/

[13] Influxdata, influxdata.com. Customers. Retrieved from: https://www.influxdata.com/customers

[14] Timescale, timescale.com. Success Stories. Retrieved from: https://www.timescale.com/success-stories

[15] Influxdata, Influxdata.com. InfluxDB storage engine. Retrieved from: https://docs.influxdata.com/influxdb/v2.0/reference/internals/storage-engine/

[16] Yandex LLC, clickhouse.tech. Overview of ClickHouse Architecture. Retrieved from: https://clickhouse.tech/docs/en/development/architecture

[17] Yandex LLC, clickhouse.tech. Custom Partitioning Key. Retrieved from: https://clickhouse.tech/docs/en/engines/table-engines/mergetree-family/custom-partitioning-key/

[18] Steven Yeh et al. Simple Wrapper around Npgsql for using PostgreSQL COPY functions. Retrieved from: https://github.com/PostgreSQLCopyHelper/PostgreSQLCopyHelper

[19] Andrey Zakharov et al. Yandex ClickHouse fully managed .NET client. Retrieved from: https://github.com/killwort/ClickHouse-Net

[20] Jakub Bednar et al. InfluxDB 2.0 C# Client. Retrieved from: https://github.com/influxdata/influxdb-client-csharp

[21] Influxdata, Influxdata.com. Line protocol, Special Characters. Retrieved from: https://docs.influxdata.com/influxdb/v2.0/reference/syntax/line-protocol/#special-characters

[22] Influxdata, Influxdata.com. Optimize writes to InfluxDB, Batch writes. Retrieved from: https://docs.influxdata.com/influxdb/v2.0/write-data/best-practices/optimize-writes/#:~:text=InfluxDB%20scrapers-,Batch%20writes,5000%20lines%20of%20line%20protocol.