ECOLE POLYTECHNIQUE DE BRUXELLES

INFO-H-415: Advanced Databases

Time series databases with TimeScaleDB and InfluxDB

Kristóf Balázs

Stefanos Kypritidis

Nishant Sushmakar

Olha Baliasina

Prof. Esteban Zimányi

2024

Contents

Introduction

In the era of rapid technological advancement, time series data has emerged as a critical component across a range of applications, including financial forecasting, climate studies, healthcare analytics, and more. A time series is a sequence of data points recorded over time, where the order of observations is critical for analysis. Unlike static datasets, time series capture temporal dependencies, revealing trends, cycles, and seasonal patterns. Key features include temporal dependence, data frequency (regular or irregular), and challenges like noise or outliers. Traditional databases often struggle with these complexities, but time series databases (TSDBs) address them through time-based partitioning, efficient compression, and specialized indexing, optimizing storage and analysis for time-oriented data.

Within this context, two tools, TimescaleDB and InfluxDB, stand out as prominent time series database solutions. TimescaleDB, built on PostgreSQL, combines relational database robustness with time series-specific optimizations. It is widely adopted in applications requiring seamless SQL integration alongside temporal data processing. On the other hand, InfluxDB, written in Go, offers a purpose-built architecture tailored to high-throughput ingestion and real-time analytics, making it a popular choice for DevOps monitoring and IoT systems. These tools represent distinct approaches to addressing the challenges posed by the growing scale and complexity of time-series data.

This study seeks to evaluate the comparative performance of TimescaleDB and InfluxDB. Moreover, it uses the TSBS benchmark on its self-generated data, and a custom benchmark on a real-world medical sensor dataset to achieve that. In detail, the sensor dataset includes various physiological metrics such as heart rate, accelerometer data, glucose levels, and skin temperature. By assessing their handling of diverse data types, query efficiencies, and scalability, the findings aim to provide actionable insights into the suitability of each tool.

1

Chapter 1: Time series databases

1.1 Time series

1.1.1 Definition

A time series is a sequence of observations collected or recorded at successive points in time, often at consistent intervals. Unlike a static dataset, where rows can be rearranged without changing the meaning, a time series carries an inherent temporal order, making the timing of each data point critical to its interpretation. Historical financial prices, daily rainfall amounts, server load metrics sampled every minute, or human heart rate readings taken every second are all examples of time series data. The primary focus in working with time series is understanding how values evolve over time, rather than just examining relationships between variables at a single moment. The key characteristics of time series data are the following:

- **Temporal dependence:** The value of each data point often depends on preceding values. Patterns such as trends, cycles, and seasonal variations emerge precisely because past behavior influences future observations. For example, energy consumption may rise during the evening due to predictable human activity patterns, and temperature readings may follow a daily cycle.

- **Data frequency and regularity:** Time series can be evenly spaced (e.g., daily sales data) or irregular (e.g., event logs generated sporadically). Regular, fixed-frequency data simplifies certain analytical techniques, while irregular data often requires interpolation or resampling methods to align observations before analysis.

- **Complexity of noise and outliers:** Real-world time series often contain noise—fluctuations that do not follow a simple pattern. They may also feature sudden spikes or drops that deviate from historical behavior, known as outliers. Identifying and handling such anomalies can be crucial for reliable forecasting and decision-making.

- **Dimensionality and multivariate structures:** Although the simplest time series tracks a single variable over time, many real applications generate multivariate time series, recording multiple related measurements simultaneously. For instance, a manufacturing line might produce synchronized temperature, pressure, and vibration readings. Understanding how these variables interact temporally can provide richer insights than analyzing them in isolation.

1.1.2 Analytical objectives

- **Trend Analysis:** Analysts seek to detect whether values tend to increase, decrease, or remain stable over long periods. Identifying a slowly rising trend in product demand can guide capacity planning and inventory management.

- **Seasonality Detection:** Many processes exhibit predictable, recurring patterns. Retail sales may surge during holidays, or web traffic may peak in the afternoon. Recognizing these periodicities supports more accurate forecasting and resource allocation.

- **Forecasting Future Values:** One of the most prominent uses of time series data is making predictions about future behavior. Forecasting methods — ranging from simple moving averages to sophisticated machine learning techniques — help organizations anticipate outcomes such as future stock prices, weather conditions, or equipment failure times.

- **Event Detection and Diagnostics:** Sudden deviations from historical patterns, such as a spike in network latency or an unexpected dip in production yield, may signal an underlying issue. Analyzing time series data allows for early detection of anomalies and proactive intervention.

1.2 Time Series Databases

Time-series databases (TSDBs) are specialized systems designed to manage data sequences where each entry is associated with a specific timestamp. Unlike general-purpose relational or NoSQL databases, TSDBs are optimized to handle the unique demands of temporally ordered data. Traditional databases often struggle with the challenges posed by time series workloads, such as the need for efficient storage, rapid ingestion, and targeted retrieval within specific time frames.

TSDBs address these challenges by tailoring their architecture to the characteristics of time-dependent data. They automatically partition data into time-based segments, which enhances both write performance — by appending new records naturally in chronological order — and read efficiency, as queries frequently target defined time frames. This specialized approach to storage, indexing, and querying enables TSDBs to deliver superior performance and intuitive data management for time-centric workloads. As a result, they have become an essential tool for applications that rely on large-scale time series data, such as monitoring systems, analytics, and predictive modelling.

1.2.1 Key Characteristics and Architectural Principles

Time series databases center their designs on one critical aspect: the notion that data points are fundamentally organized by time. This temporal dimension underpins most of the architectural and performance optimizations found in TSDBs. Instead of structuring records into normalized tables with arbitrary key relationships, a TSDB emphasizes continuous, append-only writes that align with the chronological flow of incoming data.

A cornerstone of many time series databases is their ability to partition data automatically into time-based chunks. For example, entries might be grouped by hour, day, or month. These time-partitioned segments serve multiple purposes. First, they align with how data is typically queried. Users often seek data from a particular time window—say, all sensor readings for a given day. With temporal partitioning, the database can quickly locate and return only the relevant segments, improving both speed and efficiency. Second, these partitions provide a natural mechanism for data lifecycle management. Aging data can be moved to cheaper storage tiers or even dropped altogether when it no longer holds operational value.

Another defining characteristic of TSDBs is their emphasis on compression and encoding. Time series data, particularly from sensors or machine logs, can generate massive volumes at high velocity. To mitigate storage costs, TSDBs commonly deploy specialized compression algorithms that leverage the predictable and often slowly changing nature of time series values. Columnar storage formats and delta encoding—storing changes in values rather than absolute values—enable high compression ratios without drastically compromising read performance. Efficient compression not only saves disk space but can also speed up queries by reducing the amount of data read from storage.

Furthermore, indexing strategies in time series databases are often optimized for temporal queries rather than complex joins or arbitrary filter conditions. Indexes focus on quickly finding data by time interval and, in some cases, by a limited set of tags or dimensions. While TSDBs typically do not aim to replicate the full flexibility of relational databases in terms of arbitrary relational joins or complex transformations, their specialized indexing allows them to excel at the tasks most crucial to time series workloads.

1.2.2 Advantages of Using Time Series Databases

By adopting a time series database, organizations gain a number of concrete benefits, notably in terms of performance, scalability, and data management convenience.

Performance Tailored for Temporal Data

Traditional relational databases can store time series data, but their generic indexing mechanisms and table structures are not inherently optimized for chronological reads and writes. In contrast, TSDBs are built to handle sequential writes efficiently. As new data points arrive—often in chronological order—the database appends them directly to the appropriate time partition. This reduces the overhead of constant random I/O

operations and can minimize write amplification. The result is that ingesting millions of measurements per second becomes not just feasible, but routine for mature time series solutions.

On the querying side, the ability to quickly access a particular time slice improves latency. Instead of scanning broadly through general-purpose indexes, a TSDB can leverage time-based partitioning and compact data encoding to rapidly fetch relevant segments. Frequent queries like “return the last ten minutes of sensor data” or “fetch daily aggregates for the past month” run much faster than they might in a system that must inspect large, unpartitioned datasets.

Scalability and Retention Policies

Time series data often grows without bound. Sensor networks, server logs, financial tick data—all continuously produce new entries. As a result, systems must scale seamlessly to accommodate long histories and high data rates. Many TSDBs are designed for distributed architectures, allowing them to scale horizontally by adding more nodes. By distributing time partitions across multiple machines, a TSDB can increase write throughput and storage capacity linearly with hardware. This approach accommodates billions of data points without a steep decline in performance.

In addition, time series databases typically incorporate data retention policies at their core. Administrators can define how long data should remain in the system and at what “age” data can be compressed more aggressively or offloaded to cold storage. Automating these lifecycle management tasks relieves the operational burden of manually pruning old rows, ensuring that the database remains focused on serving current analytic needs without becoming weighed down by irrelevant historical data.

Simplified Analytics and Integrations

A time series database naturally aligns with many common analytical scenarios, making it easier to extract insights. Because the data is already structured around time, computing aggregates, detecting trends, or applying time-based transformations is more straightforward. Many TSDBs provide built-in functions for downsampling (reducing high-frequency data to coarser granularity), calculating moving averages, or identifying anomalies over specific intervals. This shortens the path from raw data ingestion to actionable knowledge.

Furthermore, TSDBs often integrate well with visualization and monitoring tools. Dashboards such as Grafana or Kibana are commonly used in conjunction with time series databases to produce real-time charts and alerts. These integrations allow non-technical stakeholders—such as operations managers or product owners—to easily understand temporal dynamics without constantly relying on engineering teams to run complex SQL queries.

1.2.3 Modern state of Time Series Databases

Over the past decade, time series databases (TSDBs) have evolved significantly, moving from niche solutions to widely recognized tools in data management architectures. Their growing prominence is tied to the rise of IoT devices, real-time analytics, DevOps monitoring, and financial market data processing. As organizations increasingly rely on continuous and high-volume data streams, TSDBs have become indispensable for handling the ingest, storage, and analysis of large-scale, time-stamped datasets.

Modern TSDBs address a broad range of use cases by offering features such as integrated compression, efficient indexing for time-based queries, flexible data models, and seamless scalability. These databases are now routinely benchmarked and compared against other database management systems, illustrating both their growing popularity and the competitive landscape among leading time series solutions.

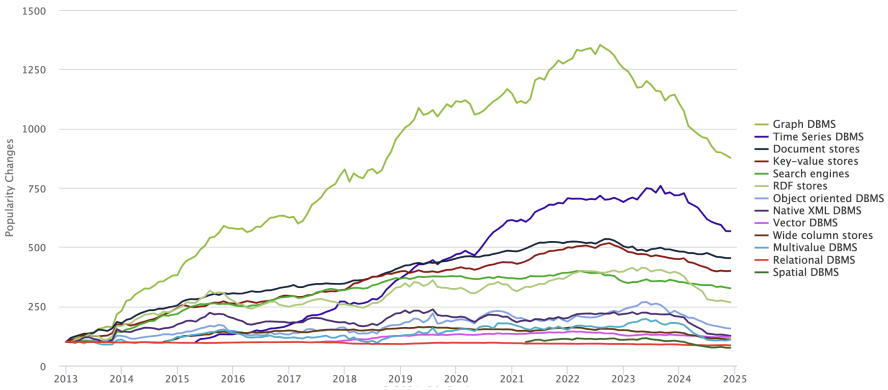


Figure 1.1: DBMS popularity (since 2013)

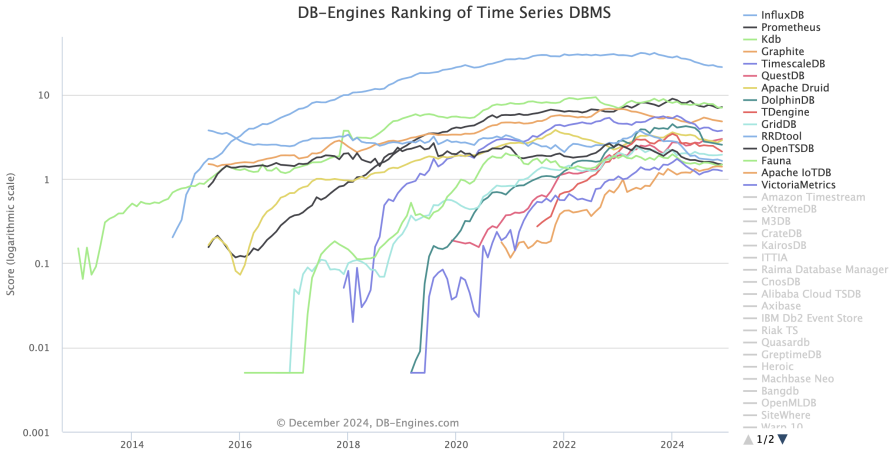


Figure 1.2: Time series DBMS ranking (since 2013)

☐ include secondary database models — 44 systems in ranking, December 2024

Rank Dec 2024	Rank Nov 2024	Rank Dec 2023	DBMS	Database Model	Score Dec 2024	Score Nov 2024	Score Dec 2023
1.	1.	1.	InfluxDB	Time Series, Multi-model	21.23	-0.24	-6.88
2.	↑ 3.	2.	Prometheus	Time Series	7.12	+0.20	-1.21
3.	↓ 2.	3.	Kdb	Multi-model	6.99	-0.07	-1.29
4.	4.	4.	Graphite	Time Series	4.81	-0.10	-0.65
5.	5.	5.	TimescaleDB	Time Series, Multi-model	3.74	+0.07	-1.55
6.	6.	↑ 10.	QuestDB	Time Series, Multi-model	2.98	+0.07	+0.67
7.	7.	7.	Apache Druid	Multi-model	2.88	+0.17	-0.45
8.	8.	↓ 6.	DolphinDB	Multi-model	2.54	-0.05	-1.37
9.	9.	↓ 8.	TDengine	Time Series, Multi-model	2.11	-0.16	-1.09
10.	10.	↑ 11.	GridDB	Time Series, Multi-model	1.94	+0.02	-0.30
11.	11.	↓ 9.	RRDtool	Time Series	1.64	-0.07	-1.18
12.	12.	12.	OpenTSDB	Time Series	1.48	-0.03	-0.57
13.	13.	13.	Fauna	Multi-model	1.48	+0.04	-0.28
14.	14.	↑ 15.	Apache IoTDB	Time Series	1.40	-0.01	+0.08
15.	15.	↓ 14.	VictoriaMetrics	Time Series	1.24	-0.04	-0.22
16.	16.	↑ 17.	Amazon Timestream	Time Series	1.23	+0.01	+0.04
17.	17.	↑ 20.	eXtremeDB	Multi-model	0.99	+0.05	+0.04
18.	18.	↓ 16.	M3DB	Time Series	0.99	+0.05	-0.20
19.	19.	↓ 18.	CrateDB	Multi-model	0.66	+0.01	-0.35
20.	20.	↓ 19.	KairosDB	Time Series	0.62	+0.05	-0.34
21.	↑ 22.	↑ 22.	ITTIA	Time Series, Multi-model	0.42	+0.04	-0.26
22.	↓ 21.	↑ 24.	Raima Database Manager	Multi-model	0.37	-0.06	-0.21
23.	23.	23.	CnosDB	Time Series	0.32	+0.03	-0.30
24.	24.	↑ 37.	Alibaba Cloud TSDB	Time Series	0.27	+0.03	+0.15
25.	25.	25.	Axibase	Time Series	0.22	+0.02	-0.15
26.	26.	↑ 31.	IBM Db2 Event Store	Multi-model	0.19	+0.01	-0.02
27.	27.	↑ 28.	Riak TS	Time Series	0.17	+0.01	-0.14
28.	28.	↑ 29.	Quasardb	Time Series	0.14	+0.01	-0.15
29.	29.	↑ 33.	GreptimeDB	Time Series	0.13	+0.02	-0.04
30.	↑ 38.	↓ 21.	Heroic	Time Series	0.11	+0.09	-0.72
31.	↑ 33.	↓ 26.	Machbase Neo	Time Series	0.09	+0.03	-0.26
32.	↓ 31.	↓ 27.	Bangdb	Multi-model	0.08	+0.02	-0.25
33.	↓ 30.		OpenMLDB	Time Series, Multi-model	0.08	+0.00	
34.	↓ 32.	↑ 35.	SiteWhere	Time Series	0.06	0.00	-0.09
35.	↓ 34.	↑ 39.	Warp 10	Time Series	0.06	0.00	-0.04
36.	↓ 35.	36.	Blueflood	Time Series	0.06	0.00	-0.08
37.	↓ 36.	↓ 32.	Tigris	Multi-model	0.05	0.00	-0.14
38.	↑ 41.	↑ 40.	SiriDB	Time Series	0.04	+0.04	-0.03
39.	↓ 37.	↓ 30.	ArcadeDB	Multi-model	0.04	+0.00	-0.17
40.	↓ 39.		ReductStore	Time Series	0.02	+0.02	
41.	41.	↑ 42.	Newts	Time Series	0.02	+0.02	+0.02
42.	↓ 41.	↓ 34.	openGemini	Time Series	0.02	+0.02	-0.14
43.	↓ 40.	↓ 38.	Hawkular Metrics	Time Series	0.01	+0.01	-0.10
44.	↓ 41.	↓ 41.	NSDb	Time Series	0.00	±0.00	-0.04

Figure 1.3: Time series DBMS ranking (December 2024)

The comparison of overall DBMS popularity with specific time series DBMS rankings highlights the increasing adoption of dedicated time series platforms. While general-purpose databases still dominate many operational and analytical workloads, specialized TSDBs stand out for their tailored performance optimizations and ecosystem integrations. As new entrants appear, and established players refine their capabilities, the TSDB market continues to advance, driving innovation in data management paradigms centered on time-oriented data.

2

Chapter 2: Database management systems

2.1 TimescaleDB

TimescaleDB is an open-source, relational time series database built atop the proven foundations of PostgreSQL. Its design goal is to combine the reliability, compatibility, and familiar tooling of PostgreSQL with specialized functionalities engineered for the storage, management, and analysis of large-scale time series datasets. By seamlessly integrating time-series optimizations into a relational framework, TimescaleDB provides a scalable and high-performance environment that allows organizations to handle continuous data streams without sacrificing the advantages of a mature SQL ecosystem. Majority of this chapter is based on timescale docs [Tim24b]. In case not, the corresponding citation will be provided.

TimescaleDB was officially launched on April 4, 2017. That same year, PostgreSQL emerged as the fastest-growing database globally, while time-series databases remained the fastest-growing category. Just a few months after its release, TimescaleDB saw successful adoption by users deploying it into production environments and developing new applications around the database. Applications built on TimescaleDB span various industries, including complex monitoring systems, industrial machine data analysis, geospatial asset tracking, operational data warehousing, and financial risk management. Currently, TimescaleDB has earned over 18,000 stars on its GitHub repository [Sta18].

TimescaleDB can be run either on Timescale Cloud or using one's own PostgreSQL installation. Timescale Cloud is a high-performance cloud platform tailored for developers, offering PostgreSQL services enhanced with fast and efficient vector search capabilities. On the other hand, self-hosting can be accomplished using Docker or Kubernetes, by rebuilding the GitHub source code, or by directly installing it on Linux, Windows, or macOS, providing a plethora of options.

2.1.1 Key Features and Advantages

- **Built on PostgreSQL:** TimescaleDB inherits PostgreSQL's robust ACID guarantees, mature ecosystem, and stable performance characteristics. This foundation ensures developers can leverage existing PostgreSQL tools, extensions, and knowledge, reducing the learning curve and simplifying integration

with current infrastructures.

- **Hypertables:** At the core of TimescaleDB's architecture are hypertables, logical abstractions that partition large time-series datasets into smaller, more manageable chunks, typically based on time intervals. These partitions remain transparent to end-users, who interact with hypertables much like regular PostgreSQL tables. This approach optimizes insertion rates, speeds up queries targeted at specific time ranges, and improves storage efficiency.

- **SQL Compatibility:** Because TimescaleDB embraces standard SQL, users can write queries using familiar constructs without adopting proprietary query languages. This compatibility streamlines development workflows, eases onboarding, and encourages adoption in organizations already conversant with SQL.

- **Scalability and Performance:** TimescaleDB's architecture supports efficient handling of high write loads common in time-series workloads. Automated partitioning, parallelization of queries, and advanced indexing strategies enable real-time analytics over large datasets. While TimescaleDB's initial scaling focus is often vertical (enhancing resources on a single node), horizontal scaling is also possible using multi-node deployments, ensuring continuous performance as data volumes grow.

- **Stability and Core Focus:** TimescaleDB prioritizes a stable, maintainable platform over introducing every possible feature. Its development centers on essential functionalities like hypertables, continuous aggregates, and compression rather than spreading effort thinly. This stability-centric philosophy results in a reliable platform with fewer maintenance headaches.

- **Automated Data Management:** Features like data retention policies and continuous aggregates reduce manual intervention. Data retention policies let users automatically expire older data, controlling storage costs and dataset size. Continuous aggregates incrementally update precomputed summaries as new data arrives, offloading the computational burden of generating aggregates on-demand and speeding up analytics.

2.1.2 TimescaleDB Architecture

Figure 2.1 illustrates an overall architecture example of PostgreSQL with TimescaleDB. When data is generated from various sources, TimescaleDB continuously aggregates this data for further processing. Once aggregated, the data is categorized by its respective stream and recorded into a specialized table called a Hypertable. A Hypertable is divided into multiple chunks, each containing a subset of the data from a particular stream and corresponding to a specific time range. If no chunk exists for a given time range, TimescaleDB automatically creates a new chunk to store the incoming data, ensuring that both temporal and stream-specific data are organized for analysis. This approach preserves the traditional RDBMS interface while supporting advanced time-series data handling.

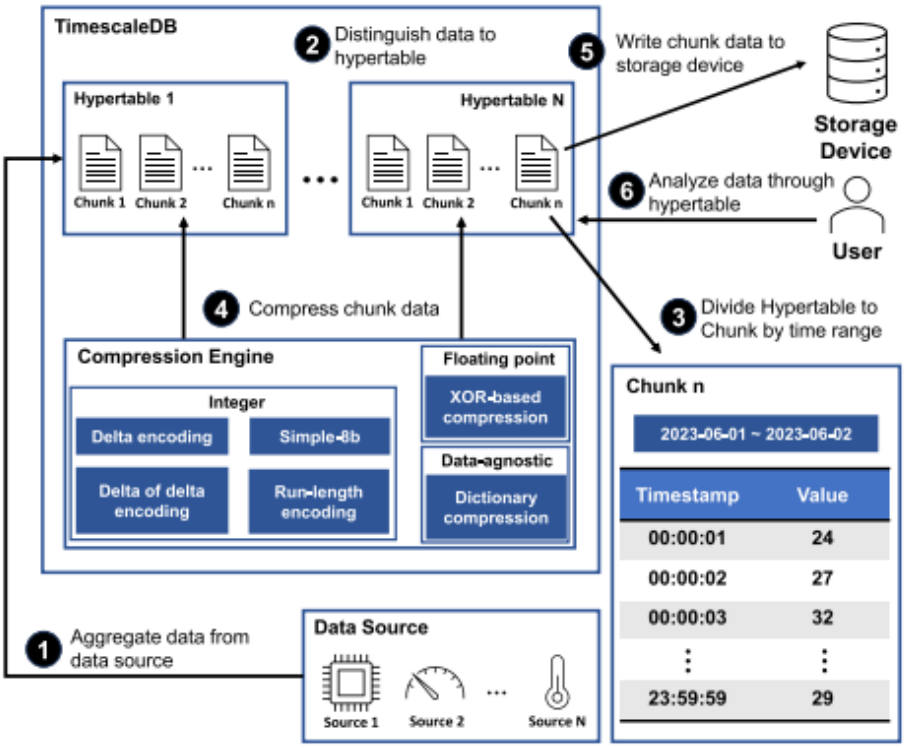


Figure 2.1: TimescaleDB architecture (reprinted from [LSK23])

As shown in the figure, chunk N is designed to store data within the time range of 2023-06-01 to 2023-06-02. Any data beyond this range, such as from 2023-06-03, is stored in a separate chunk. Once the chunks are created, TimescaleDB can apply compression to reduce storage overhead, if compression is enabled. This compression process further optimizes the management of large volumes of time-series data, ensuring that the system remains efficient even as data scales. By combining these features, TimescaleDB enables the effective analysis and management of time-series data while ensuring high performance and low storage costs [LSK23].

2.1.3 Hypertables in TimescaleDB

TimescaleDB introduces the concept of the previously mentioned hypertables, which function as logical abstractions over many underlying tables, collectively storing the time-series data. Although hypertables appear and behave like standard PostgreSQL tables, they are transparently composed of numerous smaller tables, referred to as chunks. These chunks are created by partitioning data along one or two dimensions: by time interval and, optionally, by another partition key such as a device identifier, geographic location, or user identifier.

All typical database operations—creating and altering tables, inserting rows, creating indexes, and selecting data—are performed on hypertables. From the perspective of queries and management tasks, TimescaleDB thus maintains the familiar look and feel of PostgreSQL. Users can leverage standard SQL commands, rely on existing PostgreSQL ecosystem tools, and integrate seamlessly with established workflows, while the

database automatically handles the complexity of partitioning, storage optimization, and indexing behind the scenes.

By employing hypertables for storing time-series data, TimescaleDB improves insertion and query performance. This is achieved by automatically segmenting data into manageable intervals aligned with the dataset's temporal characteristics. Meanwhile, non-time-series data can remain in regular PostgreSQL tables, preserving the flexibility and power of relational modeling. In this way, TimescaleDB offers a unified and intuitive approach to managing time-series information without abandoning the benefits of a trusted relational database environment.

2.1.4 Continuous aggregates

Time-series data grows rapidly, and as the volume of data increases, traditional aggregation methods can become inefficient and slow. For instance, if data is being collected frequently, such as temperature readings every second, calculating summaries like average temperature by hour can be a resource-intensive process. A key timescaleDB feature is continuous aggregates which address this challenge by automating the aggregation process. They are a type of hypertable that is continuously refreshed in the background, incrementally updating as new data is added or old data is modified, which ensures that the aggregation is always up-to-date without requiring manual intervention.

Continuous aggregates offer significant performance benefits compared to regular materialized views, as they do not need to be rebuilt from scratch with each update. Instead, only the changes to the dataset are applied, reducing maintenance overhead. These aggregates can be queried like regular tables, enabling efficient data access while also supporting advanced features like compression and tiered storage. Furthermore, continuous aggregates can be built on top of other continuous aggregates, providing flexibility for complex data analysis. By combining pre-aggregated data with recent, unaggregated raw data, continuous aggregates ensure real-time data is available for querying, offering both high performance and up-to-date results. Those continuous aggregates are called real-time aggregates.

Continuous aggregates in TimescaleDB offer significant performance benefits by pre-computing aggregates, allowing for much faster query execution compared to raw data queries. These aggregates are optimized for time-series data, supporting common time-bucketed queries such as daily, weekly, or monthly summaries. However, they come with limitations, such as modification restrictions—direct updates or deletes are not possible on continuous aggregates. Additionally, there may be data lag, as pre-computed aggregates might not reflect the most recent data unless real-time queries are used to combine them with raw data. Without an automatic refresh policy, continuous aggregates can slow down, as queries will need to perform on-the-fly aggregation of any unmaterialized data, negating the performance gains of pre-computed aggregates and causing slower queries overall.

2.1.5 Compression

Compressing time-series data can significantly reduce storage requirements, achieving reductions in chunk size of over 90%. This not only lowers storage costs but also ensures that data queries are processed rapidly. By grouping multiple records into a single row, compression restructures the data, replacing numerous rows with array-like structures within the columns of these rows. This format minimizes disk space usage and optimizes query performance.

The compression process operates at the chunk level within a hypertable, which is divided based on its primary dimension—typically designated during the hypertable's creation. In time-series contexts, time often serves as the primary dimension, organizing data into chunks. Additional columns, such as temperature, device_id depending on the dataset capture metrics and device-specific details over time. These columns can be utilized as lookup keys, partitioning tools, or filters, depending on the requirements of specific queries. For instance, a device_id column enables quick retrieval of data for individual devices, while a time column can partition and filter data by periods.

Compressed data in a hypertable is stored in a column-order structure, necessitating a schema distinct from the uncompressed version. This arrangement is managed by TimescaleDB, but achieving optimal compression ratios and query performance requires careful consideration of data organization and specifically the setting of `chunk_time_interval`. Notably, the layout of compressed data influences the efficiency of filters and queries. Strategies like ordering and segmenting data play a critical role in maximizing both compression and accessibility. For instance, ordering data by time exploits predictable trends, enabling effective encoding and efficient storage.

Segmenting data according to its access patterns further enhances performance. For example, segmenting by device_id allows faster analytical queries targeting specific devices. Ultimately, the effectiveness of compression and query speed hinges on aligning the data structure with the intended use cases, ensuring streamlined data retrieval and robust performance.

TimescaleDB employs a range of compression algorithms tailored to the data type being compressed. For integers, timestamps, and other integer-like types, it uses a combination of delta encoding, delta-of-delta encoding, simple-8b, and run-length encoding to optimize storage. When columns contain data with minimal repetition, XOR-based compression is applied, often in conjunction with dictionary compression. For all other data types, dictionary compression serves as the default method. In conclusion, timescaleDB automatically selects and applies the optimal compression algorithm, eliminating the need for user intervention or concern .

Delta Encoding

Delta encoding is a method designed to compress data by storing the difference (delta) between sequential data points rather than their absolute values. This approach works well for time-series data, where values

often change gradually. For instance, instead of saving exact timestamps, delta encoding records the interval between events. This significantly reduces storage requirements as the deltas, typically smaller in magnitude, require fewer bits to store. In practice, delta encoding can achieve substantial savings across large datasets, particularly when changes in values are minimal.

Delta-of-Delta Encoding

Delta-of-delta encoding extends the concept of delta encoding by storing the difference between successive deltas, effectively representing the second derivative of the dataset. This method is particularly advantageous for time-series data with consistent intervals, as it compresses repetitive deltas into minimal values, such as zeros. For example, in a time-series dataset with regular five-second intervals, only the initial deltas need storage, while subsequent entries can be reduced to zeros. This method achieves remarkable compression ratios, reducing storage needs for timestamps by factors as high as 64x.

Simple-8b

Simple-8b encoding is a highly efficient technique for compressing small integers by grouping them into fixed-size blocks. Within each block, the minimal bit-length required to represent the largest integer dictates the encoding. This minimizes redundancy as the bit-length is recorded only once per block rather than for each integer. By efficiently packing integers of varying lengths, simple-8b encoding maximizes storage savings while allowing fast decompression. This technique is especially effective when combined with delta encoding, enabling seamless handling of variable-length data within time-series datasets.

Run-Length Encoding

Run-length encoding (RLE) is a classic compression technique that excels in datasets with repeated values. Instead of storing each repeated value individually, RLE encodes the value alongside its frequency of repetition. For instance, a sequence of identical temperature readings can be compressed into compact run; value pairs. This method is highly effective for datasets with prolonged runs of identical values or minimal variation, as seen in time-series applications. RLE is often combined with other methods, such as Simple-8b, to further optimize compression in TimescaleDB.

XOR-Based Compression

For floating-point numbers, XOR-based compression compares consecutive values using the XOR (exclusive or) operation to identify bit-level differences. Only the changes are stored, significantly reducing redundancy in the data. Unlike delta encoding, XOR-based compression retains all significant bits, making it suitable for floating-point data where precision is critical. TimescaleDB leverages this method to ensure efficient compression while enabling rapid decompression, especially for workloads requiring backward scans—a common feature in time-series queries.

Dictionary Compression

Dictionary compression reduces storage requirements by replacing recurring data entries with shorter indices pointing to a dictionary of unique values. For example, instead of storing city names like "New York" or "San Francisco" repeatedly, the dataset stores indices referencing these names in a dictionary. This approach is highly effective for datasets with a limited set of frequently repeated values. In TimescaleDB, dictionary compression integrates with methods like Simple-8b+RLE to further enhance storage efficiency. If the data lacks sufficient repetition, TimescaleDB intelligently avoids dictionary compression to maintain optimal performance.

2.1.6 Data retention

In time-series analyses, data often loses relevance as it ages. If historical data is no longer needed, it can be removed once it surpasses a defined retention period. TimescaleDB offers automated data retention policies to streamline the deletion of outdated data and also allows for manual control by enabling users to selectively remove specific data chunks. Lastly, in timescaleDB data retention works on chunks, not on rows.

In many cases, retaining summaries of historical data is valuable, while the raw data itself may not be necessary. This can be achieved through data downsampling, which integrates retention policies with continuous aggregates to create summarized datasets. In this case, retention policies and refresh policies should be synchronized with an appropriate time offset in order to avoid losing any aggregated data.

2.1.7 Choosing the right chunk size

As described, hypertables are a core feature of TimescaleDB, providing a powerful way to handle time-series data. The chunking process is governed by setting the `chunk_time_interval`, which determines how much data is stored in each chunk. By default, this interval is set to 7 days, ensuring that users can start using hypertables without needing to fine-tune the configuration. However, while the default settings work for general use cases, adjusting the chunk time interval can be important for optimizing performance for specific applications. A well-chosen chunk size ensures efficient data management and query performance, making TimescaleDB more effective for varied workloads.

Properly configuring the chunk size is essential for maximizing the performance of key TimescaleDB features, including compression, continuous aggregates, and data retention policies. These features rely on the chunking mechanism, with compression applied at the chunk level and retention policies operating by deleting entire chunks. Additionally, continuous aggregates inherit the chunk configuration from their parent hypertables, making it crucial to align the chunk size with the needs of these features. Therefore, fine-tuning the chunk time interval can significantly impact the efficiency and effectiveness of TimescaleDB in specific use cases.

TimescaleDB features are interconnected, and optimizing for one may conflict with others due to how `chunk_time_interval` influences their functionality. For example, when prioritizing the best compression ratio, larger chunks that store more rows per device may be ideal. However, if the goal is to drop raw data quickly after materializing it into a continuous aggregate, smaller chunks aligned with the time_bucket interval of the aggregate definition work better. To efficiently utilize server resources, finding a balance in chunk_size that accommodates other priorities while ensuring that active chunks from all hypertables fit within 25% of the PostgreSQL memory allocation is essential [Boo22].

2.1.8 Indexes on hypertables

When a hypertable is created on a regular table, TimescaleDB converts it into a hypertable consisting of chunks. To efficiently query, insert, and manage these chunks based on the time range, TimescaleDB requires an index on the time column.

As part of the creation of hypertable process, TimescaleDB automatically creates a B-tree index on the time column. This index allows TimescaleDB to quickly identify the appropriate chunk to query or update for a given time range.

It is also possible to create unique indexes on the hypertable ensuring uniqueness and speeding up queries. However, the index must include all the partitioning columns defined when creating the hypertable (time column and others if used to partition hypertable). Additional non-partitioning columns can be included, and their order can be customized.

2.1.9 Query Optimization

When a query is executed on a hypertable, the query does not scan the entire table. Instead, it only accesses the chunks necessary to fulfill the request, making the process more efficient. This method is particularly effective when the query's WHERE clause uses the column by which the hypertable is partitioned.

However, many queries involve columns that are not the partitioning column, which can lead to inefficiencies. For instance, a satellite company might have a hypertable with two columns: one for the date the data was gathered and another for the date it was added to the database. If the hypertable is partitioned by the gathering date, a query based on the date the data was added will result in scanning all the chunks, which can significantly reduce performance.

To address this, TimescaleDB provides a feature called chunk skipping which improves query performance by allowing the system to skip chunks based on non-partitioning columns. Chunk skipping can be enabled on a column, and TimescaleDB will track the minimum and maximum values for that column in each chunk. These values are stored in the chunk_column_stats catalog table. When a query includes a WHERE filter with conditions on that column, TimescaleDB uses these stored ranges to dynamically exclude irrelevant chunks, speeding up the query. It's recommended to enable chunk skipping on columns

that are both correlated with the partitioning column (timestamp column) and commonly used in WHERE filter clauses.

2.1.10 Querying in timescaleDB

In TimescaleDB, all standard PostgreSQL queries and functions are fully supported, with additional functionality available for enhanced use. In the following examples, a table of `weather_conditions` is assumed with columns of timestamp_column, temperature and location_name.

Enabling timescaleDB

In order to enable timescaleDB functionality for postgreSQL database, it is only needed to execute the following command. Of course that requires having TimescaleDB extension installed on the PostgreSQL instance.

```
CREATE EXTENSION IF NOT EXISTS timescaledb;
```

Hypertable creation

Converting a regular postgreSQL table into a hypertable can be achieved by using the `create_hypertable()` function of Timescale. As seen below:

```
SELECT create_hypertable('weather_conditions', by_range('timestamp_column', INTERVAL '1 day'));
```

In the previous query, the parameter INTERVAL corresponds to the `chunk_time_interval` discussed previously and can be customized to a user needs. Instead of the keyword `day`, any of the following keywords can be used: `microsecond`, `millisecond`, `second`, `minute`, `hour`, `week`, `month`, `year`.

Timebucket use

The time_bucket function allows grouping data in a hypertable, enabling aggregate calculations over custom time intervals. It is typically combined with the use of GROUP BY.

```
SELECT time_bucket('1 day', timestamp_column) AS bucket,
  avg(temperature) AS avg_temp
FROM weather_conditions
GROUP BY bucket
ORDER BY bucket ASC;
```

The query returns the average temperature per day with with each row representing a single day. In the above query, any of the postgreSQL aggregators can be used, for instance MIN, MAX, SUM or COUNT.

First or last value

The Timescale first and last functions enable the retrieval of the value of one column based on the order defined by another column:

```
SELECT location_name, last(temperature, timestamp_column)
FROM weather_conditions
GROUP BY location_name;
```

The above example calculates the last chronological temperature for each location.

Continuous aggregates

Creating a continuous aggregate involves two steps. First, the view is created, and then a policy is enabled to ensure it stays refreshed. The view can be created either directly on a hypertable or on top of another continuous aggregate. It's possible to have multiple continuous aggregates for each source table or view.

For continuous aggregates, a time_bucket is required on the time partitioning column of the hypertable:

```
CREATE MATERIALIZED VIEW conditions_summary_daily
WITH (timescaledb.continuous) AS
SELECT location_name,
   time_bucket(INTERVAL '1 day', timestamp_column) AS bucket,
   AVG(temperature),
   MAX(temperature),
   MIN(temperature)
FROM weather_conditions
GROUP BY location_name, bucket;
```

Next, an automatic refresh policy is set up which is also the suggested refresh policy by timescale:

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

Start_offset specifies how far back the system should look when refreshing the continuous aggregate and end_offset specifies how far ahead in time the refresh will go. Also, schedule_interval specifies how often the continuous aggregate should be refreshed. In our example, the refresh policy will include data from one month before the current time until one day before the current time and will refresh every hour.

Compression

Ordering the data significantly affects both compression ratios and query performance, with rows that change over a dimension ideally being stored close together. Time-series data, where changes follow predictable trends, is a good candidate for ordering by time to optimize both compression and query efficiency.

Using the following configuration setup on the example table:

```
ALTER TABLE weather_conditions
SET (
        timescaledb.compress,
        timescaledb.compress_orderby='timestamp_column'
);
```

Next, an automatic compression policy is created to compress chunks that are older than seven days. An automatic compression policy is a recurrent compression job that will continuously compress the data as time goes on and is the suggested compression policy by timescale:

```
SELECT add_compression_policy('weather_conditions', INTERVAL '7 days');
```

Retention policy

Setting up automatic retention policy that removes data older than 2 months. This policy is recurring and runs daily:

```
SELECT add_retention_policy('weather_conditions', INTERVAL '2 months');
```

An alternative option is the manual retention:

```
SELECT drop_chunks(
  'weather_conditions',
  older_than => INTERVAL '3 months',
  newer_than => INTERVAL '4 months'
)
```

This query drops data newer than 4 months and older than 3 months. A key detail is that this is a one-time action and will not create a recurring policy.

Index creation

Next, a unique index is created on the hypertable for the columns location_name and timestamp_column. This improves query performance when filtering by timestamp and location, ensures global uniqueness, and prevents duplicate data. An important detail is that unique indexes on hypertables can only be created when compression is not enabled.

```
CREATE UNIQUE INDEX idx_location_time
  ON weather_conditions(location_name, timestamp_column);
```

Chunk skipping

As described in the query optimization subchapter, chunk skipping is a very useful tool for query optimization when filtering by non-partitioning columns. In our example, if many queries are filtered by temperature (for example finding timestamps where temperature is less than 0 degrees), enabling chunk skipping for that column could significantly improve the performance of those queries:

```
SELECT enable_chunk_skipping('weather_conditions', 'temperature');
```

2.1.11 Comparison with Other Time Series Databases

While TimescaleDB shares the same overarching goal as other TSDBs—managing time-stamped data efficiently—its approach and underlying technologies differ:

- **Data Model:** Unlike InfluxDB or other non-relational TSDBs, TimescaleDB is fundamentally relational, leveraging PostgreSQL's structured data model. This choice provides strong consistency, a mature optimizer, and SQL compliance while still optimizing for time-based data.
- **Scalability Approach:** TimescaleDB initially focuses on vertical scalability and can later extend to multi-node deployments. InfluxDB and other similar systems often favor a horizontally distributed architecture from the start. The right approach depends on the user's scaling strategy and infrastructure.
- **Query Language:** TimescaleDB relies on standard SQL, allowing immediate productivity for SQL-savvy teams. In contrast, InfluxDB uses InfluxQL and Flux, which may offer time-series specific functions but require learning a new syntax and retooling existing analytics pipelines.

2.1.12 Factors to Consider When Choosing TimescaleDB

Before adopting TimescaleDB, it is prudent to evaluate its alignment with project objectives and constraints:

- **Data Volume and Velocity:** Determine whether TimescaleDB can comfortably handle both the amount and rate at which your data arrives. Time-series applications with rapid ingestion can benefit from TimescaleDB's hypertable-based optimizations.
- **Query Complexity:** Assess the complexity and frequency of your queries. If advanced filtering, windowing, or complex joins are required, TimescaleDB's SQL capabilities offer powerful querying options out-of-the-box.

- **Integration Requirements:** Consider how well TimescaleDB fits into your existing infrastructure. Leveraging PostgreSQL's ecosystem simplifies integration, allowing you to reuse existing tooling, libraries, and expertise.
- **Deployment Options:** TimescaleDB supports flexible deployment modes, including self-managed installations or fully managed cloud services. Select the model that aligns with your operational preferences, compliance needs, and scaling strategy.

2.2 InfluxDB

InfluxDB is an open-source time series database (TSDB) specifically designed to address the challenges associated with managing time-stamped datasets. Written in Go, it emphasizes high-throughput ingestion, fast queries, and efficient storage, making it particularly well-suited for scenarios involving DevOps monitoring, IoT sensor readings, application metrics, and real-time analytics. By providing a flexible data model, dedicated query languages, and extensive integration options, InfluxDB supports both straightforward and complex analytical tasks on continuously evolving datasets.

2.2.1 Key Features and Advantages

InfluxDB's architecture and feature set target the needs of time series workloads:

- **Purpose-Built for Time Series Data:** Rather than adapting a general-purpose database, InfluxDB is engineered from the ground up for time-stamped data. This specialization in write-optimized storage and querying ensures that time-centric analysis runs efficiently, even at substantial scale.
- **High Write and Query Throughput:** InfluxDB can ingest millions of data points per second, accommodating large volumes of high-frequency data. Its indexing and query processing systems are optimized to deliver low-latency retrieval, ensuring that analytical tasks on rapidly changing datasets remain responsive.
- **Schema-Less Flexibility:** The database employs a tag-based, schema-less model. Measurements, fields, and tags can be added without prior schema definitions. This approach adapts to dynamic data structures and minimizes administrative overhead related to structural changes.
- **Multiple Query Languages:** InfluxDB supports InfluxQL, a SQL-like language for those familiar with relational database queries, as well as Flux, a more expressive functional scripting and query language. Flux enables advanced data transformations, joins, and analytics, thereby addressing both simple and sophisticated use cases.
- **Built-in Data Retention and Lifecycle Management:** Time series data frequently diminishes in relevance as it ages. InfluxDB's retention policies enable automatic expiration or downsampling of older

data, optimizing storage costs and maintaining efficient access to the most pertinent and recent measurements.

- **Scalability and High Availability:** While the open-source InfluxDB instance operates on a single node, enterprise and cloud editions introduce clustering capabilities for horizontal scalability and improved resilience. These features support growth in data volume and concurrency demands without sacrificing query performance.
- **Comprehensive Ecosystem (TICK Stack):** InfluxDB is part of the TICK stack, which includes:
  - **Telegraf:** A lightweight agent for data collection.
  - **Kapacitor:** A real-time data processing and alerting engine.
  - **Chronograf:** A visualization and administrative tool for managing InfluxDB and Kapacitor.

  This tightly integrated environment streamlines the entire lifecycle—from data collection and storage to processing and visualization.
- **Extensive Integrations and Client Libraries:** InfluxDB's wide array of input plugins, client libraries, and APIs simplifies integration with various environments and programming languages. Compatibility with popular visualization solutions, such as Grafana, facilitates the creation of interactive dashboards and continuous monitoring interfaces.

2.2.2 Comparison with Other Time Series Databases

InfluxDB offers several distinguishing factors:

- **Custom, Non-Relational Data Model:** Unlike databases that build on relational foundations, InfluxDB uses a custom columnar model that is optimized for time-series workloads. This approach often leads to higher performance for append-heavy scenarios and facilitates schema-less flexibility.
- **Focus on Horizontal Scalability:** InfluxDB's enterprise and cloud editions emphasize horizontal scaling by adding more nodes, rather than relying solely on vertical scaling. This approach is beneficial when facing rapidly increasing data volumes or elevated concurrency requirements.
- **Specialized Query Languages:** InfluxQL provides a familiar SQL-like syntax, while Flux enables more complex data transformations. This dual-language model differentiates InfluxDB from solutions that rely exclusively on SQL or proprietary languages less suited for time-based operations.

2.2.3 Factors to Consider When Choosing InfluxDB

Several criteria influence the decision to adopt InfluxDB:

- **Scalability Needs:** Applications projecting substantial increases in data volume or ingestion rates may find the enterprise or cloud clustering options essential for maintaining performance as workloads expand.
- **Data Retention and Lifecycle Management:** It is important to assess the relevance of historical data. InfluxDB's retention policies simplify automatic data expiry or downsampling, maintaining a lean and performant dataset.
- **Integration with Existing Systems:** Compatibility with an organization's monitoring, analytics, and visualization ecosystem can streamline implementation. InfluxDB's APIs, client libraries, and numerous plugins facilitate seamless integration.
- **Open Source vs. Enterprise Features:** Deciding between the open-source edition and the enterprise or cloud solutions involves evaluating additional features—such as clustering, security enhancements, and vendor support—against cost and operational complexity.

2.2.4 Data Model and Series Representation

At the heart of InfluxDB's data model are *measurements*, *tags*, and *fields*, each playing a distinct role:

Measurement: Conceptually similar to a table name in relational databases, a measurement defines a logical grouping of time-series data.

Tags: Key-value pairs that provide indexed metadata about the series. Tags are stored as strings and are indexed to enable efficient filtering on high-cardinality attributes such as device IDs, regions, or data sources.

Fields: The actual values (often numeric or string) associated with each time-stamped point. Fields are not indexed by default, but they are compressed and stored for fast sequential reads and scans.

A combination of a measurement and a specific set of tag key-value pairs defines a *series*. Each series is essentially a unique time series that InfluxDB tracks. Internally, this approach allows the database to handle large volumes of measurements efficiently, with tags enabling selective queries based on metadata.

2.2.5 Storage Engine and File Structure

InfluxDB uses a custom-built storage engine tailored for time-series data. Early versions relied on in-memory indexes and various storage formats, but modern InfluxDB (particularly from version 1.x onward) introduced the *Time-Structured Merge Tree (TSM)* engine to improve write and query performance, durability, and resource efficiency.

Key Aspects of the TSM-Based Storage

Shard-Based Partitioning: Data is organized into shards, each corresponding to a specific time interval (e.g., one shard per day, week, or month depending on the configuration). Sharding by time allows InfluxDB to:

- Quickly drop old data by simply removing entire shard files once they pass retention limits.
- Distribute writes and queries across different time segments, reducing file contention and improving concurrency.

Write-Ahead Log (WAL) and Caching: When new data points arrive (via line protocol writes), InfluxDB first writes them to an in-memory cache and a Write-Ahead Log on disk. The WAL ensures durability—if the server shuts down unexpectedly, the WAL can be replayed to recover recently ingested data. Periodically, InfluxDB flushes these writes from the WAL and memory cache into immutable, on-disk TSM files.

TSM Files (Immutable, Compressed Storage): The TSM file format stores data in a columnar, compressed form. Each TSM file contains sorted series data (by time) and associated indexes that map series keys (measurement + tag set) to data blocks on disk. Since these files are immutable once written, no updates occur in place. Instead, new data leads to the creation of new TSM files, and background processes periodically compact and merge these files.

This log-structured, append-only approach reduces write amplification and fragmentation. Compaction merges multiple smaller TSM files into larger ones, removing deleted or expired data and improving read efficiency.

Compression and Encoding: Time-series data often exhibits patterns and correlations between consecutive points. InfluxDB's TSM engine uses compression algorithms (such as run-length encoding, delta-of-delta encoding for timestamps, and Gorilla compression for floating-point values) to reduce storage footprint while maintaining quick decompression for queries.

2.2.6 Indexing and Metadata Management

InfluxDB maintains indexes mapping series identifiers to their location in TSM files. This index is critical for quick lookups of specific series or tag combinations. Internally, indexing involves:

- **Tag-Based Indexing:** Tags are indexed to allow filtering queries such as "find all series for device=X in region=Y." The index reduces the search space before reading data blocks from disk.
- **In-Memory Index Structures:** Many components of the index are kept in memory to speed up series lookups. As the number of series grows, so does the memory footprint. High-cardinality tag sets can challenge scalability, as each unique combination of tags defines a distinct series in the index.

Because indexes are central to performance, InfluxDB continuously refines indexing strategies. For large deployments, enterprise and cloud versions add features like distributed indexing to handle very high cardinalities.

2.2.7 Query Execution

When executing a query, InfluxDB follows these steps internally:

Tag and Time Filters: The query engine first uses the tag index and time range specified in the query to identify which shards and series might contain relevant data. Because data is time-sharded and series are indexed by tags, this filtering step reduces the amount of data that must be scanned.

TSM File Scans: Once the candidate series are identified, InfluxDB locates the corresponding blocks of data in the TSM files. Since TSM files store data chronologically and use compression schemes optimized for sequential scans, reading the requested data is efficient. The engine decompresses the data blocks as needed and applies any aggregations (e.g., SUM, MEAN) or transformations (e.g., GROUP BY intervals) required by the query.

Aggregations and Downsampling: For queries involving aggregated metrics over time, the engine can leverage downsampling or continuous queries if configured. Downsampled data is precomputed and stored at a coarser resolution, allowing faster responses to queries that do not require full-resolution historical data.

2.2.8 Lifecycle Management and Retention Policies

Time series often has diminishing value as it ages. InfluxDB integrates data retention and lifecycle management directly:

Retention Policies (RPs): Define how long data stays in the database before being automatically expired. When data ages beyond its retention period, InfluxDB can simply drop the corresponding shard files, instantly freeing storage and removing stale points without expensive table scans or manual cleanup steps.

Continuous Queries and Downsampling: InfluxDB can automatically run continuous queries to create aggregated time-series at longer intervals (for example, storing hourly averages instead of raw second-by-second values after a certain period). This approach reduces storage usage for long-term historical data while retaining the essential trends.

2.2.9 Horizontal Scalability and Clustering

While the open-source edition of InfluxDB is primarily single-node, enterprise and cloud offerings introduce clustering. In this architecture:

- **Data Sharding Across Nodes:** Series are distributed across multiple nodes based on a hashing scheme. Queries are routed to the relevant nodes holding the required series data.
- **High Availability (HA):** Replication ensures that each series is stored on multiple nodes. If one node fails, another copy remains accessible.
- **Load Balancing:** Writes and queries can be spread across multiple machines, increasing throughput and capacity linearly with added nodes.

2.3 Visualization instrument: Grafana

Grafana is an open-source visualization and analytics platform. It is used in most cases for monitoring of time-series data, but it is also used with logging tools like Loki and Elasticsearch, NoSQL/SQL databases like Postgres, and CI/CD tooling like GitHub. It supports many data sources, which includes both TimescaleDB and InfluxDB [Gra24]. It is mentioned in this report as it is often used with the time-series databases of our choice.

Grafana integrates with InfluxDB through a built-in data source plugin. It also supports both InfluxDB query languages (InfluxQL or Flux). For TimescaleDB, Grafana has a PostgreSQL data source plugin, which can be used together with TimescaleDB. Integration can also be done using docker-compose [Inf24a].

Grafana has an alerting system (Grafana Alerting) that monitors query results and it can trigger alerts based on thresholds, trends, or patterns in time-series data that we specify. Other features that make Grafana suited for time-series data are downsampling, aggregation, and continuous queries. As an example, we can use the `time_bucket()` function for TimescaleDB and the `GROUP BY time()` function for InfluxDB to group data into fixed intervals for visualization. In addition, Grafana also supports streaming data visualization and the visualization of geospatial data stored in Timescale [Tim24c].

3 Tool comparison

3.1 Time Series Benchmark Suite (TSBS)

3.1.1 Hardware specifications

In this subsection, we display the hardware specifications of the machine used to benchmark both tools. Note that this machine is used for performing both benchmarks to ensure accurate results.

Processor	Intel® Core™ i5-13500H, 12C (4P + 8E) / 16T, P-core 2.6, 4.7GHz, E-core 1.9 / 3.5GHz, 18MB
RAM	32.0 GB LPDDR5-5200
GPU	NVIDIA GeForce RTX 3050 6GB GDDR6 Laptop GPU
Storage	1TB SSD M.2 2242 PCIe 4.0x4 NVMe

Table 3.1: Hardware Specifications

3.1.2 Overview

The Time Series Benchmark Suite (TSBS) offers a toolkit for benchmarking the performance of time-series databases. It was written in Go and it supports a variety of databases such as TimescaleDB, InfluxDB, Cassandra, QuestDB, and MongoDB, among others. TSBS targets two main performance metrics: **bulk data loading** and **query execution performance**. It can compare databases in terms of suitability for specific use cases. It offers three options: `cpu-only`, `devops`, or `iot`. As our database requirements fit the `iot` use case the most, we will only focus on that.

The IoT use case is modeled around a fleet management system, and it simulates data streams from trucks in a fictional logistics company. It has realistic data challenges, such as out-of-order events, batch ingestion from offline devices, and missing entries. Metrics such as fuel levels, geolocation, load capacity, and operational status are used to mimic the diagnostic and telemetry data typically used in IoT systems [Tim24a]. The data is produced deterministically using a pseudo-random number generator (PRNG). In Table 3.1, a sample of the generated data is visible.

ID	Name	Fleet	Driver	Model	Version	Load Cap.	Fuel Cap.	Fuel Cons.
1	truck_0	South	Trish	H-2	v2.3	1500	150	12
2	truck_2	North	Derek	F-150	v1.5	2000	200	15
3	truck_3	East	Albert	F-150	v2.0	2000	200	15
4	truck_6	East	Trish	G-2000	v1.0	5000	300	19
5	truck_10	North	Albert	F-150	v2.3	2000	200	15

(a) Tags Table

Time	Tags ID	Fuel State	Load	Status	Add. Tags
2016-01-01 01:00	19035	1	0	0	
2016-01-01 01:00	18568	1	0	0	
2016-01-01 01:00	21277	1	0	0	
2016-01-01 01:00	18558	1	0	0	
2016-01-01 01:00	22636	1	0	0	

(b) Diagnostics Table

Time	Tags ID	Lat.	Long.	Elev.	Vel.	Head.	Grade	Fuel Cons.	Add. Tags
2016-01-01 01:00	1785	14.07	110.77	152	0	69	0	25	
2016-01-01 01:00	4274	52.31	4.72	124	0	221	0	25	
2016-01-01 01:00	9	66.68	105.76	222	0	252	0	25	
2016-01-01 01:00	1875	6.78	166.86	1	0	112	0	25	
2016-01-01 01:00	458	81.93	56.12	236	0	335	0	25	

(c) Readings Table

Figure 3.1: Sample from TSBS Tables for the `iot` Use Case

There are 12 query options for benchmarking the `iot` use case, each designed to showcase a different scenario. This means that query performance and behavior can vary between DBMS systems. This will be investigated in detail in Section 3.1.4. The descriptions of the queries are shown in Table 3.2.

Query Type	**Description**
last-loc	Fetch real-time (i.e., last) location of each truck.
low-fuel	Fetch all trucks with low fuel (less than 10%).
high-load	Fetch trucks with high current load (over 90% load capacity).
stationary-trucks	Fetch all trucks that are stationary (low average velocity in the last 10 minutes).
long-driving-sessions	Get trucks which haven't rested for at least 20 minutes in the last 4 hours.
long-daily-sessions	Get trucks which drove more than 10 hours in the last 24 hours.
avg-vs-projected-fuel-consumption	Calculate average vs. projected fuel consumption per fleet.
avg-daily-driving-duration	Calculate average daily driving duration per driver.
avg-daily-driving-session	Calculate average daily driving session per driver.
avg-load	Calculate average load per truck model per fleet.
daily-activity	Get the number of hours a truck has been active (vs. out-of-commission) per day per fleet.
breakdown-frequency	Calculate breakdown frequency by truck model.

Table 3.2: IoT Query Types and Descriptions

3.1.3 Setup

This section discusses the setup process for TSBS, going over the main configuration steps. These steps have an impact on the final results, and therefore they should be studied in detail. In Table 3.6 are the software versions and requirements used to set the benchmark. Later versions of TimescaleDB and Influx could not be tested as support for most TSBS functionalities has not been updated for more than three years.

Software	Version
PostgreSQL	13.17
TimescaleDB	2.6.1
InfluxDB	1.11.8
Ubuntu	22.04.5 LTS
Go	1.18.1
Python	3.10.12

Table 3.3: Software Versions Used for Testing

After installing the required software, we set up TSBS itself. We only show the setup steps in detail for TimescaleDB as the setup for Influx follows the same process, with only minor syntax differences. Both setup processes can be found in our GitHub repository at `tsbs/additional_scripts /workflow.md`.

The first step is data generation using the following command, where `sfX` represents the scale factor:

```
./tsbs_generate_data --use-case="iot" --seed=123 --scale=SF \
  --timestamp-start="2016-01-01T00:00:00Z" --timestamp-end="2016-01-10T00:00:00Z" \
  --log-interval="1s" --format="timescaledb" > ../tmp/data_ts_sfX.txt
```

The above command generates 10 days worth of data, with a log interval of one second. The scale factor in the `iot` use case is based on the number of trucks tracked.

The second step is loading the data into the databse, with the following command:

```
cat ../tmp/data_ts_sfX.txt | ./tsbs_load_timescaledb \
  --host="localhost" --port=5432 --user="postgres" --pass="xxx" \
  --db-name="benchmark" --workers=8
```

This command loads the data into TimescaleDB, with eight concurrently inserting clients. Note that we wanted to test both databases with the default settings, and thus some additional flags are omitted. The most notable is `-chunk-time`, which uses the default 12 hours in this case (default), which is not optimal as this should be adjusted based on the dataset size. This directly impacts the size on disk capabilities compared to Influx. We later demonstrate manual adjustments to this and their impact on the results. The `-batch-size` is not specified, so it uses the default of 10,000.

We wrote custom bash scripts for generating the queries as we had issues with the default ones. The `generate_timescale_queries.sh` and `generate_influx_queries.sh` scripts can also be found in our

GitHub repository.

To test every query available in TSBS, we use the `scripts/generate_run_script.py` which creates a bash script with commands to run multiple query types in a row [Tim24a]. Below is the configuration:

```
python3 generate_run_script.py \
  -b 500 \
  -d timescaledb \
  -e '--postgres="host=localhost user=postgres password=xxx dbname=benchmark sslmode=disable"' \
  -f ./queries.txt \
  -l ../tmp \
  -n 500 \
  -o ../tmp/queries_timescaledb_sfX \
  -s localhost \
  -w 8 > ../tmp/query_test_timescaledb.sh
```

After running the command above, a `query_test_timescaledb.sh` is generated, which we run to obtain the results. Lastly, we check the disk size used by the database. We repeat this process for each of the four scale factors and for both databases.

The results are saved in separate files, such as `query_timescaledb_timescaledb-last-loc-queries.out`. We aggregate the results using `aggregate_results.sh`.

3.1.4 Results

The data containing the results are available in our repository at `tsbs/results`. To analyze the results and the behavior of the databases on different query types, we established relationships between the two main tables and created a dashboard in Power BI.

The following scale factors were tested: 5, 10, 25 and 50. Figure 3.2 showcases the load times by scale factor in milliseconds. The results are very similar between databases and linear behavior is shown.

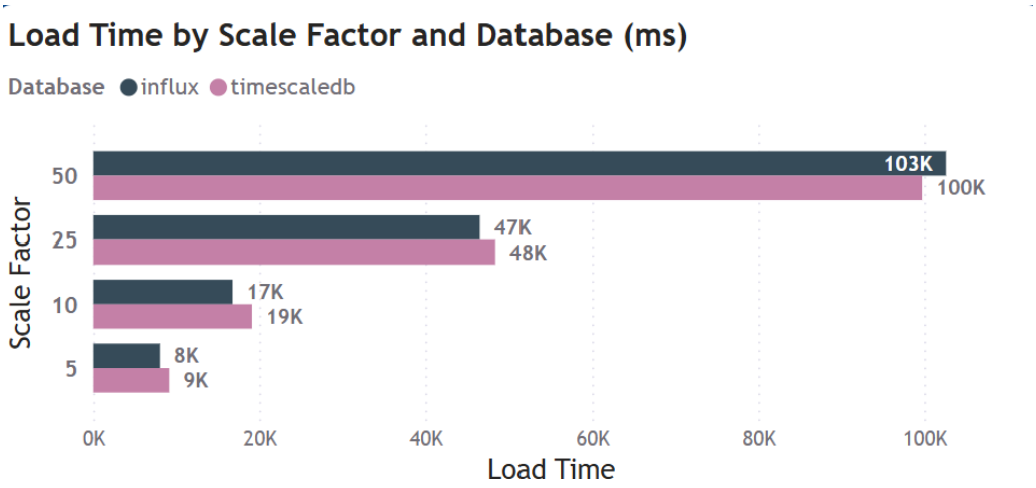


Figure 3.2: Load Time Comparison

In terms of size on disk, Influx shows much better results, with ten times better compression. The results are visible on Figure 3.3. Note that as mentioned earlier, we used the default setting with TimescaleDB which does not compress data in its hypertables. To enable compression, users must activate compression, and define a compression policy to compress data after a specific time interval. It's important to note that the effectiveness of compression heavily depends on the chunk_time_interval setting as discussed in the previous chapter. This means timescaleDB users need to understand and configure these key details in advance for optimal results.

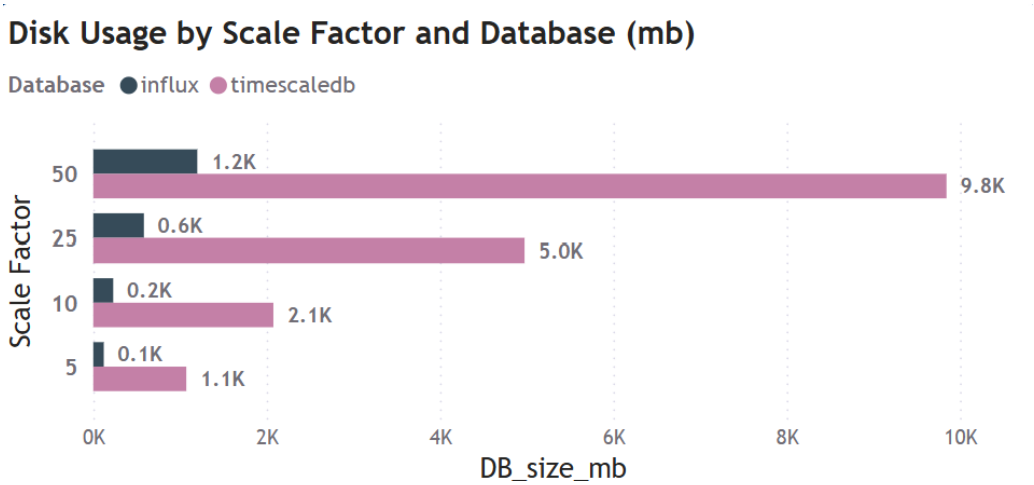


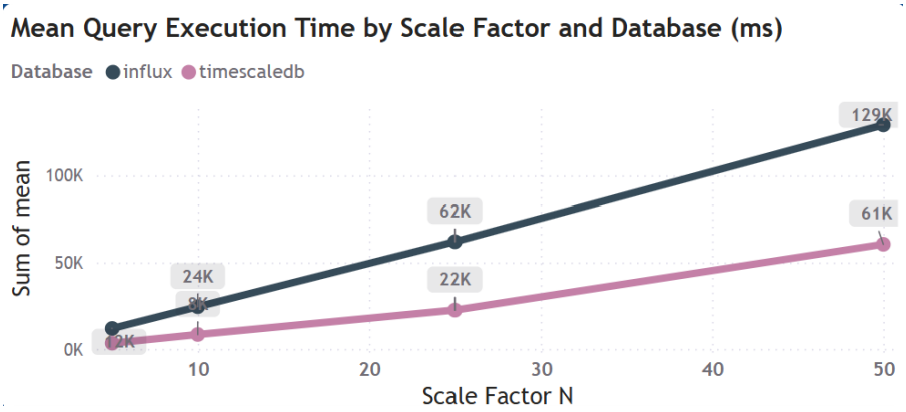
Figure 3.3: Disk Usage Comparison

We wanted to take it a step further and see how TimescaleDB's compression stacks up against Influx. To achieve this, we enabled compression to the hypertables with chunk_time_interval of 1 day for the SF50 (which creates ten or less chunks for each hypertable since there are 10 days of data). As a result, the disk usage reduces from 9.8 GB to 1.7 GB, which is only around 1.4 times worse than Influx, which is much closer to the results obtained by the TSBS team when comparing the two databases [Tim21]. Nevertheless, we can conclude that Influx performs significantly better in compressions than TimescaleDB and it does so without requiring detailed knowledge and complex configurations.

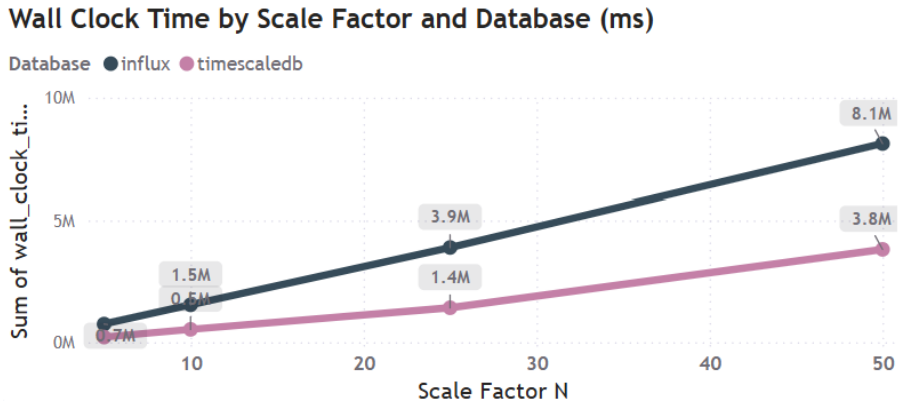
Each of the 12 queries was run 500 times. The following results showcase the mean execution times. We encourage the reader to use the Power BI report when reading this part as it provides additional metrics such as min, max, sum, standard deviation, mean, and median in the tooltips. Figure 3.4 shows the sum of the mean execution times for all queries.

Both databases show behavior that is almost exactly linear, with TimescaleDB performing better on average. However, performance mostly depends on the query type that is being tested. In this report, we showcase two query examples, where either TimescaleDB or Influx shows much better performance. The reader is encouraged to experiment further, using the dashboard and the `tsbs/query_examples/influx.md` and `tsbs/query_examples/timescaledb.md` files in our repository, where we extracted the queries from the TSBS source code.

The two queries are `avg-load` and `long-driving-sessions`. Their description is included in Table 3.2.



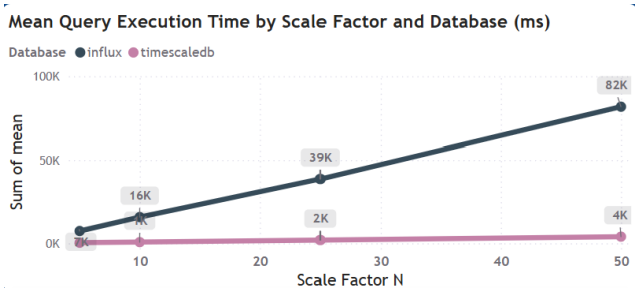
(a) SUM of Mean Execution Times



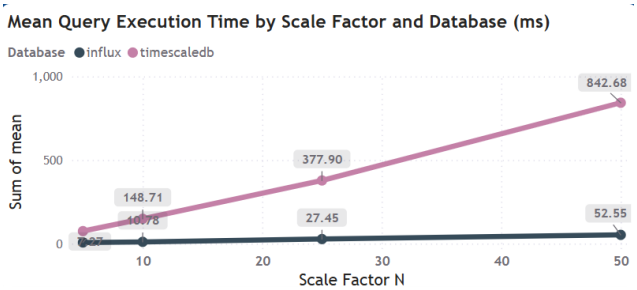
(b) Wall Clock Times

Figure 3.4: Comparison of Execution for Every Query

The results obtained are visible in Figure 3.12.



(a) Query Results - `avg-load`



(b) Query Results - `long-driving-sessions`

Figure 3.5: Comparison of Query Results for `avg-load` and `long-driving-sessions`

To explain the differences in performance, we first have to take a look at the corresponding queries that we extracted from the TSBS source code. They are visible on Figure 3.13.

TimescaleDB - avg-load Query

```
SELECT t.fleet, t.model, t.load_capacity,
    avg(d.avg_load / t.load_capacity)
    AS avg_load_percentage
FROM tags t
INNER JOIN (
    SELECT tags_id, avg(current_load)
    AS avg_load
    FROM diagnostics
    GROUP BY tags_id
) d ON t.id = d.tags_id
GROUP BY fleet, model, load_capacity;
```

InfluxDB - avg-load Query

```
SELECT mean("ml") AS mean_load_percentage
FROM (
    SELECT "current_load" / "load_capacity"
    AS "ml"
    FROM "diagnostics"
    GROUP BY "name", "fleet", "model"
)
GROUP BY "fleet", "model";
```

TimescaleDB - long-driving-sessions Query

```
WITH driving_sessions AS (
    SELECT time_bucket('10 minutes', time)
    AS ten_minutes, tags_id
    FROM readings
    WHERE velocity > 1
    GROUP BY ten_minutes, tags_id
),
long_driving_sessions AS (
    SELECT time_bucket('4 hours', ten_minutes)
    AS session,
           tags_id, count(*) AS periods
    FROM driving_sessions
    GROUP BY session, tags_id
)
SELECT t.name, t.driver
FROM tags t
INNER JOIN long_driving_sessions d
ON t.id = d.tags_id
WHERE d.periods > calculated_value;
```

InfluxDB - long-driving-sessions Query

```
SELECT "name", "driver"
FROM (
    SELECT count(*) AS ten_min
    FROM (
        SELECT mean("velocity")
        AS mean_velocity
        FROM "readings"
        WHERE "fleet" = 'random fleet'
              AND time > 'start time'
              AND time <= 'end time'
        GROUP BY time(10m), "name", "driver"
    )
    WHERE "mean_velocity" > 1
)
WHERE ten_min > calculated_value;
```

Figure 3.6: Comparison of Query Implementations for `avg-load` and `long-driving-sessions`

The `avg-load` query performs better in TimescaleDB compared to Influx. TimescaleDB benefits from JOIN performance optimization. On the other hand, Influx is designed for append-only workloads, which are optimized for fast writes and real-time queries on time-series data. Influx emulates JOINs by grouping data on common tags or fields. This causes overhead for queries like `avg-load` using `current_load` and `load_capacity`, where both the `tags` and `diagnostics` tables are needed.

The results are the opposite in the `long-driving-sessions` query. Influx outperforms TimescaleDB here as hierarchical aggregations over fixed time intervals is natively optimized and there is no CTE materialization needed like in TimescaleDB. The results are similar in queries where time-based filtering and time-based aggregations are used (mainly due to the Time-Structured Merge Tree (TSM) used in Influx [Inf24c]).

3.2 Custom Benchmark

3.2.1 Hardware Specifications

In this subsection, we display the hardware specifications of the machine used to benchmark both the tools.

Processor	Apple M3 Pro
RAM	18.0 GB
Storage	512GB SSD

Table 3.4: Hardware Specifications

3.2.2 Dataset

In this subsection, we describe the custom database developed for this benchmarking process. The dataset comprises glycemic variability data [Ben+21] collected using wearable devices (the dataset is referred to as medical sensor data throughout this study). Specifically, it includes information from 16 participants with elevated but normal-range blood glucose levels, monitored over 8–10 days using two devices: the Dexcom G6 continuous glucose monitor (CGM) and the Empatica E4 wristband. To preserve participant privacy, all data were time-shifted by date to prevent reidentification.

The Dexcom G6 measured interstitial glucose concentrations (mg/dL) at 5-minute intervals, while the Empatica E4 captured multiple physiological signals at varying sampling frequencies. The resulting dataset encompasses seven key features:

- **Photoplethysmography (PPG):** Sampled at 64 Hz, providing continuous blood volume pulse (BVP) data. From this, heart rate (HR) was derived every second, and interbeat interval (IBI) data were computed.
- **Electrodermal Activity (EDA):** Sampled at 4 Hz, representing skin conductance.
- **Skin Temperature:** Sampled at 4 Hz, reflecting thermal fluctuations.
- **Tri-Axial Accelerometry:** Sampled at 32 Hz, capturing motion data along the X, Y, and Z axes.

Table Name	Description
ACC	Data includes the `Timestamp` as a datetime value, with accelerometer data for the `X`, `Y`, and `Z` orientations.
BVP	Refers to blood volume pulse; data includes `Timestamp` as a datetime value and `Value` as the measurement recorded at the time.
Dexcom	Refers to interstitial glucose concentration; data includes `Timestamp` as a datetime value and `Value` as the measurement recorded at the time.
EDA	Refers to electrodermal activity; data includes `Timestamp` as a datetime value and `Value` as the measurement recorded at the time.
TEMP	Refers to skin temperature; data includes `Timestamp` as a datetime value and `Value` as the measurement recorded at the time.
IBI	Refers to interbeat interval; data includes `Timestamp` as a datetime value and `Value` as the measurement recorded at the time.
HR	Refers to heart rate; data includes `Timestamp` as a datetime value and `Value` as the measurement recorded at the time.
Food Log	Refers to food items consumed by the participant throughout the study; data includes: • `date`: as a date value • `time_of_day`: as a time value • `time_begin` and `time_end`: as datetime values • `logged_food`: as a string value • `amount`: as a numeric value • `unit`: as a string value • `searched_food`: as a string value • `calorie`: as a numeric value • `total_carb`, `dietary_fiber`, `sugar`, `protein`, and `total_fat`: as numeric values

Table 3.5: Dataset information

The total uncompressed size of the data is 34.1 GB. On an average each folder is around 1.5GB in size.

3.2.3 Setup

To utilize the custom dataset effectively, modifications were required in the raw datasets. Since it was not possible to distinguish between different participants within the same dataset, a separate column named `participant_id` was introduced. This additional column allowed us to uniquely identify and associate the data with individual participants, enabling more structured and participant-specific analysis. The Dexcom table initially contained irrelevant information in the first few rows, which needed to be removed to ensure consistency in the dataset's structure. After applying these preprocessing steps to each user's data using a

Python script, we proceeded with the initial setup for each of the databases.

Software	Version
PostgreSQL	14.2
TimescaleDB	2.6.1
InfluxDB	1.11.8
MacOS	Sonoma 14.6.1
Python	3.13

Table 3.6: Software Versions Used for Testing

TimeScaleDB

We created seperate schemas for all the different tables and also used a compression policy for each of them.

Figure 3.7: Timescale Database Schema

Example Code to create a hypertable

```
SELECT create_hypertable('accelerometer_data','ts',chunk_time_interval => INTERVAL '1 days');
```

We created hypertables for each of the tables in the dataset using the column representing the timestamp. The `chunk_time_interval` was set to `1 day` for each hypertable, resulting in 10 chunks of data. This

approach was designed to enhance the operational efficiency of the database by improving data partitioning and reducing the time required for query execution, compression, and other database operations. With this configuration, the system can manage data more efficiently while keeping chunks at an optimal size for both storage and performance.

Example Code to enable compression on the tables

```
ALTER TABLE accelerometer_data SET (timescaledb.compress,
timescaledb.compress_segmentby = 'participant_id',
timescaledb.compress_orderby='ts DESC');
SELECT add_compression_policy('accelerometer_data', INTERVAL '2 weeks');
```

TimescaleDB does not automatically compress your data. To enable compression, we need to alter the table settings and explicitly enable compression. Additionally, it is essential to define how the data should be *segmented* and *ordered* to ensure uniformity during compression, which significantly improves the efficiency of both compression and decompression. Proper segmentation and ordering ensure that the data is stored in a way that allows the compression algorithm to achieve optimal results.

Moreover, we can set up a *compression policy*, which automatically compresses the data after a specific time interval. This policy helps maintain long-term storage efficiency without manual intervention.

InfluxDB

The schema design followed these principles:

- Each participant's data file was treated as a separate measurement, ensuring that each measurement represents a distinct logical data type (e.g., `accelerometer_data`, `temperature`).
- Data files from different participants with similar content (e.g., heart rate recordings) were grouped into the same measurement during the data loading process.
- Attributes with low cardinality that are frequently queried or grouped (e.g., `participant_id`) were stored as tags to optimize filtering and grouping operations.
- Numerical features (e.g., `acc_x`, `hr`, `temp`) were stored as fields to enable efficient storage and aggregation.
- The default retention policy, `autogen`, was used with an infinite duration, a replication factor set to one, and a shard group duration set to seven days.

The complete schema is presented in the Table 3.7.

Measurement	Tags	Fields
demographics	ID, Gender	HbA1c
accelerometer_data	participant_id	acc_x, acc_y, acc_z
blood_volume_pulse	participant_id	bvp
interstitial_glucose	participant_id, event_type, source_device_id	event_subtype, device_info, glucose_value, insulin_value, carb_value, duration, glucose_rate_change, transmitter_time
electrodermal_activity	participant_id	eda
heart_rate_data	participant_id	hr
ibi_data	participant_id	ibi
temperature_data	participant_id	temp

Table 3.7: InfluxDB Database Schema: Measurements, Tags, and Fields

As for the benchmarking purposes, we were using InfluxDB v1. We encountered some limitations in terms of data ingestion from `.csv` files. The primary challenge was that InfluxDB v1 relies on the line protocol for data ingestion, and native support for loading `.csv` files was not yet available in this version.

To address this issue, we explored and implemented two approaches:

1. **Using the Official Python Client for InfluxDB** [Inf24b]: This approach provided a programmable way to interact with the database but required parsing `.csv` files into Pandas dataframes first, and then using the `write_points()` function of the `DataFrameClient`, adding extra steps and complexity to the data loading process.

2. **Using an Open-Source Python Library** [Bug24]: We utilized the library *Export CSV To Influx*, specifically designed to streamline the process of loading `.csv` files into InfluxDB. This library offered a simple interface where key parameters could just be passed into a single function, greatly reducing the effort and complexity of the ingestion process.

By applying these methods, we successfully loaded data into InfluxDB. However, the timings for the two approaches differed drastically:

File Name	Lines	Export CSV To Influx Time (ms)	Custom Script Using Python Client Time (ms)	Ratio
ACC_001.csv	20,296,428	778,234.36	93,371.28	8.33
BVP_001.csv	40,592,838	1,119,653.04	153,424.31	7.30
EDA_001.csv	2,537,046	65,446.39	9,652.13	6.78
IBI_001.csv	266,366	6,950.20	1,072.34	6.48
TEMP_001.csv	2,537,040	64,423.77	9,453.01	6.82
HR_001.csv	634,188	15,568.13	2,390.57	6.51
Dexcom_001.csv	245	245.72	45.64	5.39
Total	**66,863,151**	**2,050,521.61**	**269,409.28**	**7.61**
Total (min)	**-**	**34.175**	**4.490**	**-**

Table 3.8: Comparison of Insertion Times Between Methods

The results demonstrate that our custom script utilizing the Python client and Pandas outperformed the open-source library, achieving an approximately 7-fold improvement in data insertion speed while successfully handling the same dataset. This significant efficiency gain highlights the effectiveness of our approach for benchmarking purposes.

3.2.4 Queries

While developing queries for this custom benchmark, our primary objective was to approach the task from the perspective of a potential user of our application, such as an owner of an IoT device (e.g., a fitness bracelet) or a medical professional. We aimed to address questions and insights they might find valuable when analyzing health data. Accordingly, we designed our queries to showcase practical and meaningful examples of such insights.

However, our efforts were constrained by the limitations of the software we used. To ensure consistency with the TSBS benchmarking component of this project, we utilized the same version of InfluxDB (v1.11.8). This version's primary query language, InfluxQL, is an SQL-like language specifically designed for interacting with InfluxDB. While InfluxQL offers features tailored for time series data storage and analysis, it lacks some of the more advanced operations found in standard SQL, such as UNION, JOIN, and HAVING. These limitations required us to adapt our queries to fit the capabilities of InfluxQL while still achieving our benchmarking goals.

Below, we present the set of queries used in our benchmark, designed to highlight the utility and performance of InfluxQL for typical health-related time series data analyses.

InfluxDB Query - Highest Heart Rates for Each Day

```
SELECT TOP("hr", 3), "participant_id"
FROM "heart_rate_data"
WHERE time >=
'2020-02-14T00:00:00Z'
  AND time <=
  '2020-02-20T23:59:59Z'
GROUP BY time(1d), "participant_id"
```

TimescaleDB Query - Highest Heart Rates for Each Day

```
SELECT time_bucket('1 day', datetime) AS day,
       participant_id, hr
FROM heart_rate_data
WHERE datetime BETWEEN TIMESTAMP
                '2020-02-14 00:00:00'
                   AND TIMESTAMP
                '2020-02-20 23:59:59'
ORDER BY day, participant_id, hr DESC
LIMIT 3;
```

InfluxDB Query - Blood Volume Pulse Rate of Change

```
SELECT derivative(mean("bvp"), 1s)
    AS bvp_rate
FROM "blood_volume_pulse"
WHERE time >= '2020-02-13T22:29:00Z'
  AND time < '2020-02-13T22:30:00Z'
GROUP BY time(5s), "participant_id"
```

TimescaleDB Query - Blood Volume Pulse Rate of Change

```
WITH mean_bvp AS (
  SELECT time_bucket('5 seconds', ts) AS bucket,
         participant_id, AVG(bvp) AS mean_bvp
  FROM blood_volume_pulse
  WHERE ts BETWEEN TIMESTAMP
  '2020-02-13 22:29:00'
            AND TIMESTAMP
              '2020-02-13 22:30:00'
  GROUP BY bucket, participant_id
)
SELECT bucket, participant_id,
       mean_bvp - LAG(mean_bvp) OVER (
           PARTITION BY participant_id
           ORDER BY bucket) AS bvp_rate
FROM mean_bvp;
```

InfluxDB Query - Glucose Statistics

```
SELECT MEAN("glucose_value") AS "mean_g",
       STDDEV("glucose_value") AS "std_g"
FROM "interstitial_glucose"
WHERE ({list_of_participants})
  AND time >= '2020-02-15T00:00:00Z'
  AND time < '2020-02-25T23:59:59Z'
GROUP BY time(1h)
```

TimescaleDB Query - Glucose Statistics

```
SELECT time_bucket('1 hour', ts) AS hour,
       AVG(glucose_value) AS mean_g,
       STDDEV(glucose_value) AS std_g
FROM interstitial_glucose
WHERE participant_id IN {{list_of_participants}}
  AND ts >= TIMESTAMP '2020-02-15 00:00:00'
  AND ts < TIMESTAMP '2020-02-25 23:59:59'
GROUP BY time_bucket('1 hour', ts)
ORDER BY hour;
```

InfluxDB Query - Minimum Daily Average EDA

```
SELECT MIN("daily_avg_eda")
    AS "min_daily_avg_eda"
FROM (
    SELECT MEAN("eda") AS "daily_avg_eda"
    FROM "electrodermal_activity"
    WHERE ({list_of_participants})
    GROUP BY time(1d)
)
```

TimescaleDB - Minimum Daily Average EDA

```
SELECT MIN(daily_avg_eda) AS min_daily_avg_eda
FROM (
    SELECT AVG(eda) AS daily_avg_eda
    FROM electrodermal_activity
    WHERE participant_id
        IN {{list_of_participants}}
    GROUP BY time_bucket('1 day', ts)
) subquery;
```

InfluxDB Query - Maximum Movement

```
SELECT MAX("movement"), "acc_x", "acc_y",
    "acc_z"
FROM (
    SELECT SQRT(POW("acc_x", 2) +
                POW("acc_y", 2) +
                POW("acc_z", 2))
           AS "movement", "acc_x",
            "acc_y", "acc_z"
    FROM "accelerometer_data"
)
```

TimescaleDB Query - Maximum Movement

```
WITH movement_data AS (
  SELECT ts, participant_id,
         acc_x, acc_y, acc_z,
         SQRT(POWER(acc_x, 2) +
                POWER(acc_y, 2) +
                POWER(acc_z, 2)) AS movement
  FROM accelerometer_data
)
SELECT ts, participant_id,
        acc_x, acc_y, acc_z,
       MAX(movement) AS max_movement
FROM movement_data
GROUP BY ts, participant_id,
        acc_x, acc_y, acc_z;
```

InfluxDB Query - Temperature Difference

```
SELECT SPREAD("temp") AS temp_diff
FROM "temperature_data"
GROUP BY "participant_id"
```

TimescaleDB Query - Temperature Difference

```
SELECT participant_id,
    MAX(temp) - MIN(temp) AS temp_diff
FROM temperature_data
GROUP BY participant_id;
```

InfluxDB Query - IBI Count

```
SELECT COUNT("ibi") AS ibi_count
FROM "ibi_data"
WHERE "ibi" > 1
GROUP BY "participant_id"
```

TimescaleDB Query - IBI Count

```
SELECT participant_id, COUNT(ibi) AS ibi_count
FROM ibi_data
WHERE ibi > 1
GROUP BY participant_id;
```

InfluxDB Query - Glucose Difference

```
SELECT non_negative_difference(
        MAX(glucose_value)) AS glucose_diff
FROM "interstitial_glucose"
WHERE time >= '2020-02-18T00:00:00Z'
  AND time < '2020-02-28T00:00:00Z'
GROUP BY time(1d), "participant_id"
```

TimescaleDB Query - Glucose Difference

```
WITH daily_max_glucose AS (
  SELECT time_bucket('1 day', ts) AS day,
         participant_id,
         MAX(glucose_value) AS daily_max
  FROM interstitial_glucose
  WHERE ts BETWEEN '2020-02-18T00:00:00Z'
               AND '2020-02-28T00:00:00Z'
  GROUP BY day, participant_id
)
SELECT day, participant_id,
       daily_max - LAG(daily_max) OVER (
           PARTITION BY participant_id
           ORDER BY day) AS glucose_diff
FROM daily_max_glucose;
```

InfluxDB Query - Median Heart Rate

```
SELECT percentile("hr", 50) AS median_hr,
        "participant_id"
FROM "heart_rate_data"
WHERE time >= '2020-02-18T00:00:00Z'
  AND time < '2020-02-20T00:00:00Z'
GROUP BY time(6h), "participant_id"
```

TimescaleDB Query - Median Heart Rate

```
SELECT time_bucket('6 hours', datetime)
    AS interval,
       participant_id,
       PERCENTILE_CONT(0.5)
        WITHIN GROUP (ORDER BY hr)
            AS median_hr
FROM heart_rate_data
WHERE datetime BETWEEN '2020-02-18T00:00:00Z'
                   AND '2020-02-20T00:00:00Z'
GROUP BY interval, participant_id;
```

Figure 3.8: Comparison of Query Implementations Between InfluxDB and TimescaleDB

Query Name	Description and Explanation
Highest Heart Rates for Each Day	Identifies the top three highest heart rates for each day within a specific week, grouped by participants. Useful for monitoring peak activity or potential stress events.
Rate of Change in Blood Volume Pulse (BVP)	Calculates the rate of change in BVP over five-second intervals, grouped by participant. Indicates cardiovascular changes during stress or exercise.
Hourly Glucose Levels with Statistics	Computes hourly mean and standard deviation of glucose levels, grouped by participants. Provides insight into blood sugar trends and abnormalities.
Minimum Daily Average Electrodermal Activity (EDA)	Finds the lowest daily average EDA values, grouped by participants. Identifies calm or inactive periods based on stress-related skin conductance.
Maximum Movement	Calculates the peak movement magnitude for each participant from accelerometer data. Highlights the highest physical activity levels.
Temperature Difference	Determines the range between the highest and lowest temperatures for each participant. Useful for detecting environmental changes or thermoregulation.
Count of Interbeat Intervals (IBI) Above Threshold	Counts IBIs greater than one second, grouped by participants. Key for understanding heart rate variability related to stress or health conditions.
Daily Glucose Level Changes	Captures daily non-negative differences in maximum glucose values, grouped by participants. Tracks daily glucose trends and responses to dietary or medical adjustments.
Median Heart Rate Over Six-Hour Intervals	Calculates the median heart rate over six-hour intervals, grouped by participants. Summarizes central cardiovascular activity while minimizing outlier effects.

Table 3.9: Benchmarking Queries

3.2.5 Results

Benchmarking was conducted using scale factors 2, 3, 4, and 5, where each scale factor represents the number of participant's data included from the Glycemic Dataset. The estimated size of the data for 5 participants is approximately 11.15 GB. To obtain consistent results, all 9 queries were executed 5 times for each scale factor on both tools.

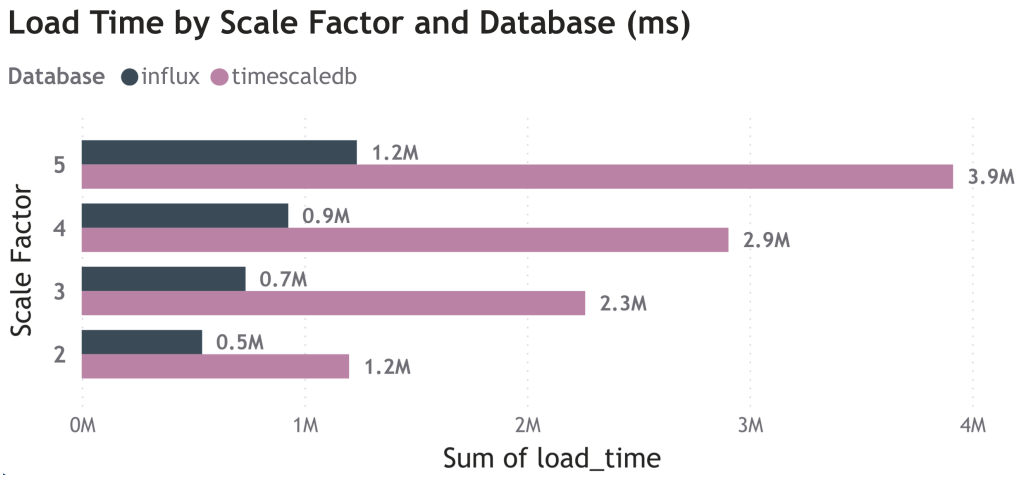


Figure 3.9: Load Performance

As observed, InfluxDB outperforms TimescaleDB in terms of load insertion time, with TimescaleDB being approximately three times slower. This performance difference can be attributed to InfluxDB's superior efficiency in handling low cardinality data. Additionally, the compression settings in TimescaleDB may not be optimized for the dataset, contributing to the significantly longer insertion times.

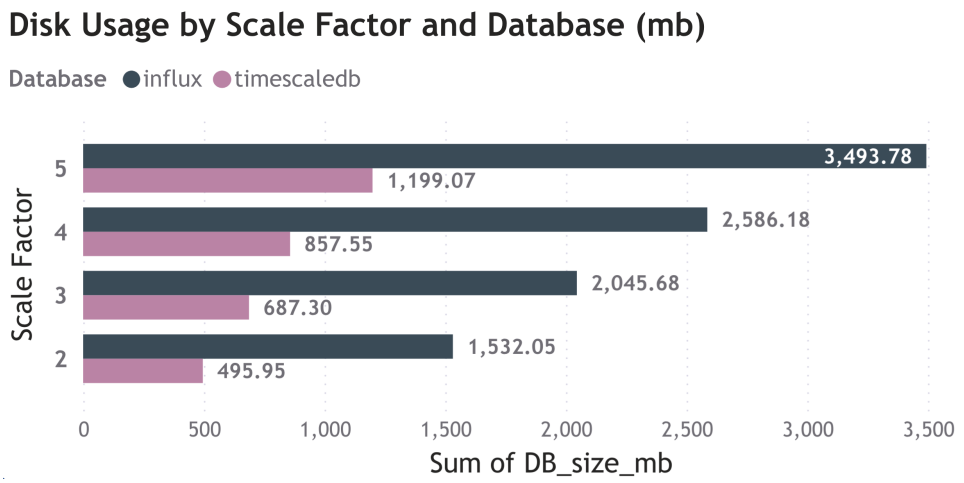


Figure 3.10: Disk Usage

In terms of disk usage, TimescaleDB demonstrates compression performance that is three times better than InfluxDB. Both time-series databases efficiently compress the data, reducing the size from 11.15 GB to approximately 3.5 GB in InfluxDB and 1.2 GB in TimescaleDB.

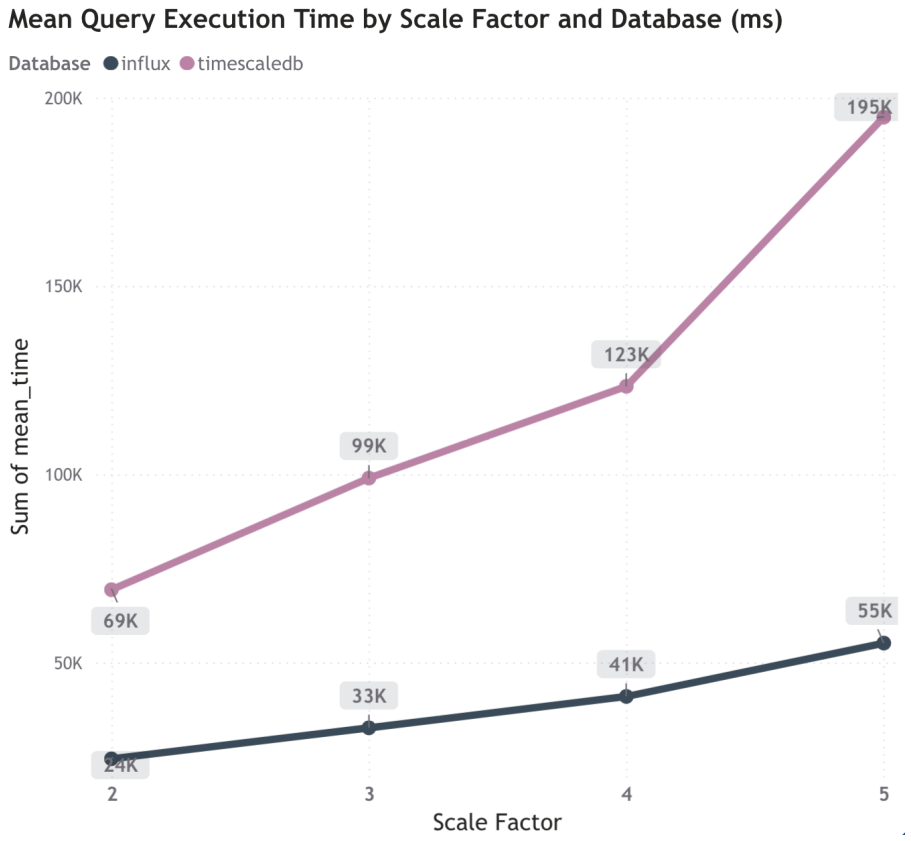
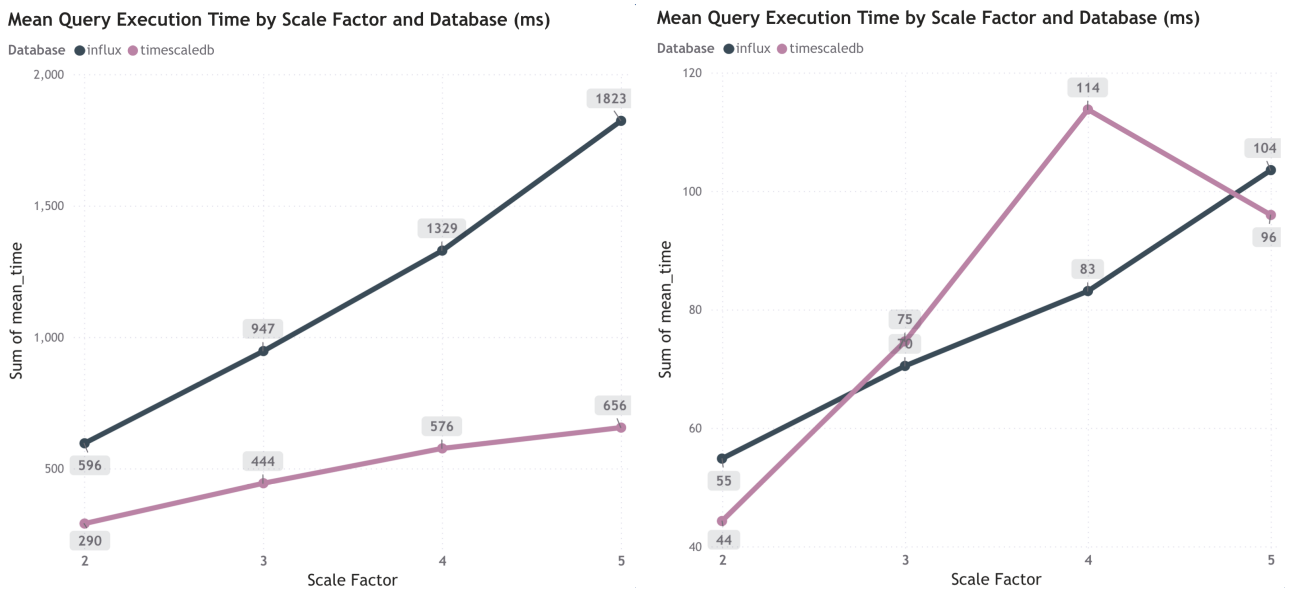


Figure 3.11: Mean Query Execution Time

The mean query execution time exhibits an exponential growth as the scale increases, which aligns with the expected behavior. The efficient compression in TimescaleDB contributes to slower query execution compared to InfluxDB. This effect becomes more pronounced as the scale increases, with TimescaleDB showing a significant impact, while InfluxDB is less affected by the increasing scale.

The results show that, although InfluxDB does not achieve significantly better compression, it outperforms TimescaleDB in other aspects. This can be attributed to the fact that the benchmark queries were designed with the limitations of InfluxQL in mind.

While the overall results indicate better query performance for InfluxDB, there are some anomalous cases where TimescaleDB shows significantly better performance.



(a) Query-5 Execution Time

(b) Query-6 Execution Time

Figure 3.12: Anomalous Behaviour Queries

TimescaleDB - Query-5

```
SELECT participant_id, MAX(temp) - MIN(temp)
AS temp_diff
FROM temperature_data
GROUP BY participant_id;
```

InfluxDB - Query-5

```
SELECT SPREAD("temp") AS temp_diff
FROM "temperature_data"
GROUP BY "participant_id"
```

TimescaleDB - Query-6

```
SELECT participant_id, COUNT(ibi)
AS ibi_count
FROM ibi_data
WHERE ibi > 1
GROUP BY participant_id;
```

InfluxDB - Query-6

```
SELECT COUNT("ibi") AS ibi_count
FROM "ibi_data"
WHERE "ibi" > 1
GROUP BY "participant_id"
```

Figure 3.13: Comparison of Query Implementations for Query-5 and Query-6

In Query-5, the more complex implementation of the `SPREAD` function may lead to inefficient performance compared to the TimescaleDB version, which uses the `MAX` and `MIN` functions. The latter are opti-

mized for efficiency, resulting in better performance in this case.

In Query-6, the performance difference is likely due to how the `WHERE` clause is handled by each database. InfluxDB may be performing a full scan to match the condition, whereas TimescaleDB executes the operation more efficiently, possibly leveraging indexing or other optimization techniques.

3.2.6 Future Enhancements

As we designed queries, many analyses were disrupted due to InfluxQL's lack of support for several operations. This benchmarking was performed to maintain consistency with the tools versions used in the TSBS Benchmarks.

Glucose spike detection and correlation trend analysis are critical aspects of medical data analysis. However, such support was largely absent in InfluxQL, whereas these tasks can be easily performed with TimescaleDB. This benchmark can be further improved by conducting the analysis using newer versions of the tools that provide the necessary support.

4 Conclusion and insights

Running the two benchmarks, TSBS and the custom benchmark, provides valuable insights into the strengths and weaknesses of InfluxDB and TimescaleDB. A significant observation is that the tools yield opposite outcomes in the two benchmarks regarding loading time performance, disk usage, and query execution.

InfluxDB uses its proprietary query language, InfluxQL, while TimescaleDB extends PostgreSQL, leveraging SQL with added features for time-series data. Many of the initial custom benchmark queries could not be implemented in InfluxQL, highlighting its limitations for complex data analysis. In contrast, TimescaleDB handles complex analytical tasks effectively, taking advantage of PostgreSQL features and its own time-series features.

4.1 Tool suitability

4.1.1 InfluxDB

InfluxDB is well-suited for scenarios requiring high-performance time-series data ingestion. As indicated by the benchmark results, InfluxDB outperforms TimescaleDB in load time across 7 out of 8 scale factors in both benchmarks. Furthermore, when tested with a real-world dataset, InfluxDB demonstrates superior performance, requiring only half or even one-third of the loading time compared to TimescaleDB.

InfluxDB typically uses less disk space than TimescaleDB, particularly for time-series workloads. As seen from the TSBS benchmark, which focuses on time-series data, InfluxDB outperforms TimescaleDB, requiring 1.4 times less storage even when TimescaleDB utilizes compression. This efficiency stems from InfluxDB's purpose-built design for time-series data and its advanced compression techniques optimized for minimal storage consumption. However, in the custom benchmark that contains more relational style data, influx performs poorly compared to timescale requiring three times more disk space.

Its custom-built query language, InfluxQL, and native support for time-series data structures make it efficient for handling continuous queries and aggregation queries. As seen, in the query execution of the custom benchmark, influxDB outperforms timescale requiring one-third of the execution time in lower scale factors while that gap increases further when scale factors increase. However, its limitations become

apparent in scenarios requiring complex queries or integrations with relational data, as InfluxQL lacks the expressiveness and flexibility of SQL. In detail, most of the advanced queries of the following exercise chapter were not possible to translate into InfluxQL. InfluxDB excels in environments focused on straightforward time-series workloads with minimal need for advanced data analysis or joins.

4.1.2 TimescaleDB

TimescaleDB is an excellent choice for use cases that demand robust time-series data analysis alongside relational database capabilities. Extending PostgreSQL inherits the full power of SQL and supports advanced analytical queries, joins, and complex operations. In the following chapter of the exercises, the capabilities of timescaleDB are showcased with advanced queries that detect valuable intelligence for the medical sensor data application. This is not feasable using influxDB.

Timescale is ideal for applications that require both high-performance time-series data processing and integration with broader datasets, such as medical or financial analytics, or operational intelligence. While TimescaleDB may not match InfluxDB in raw ingestion speed and disk storage of solely time-series data, its flexibility and scalability make it a versatile solution for hybrid workloads and sophisticated data analysis tasks. Lastly, Timescale outperforms Influx in storing time-series relational tables, as demonstrated in the custom benchmark, when it efficiently utilizes compression of the hypertables.

4.2 Key takeaways

The comparison between TimescaleDB and InfluxDB highlights the differing strengths of the two databases. InfluxDB demonstrates superior performance for high-volume ingestion of pure time-series data. Additionally, its optimized storage mechanisms result in lower disk usage for such workloads. However, InfluxDB's limitations with relational-style data are obvious. Query execution times favor InfluxDB for straightforward aggregations, particularly at larger scales, but its proprietary InfluxQL lacks the flexibility for complex or relational queries.

Conversely, TimescaleDB's strength lies in its versatility and analytical power, extending PostgreSQL to support sophisticated time-series applications. By leveraging SQL and its time-series features, it excels in handling complex queries. Moreover, one of its advantages is storing and processing hybrid workloads, efficiently utilizing compression, and delivering robust query capabilities that are unfeasible in InfluxDB. While it does not match InfluxDB's raw ingestion speed or minimal storage needs for pure time-series data, TimescaleDB's broader dataset compatibility and scalability make it a strong candidate for use cases requiring deeper analytical insight and integration with relational data.

In conclusion, TimescaleDB is the more suitable choice for this specific medical sensor application. Its efficient disk usage for storing time-series and relational data, particularly at larger scales, combined with its advanced query capabilities, makes it an excellent fit for uncovering critical insights such as glucose spikes,

stress detection, or exercise patterns. Moreover, in future applications, these sophisticated queries could play a pivotal role in identifying patients at high risk for conditions like diabetes or cardiovascular diseases, especially within the field of geriatrics.

4.3 Exercises

This section includes queries designed as exercise material for this course. These queries utilize the medical sensor dataset, which is the basis for the custom benchmark report. They serve as a preview of a potential final product for a TimescaleDB chapter. The complete exercises, including table creation, ETL processes, and more, can be developed at a later stage.

Filtering period

Calculating the average temperature of participant 1 for February 2020:

```
SELECT AVG(temp)
FROM temperature_data
WHERE participant_id = 1 AND
        event_time BETWEEN '2020-02-01 00:00:00' AND '2020-02-29 23:59:59';
```

The same can be calculated using EXTRACT:

```
SELECT AVG(temp)
FROM temperature_data
WHERE participant_id = 1 AND
        EXTRACT(YEAR FROM event_time) = 2020 AND
        EXTRACT(MONTH FROM event_time) = 2;
```

Also, calculating the average temperature of participant 1 for the last 10 days of February 2020:

```
SELECT AVG(temp)
FROM temperature_data
WHERE participant_id = 1 AND
        event_time >= TIMESTAMP '2020-03-01 00:00:00' - INTERVAL '10 day' AND
        event_time < TIMESTAMP '2020-03-01 00:00:00';
```

Aggregating for different time intervals

Average monthly temperature for each participant and for each month:

```
SELECT participant_id,
        time_bucket('1 month', event_time) AS month,
        AVG(temp) AS temperature
FROM temperature_data
GROUP BY participant_id, month
ORDER BY participant_id, month DESC;
```

Finding the first and last attribute values

Finding the first and last temperature measurement for each participant in the span of the 1st February 2020 and 1st March 2020:

```
SELECT participant_id,
        FIRST(temp, event_time) AS first_temperature_measurement,
        LAST(temp, event_time) AS last_temperature_measurement
FROM temperature_data
WHERE event_time BETWEEN '2020-02-01 00:00:00' AND '2020-03-01 23:59:59'
GROUP BY participant_id
ORDER BY participant_id;
```

Continuous aggregate

Creation of a continuous aggregate on the temperature attribute that calculates the first, last, min, max and average of temperature measurement for each day and for each participant:

```
CREATE MATERIALIZED VIEW temperature_stats_daily
WITH (timescaledb.continuous) AS
SELECT participant_id,
        time_bucket('1 day', event_time) AS day,
        FIRST(temp, event_time),
        LAST(temp, event_time),
        MIN(temp),
        MAX(temp),
        AVG(temp)
FROM temperature_data
GROUP BY participant_id, day;
```

The aggregate should refresh automatically every day and the refresh should only include data from 1 week before the current time until one day before the current time (we assume that data older than 1 week does not change):

```
SELECT add_continuous_aggregate_policy('temperature_stats_daily',
    start_offset => INTERVAL '1 week',
    end_offset => INTERVAL '1 day',
    schedule_interval => INTERVAL '1 day');
```

Daily and weekly step count

Calculating the daily and weekly step count for each participant using accelerometer data:

```
WITH accelerometer_magnitude AS (
    SELECT
        participant_id,
        ts,
        SQRT(POWER(acc_x, 2) + POWER(acc_y, 2) + POWER(acc_z, 2)) AS acceleration_magnitude
    FROM accelerometer_data
),
step_detection AS (
    SELECT
        participant_id,
        time_bucket('1 day', ts) AS day,
        time_bucket('1 week', ts) AS week,
        COUNT(*) AS potential_steps
    FROM accelerometer_magnitude
    WHERE acceleration_magnitude > 100
    GROUP BY participant_id, day, week
)
-- Daily Step Count
SELECT
    participant_id,
    day AS period,
    'daily' AS period_type,
    potential_steps AS step_count
FROM step_detection
UNION ALL
-- Weekly Step Count
SELECT
    participant_id,
    week AS period,
    'weekly' AS period_type,
    SUM(potential_steps) AS step_count
FROM step_detection
```

```
GROUP BY participant_id, week
ORDER BY participant_id, period_type, period;
```

Glucose Spike Detection

Identifying glucose spikes for participants using interstitial glucose data:

```
WITH glucose_changes AS (
    SELECT
        participant_id,
        glucose_value AS current_glucose,
        LEAD(ts) OVER (PARTITION BY participant_id ORDER BY ts) AS next_time,
        LEAD(glucose_value) OVER (PARTITION BY participant_id ORDER BY ts) AS next_glucose,
        LEAD(ts) OVER (PARTITION BY participant_id ORDER BY ts) - ts AS time_diff,
        LEAD(glucose_value) OVER (PARTITION BY participant_id ORDER BY ts) - glucose_value
            AS glucose_change
    FROM
        interstitial_glucose
    WHERE
        glucose_value IS NOT NULL
)
SELECT
    participant_id,
    next_time,
    current_glucose,
    next_glucose,
    glucose_change,
    time_diff
FROM
    glucose_changes
WHERE
    glucose_change > 14
    AND time_diff <= INTERVAL '30 minutes'
ORDER BY
    participant_id,
    current_time;
```

Glucose and Heart Rate Correlation

Identifying the correlation trend between glucose and heart rate:

```sql
SELECT
    hrd.participant_id,
    time_bucket('1 hour', hrd.datetime) AS hour,
    ROUND(AVG(hrd.hr), 2) AS avg_heart_rate,
    ROUND(AVG(ig.glucose_value), 2) AS avg_glucose,
    CORR(hrd.hr, ig.glucose_value) AS hr_glucose_correlation
FROM heart_rate_data hrd
JOIN interstitial_glucose ig
    ON hrd.participant_id = ig.participant_id
    AND hrd.datetime BETWEEN ig.ts - INTERVAL '30 minutes' AND ig.ts + INTERVAL '30 minutes'
GROUP BY hrd.participant_id, hour;
```

Anomalies in glucose readings

Detecting anomalies in glucose readings from interstitial glucose data. To achieve that the mean and standard deviation of measurements of each participant are used to identify normal and anomaly cases:

```sql
WITH glucose_stats AS (
    SELECT
        time_bucket('1 hour', ts) AS hour,
        participant_id,
        AVG(glucose_value) AS mean_glucose,
        STDDEV(glucose_value) AS std_glucose
    FROM interstitial_glucose ig
    GROUP BY hour, participant_id
)
SELECT
    hour,
    g.participant_id,
    mean_glucose,
    std_glucose,
    CASE
        WHEN ABS(glucose_value - mean_glucose) > 2 * std_glucose
        THEN 'Anomaly'
        ELSE 'Normal'
    END AS glucose_status
FROM interstitial_glucose g
JOIN glucose_stats s ON
    time_bucket('1 hour', g.ts) = s.hour AND
    g.participant_id = s.participant_id
```

```
WHERE ABS(g.glucose_value - s.mean_glucose) > 2 * s.std_glucose;
```

Rolling average analysis

Analyzing rolling average using window functions over time-series heart rate data (useful for trend analysis):

```
SELECT
    hrd.participant_id,
    time_bucket('1 hour', hrd.datetime) AS time_bucket,
    hrd.datetime,
    hrd.hr,
    AVG(hrd.hr) OVER (
        PARTITION BY hrd.participant_id
        ORDER BY hrd.datetime
        RANGE BETWEEN INTERVAL '1 hour' PRECEDING AND CURRENT ROW
    ) AS rolling_avg_hr
FROM heart_rate_data hrd
ORDER BY time_bucket, hrd.datetime;
```

Most Active Interval of each day

Identifying the most Active Interval of each day using accelerometer data:

```
WITH activity_intervals AS (
    SELECT
        participant_id,
        time_bucket('5 minutes', ts) AS time_interval,  -- 5-minute intervals for activity analysis
        DATE(ts) AS activity_date,  -- Extract the date part from timestamp for daily grouping
        SUM(SQRT(acc_x^2 + acc_y^2 + acc_z^2)) AS total_activity
    FROM accelerometer_data
    GROUP BY participant_id, activity_date, time_interval
)
SELECT
    participant_id,
    activity_date,
    time_interval,
    total_activity
FROM activity_intervals
WHERE (participant_id, activity_date, total_activity) IN (
    SELECT participant_id, activity_date, MAX(total_activity)
    FROM activity_intervals
```

```
    GROUP BY participant_id, activity_date
)
ORDER BY participant_id, activity_date, time_interval;
```

Stress detection

Identifying the stress of participants using heart rate data and electrodermal activity data:

```
WITH aggregated_heart_rate AS (
    SELECT
        participant_id,
        time_bucket('1 hour', datetime) AS time_bucket,
        AVG(hr) AS avg_heart_rate
    FROM heart_rate_data
    GROUP BY participant_id, time_bucket
),
aggregated_eda AS (
    SELECT
        participant_id,
        time_bucket('1 hour', ts) AS time_bucket,
        AVG(eda) AS avg_eda
    FROM electrodermal_activity
    GROUP BY participant_id, time_bucket
)
SELECT
    hr.participant_id,
    hr.time_bucket,
    hr.avg_heart_rate,
    eda.avg_eda,
    CASE
        WHEN hr.avg_heart_rate > 100 AND eda.avg_eda > 2 THEN 'Stress'
        ELSE 'Normal'
    END AS stress_status
FROM aggregated_heart_rate hr
JOIN aggregated_eda eda
    ON hr.participant_id = eda.participant_id
    AND hr.time_bucket = eda.time_bucket
ORDER BY hr.time_bucket;
```

Interpolation and then aggregation

Interpolation on the biggest data and then aggregation. This query processes blood volume pulse data of the period 1st February 2020 till 21st February 2020, interpolates missing values into 1-second intervals, and calculates smoothed averages and standard deviation metrics for each participant:

```
WITH interpolated_data AS (
    SELECT
        participant_id,
        time_bucket_gapfill(
            '1 second',
            ts,
            TIMESTAMP '2020-02-01 00:00:00',
            TIMESTAMP '2020-02-20 23:59:59'
        ) AS bucketed_time, -- Specify your desired time range explicitly as TIMESTAMP
        interpolate(avg(bvp)) AS interpolated_bvp
    FROM blood_volume_pulse
    WHERE ts BETWEEN TIMESTAMP '2020-02-01 00:00:00' AND TIMESTAMP '2020-02-20 23:59:59'
    GROUP BY participant_id, time_bucket_gapfill(
        '1 second',
        ts,
        TIMESTAMP '2020-02-01 00:00:00',
        TIMESTAMP '2020-02-20 23:59:59'
    )
)
SELECT
    participant_id,
    AVG(interpolated_bvp) AS smooth_bvp,
    STDDEV(interpolated_bvp) AS bvp_variation
FROM interpolated_data
GROUP BY participant_id;
```

Exercise detection

Detection of exercising timestamps using accelerometer data and heart rate data:

```
WITH movement_activity AS (
    SELECT
        participant_id,
        ts,
        GREATEST(ABS(acc_x), ABS(acc_y), ABS(acc_z)) AS max_acc,
```

```
        CASE
            WHEN GREATEST(ABS(acc_x), ABS(acc_y), ABS(acc_z)) > 70 THEN 1
            ELSE 0
        END AS is_exercise
    FROM
        accelerometer_data
    WHERE
        acc_x IS NOT NULL AND acc_y IS NOT NULL AND acc_z IS NOT NULL
),
heart_rate_activity AS (
    SELECT
        participant_id,
        datetime AS ts,
        hr,
        CASE
            WHEN hr > 120 THEN 1  -- Threshold for exercise heart rate
            ELSE 0
        END AS is_exercise
    FROM
        heart_rate_data
    WHERE
        hr IS NOT NULL
),
combined_activity AS (
    SELECT
        COALESCE(m.participant_id, h.participant_id) AS participant_id,
        COALESCE(m.ts, h.ts) AS ts,
        COALESCE(m.max_acc, 0) AS max_acc,
        COALESCE(h.hr, 0) AS hr,
        COALESCE(m.is_exercise, 0) + COALESCE(h.is_exercise, 0) AS exercise_score
    FROM
        movement_activity m
    FULL OUTER JOIN
        heart_rate_activity h
    ON
        m.participant_id = h.participant_id AND m.ts = h.ts
),
exercise_detection AS (
    SELECT
        participant_id,
```

```
        ts,
        max_acc,
        hr,
        exercise_score,
        CASE
            WHEN exercise_score >= 2 THEN 'Exercise'
            ELSE 'Rest'
        END AS activity_state
    FROM
        combined_activity
)
SELECT
    participant_id,
    ts,
    max_acc,
    hr,
    activity_state
FROM
    exercise_detection
WHERE
    activity_state = 'Exercise'
ORDER BY
    participant_id, ts;
```

Bibliography

[Ben+21] Bent, B. et al. (2021). *Engineering digital biomarkers of interstitial glucose from noninvasive smartwatches*. https://doi.org/10.1038/s41746-021-00465-w. [Accessed 19-12-2024].

[Boo22] Booz, Ryan (2022). *Timescale Tips: Testing Your Chunk Size for Enhanced PostgreSQL Performance*. Accessed: 2024-12-15. URL: https://www.timescale.com/blog/timescale-cloud-tips-testing-your-chunk-size/.

[Bug24] Bugazelle (2024). *GitHub - Bugazelle/export-csv-to-influx: A Tool for Loading CSV Data into InfluxDB*. https://github.com/Bugazelle/export-csv-to-influx. [Accessed 19-12-2024].

[Gra24] Grafana (2024). *Grafana OSS and Enterprise | Grafana documentation — grafana.com*. https://grafana.com/docs/grafana/latest/. [Accessed 16-12-2024].

[Inf24a] InfluxData (2024a). *Getting Started with InfluxDB and Grafana | InfluxData — influxdata.com*. https://www.influxdata.com/blog/getting-started-influxdb-grafana/. [Accessed 16-12-2024].

[Inf24b] — (2024b). *GitHub - influxdata/influxdb-python: The Official Python Client for InfluxDB*. https://github.com/influxdata/influxdb-python. [Accessed 19-12-2024].

[Inf24c] — (2024c). *In-memory indexing and the Time-Structured Merge Tree (TSM)*. https://docs.influxdata.com/influxdb/v1/concepts/storage_engine/. [Accessed 16-12-2024].

[LSK23] Lee, Sangmyung, Yongseok Son, and Sunggon Kim (2023). "Analyzing I/O Characteristics of Time-Series Data Using High Performance Storage Devices". In: *IEEE Access*.

[Sta18] Staller, Andrew (2018). *3,400+ stars, 20+ releases, 30+ talks, and production deployments worldwide: Here's a recap of Timescale's first year*. Accessed: 2024-12-15. URL: https://www.timescale.com/blog/timescaledb-2017-recap-aa9b593e10cf/.

[Tim21] Timescale (2021). *TimescaleDB vs. InfluxDB: Purpose-built for time-series data — timescale.com*. https://www.timescale.com/blog/timescaledb-vs-influxdb-for-time-series-data-timescale-influx-sql-nosql-36489299877. [Accessed 16-12-2024].

[Tim24a] Timescale (2024a). *GitHub - timescale/tsbs: Time Series Benchmark Suite, a tool for comparing and evaluating databases for time series data — github.com*. https://github.com/timescale/tsbs. [Accessed 16-12-2024].

[Tim24b] — (2024b). *Timescale Documentation*. Accessed: 2024-12-16. URL: https://docs.timescale.com/.

[Tim24c] — (2024c). *Timescale Documentation | Use Grafana to visualize geospatial data — docs.timescale.com* https://docs.timescale.com/use-timescale/latest/integrations/observability-alerting/grafana/geospatial-dashboards/. [Accessed 16-12-2024].