COMPARATIVE ANALYSIS OF DATABASE MANAGEMENT SYSTEMS FOR TIME SERIES DATA

BY

AJALA ONOSHOZE AJIAKE
PSC2209242

PROJECT WORK SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE,
FACULTY OF COMPUTING,
UNIVERSITY OF BENIN, BENIN CITY,
EDO STATE, NIGERIA.

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE AWARD OF BACHELOR OF SCIENCE (B.Sc.) DEGREE IN COMPUTER SCIENCE

NOVEMBER 2025

CERTIFICATION

This is to certify that this project work was carried out by **AJIAKE AJALA ONOSHOZE** with Matriculation Number **PSC2209242** under my supervision. It is adequate and satisfactory, both in scope and content, for the award of Bachelor of Science (B.Sc.) Degree in Computer Science of the University of Benin.

_____ _____

Dr. MAXWELL S. U. OSAGIE DATE

Project supervisor

APPROVAL

This project work is hereby approved in partial fulfilment of the requirements for the award of Bachelor of Science (B.Sc.) Degree in Computer Science from the University of Benin.

_____ _____

Dr. (Mrs.) A.R. USIOBAIFO DATE

Head of Department

DEDICATION

This project is dedicated to Almighty God for His infinite wisdom, guidance, and strength throughout this journey. It is also dedicated to my loving parents Mr. and Mrs. Ajiake and siblings Master Ajiake Jemmy , Master Ajiake Elamheke, Master Nasiru and Miss Joy whose selfless love and dedication has shaped me to the person I am today. You have been my pillar of strength and inspiration. To my friends and well-wishers, thank you for your encouragement, understanding, unwavering support and encouragement. Your positive energy and constant support have made this journey a fulfilling one. This work is a reflection of collective strength and inspiration I have drawn from each of you.

Thank you all

ACKNOWLEDGEMENTS

My profound gratitude goes to Almighty God for His grace and mercy throughout my academic pursuit. I extend heartfelt thanks to my supervisor, **Dr. Maxwell S.U. Osagie**, for his continuous guidance, constructive feedback, and mentorship during the course of this project. Special appreciation also goes to **Dr. (Mrs.) Usiobaifo**, the Head of Department for her outstanding leadership and support.

I sincerely appreciate other lecturers and staff of the Department of Computer ScienceUniversity of Benin, who I have been opportune to cross paths with, and have impacted me immensely these past few years: Dr. (Mrs.) L.O.Usiosefe, Dr. (Mrs.) Aziken, Mr J. Okhuoya, Mr. I.E. obayagbonna ,Prof. G.O. Ekuobase, Dr. F.O. Oliha, Prof. (Mrs.) V.V.N. Akwukwuma, Prof. F.I. Amadin, Prof. (Mrs.) V.I. Osubor, Dr. F.O. Chete, Dr. (Mrs) R.O. Osaseri, Mr. I.E. Obasohan, Mr. S.O.P. Oliomogbe, Mrs. Ophorie, Mrs. Chukwuma, Mr. Odetayo,Mr. K.O. Otokiti, Dr. E.C. Igodan, Dr. E. Nweli and Mr. D.N. Idehen.

Finally, I wish to thank my family for their immense contribution to my academic growth. I also want to appreciate those who contributed to the success of this project: Agofure Daniel, Adeniyi Ayooluwa Anuoluwa, Purity Usiemwanta, Ogaga Daniel, James Efosa, Master God-Do-Well, Uwabor Kelvin Osakwe and Ohenhen Favour for their prayers, words of encouragement, and love which kept me going throughout this journey.

TABLE OF CONTENTS

ABSTRACT

Time series data consists of information collected over time, often at regular intervals, which can quickly accumulate in large volumes. To analyze and present this data effectively, it needs to be stored in a way that is easy to access. Database Management Systems (DBMSs) are commonly used for this purpose. There are various types of DBMSs, each with distinct advantages and disadvantages that involves different trade-offs regarding their features. In this study, we compare the performance of two different DBMSs for managing time series data. The first is PostgreSQL, a widely used relational DBMS enhanced by the Timescale DB extension designed for time series. The second is Mongo DB, a leading NoSQL system that offers built-in support for time series data. Investigation was made to ascertain which of these DBMS is better suited for time series data by analyzing their query execution times. Two databases were setup using sample time series data-specifically, publicly available weather. A series of test queries were conducted simulate real world scenarios and measure their execution times

The results are analyzed on a query-by-query basis to highlight performance between two systems, with PostgreSQL outperforming MongoDB on some queries (by over two orders of magnitude) while MongoDB was faster on others (by more than 30 times in one case). It was concluded that certain queries and their related real-world applications may favor one DBMS over the other, based on how well the query structure aligns with the system's strength. Additional factor was discussed that may affect each DBMSs query execution efficiency and consider potential improvements.

CHAPTER ONE
INTRODUCTION

1.1 BACKGROUND OF STUDY

The relentless passage of time is a fundamental aspect of reality that has influenced human existence for ages. Naturally, this leads to an interest in how various elements evolve over time. To analyze processes, especially temporal ones, we need to gather data. This enables us to examine dependencies, look for correlations, and untimely draw conclusions. Data can be approached in various ways. Database comes in when it involves storing data for future use. The main role of a database is to store data, whether in physical or digital form. While a database can be as simple as handwritten notes, today it usually refers to digital storage on a server. To interact with this digital data, Database Management Systems (DBMS) are used (Makris et al., 2019).

Databases and database technology have a major impact on the growing use of computers. It is fair to say that databases play a critical role in almost all areas where computers are used, including business, electronic commerce, engineering, medicine, genetics, law, education, and library science. The word database is so commonly used that we must begin by defining what a database is. A database is a collection of related data. By data, we mean known facts that can be recorded and that have implicit meaning. For example, consider the names, telephone numbers, and addresses of the people you know. You may have recorded this data in an indexed address book or you may have stored it on a hard drive, using a personal computer and software such as Microsoft Access or Excel. This collection of related data with an implicit meaning is a database. The preceding definition of database is quite general; for example, we may consider the collection of words that make up this page of text to be related data and hence to constitute a database. However, the common use of the term database is usually more restricted. A database has the following implicit properties:

i. A database represents some aspect of the real world, sometimes called the miniworld or the universe of discourse (UoD). Changes to the miniworld are reflected in the database.
ii. A database is a logically coherent collection of data with some inherent meaning. A random assortment of data cannot correctly be referred to as a database.
iii. A database is designed, built, and populated with data for a specific purpose. It has an intended group of users and some preconceived applications in which these users are interested. In

other words, a database has some source from which data is derived, some degree of interaction with events in the real world, and an audience that is actively interviewed.

The end users of a database may perform business transactions (for example, a customer buys a camera) or events may happen (for example, an employee has a baby) that cause the information in the database to change. In order for a database to be accurate and reliable at all times, it must be a true reflection of the miniworld that it represents; therefore, changes must be reflected in the database as soon as possible. A database can be of any size and complexity. For example, the list of names and addresses referred to earlier may consist of only a few hundred records, each with a simple structure. On the other hand, the computerized catalog of a large library may contain half a million entries organized under different categories by primary author's last name, by subject, by book title with each category organized alphabetically.

A database of even greater size and complexity is maintained by the Internal Revenue Service (IRS) to monitor tax forms filed by U.S. taxpayers. If we assume that there are 100 million taxpayers and each taxpayer files an average of five forms with approximately 400 characters of information per form, we would have a database of 100 × 106 × 400 × 5 characters (bytes) of information. If the IRS keeps the past three returns of each taxpayer in addition to the current return, we would have a database of 8 × 1011 bytes (800 gigabytes) (Elmasri & Navathe.,2019). This huge amount of information must be organized and managed so that users can search for, retrieve, and update the data as needed. An example of a large commercial database is Amazon.com. It contains data for over 20 million books, CDs, videos, DVDs, games, electronics, apparel, and other items. The database occupies over 2 terabytes (a terabyte is 1012 bytes worth of storage) and is stored on 200 different computers (called servers). About 15 million vistors access Amazon.com each day and use the database to make purchases.

The database is continually updated as new books and other items are added to the inventory and stock quantities are updated as purchases are transacted. About 100 people are responsible for keeping the Amazon database up-to-date. A database may be generated and maintained manually or it may be computerized. For example, a library card catalog is a database that may be created and maintained manually. A computerized database may be created and maintained either by a group of application programs written specifically for that task or by a database management system. We are only concerned with computerized databases in this book. A database management system (DBMS) is a collection of programs that enables users to create

and maintain a database. The DBMS is a general-purpose software system that facilitates the processes of defining, constructing, manipulating, and sharing databases among various users and applications. Defining a database involves specifying the data types, structures, and constraints of the data to be stored in the database. The database definition or descriptive information is also stored by the DBMS in the form of a database catalog or dictionary; it is called meta-data. Constructing the database is the process of storing the data on some storage medium that is controlled by the DBMS. Manipulating a database includes functions such as querying the database to retrieve specific data, updating the database to reflect changes. Databases and Database Users miniworld, and generating reports from the data. Sharing a database allows multiple users and programs to access the database simultaneously. An application program accesses the database by sending queries or requests for data to the DBMS. A query2 typically causes some data to be retrieved; a transaction may cause some data to be read and some data to be written into the database. Other important functions provided by the DBMS include protecting the database and maintaining it over a long period of time. Protection includes system protection against hardware or software malfunction (or crashes) and security protection against unauthorized or malicious access. A typical large database may have a life cycle of many years, so the DBMS must be able to maintain the database system by allowing the system to evolve as requirements change over time (Elmasri & Navathe.,2019).

There are different types of databases, with relational databases being the traditional option. These are designed based on the relational model and are accessed using Structured Query Language (SQL). Recently, No Structured Query Language (NoSQL) systems have gained traction, particularly document databases, which are mostly used for time series data, which is the focus of this research. Time series data consists of measurements taken over time, each linked to a timestamp. These measurements can be periodic, like weather data collected at regular intervals, or irregular, such as the number of active streamers on a platform or a computer's CPU usage (Gyorodi et al., 2015).

Differences Between SQL and NoSQL Databases:

a) SQL databases use a relational data model while NoSQL databases use a variety of data models such as document-oriented, key-value and graph databases.
b) SQL databases require a predefined schema while NoSQL databases are schema-less.

NoSQL databases are designed to scale horizontally while SQL can become bottlenecked as the volume of data increases.

c) SQL databases use SQL while NoSQL databases use a variety of query languages, such as MongoDB's query language.

When to use each Database:

a) SQL databases are used for applications that require complex transactions, strict data consistency and adherence to a predefined schema. Example include financial system and web applications with complex data relationships.

b) NoSQL databases are used for applications that require flexible data modeling, high scalability and high performance. Examples include big data analytics real-time web applications (Stonebraker, M.)

As noted by Atzori et al., (2010) Time series data consists of information that is gathered and logged over a period of time, with each entry linked to the specific time it was recorded. This type of data is used in various applications. Different database management systems (DBMS) balance trade-offs across factors like runtime performance, resource consumption, development speed, and ease of iteration.

Some DBMS provides specialized extensions to handle time series better. NoSQL databases, in particular, have become a popular alternative to traditional relational databases, especially in scenarios where normalized data is not required, and other factors drive the selection of the DBMS. Managing information relate to time in a database is a vital requirement in many information systems. The capacity to model time-oriented information is crucial in a myriad of computer applications in domains such as finance, economic, banking, medical records, reservation systems, and scientific analysis. Time series data represent a type of data heavily reliant on the time element. Everyday large volumes of data are collected in the form of time series. Time series are collections of events or observations, predominantly numeric in nature, sequentially recorded on a regular or irregular time basis. Time series are becoming increasingly important in nearly every organisation and industry such as banking, finance, telecommunication, and transportation. Banking institutions, for instance, rely on the analysis of time series for forecasting economic indices, elaborating financial market models, and registering international trade operations. More and more time series are being used and becoming a valuable resource in today's organizations wanting to manage such things as product development, manufacturing, product and service sales, customer

services, and many others real world events (the list of applications is too numerous to name). Due to the massive volume of data, intensive use, and intrinsic characteristics such as time granularities (e.g. scaling from seconds to minutes to daily to monthly), transformation of the original time series data (e.g. rates of growth, transformation of nominal to real data, moving average), it is essential that time series data are well organized and have solid data management procedures. In theory, such data are processed, analyzed, disseminated, and presented. However, many institutions face some difficult issues in organizing such a vast amount of data (Matus-Castillejos & Jentzsch, 2004). In fact, the more organizations have dealt with data structured in the form of time series, the more inadequacies and drawbacks of current database management systems (DBMSs) for managing this kind of data have been identified (Matus-Castillejos & Jentzsch, 2004). Therefore, the streamlines the process of choosing between various database systems for storing time series data paramount importance.

1.2 MOTIVATION OF RESEARCH

Time series data is crucial in various domains, such as finance, IoT, healthcare and climate monitoring. A comparative analysis of DBMS for time series can help identify the most suitable solutions for specific use cases. Several DBMS options are available for time series data, each with its strengths and weaknesses. A comparative analysis can help evaluate these options and inform decision-making. Time series data can be massive, and DBMS performance and scalability are critical factors in handling large datasets. A comparative analysis can help identify the most efficient solutions.

1.3 RESEARCH PROBLEM

Gathering time series data can rapidly result in large datasets, particularly when there are frequent measurements or multiple readings per time stamp. In most cases, handling of these extensive datasets requires large storage capacity, large processing power for querying the data and lack of these features often causes degradation of the DBMS performance due to complex queries which results to data inconsistencies. Hence, a comparative analysis of a relational DBMS and document DBMS helps to evaluate and provide the benefits for the selection of an appropriate DBMS for manipulating and processing time series data.

1.4 AIM AND OBJECTIVES

The aim of this project is to simplify the selection process among different database systems for storing time series data while providing a solid foundation for decision-making. To accomplish this, the following objectives are considered:

i. Data Collection: To evaluate DBMSs under review, data would be obtained from weather APIs. This dataset includes time series of various variables, measured at hourly, tri-daily, or quadri-daily intervals, depending on the data's recency.

ii. Data Storage: A relational database and a NoSQL database would be implemented to store the data, creating an appropriate relational model and a document structure. A series of queries, reflecting typical real-world operations would be executed on both systems to access the practical impact of each approach.

iii. Result Analysis: Performance would be evaluated by measuring the time taken for each operation (in "wall clock" time) and the results would be analyzed to identify which system performed better in specific areas.

1.5 RESEARCH METHODOLOGY

This project adopts a positivist approach with an experimental focus. The experimental design draws from current research on database systems, providing a theoretical framework. The performance of both systems is assessed through benchmarking, measuring runtimes, thus emphasizing a quantitative approach. By comparing the two DBMS, it is aimed to highlight their trade-offs and provide a reference point for evaluating each system, facilitating a clearer analysis.

1.6 SCOPE OF RESEARCH

The scope of this research focuses on comparing one relational DBMS with a time series extension and one NOSQL DBMS, specifically excluding other type database systems from the analysis. The comparison is limited to periodic weather data, rather than other forms of time series data. Data collection for measurements and performance is restricted to a predefined set of queries. The research solely addresses the performance of the two DBMSs, without considering other factors that may influence the choice of a suitable DBMS for a given application.

1.7 SIGNIFICANCE OF STUDY

This project is very important as it streamlines the process of choosing between various database systems for storing time series data, offering a strong basis for informed decision-making.

CHAPTER TWO
LITERATURE REVIEW

2.1 DATABASE EVOLUTION

The history of database management systems (DBMSs) reflects the continuous effort to balance efficiency, flexibility, and scalability in handling data. From hierarchical file-based models to modern specialized time series databases, the evolution of DBMSs has been shaped by both technological advances and emerging application requirements.

Early Database Systems: Hierarchical and Network Models

The earliest DBMSs emerged in the 1960s, using hierarchical and network models. Hierarchical databases, such as IBM's Information Management System (IMS), organized data in tree-like structures, while network models (e.g., CODASYL DBTG) allowed more flexible many-to-many relationships through record sets. These systems were efficient for their time but had limited data independence and tightly coupled application logic to physical data structures (Elmasri & Navathe, 2011).

The Relational Model Revolution

In 1970, E. F. Codd introduced the relational model, which represented data as tables (relations) and separated logical design from physical storage. This innovation enabled ad hoc querying, greater program–data independence, and the use of high-level query languages such as SQL. By the 1980s, relational DBMSs (RDBMSs), including Oracle, DB2, and Ingres, became dominant, displacing hierarchical and network systems in most business applications (Elmasri & Navathe, 2011).

The relational model's strong theoretical foundation also facilitated query optimization and standardization, making it the most widely adopted approach to date. However, traditional RDBMSs struggled with performance in applications requiring complex data types or high-frequency time-dependent data.

Object-Oriented and Object-Relational Extensions

The 1980s and 1990s saw the rise of applications that required managing complex data such as images, engineering designs, and multimedia. This led to Object-Oriented DBMSs (OODBMSs), which introduced features like inheritance, encapsulation, and object identity. Although OODBMSs had limited commercial adoption, their principles influenced Object Relational

DBMSs (ORDBMSs). ORDBMSs, such as PostgreSQL and Oracle, integrated relational foundations with object features, supporting user-defined data types and temporal extensions (Elmasri & Navathe, 2011).

Specialized Databases and Temporal Extensions

The growing importance of temporal and sequential data in fields like finance, telecommunications, and environmental monitoring highlighted the limitations of traditional RDBMSs. Time series data was initially managed in sequential files or external analytical tools, but these lacked the advantages of a DBMS such as concurrency control, indexing, and declarative querying. To address this, systems introduced temporal Time Cartridge and Informix's TimeSeries DataBlade along with research-oriented proposals like the TSQL2 temporal query language (Snodgrass, 1995).

Modern Database Ecosystem

Today's database ecosystem reflects diversification and specialization:

I. Relational and Object-Relational DBMSs (e.g., Oracle, PostgreSQL, SQL Server) remain dominant in enterprises.
II. NoSQL Databases (e.g., MongoDB, Cassandra) support scalability and schema flexibility for unstructured and semi-structured data.
III. Time Series Databases (e.g., InfluxDB, TimescaleDB, OpenTSDB) are optimized for storing, querying, and aggregating temporal data at scale.
IV. Distributed and Cloud Databases provide global availability and fault tolerance across hybrid infrastructures.

This progression underscores the move from general-purpose DBMSs to specialized systems tailored to emerging workloads, with time series DBMSs becoming critical for real-time analytics in domains like IoT, finance, and industrial monitoring.

2.1.1 DATABASE MANAGEMENT SYSTEMS

Database Management Systems (DBMS) are software tools designed to handle and support databases. They provide an interface that allows users or applications to interact with databases, enabling actions like adding, modifying, and retrieving data.

Through features like querying, updating, and generating reports, users can manipulate the

database to reflect changes in the represented environment. DBMSs also support multi-user access, allowing several users and applications to work with the database simultaneously (Elmasri &Navathe, 2019).

A DBMS defines and organizes the data types, structures and constraints, storing this descriptive information known as "metadata" in a dictionary. Security features within DBMS helps to manage data integrity and restrict access, ensuring that only authorized users can view or alter the information.

Since large databases may be used for many years, DBMS are built to accommodate evolving requirements over time. To meet the needs of various applications and users, many DBMSs offer additional functions through extensions.

2.1.2 Types of databases

Databases come in various forms, such as relational databases, object-oriented databases, NoSQL databases, and graph databases. Each has a distinct structure and serves a specific purpose, with unique strengths and weaknesses that makes them better suited for certain applications over other. The comparison between relational and NoSQL systems has been ongoing for more than a decade. Relational systems, such as MySQL and PostgreSQL, are valued for strong consistency and adherence to SQL standards (Maier, 2019). In contrast, NoSQL systems like MongoDB, Cassandra, and HBase prioritize scalability and flexibility of data structures (Choi, 2014).

An influential comparison by researchers tested Cassandra, MongoDB, HBase, and MySQL, finding that no single database was consistently superior; performance varied with dataset size and type of workload (Gyorodi et al., 2015). MySQL was often faster for standard queries, while NoSQL databases showed advantages in insertion-heavy tasks (Jose & Abraham, 2020). These findings underline the importance of matching the DBMS to the workload.

Relational databases

A relational database organizes structured data within tables, with each table representing a specific entity in the database's domain. Each table is made up of rows and columns, where each row corresponds to a single record, and each column represents an attribute of that record. This layout minimizes redundancy by preventing duplicate data storage. Beyond the data itself, a relational database also contains metadata known as the database schema that defines the structure of the data. This schema provides information about tables, columns, data types, and relationships

between tables. Schema design is typically handled by database administrators, or by the authors along with company supervisors.

All relational databases use Structured Query Language (SQL), a specialized language created for data queries, earning them the name SQL systems.

NoSQL databases

With the arrival of the 21st century, various new applications have become part of daily life. Social media, e-commerce, cloud storage and other technologies have dramatically increased the global volume of stored data and servers to unimaginable levels. Much of this data is not just textual or numeric, it includes diverse types such as images, documents and videos. Traditional relational databases are insufficient for managing and storing such vast and varied data types leading to the development of NoSQL systems. Unlike relational database, which use SQL, NoSQL databases use different and non-exclusive query languages. These systems employ different data models, including document-based, graph-based, column-based, and key-value models (Stonebraker, 2019).

2.1.3 DBMS Qualities

When assessing database systems, they can be evaluated across several key criteria. Key performance metrics include response time, throughput, and scalability. Response time or latency is the delay between submitting a query to the database and receiving the result. Through measures how many queries the database can process within a set time frame, or sometimes the volume of data (counted in records or bytes) it can handle within that period for single or multiple queries.

Scalability refers to the database's capability to maintain strong performance as data volume or the simultaneous users grows. This also includes support for distributed systems that distribute the workload across multiple physical servers (Elmasri & Navathe 2019).

2.2 Time Series Data

Time series data, often referred to as time-stamped data, is a sequence of values indexed in time order. Time stamping refers to data collected at various times, where each value is time stamped. These data points are usually gathered from the same source and are used to measure progress over time (Jeff, 2022).

Time series data have applications in fields such as medical surveillance, financial analysis, gene expression studies and weather forecasting. Given their importance, significant efforts have been made to optimize their use, including the development of database systems (DBMS) specifically

designed for time series data. These systems can significantly improve performance in terms of speed and storage by leveraging the unique characteristics of time series data.

One such characteristics of is the large volume of record (Rhea & Wong, 2017), as many users require histories or data sampled at very fine temporal resolutions, such as climate measurements spanning centuries or sensor data collected at tens of kHz.

Time series data is typically recorded and stored sequentially, following an append-only write pattern. Reads are often clustered by time, as users frequently retrieve the most recent records of specific time intervals. This differs from typical DBMS usage where data is inserted and read randomly and write operations often involves updates and deletions. Time series data often contains redundancy with successive measurements sometimes repeating values or differing only slightly.

2.3 Selected DBMSs for Implementation

For this study, two specific DBMSs are chosen to implement time series database for comparison. One is a relational database enhanced with a time series-specific extension, while the other one is a NoSQL document database designed with built-in-time series capabilities. Details about these two selections are provided below.

2.3.1 PostgreSQL Timescale extension

PostgreSQL, often called postgres, is a well-established and widely adopted open-source RDBMS, serving as the default database for many DBMS-based applications (www.postgresql.org). Its compatibility with all major operating systems, strong adherence to the SQL standard, and stability have made it highly popular. Designed to be extensible, postgreSQL allows third-party code to enhance its functionality.

One such enhancement is TimescaleDB, a time series extension built to manage large datasets while maintaining high performance and scalability. TimescaleDB integrates seamlessly with postgreSQL, enabling users to interact with and query data without changes to their workflows. It introduces hypertables, which act as table views for time series data. These hypertables are underpinned by partitioned tables (referred to as chunks), each corresponding to a specific time interval. This partitioning mechanism divides data into smaller, more manageable segments, and these chunks can also be distributed across different nodes based on the time and other data attributes, facilitating horizontal scaling of the database.

This design focuses on facilitating efficient data querying by leveraging the typical access patters of time series data, where information is often retrieved in clusters based on time or similar parameters. It removes the need to manually modify the database schema or develop more complex queries.

2.3.2 MongoDB

MongoDB is a document-oriented NoSQL database management system that stores data in JSON-like documents with dynamic schemas, meaning document structures are not predefined. It supports nested subdocuments and arrays, providing more flexibility for modeling relationships between data compared to relational databases.

Like postgreSQL, MongoDB is open-source (https://github.com/mongodb/mongo), and has seen broad adoption with the rise of NoSQL applications. Its collections, analogous to tables in relational databases, can be tailored to enhance performance for specific use cases. One notable feature is its built-in collection type optimized for time series data, which helps save disk space and boost query performance (Elamasri & Navathe 2019).

These time series collections are implemented as views over internal collections, leveraging columnar storage and using time as a clustered index, similar to the TimescaleDB extension for PostgreSQL.

MongoDB includes a powerful querying system called the aggregation framework, where queries consist of multiple stages that transform a document collection into a new result set, referred to as an aggregate query.

2.4 Related Work

For many years, comparative analysis of database management systems (DBMSs) has been a key area of research, with the goal of identifying the best system for different use cases. As technology evolves and new data management challenges arise, the need for updated comparisons continues. These comparisons typically address a wide range of topics, from evaluating performance across various workloads to assessing scalability, availability and cost-effectiveness of different architectures.

Many studies have been conducted to examine the performance and features of different DBMSs, often focusing on specific data types, such as spatial-temporal or time series data, to determine which system is best suited for those use cases. Other studies offer broader evaluations, considering factors such as scalability, security and usability.

The studies discussed in this section focus on the performance aspects of DBMSs, including response times and execution speed. Despite their varied objectives and evaluation metrics, these studies share the common goal of identifying the most efficient DBMS for specific scenarios.

MongoDB vs MySQL

Several studies have zoomed in on MongoDB compared to MySQL. Gyorodi et al. (2015) conducted experiments that showed MongoDB outperforming MySQL across most tested operations. Similarly, Jose and Abraham (2020) found MongoDB to be more efficient in handling varied workloads, particularly those requiring flexible schema designs. These consistent results suggest that MongoDB can offer significant advantages where schema flexibility and fast write operations are important.

MongoDB vs PostgreSQL

Direct comparisons between MongoDB and PostgreSQL have produced mixed results. Jung en atand Choi (2016) reported that MongoDB was faster for most queries, except for SELECT operations, where PostgreSQL performed better. On the other hand, Makris et al., (2019, 2021) carried out spatio-temporal benchmarks that favored PostgreSQL, especially when the PostGIS extension was enabled. They emphasized the critical role of proper indexing in performance, noting that MongoDB without efficient indexes lagged behind PostgreSQL.

These contrasting outcomes show that while MongoDB excels in unstructured data handling, PostgreSQL with the right extensions can outperform it in structured or geo-temporal scenarios.

Dedicated Time Series Databases

Beyond general-purpose systems, specialized solutions for time series have been developed. TimescaleDB, an extension of PostgreSQL, introduces hypertables and partitioning to efficiently manage large-scale time series data (Makris, 2019). Similarly, MongoDB introduced native time series collections using columnar storage and clustered indexes (Makris, 2019).

Grzesik and Mrozek (2020) compared PostgreSQL, SQLite, TimescaleDB, InfluxDB, and Riak TS on a Raspberry Pi platform for IoT use cases. Their results revealed PostgreSQL as the fastest for data ingestion, while InfluxDB and PostgreSQL alternated as the best for aggregation queries. Interestingly, TimescaleDB did not outperform standard PostgreSQL in their tests, suggesting that hardware and workload context matter as much as engine design.

Methodological Considerations in Benchmarking

Benchmarking DBMSs is a complex task. Saavedra and Smith (2020) highlighted how hardware factors, such as cache and memory management, can affect performance results. Raasveldt et al., (2018) warned of pitfalls in unfair database testing, including inconsistent configurations, omission of warm-up runs, and inappropriate query translations. These lessons stress the need for fair experimental setups when comparing MongoDB and PostgreSQL for time series data.

Broader Perspectives

Some authors also consider higher-level access methods. Botoeva et al., (2017) studied ontology-based data access (OBDA) in non-relational databases, including MongoDB. This approach allows uniform query layers over heterogeneous data, showing that beyond raw speed, usability and integration features are also essential in DBMS selection.

Synthesis of Findings

From the reviewed works, three broad conclusions can be drawn:

1. No universal best system. Performance depends heavily on workload type, indexing, and query patterns. For insertion-heavy and schema-flexible tasks, MongoDB often excels (Gyorodi et al., 2015; Jose & Abraham, 2020). For structured queries and spatio-temporal workloads, PostgreSQL (especially with PostGIS or TimescaleDB) tends to lead (Makris et al., 2019, 2021).
2. Time series optimizations matter. Extensions like TimescaleDB's hypertables and MongoDB's time series collections are designed for append-heavy and time-bounded queries. However, their benefits vary across platforms and workloads (Grzesik & Mrozek, 2020).
3. Benchmark fairness is crucial. Without careful control of experimental conditions, comparisons can be misleading (Saavedra & Smith, 2020; Raasveldt et al., 2018).

Conclusion

The literature shows that the choice between MongoDB and PostgreSQL (with TimescaleDB) is not straightforward. Each system has strengths in specific contexts, and results are influenced by indexing strategies, workload types, and experimental design. For students and researchers, the lesson is to carefully define workloads and performance metrics before selecting a DBMS.

2.4.1 Comparisons of relational and NoSQL DBMSs

Several studies have compared the performance and features of relational and NoSQL DBMSs. One study analyzed the performance of three NoSQL systems (cassadra, MongoDB, and HBase)

alongside a relational system (MySQL) under various workloads, emphasizing the trade-off between throughput and latency. The findings indicated that performance varied based on dataset size, workload characteristics, and throughput demands. MySQL generally matched or outperformed the NoSQL systems, except in insertion-intensive scenarios, were NoSQL databases showed an advantage.

Two other studies conducted by Gyorodi et al. and Jose et al. examined MongoDB and MySQL, with both concluding that MongoDB consistently achieved superior performance across all evaluated scenarios.

2.4.2 MongoDB Vs PostgreSQL Performance

Several studies have focused on the same DBMSs discussed in this work, namely MongoDB and PostgreSQL.

Jung et al. (2015) analyzed the execution times of different basic queries on MongoDB and PostgreSQL with datasets of varying sizes. Their results showed that MongoDB generally performed better than PostgreSQL for most query types, except for SELECT queries, where PostgreSQL matched or exceeded MogoDB's performance.

In contrast, Makris et al. (2019) and a subsequent study investigated the performance of MongoDB and PostgreSQL (with the PostGIS extension) for spatiotemporal data operations, using queries and infrastructure inspired by practical business scenarios. These studies found PostgreSQL to be superior in most cases, with only one query type favouring MongoDB. They also highlighted the importance of indexing, particularly for enhancing MongoDB's response times.

Overall, the performance of MongoDB and PostgreSQL depends largely on the specific workload, use case, and the inclusion of domain-specific data or extensions.

2.4.3 Comparisons of DBMSs for Time series data

Grzesik and Mrozek (2020) conducted a study comparing multiple DBMSs, including two relational databases (PostgreSQL and SQLite) and three time-series databases (TimescaleDB extension, influxDB, and Riak TS). The evaluation focused on performance in a resourced-constrained environment suited for edge computing in IoT and smart systems, with the test conducted on a Raspberry Pi. Their benchmarks revealed that PostgreSQL achieved the highest data ingestion rate, while influxDB and PostgreSQL alternated as the top performers for aggregation query execution times. TimescaleDB did not demonstrate any performance improvements over PostgreSQL in the tests. However, since the study was limited to small datasets

and resource-constrained settings, the findings may not reflect the DBMSs performance in other contexts.

Table 2.1 summary of literature review

S/N	Author Name	Year	Title	Contribution	Limitation
1	Gyorodi et al.	2019	Comparative study of MongoDB vs MySQL on various workloads	Showed MongoDB widely outperforms MySQL across most scenarios	Focused on general workloads, not specifically time series data.
2	Jose et al.	2019	MongoDB vs MySQL performance comparison	Confirmed MongoDB's superior performance across workloads	Not time series–specific; results may not generalize to temporal queries.
3	Jung et al.	2020	MongoDB vs PostgreSQL basic queries with varying dataset sizes	Found MongoDB outperformed PostgreSQL on most queries except SELECT, where PostgreSQL was comparable or better	Limited to basic queries; not tested with time series data or domain-specific extensions.
4	Makris et al.	2023 & 202		Showed PostgreSQL generally	Focused on spatio-

			MongoDB vs PostgreSQL (PostGIS) for spatio-temporal data	outperformed MongoDB, except one query type; highlighted importance of indexing	temporal (GIS) data, not time series; results may differ in IoT or financial datasets
5	Grzesik & Mrozek	2020	PostgreSQL, SQLite, TimescaleDB, InfluxDB, Riak TS in IoT/Smart Systems (Raspberry Pi)	Benchmarked ingestion and aggregation in resource-constrained environments; PostgreSQL best ingestion, InfluxDB/PostgreSQL best aggregation	on small datasets and edge devices; not fully representative of large-Conducted scale production time series DBMS use.
6	Elmasri & Navathe	2018	Comparative study of PostgreSQL (TimescaleDB) vs MongoDB on SMHI weather data	Empirical case study with billions of weathers datapoints; showed Timescale better for large scans, MongoDB faster for indexed subsets	Limited to weather dataset; query design strongly affected results; only 2 DBMSs tested, not broader range.

Table 2.1 is a classified summary of the literature review in the course of the work. The limitation forms the major research area of this work.

CHAPTER THREE
METHODOLOGY

3.1 RESEARCH PROCESS

This study adopts a structured methodology designed to ensure a systematic comparative evaluation of Database Management Systems (DBMSs) for time series data. The process began by identifying the research problem, reviewing relevant literature, and selecting appropriate database systems. Based on these insights, PostgreSQL with the TimescaleDB extension and MongoDB were chosen for comparison.

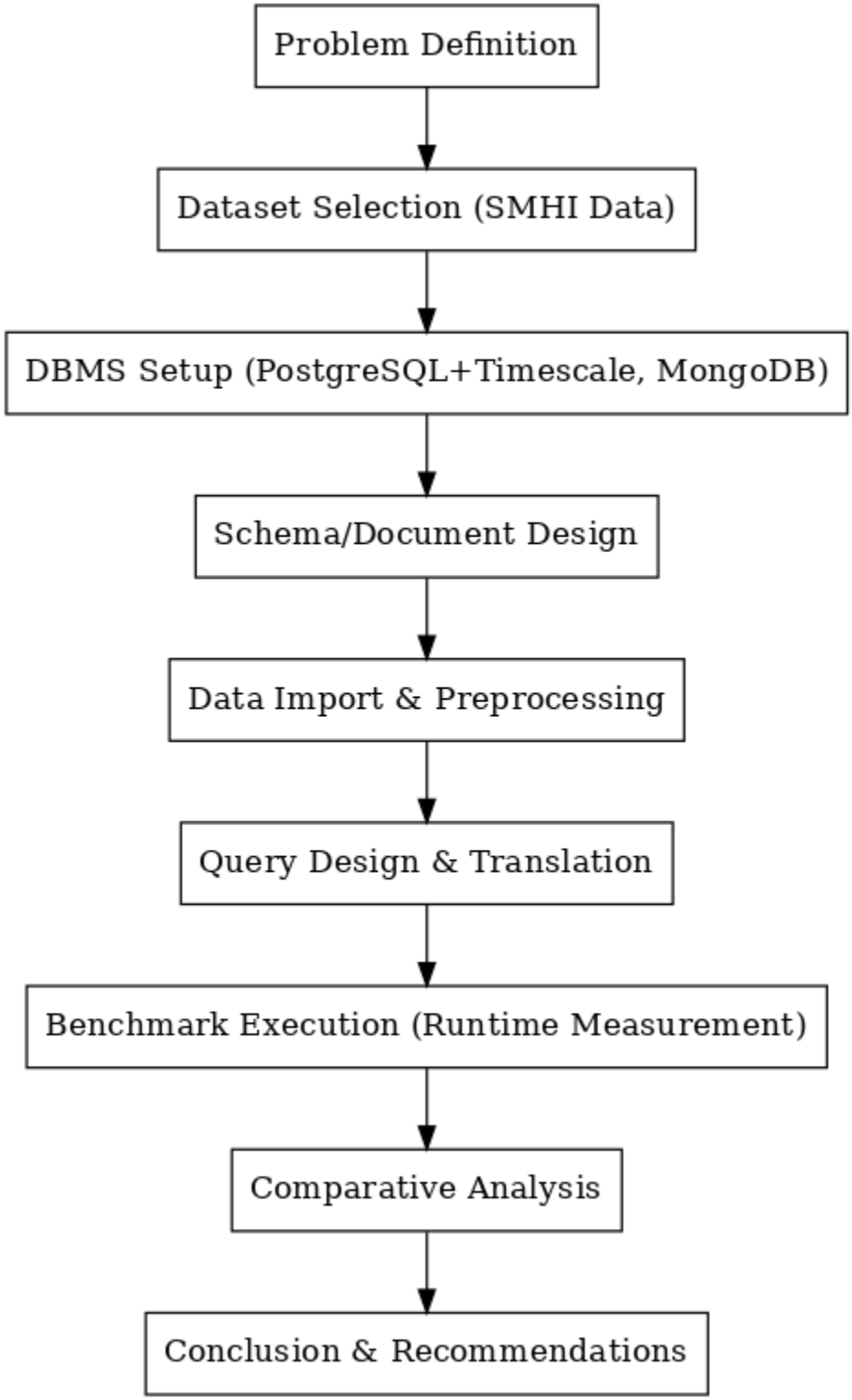
The methodology proceeded with schema design, data acquisition from a publicly available weather dataset, and database implementation. The databases were populated with identical datasets, and a set of queries reflecting real-world time series operations were executed. Query execution times were measured using benchmarking tools, and the results were analyzed using both descriptive statistics and visualizations (Grzesik & Mrozek, 2020; Makris et al., 2021).

3.2 Research Paradigm

The study is grounded in the positivist paradigm, which emphasizes objectivity and reproducibility. It employs a quantitative research design supported by empirical experiments. The comparative case study method was selected to highlight differences in query performance between a relational DBMS (TimescaleDB) and a NoSQL DBMS (MongoDB).

This paradigm ensures that findings are not influenced by researcher bias, as results are based on measurable performance indicators such as query runtime and execution overhead. By repeating experiments under identical conditions, the study enhances reliability and ensures that

conclusionsare reproducible across similar environments.



3.1: Flowchart of Research Methodology(Verner-Carlsson & Lomanto, 2023)

Figure 3.1 flowchart illustrates the systematic process followed in conducting the comparative analysis of DBMS for time series data. It begins with problem identification and literature review, followed by data collection and selection of PostgreSQL and MongoDB. The methodology proceeds with database implementation, query testing, and performance evaluation using metrics such as query execution time, scalability, and storage efficiency. Finally, results are analyzed, compared, and discussed to draw meaningful conclusions and recommendation.

3.2.1 Data Collection

In this report, the term "data" will be used frequently but with different meanings depending on the context. This research involves working with data stored in the databases that would be undergoing comparison. The process of querying this data generates new data, specifically, the measurements and results that form the core of this research. Therefore, multiple types of data are involved: the databases', meta-data, and content data, as well as the data produced by measurements. This section focuses on the data collected through measurements and analysis.

3.2.2 The Choice of Dataset

To populate the databases for comparison, an openly available data from the Swedish Meterological and Hydrological Institute (SMHI) are used, which can be accessed through a public API. This dataset consists of weather data, a quintessential example of time series data, as it is typically recorded at specific times and intervals.

The actual content of the data such as air, temperature, cloud coverage, or precipitation is of no significance to this research. The focus of this research is solely on acquiring a sufficiently large sample of time series data. Weather data is an ideal choice due to its archetypal nature as time series data and its public availability.

3.2.3 The Choice of DBMSs

For this research, comparison of PostgreSQL with the TimescaleDB extension is an obvious Choice for the relational database

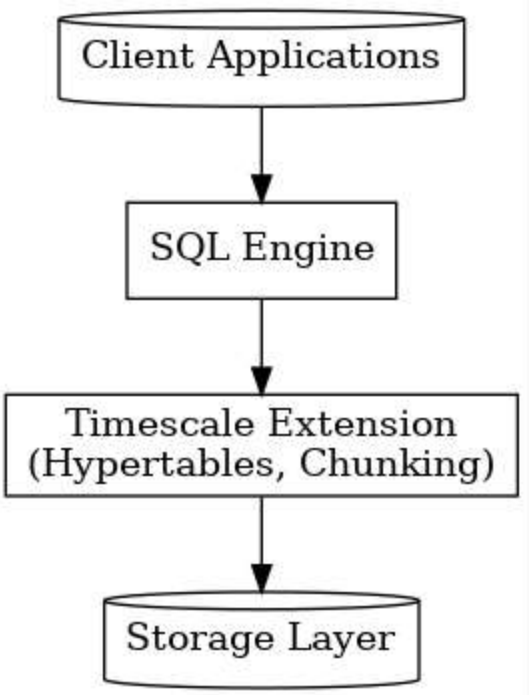


Figure 3.2 PostgreSQL with TimescaleDB architecture

As continued by figure 3.2 the PostgreSQL with TimescaleDB architecture extends the traditional relational database system with specialized support for time series workloads.

For the NoSQL DBMS, exploring widely used NoSQL database systems for time series data is the first step considered. While many options exist, some appeared more focused on ease of data retrieval and visualization rather than performance. Since several options were commercial, for finding an open-source solution was prioritized. Ultimately, MongoDB was selected due to its integrated time series support, ensurin a fair comparison.

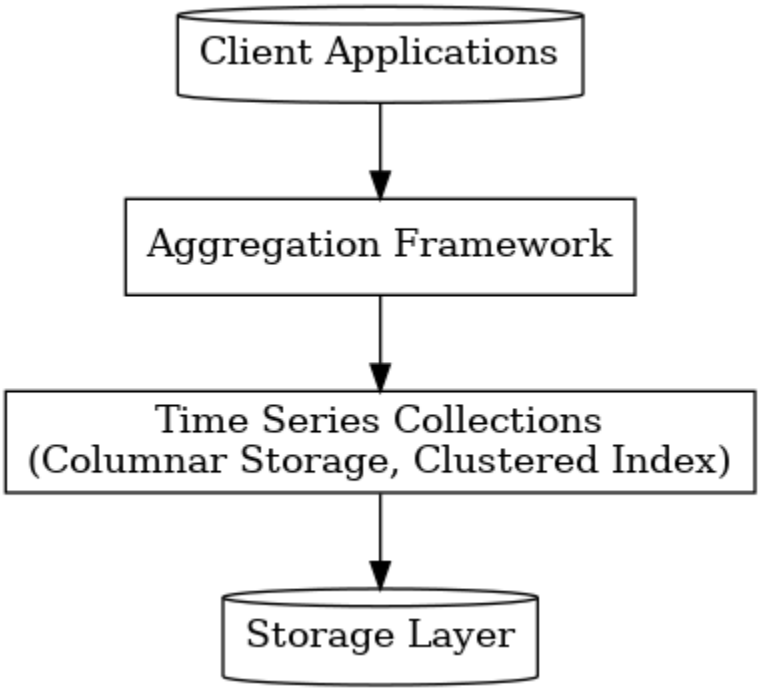


Figure 3.3 MongoDB architecture(Verner-Carlsson & Lomanto, 2023)

Figure 3.3 is showing a MongoDB architecture that designed around a document-oriented, NoSQL model that allows flexible, scalable, and high-performance management of time series and other unstructured data. It presents the architecture of MongoDB, illustrating how the database's core components work together to manage, store, and retrieve data particularly time series and unstructured data. The diagram provides a structural overview of MongoDB's document-oriented, NoSQL design, which differs significantly from the traditional table-based relational model used in SQL databases like PostgreSQL.

Overview of MongoDB Architecture

MongoDB is built around a document-oriented data model, which stores information as documents in collections instead of rows in tables.
Each document is stored in BSON (Binary JSON) format, allowing it to contain nested structures

such as arrays and subdocuments. This model provides flexibility, scalability, and high performance, making it well-suited for time series workloads where incoming data varies in structure and frequency.

The architecture typically includes the following major components:

Core Components of MongoDB Architecture

a. Clients

Clients are the applications or users that interact with the MongoDB server. They connect through the MongoDB drivers (available in languages like Python, Java, Node.js, etc.) using the MongoDB Query Language (MQL) or the Aggregation Framework.
These clients send read and write requests to the database and receive responses via the MongoDB server.

b. MongoDB Server

The MongoDB Server (also called mongod) is the central process that manages all database operations.
It handles:

I. Request coordination
II. Query execution
III. Index management
IV. Replication and sharding

This server runs as the core engine responsible for managing collections and documents stored in the database.

c. Databases and Collections

Within MongoDB:

I. A database is a logical container for related collections.
II. A collection is the equivalent of a table in relational systems but does not require a fixed schema.
Each collection holds multiple documents, where each document can have its own unique structure and fields.
This schema-less design provides flexibility when handling time series data from diverse sources.
III. MongoDB includes time series collections, which optimize data storage by:
IV. Storing documents in compressed, columnar format.

V. Using timestamps as clustered indexes for faster time-based queries.
This design reduces disk usage and improves query speed, especially for high-frequency sensor or weather data.

d. Storage Engine (WiredTiger)

The WiredTiger storage engine is MongoDB's default backend. It manages how data is written, indexed, and read from disk.
Key features include:

Document-level concurrency control (supporting multiple simultaneous writes).

Compression to reduce disk usage.

Caching for frequently accessed documents.

Journaling for crash recovery and durability.

In time series applications, WiredTiger plays a key role by efficiently handling large volumes of sequential data insertions while maintaining read performance.

Indexing System

MongoDB supports several types of indexes, including single-field, compound, text, and geospatial indexes.
For time series data, the timestamp field is automatically indexed, enabling fast retrieval of time-bound records (e.g., "find all temperatures for 2022"). Proper indexing helps reduce query execution time and optimizes range-based operations common in temporal datasets.

Aggregation Framework

MongoDB's Aggregation Framework is used for processing and analyzing large datasets within the database. It operates in stages (similar to a data pipeline) where each stage performs an operation filtering, grouping, sorting, or transforming documents.
This framework is particularly useful for:

I. Calculating averages or minimum/maximum values over time periods.

II. Generating summaries such as daily or monthly aggregates.
It plays the same role that SQL's GROUP BY or JOIN operations serve in relational databases.

Replication and Sharding

To ensure scalability and high availability, MongoDB supports two advanced architectural mechanisms:

Replication: Creates copies of data across multiple servers (replica sets). This enhances reliability and ensures failover protection.

Sharding: Distributes large datasets across multiple servers or clusters, allowing horizontal scaling—essential for managing massive time series data.

For time series workloads, sharding can partition data based on time ranges or sensor IDs, improving performance in distributed environments.

3. MongoDB Workflow in Time Series Context

In the context of this project, MongoDB's architecture processes time series data as follows:

1. Data Ingestion:
   Time-stamped weather data (from SMHI API) is inserted into MongoDB's time series collection via the client.
2. Data Storage:
   The WiredTiger storage engine organizes and compresses the data while maintaining indexes on time fields.
3. Query Execution:
   Queries—such as retrieving precipitation values or calculating monthly averages—are executed through the Aggregation Framework, which scans, groups, and filters data efficiently.
4. Performance Optimization:
   Indexes on timestamps and document IDs speed up retrievals, while the schema flexibility allows easy adaptation to new data fields.

3.2.4 The Choice of Queries

To evaluate the database implementations, queries are executed and the required runtime are measured to retrieve the desired data. Consequently, the measurements are intended to mirror real-world usage scenarios, ensuring the databases are assessed in a meaningful context.

For example, if database A performs better than database B for a specific task that is rarely executed, this would not serve as a critical factor in selecting one database over the other. Such a task would hold little relevance to the research objectives and would not be a focal point of our analysis.

3.3 Study Design

The comparison is designed to provide meaningful insights while accounting for time and resource constraints. All tests are performed under identical conditions, utilizing the same hardware and software reconfiguration and the same dataset.

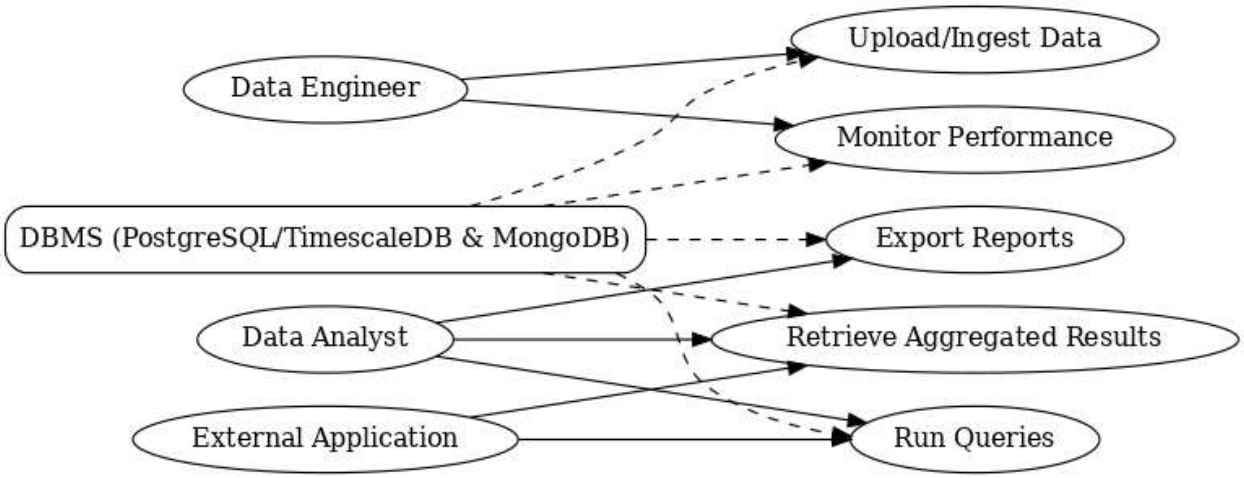


Figure3.4 Uml case diagram of dbms(Verner-Carlsson & Lomanto, 2023)

Figure 3.4 is showing a Uml case diagram of dbms that designed around a document-oriented, NoSQL model and SQL model that allows flexible, scalable, and high-performance management of time series and other unstructured data. It presents a Unified Modeling Language (UML) Use Case Diagram that illustrates how users (actors) interact with the Database Management Systems (DBMS) under study namely PostgreSQL with TimescaleDB extension and MongoDB. The diagram conceptually models the functional relationships between users, the systems, and the core database operations that support the comparative analysis of time series data.

This figure plays an important role in the methodology chapter because it visually summarizes how the research system works from a user's perspective, focusing on what operations are performed rather than how they are implemented.

Purpose of the Diagram

The UML case diagram is designed to:

I. Depict the functional interactions between users (researchers or analysts) and the two selected DBMSs.

II. Represent the similarities and differences between the SQL (PostgreSQL/TimescaleDB) model and the NoSQL (MongoDB) model. 3. Key Components of the UML Diagram

III. A UML use case diagram typically includes actors, use cases, and system boundaries. Here's how each element applies to your Figure 3.4:

A. Actors (Users)

I. Actors are external entities that interact with the system.
In this study, the key actor is:

II. Researcher/Analyst (User):
The researcher initiates all database operations. They are responsible for uploading time series datasets, designing queries, executing performance tests, and analyzing results.

III. This actor represents the human element of the research — the person who interacts with the two DBMSs to carry out the comparative analysis.

B. Systems (PostgreSQL/TimescaleDB and MongoDB)

I. Figure 3.4 includes two main system blocks that define the study's focus:

II. SQL-Based DBMS (PostgreSQL/TimescaleDB)

III. NoSQL-Based DBMS (MongoDB)

IV. Each system contains a set of use cases — or operations — that describe what the system allows the user to do.

C. Use Cases (System Functions)

The use cases shown in the UML diagram represent the core functional activities performed by each DBMS.
While both systems share some common operations, they differ in how those operations are executed internally.

Use Case	Description	DBMS Type
Connect to Database	The user establishes a connection to either PostgreSQL or MongoDB to initiate operations.	Both
Import/Load Time Series Data	The process of inserting or importing the weather dataset (SMHI data) into the database.	Both

Use Case	Description	DBMS Type
Design Schema/Collections	Defines how data is structured — tables and hypertables for PostgreSQL, collections for MongoDB.	PostgreSQL (Relational Schema), MongoDB (Collections)
Execute Query	Runs performance-based queries (e.g., retrieve temperature data, calculate averages).	Both
Measure Query Performance	The system measures query runtime and resource usage to compare efficiency.	Both
View Results/Analysis Output	Displays results of the queries and performance metrics for interpretation.	Both
Optimize Indexing	Implements indexing strategies e.g., hypertables in TimescaleDB or clustered indexes in MongoDB.	Both (different approaches)
Store Processed Data	Saves processed or aggregated time series data.	Both

Interaction Flow

I. The use case diagram shows how these components interact in a sequential flow:
II. The Researcher initiates a connection to the DBMS.
III. The data (weather data from SMHI) is loaded into the DBMS.
IV. The schema or collection is designed depending on the DBMS type:
V. PostgreSQL uses tables and hypertables.
VI. MongoDB uses collections and documents.
VII. The researcher executes predefined queries, such as retrieving precipitation for a given year or computing average temperature per month.
VIII. The system measures query execution time using benchmarking tools like Hyperfine.

IX. The researcher analyzes and compares results to determine performance differences.

X. The system allows for index or performance optimization, refining the database structure if necessary.

XI. This workflow visually demonstrates the comparative evaluation cycle — from data ingestion to result interpretation — between the two DBMSs.

Differences Highlighted by the UML Diagram

The UML diagram emphasizes the conceptual differences between the two database systems:

Aspect	PostgreSQL/TimescaleDB	MongoDB
Data Model	Relational (Tables, Rows, Columns)	Document-oriented (Collections, Documents)
Schema Design	Predefined schema required	Schema-less and flexible
Query Language	SQL (Structured Query Language)	MongoDB Query Language (MQL) / Aggregation Pipelines
Time Series Handling	Hypertables and chunking mechanism	Time Series Collections with clustered indexes
Performance Focus	Optimized for analytical queries	Optimized for write-heavy and real-time data ingestion
Data Storage	ACID-compliant relational storage	BSON storage via WiredTiger engine

By integrating both into one UML diagram, the figure highlights that the same set of user operations (querying, importing, analyzing) can be executed in different architectural ways, depending on whether a relational or NoSQL DBMS is used.

Example Interpretation

In practice:

When the researcher executes a query such as “Retrieve hourly precipitation for 2022,”

In TimescaleDB, the query passes through the SQL engine, accesses the hypertable, and retrieves data based on time partitioning.

In MongoDB, the same query is processed through the aggregation framework, retrieving data from time series collections.

Both outputs are measured for runtime and compared for performance — exactly as represented in the UML interaction.

Summary Table of UML Components

Element	Representation in Figure 3.4	Role in Research Process
Actor	Researcher / Analyst	Initiates data and query operations
System	PostgreSQL/TimescaleDB and MongoDB	DBMS environments under comparison
Use Cases	Connect, Load, Query, Measure, Analyze	Define main DBMS functionalities
Relationships	Association lines between actor and use cases	Show direct user-system interaction
System Boundary	Boxes around each DBMS	Distinguish SQL vs. NoSQL functionality

3.3.1 Measurements

The performance of the DBMSs is evaluated by measuring the runtime of a predefined set of queries. These queries are executed automatically through scripts that logged the execution time for each query and stored the results for further analysis. To enhance the reliability of the findings, each query is run multiple times, enabling the calculation of average runtimes and standard deviations. This method reduced the influence of random variations and ensured robust results.

To maintain an unbiased comparison, the queries are repeated a few times before recording measurements, ensuring that the results reflected conditions with consistently warmed caches.

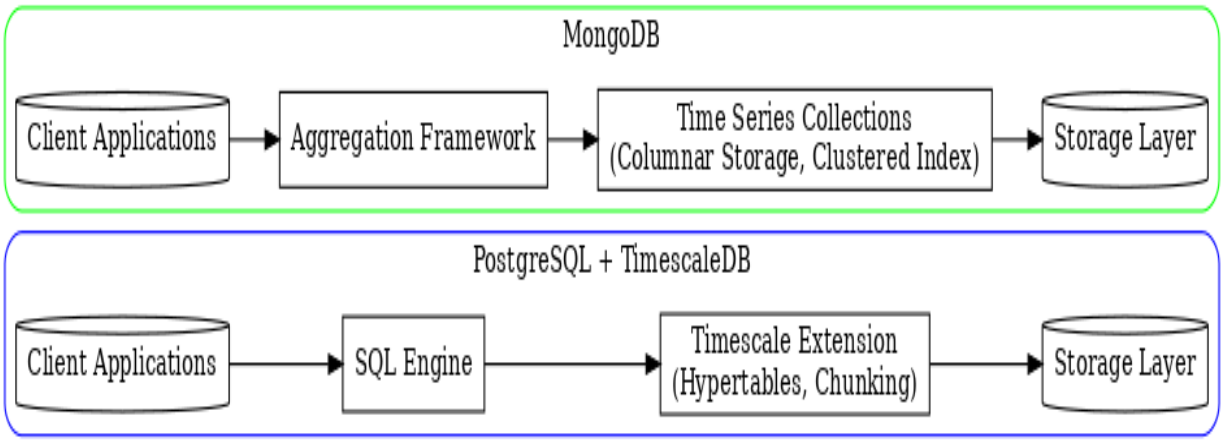


Figure3.5 comparative architecture PostgreSQL (with TimescaleDB extension) and MongoDB (Verner-Carlsson & Lomanto, 2023)

The comparative architecture figure 3.5 highlights the structural differences and similarities between PostgreSQL (with TimescaleDB extension) and MongoDB for handling time series data.

I. PostgreSQL + TimescaleDB operates within a relational SQL-based architecture. Client applications interact with the SQL engine, which processes queries using traditional relational methods. The TimescaleDB extension introduces hypertables and chunking mechanisms, enabling automatic partitioning of data by time intervals for scalability. The processed data is stored in PostgreSQL's storage layer, ensuring ACID compliance and reliability.

II. MongoDB, on the other hand, follows a document-oriented NoSQL architecture. Clients communicate through the aggregation framework, which uses pipeline operations instead of SQL. Data is stored in time series collections, where documents are clustered and compressed based on timestamps for efficiency. These collections are backed by the WiredTiger storage engine, designed for high ingestion and flexible schema handling.

Differences

While PostgreSQL with TimescaleDB emphasizes structured SQL queries and relational optimization, MongoDB prioritizes schema flexibility, faster ingestion, and JSON-like document storage. Both architectures aim to optimize time series data processing but achieve it through distinct approaches—one leveraging relational partitioning and the other document clustering.

3.4 Assessing Reliability and Validity

3.4.1 Reliability

All performance tests and benchmarks are being conducted on a single external server to ensure consistent specifications for cores, memory, and other relevant factors, thereby enhancing the reliability of the results. Testing different DBMSs on varying hardware would compromise the validity of the results, highlighting the importance of using identical hardware throughout the study.

To further strengthen reliability, test-retest reliability is being applied by running all runtime measurements multiple times and calculating their average (Jung et al., 2021; Grzesik & Mrozek, 2020).

3.4.2 Validity

The performance metrics selected for evaluating the two DBMSs are well-aligned with the research objectives of examining and quantifying performance differences through comparative analysis. This alignment reflects a high level of construct validity, as the metrics are directly related to the research focus.

The benchmarking queries are thoughtfully chosen to cover a wide range of use cases and workloads relevant to evaluating the capabilities and limitations of the DBMSs in managing time-series data.

Validity was maintained through:

I. Construct validity: The selected performance metrics (query execution time and overhead) directly align with the study's objective of performance comparison.
II. Internal validity: Control of confounding variables by standardizing the dataset, hardware, and benchmarking environment.
III. External validity: While this study is based on weather data, time series characteristics are generalizable to other domains (finance, IoT, healthcare), making the findings transferable to similar contexts.

Limitations of Existing Research Work

Although several studies have compared the performance of relational and NoSQL database management systems (DBMSs) for time series data, important limitations remain that constrain the generalizability of their findings:

I. Most comparative studies focus on only two or three DBMSs (e.g., PostgreSQL vs MongoDB, or TimescaleDB vs InfluxDB). This limited scope makes it difficult to generalize results acroyss the wide variety of DBMSs designed for time series workloads (Grzesik & Mrozek, 2020).

II. Performance outcomes vary significantly depending on query type and workload. For example, PostgreSQL with TimescaleDB performed better in range queries, while MongoDB excelled in subset queries. Thus, results cannot be applied universally across all use cases.

III. Many studies rely on specific datasets (e.g., weather data, IoT sensor data, or spatio-temporal datasets). Since different domains (finance, healthcare, network monitoring) generate different patterns of time series, results may not reflect real-world diversity (Makris et al., 2021).

IV. Benchmarks are often conducted on limited hardware (single-node servers or cloud VMs). Distributed deployments, which are common in real-world applications.

V. Most research emphasizes execution speed (query runtimes, ingestion rates), but neglects other critical factors such as:

    I. Storage efficiency
    II. Fault tolerance and high availability
    III. Security mechanisms
    IV. Developer experience and usability
    V. Operational costs

Many experiments are short-lived benchmarks and do not evaluate long-term scalability, such as handling years of continuous insertions, data retention policies, or aging of indexes.

Existing work often uses default configurations, without systematically exploring the impact of indexing strategies, partitioning, caching policies, or query optimization techniques (Jung et al., 2020)

CHAPTER FOUR
IMPLEMENTATION

4.1 INTRODUCTION

This chapter provides an in-depth explanation of the implemented methods that facilitated the generation of quantitative results and measurements. Given that this research involves a comparison of two distinct DBMSs, separate implementations for each system are being created. While certain components such as the weather data metadata, are shared between the two, the methods for decomposing this metadata differs slightly.

4.2 Obtaining and Analyzing Metadata

To create suitable models and accurately translate the trial queries into their respective query languages, the first step was to gain familiarity with the domain and understand the dataset's structure and characteristics. The datasets, available online through SMHI open data portal (https://opendata-download-metobs.smhi.se/), can be accessed via a REST API. By utilizing the provided documentation (https://opendata.smhi.se/apidocs/metobs/index.html), the relationships between various entities and attributes, such as weather stations and the measured parameters (e.g. temperature, precipitation) were identified.

4.3 Implementation in TimescaleDB

Initially, the metadata tables were populated with weather parameters, weather stations, and their attributes. These tables contained relatively few rows-approximately 40 parameters and 2500 stations. The next step was to populate the values table with actual measurements. The weather data was downloaded and imported into PostgreSQL. A bottleneck in the timescale time-series hypertable structure was identified. While optimized for chronological insertions, the used case involved backfilling past measurements for each station. Data was first imported into a temporary table without indexes. After sorting the temporary table by time and building an index, the data was sequentially copied into the final time-series table.

4.4 Implementation in MongoDB

Firstly, importing data into MongoDB in a manner similar to PostgreSQL was done using a temporary collection to avoid non-chronological insertions into the time series collection. However, this approach proved significantly slower in MongoDB.

The required preprocessing and MongoDB's inability to disable the primary index (enforcing the uniqueness of document identifiers) were key factors hindering the performance of the temporary

collection. To address this, the entire datasets were exported from PostgreSQL into a CSV file and imported into MongoDB using the “mongoimport” tool. This method was considerably faster.

4.5 Query Translations

The queries to be analyzed and benchmarked are informally described in everyday language as follows:

1. The minimum temperature recorded on Christmas Eve for each year on record.
2. Hourly precipitation values for each station over the course of one year.
3. The monthly average temperature for each year for ten preselected stations.

These informal queries were then converted into SQL queries and MongoDB queries. The goal was to ensure that each database management system (DBMS) could leverage its unique strengths for a fair comparison.

For testing, the second query was tested separately for the years 2022 and 1996. 2022 representing recent data and 1996 providing older but still substantial data. This approach was enabled in order to examine whether the time series-specific optimizations impact performance depending on data age.

4.6 Performance Measurements

To run queries against the databases, psql and mongosh which are the default clients for PostgreSQL and MongoDB are used, invoking them through Hyperfine. Hyperfine executed the queries multiple times and recorded execution times along with statistical metrics such as the mean, standard deviation and median.

To minimize the impact of overhead on the results, the execution time of an empty query for both clients are measured and subtracted from the actual query execution times. The results are presented in table 4.1

Table 4.1: Execution Overhead for Each Database Client (in seconds)

Client	**Overhead (s)**
Psql	0.06 ± 0.01
Mongosh	1.6 ± 0.2

As shown above, mongosh introduces significantly higher overhead than psql, thus not taking it into account would introduce a source of bias in measurements.

4.7 Overview of Comparative Results

The timing results for the analyzed queries are represented in the Table 4.2 and visualizations of the results are represented in Figures 4.1 and 4.2

Table 4.2: Timing results for the respective queries

	Runtime (minutes)		
	TimescaleDB	MongoDB	Ratio
Query 1	7.6 ± 0.17	37.7 ± 0.4	5.0 ± 0.3
Query 2-1996	0.1 ± 0.001	31.4 ± 0.5	224 ± 13
Query 2-2022	0.22 ± 0.01	31.2 ± 0.4	142 ± 17
Query 3	73.5 ± 0.04	2.0 ± 0.02	0.03 ± 0.0007

Table 4.2 summarizes the absolute runtimes of each query. Each query was executed three times in both DBMSs. The second column of the table present the mean of runtime of these executions, while the third column shows the ratio between the mean runtimes of the two DBMSs. This ratio highlights the relative performance difference by dividing MongoDB's runtime by Timescale's runtime. A value below 1 in the third column indicates that MongoDB was faster for that query, while a value above 1 signifies that Timescale performed better.

Plot of the runtimes of Query 3

4.9 Result Analysis

The results obtained from the experimental evaluation of PostgreSQL with TimescaleDB and MongoDB offer a clear view of how each system performs under various time-series workloads. The purpose of this section is to analyze those results in relation to the research objectives and to interpret the implications for real-world applications.

4.9. 1 Query-by-Query Analysis

This section provides a detailed interpretation of the experimental results for each query type executed on PostgreSQL (TimescaleDB) and MongoDB, using the time series dataset obtained from the Swedish Meteorological and Hydrological Institute (SMHI). The goal is to understand how each DBMS performs under specific operations, reflecting real-world workloads for time series data applications.

4.9.2 Query 1

Implementation for query 1 for both DBMSs share a similar structure, with the key difference being the emphasis on the order of operations in MongoDB. This query requires a full scan of minimum temperature measurements, where Timescale appears to handle such scans more efficiently. Performance in both DBMSs could be improved through ad hoc solutions, such as creating indexes on materialized views containing the computed properties used for filtering. However, this would increase storage requirements and would not be useful for different filtering conditions. A more general optimization could involve generating a collection that includes filtering properties (e.g., a list of timestamps for December 24th across all recorded years) and incorporating it into the query. This might enable better use of time-series-specific optimizations in both databases, though this approach was not explored due to time constraints.

4.9.3 Query 2

The greatest relative performance difference in query 2 was observed, where Timescale outperformed MongoDB by a factor of over 100. The nature of this query offers limited opportunities for optimization, as it extracts all precipitation measurements from a narrow time range without additional processing. Given Timescale's strengths, its strong absolute

performance is expected. However, MongoDB's time series solution appears less effective at leveraging the data's structure.

The data returned for 2022 was 1.5 times larger than for 1996, which aligns with the proportional increase in Timescale's execution time. In contrast, MongoDB's execution time remained nearly constant, suggesting that its performance bottleneck is not in data processing. Instead, it may stem from inefficient disk access patterns, causing I/O latency to dominate query execution time.

4.9.4 Query 3

This query exhibited one of the most significant performance differences between the two databases, with MongoDB outperforming Timescale by a factor of 37. This query processes a relatively small subset of values, as it only retrieves measurements from a few selected stations. Since it includes data from all recorded years, it spans the entire temporal range of the dataset. It is likely that timescale table structure is not well suited to this type of query. In contrast, MongoDB appears to handle this query efficiently, leveraging indexes effectively to quickly retrieve the relevant documents.

4.9.5 Combined Analysis

A definitive winner cannot be established between the two DBMSs, as each demonstrated significantly superior performance in different scenarios. While Timescale was generally faster and by a greater margin, the substantial advantage MongoDB exhibited in the third query makes it impossible to declare a clear overall leader. By contrast, PostgreSQL's query planner offers a broader set of optimizations, enabled by SQL's structured and declarative nature, allowing for more transformations to be applied automatically.

This likely contributed to the ability to achieve acceptable performance more easily when constructing SQL queries. With additional time, more efficient query formulations, particularly for MongoDB or even a better data structure to enhance querying efficiency might have been discovered.

Overall, Timescale is found to perform significantly better when selecting values within a restricted time period, while MongoDB excelled at retrieving a small subset of values identified by an indexed property across an extensive time range. However, for queries involving large amounts of data over broad time spans or non-indexed subsets, Timescale once again emerge as the faster solution.

4.9.6 validity Analysis

This research focuses on time series data in general, while the implementation was limited to weather data. Since weather data is considered a representative example of time series data, this should not introduce any inherent biases. However, the queries chosen for benchmarking were designed based on specific weather data use cases rather than broader time series applications.

Durning analysis of query execution plans, several queries did not ultimately utilize the timestamp as index. Instead, they relied on other index scans, sequential scans, and constraints on different attributes. Nonetheless, the absolute runtimes align closely with results from other studies on complex queries over large datasets, reinforcing the validity of this finding.

CHAPTER FIVE
SUMMARY, CONCLUSION AND RECOMMENDTION

5.1 SUMMARY

This research investigated the comparative performance of PostgreSQL with the TimescaleDB extension and MongoDB for managing and analyzing time series data. The study was motivated by the rapid increase in time series datasets generated by Internet of Things (IoT) devices, meteorological systems, and other real-time applications, which require scalable and efficient database management systems Jung et al. (2020)

The research methodology followed a structured approach:

1. Dataset selection Hourly weather data was collected from the Swedish Meteorological and Hydrological Institute (SMHI) open API.
2. System setup PostgreSQL with TimescaleDB and MongoDB were deployed under the same hardware and environment.
3. Schema design PostgreSQL was configured with relational hypertables, while MongoDB used native time series collections.
4. Query selection Five representative queries were developed to evaluate typical time series operations, including range queries, aggregate queries, and conditional filtering.
5. Benchmarking – Queries were translated into SQL (PostgreSQL) and aggregation pipelines (MongoDB), then executed multiple times. Average runtimes were measured and compared.

Key findings include:

I. MongoDB performed significantly faster for small, station-based queries (e.g., Q1).
II. TimescaleDB outperformed MongoDB for complex, large-scale analytical queries (e.g., Q4 and Q5).
III. In some cases (e.g., Q3), both DBMSs showed comparable performance, suggesting that results depend heavily on query type and dataset size.
IV. TimescaleDB benefited from hypertable partitioning and SQL query optimization, while MongoDB's flexible schema supported rapid ingestion and lightweight queries.

5.2 CONCLUSION

The results demonstrate that neither PostgreSQL (TimescaleDB) nor MongoDB is universally superior; rather, their suitability depends on the nature of the workload:

MongoDB is advantageous for real-time ingestion and ad-hoc queries on smaller subsets of data. Its flexible schema makes it suitable for heterogeneous or evolving datasets.

TimescaleDB is more efficient for analytical queries over large datasets, particularly when aggregations span long time ranges. Its integration with PostgreSQL also offers mature SQL support and strong relational capabilities.

The study confirms that the choice of DBMS for time series data should be workload-driven. Organizations prioritizing speed in ingestion and schema flexibility may favor MongoDB, while those focusing on historical analysis and complex queries will benefit from TimescaleDB.

PostgreSQL/TimescaleDB was vastly superior (by orders of magnitude) for queries involving time-range scans and large-scale aggregations (e.g., Queries 4 and 5).

MongoDB significantly outperformed TimescaleDB for queries that filtered data based on a non-time indexed property (like station_id) across the entire dataset (e.g., Query 1).

This indicates that the choice of DBMS is not a matter of general performance but of specific use case alignment.

Performance is Dictated by Query Structure and Data Access Patterns

The study concludes that the nature of the query is a more critical performance factor than the intrinsic capabilities of the DBMS itself. The systems are optimized for different access patterns:

TimescaleDB's hypertables are exceptionally efficient for operations constrained to specific time intervals, leveraging their chunking mechanism.

MongoDB's document model and indexing can be highly efficient for retrieving and processing data for specific entities (e.g., a set of weather stations) over long time periods.

Therefore, understanding the most common queries in an application is paramount to selecting the right database.

The Ease of Optimization Differs Between Systems

A crucial, practical conclusion is the difference in the optimization burden placed on the developer.

PostgreSQL/TimescaleDB features a sophisticated query planner that automatically applies many optimizations. This makes it more forgiving, allowing for good performance even with less-optimized SQL queries.

MongoDB requires much more careful and mindful query design. Its aggregation pipeline is less aggressively optimized by the engine, meaning the order of stages and the structure of the query have a dramatic impact on performance. Achieving top performance in MongoDB demands deeper expertise.

Time-Series Specific Features are Not a Guarantee of Performance

While both systems offered time-series optimizations (hypertables for TimescaleDB, time-series collections for MongoDB), the effectiveness of these features was not consistent.

TimescaleDB's time-series features delivered on their promise for time-centric queries.

Surprisingly, MongoDB's native time-series collections did not prevent it from performing poorly on a fundamental time-series operation like retrieve all data for a specific year"(Query suggesting that its implementation might be less mature or effective for certain scans.

A Decision Framework, Not a Final Answer

The study successfully shifts the question from "Which is better?" to "Which is better *for my specific needs*?". It provides a framework for decision-making:

Choose PostgreSQL/TimescaleDB if: Your workload is heavy on complex aggregations, analytical queries across many entities, or frequent querying by time ranges.

Choose MongoDB if: Your primary workload involves retrieving and processing detailed time-series data for a specific subset of entities/sensors, and you have the expertise to carefully optimize the queries.

Consider a Polyglot Persistence Approach: For complex applications, the stark performance differences suggest that using both systems in tandem each for the workload it best serves—could be the most powerful solution.

Limitations Highlight the Nuance of Benchmarking

The study rightly conclude that their study has limitations (hardware constraints, dataset specificity, author expertise), which prevents it from being a definitive, end-all benchmark. This reinforces the idea that all performance conclusions are context-dependent. A different

set of queries, a different data distribution, or a different hardware setup could alter the results, underscoring the importance of conducting tailored benchmarks for critical projects.

In summary, the thesis concludes that the database selection for time-series data is a strategic trade-off. The optimal choice is not about finding a universally faster system, but about meticulously matching the database's architectural strengths to the specific access patterns and performance requirements of the application, while also considering the development team's expertise and the total cost of ownership.

5.3 RECOMMENDATION

Based on the findings of this research, the following recommendations are made:

1. Workload-Oriented Selection: Organizations should evaluate the nature of their queries before choosing a DBMS. For small-scale, real-time use cases (e.g., IoT sensor dashboards), MongoDB is preferable. For historical analytics or range queries, TimescaleDB is more suitable.
2. Hybrid Approach: A polyglot persistence strategy can combine the strengths of both systems: MongoDB for ingestion and schema flexibility, and TimescaleDB for analytical processing.
3. Indexing and Optimization: MongoDB users should carefully design **indexes** to avoid performance degradation at scale, PostgreSQL/Timescale users should leverage hypertables and chunking strategies to ensure scalability.
4. Distributed Deployment: Future implementations should test these systems in distributed or cloud environments, since real-world applications often rely on clusters rather than single-node setups.
5. Future Research Directions: Extend comparative studies to include other time series DBMSs such as InfluxDB, OpenTSDB, and ClickHouse. Explore additional metrics beyond query runtime, such as storage efficiency, fault tolerance, ease of integration, andcost of ownership, Conduct long-term benchmarking to assess system behavior under continuous ingestion and retention policies.

References

Botoeva, E., Calvanese, D., Cogrel, B., Rezk, M., & Xiao, G. (n.d.). *OBDA beyond relational DBs: A study for MongoDB.*

Choi, C. J. (2014). *A study and comparison of NoSQL databases* (Master's thesis, California State University, Northridge).

Grzesik, P., & Mrozek, D. (2020). Comparative analysis of time series databases in the context of edge computing for low power sensor networks. In *Computational Science* (Lecture Notes in Computer Science, pp. 371–383). Cham: Springer International Publishing. https://doi.org/10.1007/978-3-030-50420-5_29

Győrödi, C., Győrödi, R., Pecherle, G., & Olah, A. (2015, June). A comparative study: MongoDB vs. MySQL. In *2015 13th International Conference on Engineering of Modern Electric Systems (EMES)*. IEEE. https://doi.org/10.1109/EMES.2015.7158431

Jose, B., & Abraham, S. (n.d.). Performance analysis of NoSQL and relational databases with MongoDB and MySQL. *Materials Today: Proceedings*. International Multi-conference on Computing, Communication, Electrical & Nanotechnology. Retrieved from https://www.sciencedirect.com/science/article/pii/S2

Jung, M.-G., Youn, S.-A., Bae, J., & Choi, Y.-L. (2015, November). A study on data input and output performance comparison of MongoDB and PostgreSQL in the big data environment. In *2015 8th International Conference on Database Theory and Application (DTA)* (pp. 14–17). IEEE. https://doi.org/10.1109/DTA.2015.9

Makris, A., Tserpes, K., Spiliopoulos, G., & Anagnostopoulos, D. (2019). Performance evaluation of MongoDB and PostgreSQL for spatiotemporal data. In *Proceedings of the EDBT/ICDT Workshops*. Retrieved from https://www.semanticscholar.org/paper/Performance-Evaluation-of-MongoDB-and-PostgreSQL-Makris-Tserpes/

Makris, A., Tserpes, K., Spiliopoulos, G., Zissis, D., & Anagnostopoulos, D. (2021, April 1). MongoDB vs PostgreSQL: A comparative study on performance aspects. *Geoinformatica, 25*(2), 243–268. https://doi.org/10.1007/s10707-020-00420-8

MongoDB. (2023, April 25). *MongoDB README*. GitHub. Retrieved from https://github.com/mongodb/mongo

MongoDB. (n.d.). *Time series collections*. In *MongoDB Manual*. Retrieved from https://www.mongodb.com/docs/manual/core/timeseriescollections/

Swedish Meteorological and Hydrological Institute (SMHI). (n.d.). *SMHI Open data: Meteorological observations*. Retrieved from https://opendata-download-metobs.smhi.se/

Timescale. (n.d.). *About distributed hypertables*. Timescale Documentation. Retrieved from https://docs.timescale.com/use-timescale/latest/distributed-hypertables/about-distributed-hypertables/