KTH ROYAL INSTITUTE
OF TECHNOLOGY

Degree Project in Computer Engineering[1]
and Information and Communication Technology[2]

First cycle, 15 credits

A Comparative Analysis of Database Management Systems for Time Series Data

TOVE VERNER-CARLSSON[1]
VALERIO LOMANTO[2]

Stockholm, Sweden, 2023

A Comparative Analysis of Database Management Systems for Time Series Data

TOVE VERNER-CARLSSON
Degree Programme in Computer Engineering

VALERIO LOMANTO
Bachelor's Programme in Information and Communication Technology

Date: 21st June 2023

Supervisors: Morteza Esmaeili Tavana, Per Hagström, Isabel Ghourchian
Examiner: Håkan Olsson
School of Electrical Engineering and Computer Science
Host company: Zenon AB
Swedish title: En jämförelse av databashanteringssystem för tidsseriedata

Abstract

Time series data refers to data recorded over time, often periodically, and can rapidly accumulate into vast quantities. To effectively present, analyse, or conduct research on such data it must be stored in an accessible manner. For convenient storage, database management systems (DBMSs) are employed. There are numerous types of such systems, each with their own advantages and disadvantages, making different trade-offs between desired qualities.

In this study we conduct a performance comparison between two contrasting DBMSs for time series data. The first system evaluated is PostgreSQL, a popular relational DBMS, equipped with the time series-specific extension TimescaleDB. The second comparand is MongoDB, one of the most well-known and widely used NoSQL systems, with out-of-the-box time series tailoring.

We address the question of which out of these DBMSs is better suited for time series data by comparing their query execution times. This involves setting up two databases populated with sample time series data — in our case, publicly available weather data from the Swedish Meteorological and Hydrological Institute. Subsequently, a set of trial queries designed to mimic real-world use cases are executed against each database, while measuring their runtimes. The benchmark results are compared and analysed query-by-query, to identify relative performance differences.

Our study finds considerable variation in the relative performance of the two systems, with PostgreSQL outperforming MongoDB in some queries (by up to more than two orders of magnitude) and MongoDB resulting in faster execution in others (by a factor of over 30 in one case).

Based on these findings, we conclude that certain queries, and their corresponding real-world use cases, may be better suited for one of the two DBMSs due to the alignment between query structure and the strengths of that system. We further explore other possible explanations for our results, elaborating on factors impacting the efficiency with which each DBMS can execute the provided queries, and consider potential improvements.

Keywords

Database Management Systems, PostgreSQL, TimescaleDB, MongoDB, Time Series, Database Comparison, Performance Analysis

Sammanfattning

I takt med att mängden data världen över växer exponentiellt, ökar också behovet av effektiva lagringsmetoder. En ofta förekommande typ av data är tidsseriedata, där varje värde är associerat med en tidpunkt. Det kan till exempel vara något som mäts en gång om dagen, en gång i timmen, eller med någon annan periodicitet. Ett exempel på sådan data är klimat- och väderdata.

Sveriges meteorologiska och hydrologiska institut samlar varje minut in mätvärden från tusentals mätstationer runt om i landet, så som lufttemperatur, vindhastighet och nederbördsmängd. Det leder snabbt till oerhört stora datamängder, som måste lagras för att effektivt kunna analyseras, förmedlas vidare, och bevaras för eftervärlden. Sådan lagring sker i databaser.

Det finns många olika typer av databaser, där de vanligaste är relationella databaser och så kallande NoSQL-databaser. I den här uppsatsen undersöker vi två olika databashanteringssystem, och deras lämplighet för lagring av tidsseriedata. Specifikt jämför vi prestandan för det relationella databashanteringssystemet PostgreSQL, utökat med tillägget TimescaleDB som optimerar systemet för användande med tidsseriedata, och NoSQL-systemet MongoDB som har inbyggd tidsserieanpassning.

Vi utför jämförelsen genom att implementera två databasinstanser, en per komparand, fyllda med SMHI:s väderdata och därefter mäta exekveringstiderna för ett antal utvalda uppgifter som relaterar till behandling av tidsseriedata. Studien konstaterar att inget av systemen genomgående överträffar det andra, utan det varierar beroende på uppgift. Resultaten indikerar att TimescaleDB är bättre på komplexa uppgifter och uppgifter som involverar att plocka ut all data inom ett visst tidsintervall, emedan MongoDB presterar bättre när endast data från en delmängd av mätstationerna efterfrågas.

Nyckelord

Databashanteringssystem, PostgreSQL, TimescaleDB, MongoDB, Tidsserier, Databasjämförelse, Prestandaanalys

Acknowledgments

Firstly, we would like to extend our sincere gratitude towards the people at Zenon AB, for their support, enlightening insights, and all-around hospitality. With the guidance and direction provided by Per Hagström and Isabel Ghourchian, our project became fruitful and our experience turned out a true delight.

We would also like to thank our examiner Håkan Olsson and our supervisor Morteza Esmaeili Tavana, for their consistent support throughout the project. Specifically, we appreciated constructive discussions on writing style and report structure.

Finally, we would like to acknowledge the people around us who tirelessly ensured that the world kept spinning, while we were absorbed by our thesis endeavours. You know who you are, and we are forever grateful.

Stockholm, June 2023
Tove Verner-Carlsson Valerio Lomanto

Contents

List of acronyms and abbreviations

DBMS	Database Management System
RDBMS	Relational DBMS
SMHI	Swedish Metereological and Hydrological Institute
SQL	Structured Query Language

Glossary

Entity Relational Diagram	A visual representation of a database schema, displaying entities and the relations between them in a standardised graphical notation.
Foreign Key	A column or set of columns in a table that refer to the primary key of another table, identifying a row within that table and linking the two tables together.
Workload	The type and intensity of operations or tasks that are imposed on a system. It represents the specific set of activities and demands placed on a system, either in real-world cases or simulated conditions (*e.g.*, for benchmarking).

Chapter 1

Introduction

The ceaseless flow of time is a deep and fundamental fact about reality that has shaped human life since time immemorial. It is therefore only natural to harbour an interest in how different aspects *change* over time. To analyse processes in general, or temporal processes in particular, we need to collect data. Only then can we begin to examine dependencies, search for correlations, and ultimately draw conclusions. In formal settings, data collection can be done in various ways. Once collected, it is essential to store the data in a suitable manner for future processing. This is where *databases* come into play. The primary purpose of a database is to store data, either in physical or digital form. A database could technically be a manual entity written on paper, but in modern times, the term usually refers to something digital — that is, data stored on a server. To interact with such digitally stored data, we typically employ a Database Management System (DBMS).

There are different types of databases, with the traditional approach being *relational* databases. They are named as such because data is stored following the well-studied concept of a *relational model*. These databases are constructed and interacted with using the query language SQL. However, in recent years, *NoSQL systems* have gained increasing popularity. Document databases, one type of NoSQL system, are widely used for time series data, which is the subject of this research.

Time series data is data collected over time, with each data entry associated with a timestamp. The measurements may be periodic — occurring at regular intervals — or irregularly scattered across a specific time period. An example of periodically gathered data is *weather and climate data*, which is typically measured once per time unit (*e.g.,* once per minute or once per hour). Other types of time series data include the current number of active streamers on a

streaming service, the current CPU usage of a computer or other performance measurements, and heart rate data measured by a sports watch.

In collaboration with Zenon, this study aims to investigate the characteristics of different database systems for storing and analysing time series data.

1.1 Background

Time series data refers to data that is collected repeatedly and recorded over a period of time, where each record is associated with the time of recording, and many applications make use of this kind of measurement. Different DBMSs make different trade-offs between various aspects of runtime performance, resource usage, development speed, and ease of iterations, with some additionally offering specific support to time series data through extensions. NoSQL databases have emerged as an alternative to traditional relational databases, catering to use cases where normalised relational data is not necessary and other considerations dominate the choice of DBMS.

1.2 Problem

The issue that this project aims to tackle is the evaluation of the suitability of two specific database system for storing time series data.

1.2.1 Original problem and definition

A client of Zenon is currently storing large amounts of time series data in a purely relational database, and the company is interested in evaluating the benefits of a possible migration to a different system.

1.2.2 Scientific and engineering issues

Collecting time series data quickly accumulates into very large datasets, especially if there are periodic measurements and multiple measurements per timestamp. Working with these large datasets requires a significant amount of storage space and computational power when querying the data.

1.2.3 Research question

In this study, we consider and attempt to answer the following research questions:

1. What are the general advantages and disadvantages of relational DBMSs compared to document DBMSs, in regards to time series data?

2. Which out of two chosen common DBMSs, one per considered paradigm, has better performance for common time series data workloads?

1.3 Purpose

The research conducted in this project aims to simplify the selection process among different database systems for storing time series data while providing a solid foundation for decision-making. Specifically, we focus on comparing the performance of relational and document databases in the context of time series data, using weather data as our sample dataset source.

By investigating the advantages and disadvantages of these two paradigms when applied to time series data, we seek to offer valuable insights for companies selecting the most suitable storage solution for their applications. Additionally, we hope that our work will contribute to expanding the existing knowledge within the field, and provide inspiration for future academic studies.

The effective choice of a suitable database type can lead to resource savings in terms of storage, computational usage, and man-hours. Applications such as weather forecasts and historical weather data analysis, which involve recording various quantities in this format, can particularly benefit from the selection of an appropriate DBMS for manipulating and processing time series data.

1.4 Goals

The goal of this project is to investigate the difference between relational and document databases in regards to time series data. We deliver comparative benchmark measurements for the performance of the predefined queries. To achieve this we have set the following sub-goals:

1. Provide a summary stating the advantages and disadvantages of using different database systems to assist in decision-making.

2. Expand on the existing knowledge in the field by conducting a sample study for a specific case.

1.5 Research Methodology

To address our research questions, we conducted a case study applied to weather data involving the analysis of time series data using two selected DBMSs. This study follows a positivist approach with an experimental focus. The experimental design is based on an exploration of contemporary database systems research, providing a theoretical foundation. The performance of the two systems was evaluated through benchmarking, utilising measurements of runtimes. thereby emphasising a quantitative approach. We chose to make a comparative analysis between two DBMSs to highlight trade-offs between them, and to have a reference point to evaluate each one against, allowing for a more comprehensible analysis.

1.6 Implementation

To evaluate the DBMSs under consideration, we obtained data from SMHI weather observations across Sweden over the past decades. The dataset consists of time series of various quantities measured with hourly or tri- and quadri-daily periods (depending on recency). In addition to being directly of interest to Zenon, this sample of several millions of data points with various relationships should be sufficiently representative of different kinds of time series data and possible workloads.

We set up one relational and one NoSQL database to store this data, designing a suitable relational model for the first, and a document structure for the latter. We performed a series of queries on them modelled after typical operations one might be interested in, to try to capture as closely as possible the real-world impact of the different choices.

We measured the performance of the systems under analysis throughout the study by taking measurements of the time required by the operations (in "wall clock" time, *i.e.*, the time elapsed between the start and the end of the operation).

We then analysed these metrics to identify the aspects in which some approaches proved superior to others. This will help developers in the future to make informed decisions when handling time series data.

1.7 Scope and Limitations

This thesis focuses on comparing one relational DBMS (with a time series extension) and one NoSQL DBMS. The comparison specifically excludes other database systems from consideration. We solely concentrate on periodic weather data, excluding other types of time series data. The measurements and performance data collection is limited to a predefined set of queries.

Moreover, the comparison solely addresses the performance aspect of the two DBMSs, disregarding other important considerations in the selection of a suitable DBMS for a particular application. Notably, the usability and developer experience of the two alternatives was not investigated, despite initial considerations. This decision was prompted by the realization that the available methods would lack the necessary validity to assess that perspective effectively.

1.8 Structure of the Thesis

Following this introduction, Chapter 2 will provide a comprehensive background covering the problem and various theoretical concepts within the field. The background chapter will also include detailed explanations and definitions of key concepts and terminology. Chapter 3 will be dedicated to describing the methodology we employed to address the problem outlined in Section 1.2, and why it was a suitable choice. In the subsequent Chapter 4, we will present detailed information regarding our implementation, including diagrams, code snippets, and scripts.

Chapter 5 will be dedicated to presenting the results of our measurements and comparisons. In Chapter 6, we will provide a detailed analysis and interpretation of the findings obtained from our research. We will engage in a retrospective discussion, reflecting on our insights and considering alternative approaches that could have been pursued.

Finally, in Chapter 7, we will draw overall conclusions from the results and reflect on the potential for future studies that can offer further insights into the problem. Additionally, we will provide a holistic reflection on the project as a whole, considering its significance and potential avenues for future research.

Chapter 2

Background

In the modern world, an ever more extensive amount of digital data is produced and stored to be later retrieved and manipulated. The resulting electronic collections of data — *i.e.*, databases — have thus become an increasingly relevant subject, and as the scope of their usage has expanded to immense proportions, the impact of any findings on different ways to work with them has also grown.

Database Management Systems, *i.e.*, systems to store, organise, retrieve, and manipulate digital data, have hence been the object of intense study[1, 2], with traditional solutions converging towards the so-called *relational model*[3], embodied by Relational DBMSs (RDBMSs), and non-relational approaches experiencing a resurgence in recent years under the moniker of NoSQL DBMSs. We expose general notions about databases in Section 2.1, describing DBMSs broadly (subsection 2.1.1), the relational/NoSQL distinction (subsection 2.1.2), and typical desiderata of DBMSs (subsection 2.1.3).

This study compares instances of two database types: one relational database implementation, and one NoSQL database. The systems under study are presented individually in Section 2.3.

Among the various types of data stored in these systems, a common recurring kind of entry is *time series*, *i.e.*, series of data points collected at discrete intervals and indexed by time, either coming from the physical world or produced by other digital processes. These data are sufficiently ubiquitous, and present sufficiently *sui generis* characteristics, to have spurred the development of specialised solutions to handle them[4, 5]. We go into more detail on the peculiarities of time series data in Section 2.2.

Given the importance of databases in modern computing, and the vast

amounts of time series data being generated, it is crucial to evaluate the capabilities of different DBMSs to effectively handle this type of data. Understanding the fundamental principles of how a DBMS operates, and how it can be optimised for specific workloads, is key to making informed decisions on selecting the appropriate solution. Section 2.4 presents relevant results from related works, providing an overview of the field of DBMS performance evaluation and setting the stage for our own study.

2.1 Databases

A database is a structured collection of data that is organised in a way that enables efficient storage, retrieval, and management of information. It should be logically coherent with some inherent meaning, and it cannot be considered a database if it consists of random or unrelated data. A database is designed, built, and populated for a specific purpose, with an intended group of users. It represents an aspect of the real world, also known as the miniworld or universe of discourse.

The end users of a database may perform business transactions or events may happen that cause changes to the data in the database. To ensure the accuracy and reliability of a database, it must accurately reflect the miniworld it represents, and changes therein must be reflected in the database as soon as possible[2].

When we talk about using a database, we use some well-defined terminology about different operations relating to the database. Specifically, *queries* are used to retrieve data from a database, and *to query* will also be used in the verbal form. Furthermore, *transactions* are used to modify data in a database, and *events* are changes in the real world that require updates to a database[2].

Databases are typically designed to store and manage large volumes of information that can be accessed by multiple users or applications. They are commonly used in business, research, and other fields where large amounts of data need to be organised and analysed. For instance, a retail store might use a database to keep track of its inventory levels, sales data, and customer information. Researchers also rely on databases to store and analyse large amounts of data, such as genomic data or climate data. Thus, databases are of interest both commercially and academically.

2.1.1 Database Management Systems

Database Management Systems are software programs that are designed to manage and maintain databases. They provide an interface for users or applications to interact with the database and perform operations such as adding, modifying, and retrieving data. It allows for manipulating the database through functions such as querying the database to retrieve specific data, updating the database to reflect changes in the miniworld, and generating reports from the data. Sharing the database enables multiple users and programs to access the database simultaneously[2].

The DBMS also facilitates the processes of defining the data types, structures, and constraints of the data to be stored in the database, while creating it or managing the database schematics. This descriptive information is stored by the DBMS in the form of a database catalog or dictionary, which is also called metadata[2].

Furthermore, there are security benefits of using DBMSs, in that the system manages the security and integrity of the data in the database, ensuring that only authorised users can access and modify the data. 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[2].

With new applications and areas of use, the most general purpose DBMSs have been complemented with additional functionality in the form of add-on modules and extensions.

2.1.2 Types of databases

There are several types of databases, including relational databases, object-oriented databases, NoSQL databases, and graph databases, each of which has its own unique structure and purpose. Naturally, each type of database has its own advantages and disadvantages, and will be more suitable for certain applications than others. As mentioned, this study is a comparison between a relational DBMS and a NoSQL document database.

Relational databases

A relational database consists of structured data. The data is stored in tables, where each table represents a specific entity in the miniworld. Each table consists of rows and columns, where each row represents a single record, and each column represents a specific attribute of the record. This structure

ensures that there is no redundancy in the data, such as the same value being stored twice. We will not delve into this further here, but the interested reader is urged to pick up any textbook on databases or scour the internet for details.

The database does not only include the data, but also information about the structure of the data. This is called the metadata, or the database schema. This includes information about tables, columns, data types and how data in one table can be related to data in another table. Designing the schema is up to the database administrators, or, in this study, the authors in conjunction with the company supervisors. We will elaborate in future chapters on how this was accomplished.

Database systems based on the relational data model almost universally adopt Structured Query Language (SQL), a domain-specific language developed for this explicit purpose, as their query language. Thus, they are also commonly referred to as SQL systems.

NoSQL databases

With the arrival of the 21st century, all sorts of new applications entered human daily life. Social media, e-commerce, cloud storage and more has increased the amount of stored data and servers globally to an unfathomable extent. A lot of this data is not textual-numeric, but of many types such as pictures, documents, videos and more. Traditional relational databases will simply not suffice to store all of these different types of data and manage such huge databases.

This has led to the introduction and development of NoSQL systems. While the standard query language for relational databases is SQL, these databases use no SQL, or at least not only SQL. The NoSQL systems use various data models, such as document-based, graph-based, column-based and key-value data models[2]. Since in this text we are concerned with time series data, columnar and document databases seemed the more appropriate fit; we have ultimately chosen to look at a document NoSQL system because we expect it to be a larger departure from the relational model, and thus the comparison to be more informative. Additionally, some of the popular columnar storage solutions we surveyed did not seem to offer specific support for time series data.

2.1.3 DBMS qualities

When evaluating database systems, there are several axes along which they can be compared. Some of the most commonly sought-after performance-related metrics are response time, throughput, and scalability. Response time,

also called latency, is the wait time between the submission of a query to the database and result being produced and returned. Throughput refers the number of queries the database can handle in a given time period, or sometimes to the amount of data (measured either in record count or bytes) that can be processed in a given time period by one or multiple queries. Scalability is the ability of the database to maintain good performance when the amount of data or the number of concurrent users increases, including the support for distributed database systems that enable the users to divide the load across multiple physical machines.

2.2 Time Series Data

It is often the case that whenever something is recorded or measured, the resulting piece of information is associated to the time of the recording or measurement. This information is of crucial importance in order to analyse temporal patterns in the data.

The data points might be coming from physical processes (*e.g.*, measurements registered by sensors, or miscellaneous information manually recorded), or be generated by other computational processes (*e.g.*, logs or statistics on data processed by some service, or the output of simulations).

Such collections of data comprise *time series*, and find applications in sectors as disparate as, among others, medical surveillance[6], analysis of financial instruments[7], investigation of gene expression[8], and naturally weather reporting and forecasting[9].

It is then not surprising that considerable efforts have been expended to streamline and enhance their usage in various aspects, and in particular DBMS designs have been developed to specifically accommodate time series data. This way, performance in terms of speed and storage volume can be significantly improved, as time series data possess specific characteristics that can be leveraged to optimise database design and improve performance.

One of these characteristics is the large volume of typical time series records[5], as many users need to retain a long history, or wish to store samples taken at exceedingly fine temporal resolution — one needs to look no further than climate measurements stretching back centuries[10], or readouts from sensors with sampling frequency in the tens of kHz range, to convince oneself of this.

Furthermore, the data is usually recorded in a sequential manner, and inserted into the database in the same order, resulting in an essentially append-only write pattern. Likewise, reads are typically clustered in time, as several

uses of time series data involve retrieving the most recent records, or a subset of the data in a given time interval[4]. This is in stark contrast to the typical DBMS usage pattern, where data is inserted and read in a random order, and where the write accesses include frequent updates and deletes, often concurrently.

Additionally, time series data is often subject to a high degree of redundancy, as subsequent measurement will sometimes repeat previously observed values, or differ only in some of the recorded properties.

In the following section, we will discuss various DBMS designs that have been developed to take advantage of these properties.

2.3 Selected DBMSs for Implementation

For this study, we selected two particular DBMSs in which to implement a time series database for our comparison. One is a relational database with a time series specific extension, and the other is a NoSQL document database with an out-of-the-box time series tailoring. Below will follow some specifics for these two choices.

2.3.1 PostgreSQL Timescale extension

PostgreSQL, commonly also referred to as Postgres, is a mature and widely used open-source RDBMS, and is the default database for numerous DBMS-based applications[11]. Its wide availability (being compatible with all major operating systems), its high degree of compliance with the SQL standard, and its reputation for reliability and stability have contributed to its popularity.

PostgreSQL is designed to be extensible, allowing for third-party code to expand the functionality of the database. One such extension is TimescaleDB[12], a time series specific extension to PostgreSQL that is designed to be able to handle large amounts of data while maintaining high performance and scalability, without changing the way users interact with the database and query the data.

Timescale achieves this by offering table *views* (so-called *hypertables*) of time series which are backed by separate tables, one for each time interval (*chunks*), thus using a partitioning mechanism to split the data into smaller chunks[13]. These can additionally be partitioned across different storage nodes along time and other properties of the data, allowing for horizontal scaling of the database[14].

This design is intended to allow for efficient querying of the data by leveraging the typical usage pattern of time series data (where, as mentioned, the data is often read in a clustered manner with respect to their time value or other parameters), without the need to manually restructure the database schema or design more complicated queries.

2.3.2 MongoDB

MongoDB is a document-oriented NoSQL DBMS that stores data in JSON-like documents with dynamic schemas, *i.e.*, with no predefined document structure. It supports nested subdocuments and arrays, offering greater flexibility in how to express relationships between data compared to relational DBMSs.

Like PostgreSQL, it is open-source[15], and has garnered a large user base with the rise of NoSQL-based applications.

MongoDB *collections*, its analogue to tables in relational databases, can be tweaked in various ways to optimise for specific use cases. In particular, MongoDB includes out of the box a specific collection type tailored to time series data, designed to reduce disk usage and improve query times[16].

These time series collections are implemented as views over an internal collection using columnar storage and time as a clustered index, in a way not too dissimilar to the TimescaleDB extension for PostgreSQL.

MongoDB offers a flexible and powerful querying mechanism through its aggregation framework, wherein a query is composed of a series of stages that progressively transform a collection of documents into a new set of documents. This sequence of transformations is referred to as an `aggregate` query.[17]

2.4 Related Work

Comparative analysis of DBMSs has been a popular research topic for several years, with researchers seeking to identify the most suitable system for specific use cases. As technology evolves and new data management challenges arise, the need for updated comparisons remains constant. The focus of these comparisons can vary widely, from evaluating the performance of different systems on various workloads to assessing the scalability, availability, and cost-effectiveness of different architectures.

Many comparative studies have thus been conducted to evaluate the performance and capabilities of different database management systems. These analyses often focus on a specific type of data, such as spatio-temporal or

time series data, to determine which DBMS is most suitable for applications of that specific type. Others studies perform a more general evaluation, for instance taking into account factors such as scalability, security, and ease of use.

The studies discussed in this section focus on comparative analysis of different DBMSs with regards to performance aspects, ranging from response times to execution speed. Although the studies vary in their objectives and evaluation metrics, they share the common goal of identifying the most efficient DBMS for specific scenarios.

2.4.1 Comparisons of relational and NoSQL DBMSs

There have been numerous comparative studies conducted to evaluate the performance and capabilities of relational versus NoSQL DBMSs.

In one study, researchers compared the performance of three NoSQL DBMSs (Cassandra, MongoDB, and HBase) and one RDBMS (MySQL) on various workloads, examining the trade-off between throughput and latency. The study found that the databases traded blows on different dataset sizes, workloads, and throughput levels, with MySQL typically offering better or comparable performance except in the case of insertion-heavy workloads and the NoSQL options having slightly different strengths and weaknesses[18].

Two other studies, one by Győrödi *et al.,* [19] and a more recent one by Jose *et al.,* [20], compared the performance of MongoDB and MySQL on various workloads, with both observing widely superior results with MongoDB across all scenarios.

2.4.2 MongoDB vs PostgreSQL performance

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

In a study by Jung *et al.,* [21], the authors compared the execution time of several basic queries of MongoDB and PostgreSQL with varying dataset sizes. The study found that MongoDB largely outperformed PostgreSQL with all tested query types but `SELECT` queries, where PostgreSQL offered comparable or superior performance.

On the other hand, a study by Makris *et al.,* [22] and a follow-up study by Makris *et al.,* [23] compare MongoDB and PostgreSQL (with the PostGIS extension) in terms of their response times for operations on spatio-temporal data. The queries and underlying infrastructure under analysis in

their evaluation are based on concrete business scenarios. Their results show PostgreSQL outperforming MongoDB in most cases, with only one type of query favouring the NoSQL option. Moreover, their conclusions highlight the importance of using indexes to reduce the average response time, especially for MongoDB.

Overall, the performance of MongoDB and PostgreSQL may depend on the specific use case and workload, especially when dealing with domain-specific data and extensions.

2.4.3 Comparisons of DBMSs for time series data

A study by Grzesik and Mrozek[24] compared several DBMSs including two relational databases (PostgreSQL and SQLite) and three time-series databases (the TimescaleDB extension, InfluxDB, and Riak TS). The study evaluates the performance of the selected solutions in a resource-constrained environment suitable for edge computing in IoT and Smart Systems, deploying and testing the DBMSs on a Raspberry Pi.

The results of their benchmarks showed that PostgreSQL offers the highest rate of data ingestion, and InfluxDB and PostgreSQL alternating the best performance in terms of aggregation query execution time. TimescaleDB did not improve performance over PostgreSQL in any of the tests.

As their work focused on small datasets and resource-constrained environments, the results may not be representative of the performance of the DBMSs in other scenarios.

2.5 Summary

In summary, DBMSs are a piece of IT infrastructure important enough to be worth optimising, and many different ones are available, each with their own strengths and weaknesses. In this context, time series data presents both specific requirements and opportunities for optimisation, and specialised solutions have been developed which have already been the subject of research. However, the field is still young, and there is still room for improvement; the results of previous studies are often concerned with only some aspects of the problem, and occasionally seem *prima facie* contradictory. To some extent this is to be expected, as DBMSs are complex systems with many moving parts, and the performance of a DBMS is highly dependent on the specific use case and workload. This motivates further research into the topic, even with a somewhat narrow focus, which is the purpose of the present work.

Chapter 3

Methods

Here we report on the methodology that was employed in the study, describing research methods and experiment design. We explain the data we set out to collect, present our considerations on test environment and measurements, and elaborate on evaluation criteria for the conducted work, before concluding with an overview of our planned analysis.

3.1 Research Process

To address the research questions outlined in the previous chapters, we formulated a comprehensive course of action comprising several key steps. This section provides a high-level overview of the process undertaken.

The initial step involved breaking down the task given by the company into specific research questions and establishing clear boundaries for what could be effectively addressed. Subsequently, a review of existing literature was conducted to familiarise ourselves with the current knowledge and research perspectives in the field. Building upon the insights gained from the literature study, we reassessed our initial research questions and began outlining how to address it appropriately. This phase necessitated making a series of crucial decisions, such as selecting the most suitable DBMSs for the study, among other considerations.

Once we had made the necessary decisions, it was time to begin the implementation phase. Our tasks included designing schemas for the selected DBMS instances, downloading data through the openly available API, and populating the newly created databases with the acquired data. Zenon provided us with queries that would represent stereotypical use cases for time series data, which we then translated into the query languages specific to our

chosen DBMSs. Subsequently, we executed these queries and measured their performance by benchmarking the runtimes.

Following the completion of data collection, we proceeded to compile the gathered measurements and analyse the results. This involved various steps, such as creating visual representations through graphs and computing time differences. We examined the data not only in terms of its content but also with a focus on its validity and reliability. The forthcoming sections will provide a more detailed exploration of the data evaluation process.

3.2 Research Paradigm

The primary methodology employed in this study is based on a quantitative and positivist approach. A comparative empirical case study has been chosen as the preferred method. While the review of existing literature provides valuable insights, the authors feel that relying solely on literature may not yield specific enough answers to the research questions. Therefore, the decision was made to develop our own implementation to contribute new empirical data. It is anticipated that, given similar conditions, our results will be objective and replicable. Notably, comparative conclusions are expected to be largely independent of the researcher and hardware, exhibiting consistent relative deviations.

The nature of this study as a case study implies that different results may be obtained in other cases. While the results are replicable for the specific selected DBMSs, they may differ if different ones were chosen. The decision to conduct a case study, as opposed to including several DBMSs per database paradigm, was driven by constraints on time and resources. However, measures have been taken to ensure that the chosen case is representative and fair.

The main rationale for conducting a comparative study is that it provides a relevant answer to the research questions. If only one DBMS was studied in isolation, it would be challenging to evaluate the results without a basis for comparison. Without an independent value to compare against, measurements would have limited significance. For instance, the runtime of a query would merely reflect the performance of the hardware rather than demonstrate the aptitude of the DBMS for time series data without a point of reference. Therefore, including a comparison is crucial to provide meaningful insights.

3.3 Data Collection

Firstly, we would like to add a word of warning. In this report, the word "data" will be used a lot, but has diverse meanings. Since we are comparing databases, we operate on the data in these databases. However, our measurements for querying the data in the databases, also generates data — the data we are actually interested in. In other words, the databases have meta-data, and content data. Then our own research generates data in the form of measurements and results. Accordingly, the reader is urged to be somewhat cautious about which data is referred to at different points, while the authors will do their best to make it as clear as possible. The title of this section concerns the data we gathered from our own measurements and inquiries while the upcoming subsection details our rationale for the choice of the data we used to populate the databases.

3.3.1 The choice of dataset

Openly available data from the Swedish Meteorological and Hydrological Institute (SMHI) was used to populate the databases to compare. This is weather data that can be downloaded through a public API[25]. Weather data is a perfect application of time series data, since it is very typically measured on specific times and with certain intervals. This particular dataset also happened to be of interest to the host company of this study.

We decided to download all of the publicly available data and migrate it into our databases. This was still merely a subset of the data that SMHI themselves store. While SMHI collects measurements as frequently as every minute, they only publish hourly data for the public. Nonetheless, the hourly measurements encompassed billions of recorded values for various weather parameters across numerous weather stations nationwide. Thus, it constituted a sizeable dataset suitable for conducting a representative comparison between our selected DBMSs.

Mind that in this study we have little to no interest of the content of the data — we *do not* care about the air temperatures, cloudiness levels, precipitation, *etc.,* that can be derived thereof. *We are only interested in obtaining a large enough sample of time series data,* the content is irrelevant. Since weather data is a stereotypical instance of time series data it is a good option in its own accord. Adding the facts that it is also publicly available, and of particular interest to our task providers, it makes an excellent choice as the source of our dataset to operate on. There are no ethical complications in using the SMHI

data since it is publicly available and free to use, being distributed under a Creative Commons Attribution 4.0 license[26].

3.3.2 The choice of DBMSs

We chose to compare PostgreSQL with the TimescaleDB extension and MongoDB for this study. As mentioned in section 2.4, similar comparisons have been done before. We have not found one comparing these two DBMSs specifically for time series data. PostgreSQL with the TimescaleDB extension was a given choice for the relational DBMS, due to the fact that the host company wished us to investigate this extension in particular. The current database system used by SMHI is also PostgreSQL, but without the Timescale extension.

For the choice of the NoSQL database system, we were tasked with finding a suitable contender ourselves. Here we started by looking at which were the common go-tos commercially for NoSQL database systems time series data supporting. There is a vast amount of software designed for this specific purpose, but we found that some of them seemed to be optimised for easy and convenient data retrieval and visualisation rather than performance. Several options were commercial, and we preferred an open source program. Our choice eventually fell upon MongoDB, with integrated time series support in order to obtain a fair comparison. Another option could have been InfluxDB, but a similar study comparing TimescaleDB and InfluxDB had already been conducted, and we wanted to provide a novel contribution. MongoDB is one of the most used and well-known NoSQL DBMSs, and has been around since 2009, which makes it a solid choice for representing the NoSQL systems in this study.

3.3.3 The choice of queries

To benchmark our database implementations, we executed queries and measured the runtime of retrieving the desired data. The company provided us with a set of typical queries that hold significance in the context of a time series database, in plain text form. These are presented in section 4.4. These queries were designed by the domain expert at the company, considering common use cases for such a database. Therefore, our measurements aimed to reflect real-world usage scenarios, ensuring the databases were evaluated in a relevant context. For instance, if database A outperforms database B in a specific task but that task is rarely performed, it may not be a significant criterion

for choosing database A over database B. Hence, it would not be relevant to our research question and not of interest in our analysis. Other tasks could have been chosen for the DBMS comparison, which might have given different results.

For the replicability of this study, to ensure similar results the same queries should be used. For the purpose of extensive research or future work, it could however be more interesting to explore other queries or tasks. Details of the queries and tasks that were used for the comparative analysis of this study will be provided in Chapter 4.

3.4 Study Design

We designed our comparison to be as informative as possible, mindful of our time and resource constraints. All tests were run against the same hardware and software environment (a cloud server), and on the same dataset.

3.4.1 Test environment

We conducted our measurements on a t3.large EC2 AWS instance running Ubuntu 22.04, ensuring sufficient storage space (300 GB) to fit the raw data and the backing store for the DBMSs under test. The instance has 2 vCPUs and 8 GB of volatile memory.

We installed PostgreSQL version 15.2, TimescaleDB version 2.10.3, and MongoDB version 6.0.6 on the instance. These are the latest available versions at the time of writing.

3.4.2 Measurements

The performance evaluation of the DBMSs was conducted by measuring the runtime of a predetermined set of queries. These queries were executed automatically using scripts that recorded the elapsed time for each query execution and saved the results for further analysis. To ensure the reliability of our findings, the queries were run multiple times, allowing for us to compute the average runtime and the standard deviation. This approach helped to mitigate the impact of random fluctuations and ensure the robustness of our results.

Naturally, this implies that care must be taken to consider the effects of caching at multiple levels (*e.g.*, disk caches, OS-level file caching in main memory, caching by the DBMS themselves, CPU caches, perhaps

even branch predictor warm-up effects) on the results[27–29]. To ensure an unbiased comparison, we repeated the queries a few times before recording measurements, so that our results are representative of consistently warm caches[28, 29].

3.5 Assessing Reliability and Validity

Reliability and validity are crucial considerations in any research study as they determine the credibility and trustworthiness of the findings. In this section, we will discuss the these aspects of our research methodology to ensure the accuracy and consistency of our results.

3.5.1 Reliability

In order to ensure the reliability of our results, we conducted all performance tests and benchmarks on a single external server. This approach guarantees consistent specifications in terms of cores, memory, and other relevant factors. Testing the different DBMSs on different hardware would yield meaningless results, underscoring the importance of using the same hardware throughout the study. To repeat our study, one would not need to use the same hardware that we did — the key is to maintain consistency in hardware usage across all tests. The reason for this is that we are solely interested in the relative difference in performance, rather than absolute numbers, since this study is comparative.

The general repeatability of our study can be considered high if the underlying conditions remain the same. By this, we imply that maintaining consistency in the dataset, chosen DBMSs, schemas, queries, and tests would yield, as mentioned, similar relative differences in the measured results, irrespective of the hardware on which the tests are conducted.

One assessment method that was employed for increasing the reliability of the study was the test-retest reliability, *i.e.,* , all measurements of runtimes were repeated several times, after which an average was calculated.

3.5.2 Validity

The metrics selected for evaluating the performance of the two DBMSs are directly aligned with the research objective, which is to investigate performance differences and quantify these differences through a comparative

analysis. This indicates a high level of construct validity, as the chosen metrics are conceptually linked to the research topic.

To ensure internal validity in our methods and data collection, we undertook measures to minimise the influence of confounding factors on our measurements. For instance, we considered cache behaviour, as described in section 3.4.2, and took care to prevent any interference from other processes running on the server during the tests.

The queries that were used for benchmarking were carefully chosen to span a broad variety of use cases and workloads that were all relevant for assessing the DBMSs' qualities and limitations in regards to time series data, in order to obtain results answering all of the various aspects implied by our research questions.

3.6 Planned Data Analysis

As a comparative analysis, our planned method revolved around contrasting the results obtained from the measurements of the two DBMSs. We decided to conduct this comparison at both a high level, evaluating overall performance, and more in detail, examining the performance of individual queries.

In our analysis, we planned to investigate both the absolute differences and the relative differences between the two DBMSs. This approach allowed us to gain a deeper understanding of the disparities in performance. To aid in the interpretation of our findings, we planned to employ graphical visualisations, plotting the data in intuitive formats.

We also planned to conduct simple statistical tests on the data, to provide insights into the statistical significance and magnitude of the observed disparities.

3.6.1 Software tools

We opted to make use of the popular NumPy[30] and SciPy[31] scientific computing libraries for Python to compute basic statistics and perform statistical analyses. We additionally choose the Matplotlib[32] library to generate graphs and charts in order to visualise the data.

To drive our benchmarks, we selected Hyperfine[33], a simple command-line performance measurement tool. It allows for the execution of a command multiple times, recording the execution time of each run and providing statistics on the results, and handling warmup runs and setup and cleanup procedures for each benchmark.

Chapter 4

Implementation

In this chapter, we will elaborate on the implementations of our chosen methods, which allowed us to obtain quantitative results and measurements. Given that our study is a comparison between two distinct DBMSs, we produced separate implementations for each database. While there were some shared elements, such as the metadata of our weather data, we still approached the decomposition of this metadata in slightly different ways.

4.1 Obtaining and Analysing Metadata

To construct suitable models and accurately translate the trial queries into their respective query languages, we needed to familiarise ourselves with the domain and gain an understanding of the properties and structure of the dataset, as it was made available to us.

The dataset can be accessed online through the SMHI open data portal[25], retrievable via a REST API. With the help of the documentation available alongside the data [34], we outlined the relations between different entities and attributes of the data — such as the weather stations and the different parameters that are being measured (temperature, precipitation, etc). During this process, we also had guidance from our supervisor at Zenon, who has extensive domain knowledge. This metadata as a whole was downloaded as JSON files through the script in Appendix B.1, by sending requests to the API.

4.2 Implementation in TimescaleDB

We will give a brief explanation of how we implemented our databases in PostgreSQL with the TimescaleDB extension here, and a detailed description

of how to run the SQL scripts in appendices.

4.2.1 Setup and designing a relational schema

By inspecting the metadata, studying the documentation of the API, and having thorough discussions with our company supervisors, we derived an Entity Relational Diagram. This diagram is shown in figure 4.1.

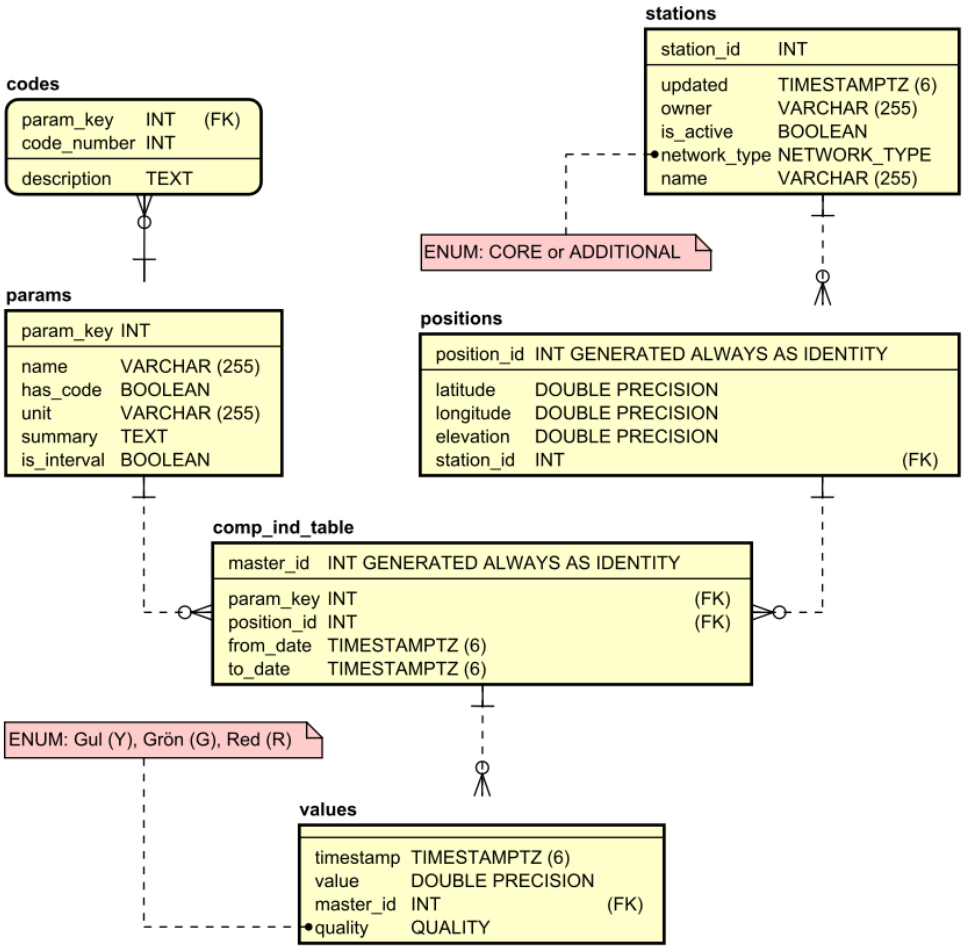

Figure 4.1: Entity Relationship Diagram for the PostgreSQL database.

In the diagram, each yellow box is an entity, while an arrow (a line) is a relation. The entities are tables in the database, and the relations represent foreign keys. The attributes above the black dividing line comprise the primary key of the table. All attributes in a box become columns in the corresponding table.

The diagram is drawn in Astah Professional, which allows for automatically exporting it as an SQL script, that can then be run to create the database according to the designed schema. We also decided to make use of some enumerables, a built-in class of custom datatypes in PostgreSQL, so we added these as a script that needs to be run before the main script. Since we are evaluating the TimescaleDB extension for time series data, we also need the values table to be a hypertable. In accordance with the TimescaleDB recommendations[13], we selected a chunk size of two years after initial testing. Running the scripts in sequence gives a database of our designed structure. The scripts are found in Appendix C.1.

4.2.2 Populating the Timescale database

First, we populated the metadata tables with the weather parameters and weather stations and their attributes. This was fairly easy, since all of these tables have a small amount of rows — about 40 parameters and 2500 stations. Admittedly, the table `comp_ind_table`, which in brief contains all combinations of parameters and stations, has over 20 000 rows, but this is still easily handled by any common relational DBMS.

Then it was time to populate the `values` table with the actual measurements, and here we are talking in total over one billion rows. The weather measurement data was downloaded and imported into PostgreSQL using the script in Appendices B.3 and C.2, respectively. Our data ingestion required slight processing to compute some of the properties of the data that were implicit in the representation obtained from the API. Due to the large volume of data, it proved crucial to use an efficient method for importing the data into the database, as naïve solutions would require several days of continuous execution*.

After discussions with our company supervisors, we concluded that the likely bottleneck was the structure of the Timescale time series hypertable, which is tailored for use cases where insertions are made in chronological order, while our data processing would result in backfilling measurements from the past for each station.

We thus adopted their suggestion to first import the data into a simple temporary table (with no indexes), then allow for sorting the table by time by building an index, and finally populate the final time series table by copying over the data from the temporary table in sequential time order. This approach

*As we found out the hard way.

proved to be considerably faster than our first attempt, and allowed for a complete import in under two hours.

4.3 Implementation in MongoDB

The MongoDB implementation differed from the TimescaleDB implementation in several ways. Instead of a relational schema, there is a semi-structure for the document collections. Instead of importing all of the CSV files obtained from the API into a relational, we took advantage of the fact that we had already conducted the relational implementation.

4.3.1 Setup and designing a document structure

In MongoDB, the documents are divided into collections. To have a rigorous schema is not as important as in the relational case, but there still needs to be some kind of structure to the dataset storage. From the structure of the metadata we designed the structure of the MongoDB collections to be used. After some experimentation, we decided to keep it similar to the relational implementation, as testing suggested that querying against embedded subdocument arrays would in practice result in worse performance. We then set up the database accordingly with the scripts in Appendix C.3.

4.3.2 Populating the MongoDB database

Initially we attempted to import the data in a similar fashion as with PostgreSQL, using a temporary collection to avoid non-chronological insertion into the time series collection, but we observed this to be much slower in MongoDB, to the point where it would not be feasible within our time limitations.

We conjectured that the required preprocessing and the inability to disable the main index in MongoDB (the one enforcing uniqueness of required document identifiers) were limiting factors in building the temporary collection. We thus opted to export the full dataset from PostgreSQL into a CSV file, and then import it into MongoDB using the `mongoimport` tool. This proved significantly faster, confirming our expectations and allowing us to complete the import within a few hours. Our choice to structure the data model similarly helped make this approach fairly straightforward to implement. Our script for migrating the data to MongoDB is found in Appendix C.4.

4.4 Query Translations

The queries that were to be examined and benchmarked can be loosely formulated in common language in the following ways, as provided by the host company's domain expert.

1. Monthly average temperature for each year for ten preselected stations, based on hourly temperature values.
2. Minimum temperature on Christmas Eve for every year on record.
3. Wind and temperature for ten preselected alpine stations when temperature is below $-20\,^{\circ}\mathrm{C}$.
4. Hourly precipitation values per station for one year.
5. Temperature and wind speed average for each week of the year and day of the week since 1996, per station.

These informally specified queries were translated into the set of SQL queries in A.1, and the set of MongoDB queries in A.2. We strove to translate the queries in a way that let each DBMS make use of its advantages and strengths, for a fair comparison. The first query was tested with the ten station having the longest records of temperatures. The fourth query was tested separately with years 2022 and 1996, the first representing a selection of recent measurements and the second a selection of more dated ones (but still containing considerable amounts of data); this allowed us to investigate whether the time series–specific solutions result in different performance based on the age of the data.

4.4.1 SQL

There are many ways to write a query that would give the same resulting table. While translating the queries, considerable effort was put into making them optimal for the database in question. Since we are using Timescale, which is constructed to be fast for time series data through making use of hypertables split up into chunks, we wanted to formulate the queries so that this could be utilised in a beneficial way. This means applying time-based filtering, in such a way that the query execution can take advantage of the hypertable structure. We also built indexes on frequently scanned columns, as to obtain the speed of index searches instead of sequential scans when possible.

In order to evaluate and improve our queries, we made use of the `EXPLAIN` and `EXPLAIN ANALYZE` commands in PostgreSQL, allowing us to inspect the execution plans and make sure that our formulations did not prevent some expected optimisation from being applied, and ran initial timing tests against a test database (containing a small subset of the full data), to get a rough idea of the performance of the queries and compare different formulations.

The queries were expressed as PostgreSQL stored function and loaded in the database, for ergonomics during testing and to allow for simple parametrisation of the queries (*e.g.*, for the two different years used with Query 4).

4.4.2 MongoDB query language

Since the structure for our MongoDB database was similar to our PostgreSQL schema, the core structure of our queries was not too far from the SQL ones either. Care was however given to adapting the queries to better serve the document database strengths, and to compensate for the shallower optimisation strategies applied by MongoDB. Chiefly, this meant picking a good ordering of the stages of `aggregate` queries, with filtering steps applied as early as possible, and ensuring that `$lookup` pipelines did not repeat expensive operations when possible.

Again we made use of the corresponding `explain()` command to inspect the execution plans of our queries, and ran tests on different formulations. However, given the lower familiarity of the authors with NoSQL databases, potential improvements may have been overlooked.

MongoDB does not offer an equivalent to stored functions[35], so the queries were simply written in the MongoDB scripting language to files.

4.5 Performance Measurements

To execute the queries against the databases we invoked (through Hyperfine) `psql` and `mongosh`, the default clients for PostgreSQL and MongoDB respectively. Hyperfine executed the commands multiple times (including warmup runs not used for the results) and reported the execution times, as well as mean, standard deviation, and median.

In the PostgreSQL case, for each query we supplied the call to the corresponding stored function as an argument to `psql`, while in MongoDB we provided the file containing the query script. It is worth noting that these

tools add some overhead to the execution of the queries, as would any other client used to issue commands to the respective database servers.

To minimise the impact of this overhead on our results, we attempted to correct for it by measuring the execution time of the empty query for both, and subtracting it from our actual query measurements. The result is reported in table 4.1.

Table 4.1: Runtime of the empty query for each DB client (in seconds).

Client	Overhead (s)
`psql`	0.06 ± 0.01
`mongosh`	1.6 ± 0.2

As can be seen, the overhead introduced by `mongosh` is much larger than that of `psql`, thus not taking it into account would introduce a source of bias in our measurements. Since our queries have fairly long execution times, the relevance to our results is likely negligible, but in another comparison where the queries under test were much faster (*e.g.*, typical web service queries with runtimes measured in fractions of a second), neglecting to account for this could completely invalidate the conclusions.

Chapter 5

Results

We devote this chapter to presenting the results of our study. We start with an overview of all our results in Section 5.1, then, in Section 5.2, we present in greater detail the results of individual queries.

5.1 Overview of Comparative Results

This section provides a summary of the comparative results obtained from the study. We present timing results for the analysed queries in Table 5.1, and visualisations of the results in Figures 5.1 and 5.2.

Table 5.1: Summary of timing results for the respective queries in elapsed minutes, and ratio of MongoDB result divided by TimescaleDB result.

	Runtime (minutes)		Ratio
	TimescaleDB	MongoDB	
Query 1	73.48 ± 0.11	1.97 ± 0.05	$0.026\,8 \pm 0.000\,7$
Query 2	6.9 ± 0.5	36.4 ± 1.2	5.3 ± 0.4
Query 3	68.88 ± 0.04	52 ± 2	0.75 ± 0.03
Query 4-1996	0.138 ± 0.004	31.7 ± 1.6	229 ± 13
Query 4-2022	0.21 ± 0.03	31.2 ± 1.1	146 ± 18
Query 5	7.7 ± 1.4	118 ± 17	15 ± 4

The overview in Table 5.1 presents the absolute runtimes in minutes for the respective queries. Each query was executed nine times in each DBMS, and the mean of these runs is displayed in the table. The second column of the table presents the mean of these three runs. Additionally, we include a third

column indicating the ratio between the runtime means of the two DBMSs. This ratio illustrates the relative difference in performance. It was calculated by dividing the absolute value obtained in MongoDB by the value obtained in Timescale. Accordingly, a value below 1 in the third column qualifies MongoDB as the faster DBMS for that query, and a value above 1 means that Timescale outperformed MongoDB — *e.g.*, if the value is 20, it means that MongoDB was 20 times slower than Timescale.

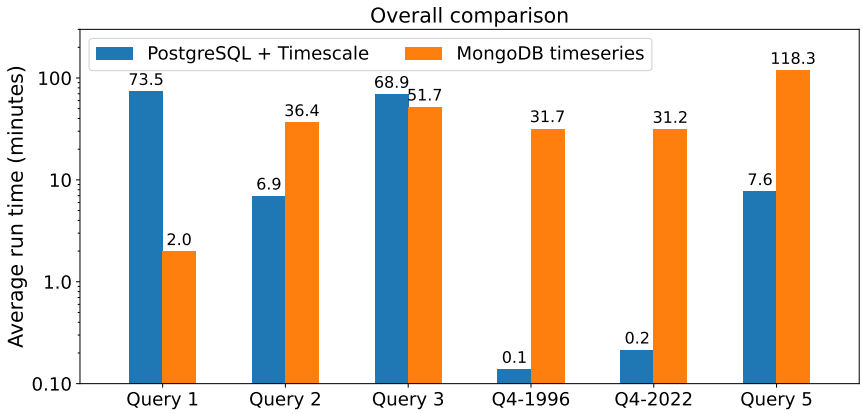

Figure 5.1: Absolute runtimes of the queries for the two databases.

Figure 5.1 depicts the same values as Table 5.1, but in the form of a bar chart. In this chart, a higher bar indicates a longer execution time. Note that the scale of the graph is logarithmic, meaning for example that although MongoDB was more than five times slower than Timescale for Query 2, the corresponding bar on the graph is less than 50 % higher. While logarithmic scales may be unintuitive, this choice was necessary due to the huge variations for the measurements combined with the desire to fit all of them within the same plot. For more visually intuitive graphs, we refer to the query-by-query results provided in the next section.

Since this study is comparative, we are interested in the relative performance of the DBMSs rather than absolute numbers. This has been previously discussed in Chapter 3. Hence, in Figure 5.2 we depict the ratio of the MongoDB value divided by the Timescale value. These ratios correspond to the values in the third column of Table 5.1. Given the disparity of the values, we have again opted for a logarithmic y-axis to encompass all values within the same graph.

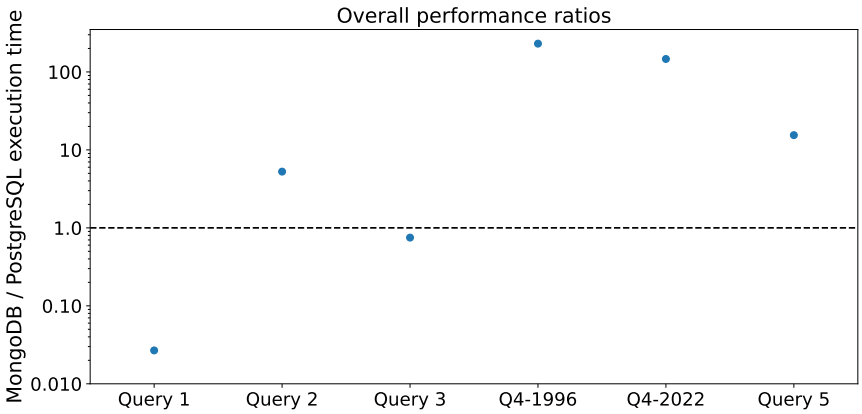

Figure 5.2: Ratio of MongoDB runtimes over PostgreSQL runtimes.

5.2 Query-by-Query Results

To enhance the visualisation of our collected results, we also look at the queries individually. The specifications for the queries were outlined in section 4.4, therefore here we will only briefly mention their contents.

For each query, we present a corresponding graph in the form of a bar chart displaying the mean of nine runs for Timescale and MongoDB. Potential factors contributing to the observed results will be discussed in Chapter 6, Analysis and Discussion.

5.2.1 Query 1

The first query requires compiling average monthly temperatures for all years on record at ten selected stations. Figure 5.3 shows a significant advantage for MongoDB, beating Timescale by a factor of 35 for this query.

5.2.2 Query 2

Query 2 concerns only Christmas Eve every year, but still requires picking this date from each year within the data. Given our translations of this query, it appears that Timescale is considerably faster at performing this task, with less than a fifth of the average runtime for MongoDB, as can be seen in Figure 5.4.

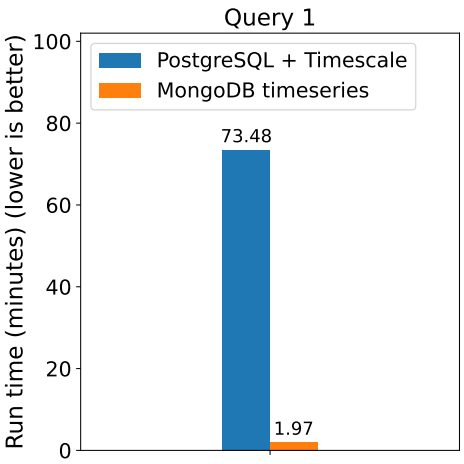

Figure 5.3: Plot of the runtimes of Query 1.

Table 5.2: Individual timing results for single runs of Query 1 (minutes).

PostgreSQL	MongoDB
73.6	1.93
73.6	1.84
73.4	1.97
73.5	2.02
73.3	2.00
73.5	1.99
73.4	1.96
73.4	2.03
73.6	1.99

Table 5.3: Individual timing results for single runs of Query 2 (minutes).

PostgreSQL	MongoDB
5.45	34.6
7.07	35.6
7.06	36.1
7.09	35.3
7.08	35.8
7.13	36.9
7.09	38.6
7.11	37.8
7.11	36.8

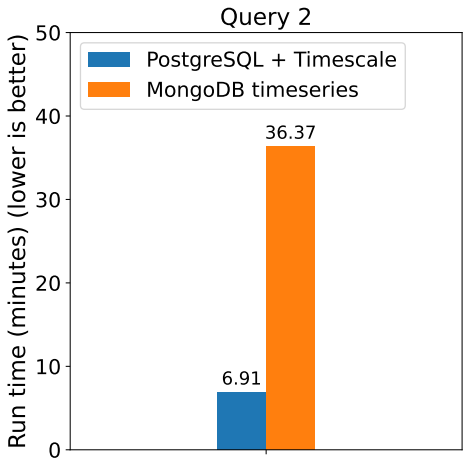

Figure 5.4: Plot of the runtimes of Query 2.

5.2.3 Query 3

Like Query 1, the third query also pre-defines ten hand-picked stations. We ask for the values of two different weather parameters (temperature and wind) for each of these ten stations, presented on the same row or document level. Furthermore, there is a limit on one of these parameters — we only include records where the temperature is below $-20\,^{\circ}\mathrm{C}$. Inspecting Figure 5.5, we see that the performances are of the same magnitude for the two DBMSs in regards to this query, with MongoDB being about $27\,\%$ faster.

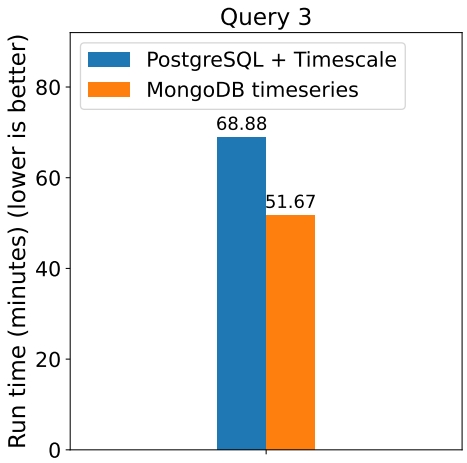

Figure 5.5: Plot of the runtimes of Query 3.

Table 5.4: Individual timing results for single runs of Query 3 (minutes).

PostgreSQL	MongoDB
68.9	50.2
68.9	50.1
68.9	51.8
68.8	52.3
68.9	57.2
68.8	51.6
68.9	50.3
68.9	50.2
69.0	51.4

5.2.4 Query 4

We ran Query 4 for two separate years, as mentioned in section 4.4. In this query, all values for a certain weather parameter (precipitation measured every hour) during a specified calendar year are selected. Figure 5.6 shows the values for both of these years for Timescale and MongoDB respectively. As can be seen, Timescale outperforms MongoDB by several magnitudes, averaging less than 15 seconds per year, while MongoDB needed more than half an hour to retrieve the requested data.

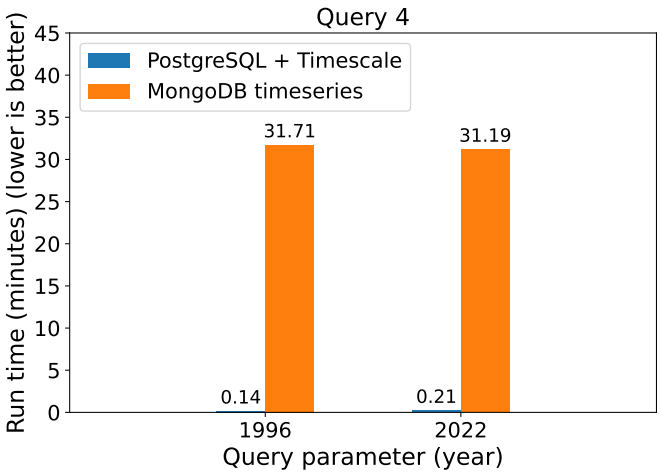

Figure 5.6: Plot of the runtimes of Query 4.

Table 5.5: Individual timing results for single runs of Query 4 (minutes).

1996		2022	
PostgreSQL	MongoDB	PostgreSQL	MongoDB
0.135	30.3	0.207	30.3
0.146	30.3	0.200	33.6
0.137	31.2	0.195	30.0
0.139	31.2	0.212	31.5
0.134	32.8	0.184	32.0
0.134	31.2	0.193	30.7
0.141	30.5	0.221	31.1
0.140	32.6	0.270	30.0
0.140	35.4	0.240	31.5

5.2.5 Query 5

The fifth query specifies limitations on the timestamps by extracting weeks and weekdays, and computing total averages for two different parameters restricted to these partitions. The data is also divided per station. All values since 1996 were included in the result set. This workload seems to have heavily favoured Timescale, for which the execution time averaged below 8 minutes, meanwhile MongoDB required more than 1.5 hours, making it more than 12 times slower. This query also seems to be the query for which the standard deviation is highest. Figure 5.7 shows a plot for the execution times of query 5.

Table 5.6: Individual timing results for single runs of Query 5 (minutes).

PostgreSQL	MongoDB
6.83	92.0
6.91	102.9
6.94	96.3
7.12	106.5
6.90	132.2
6.79	132.9
6.83	133.3
9.55	135.5
11.0	132.8

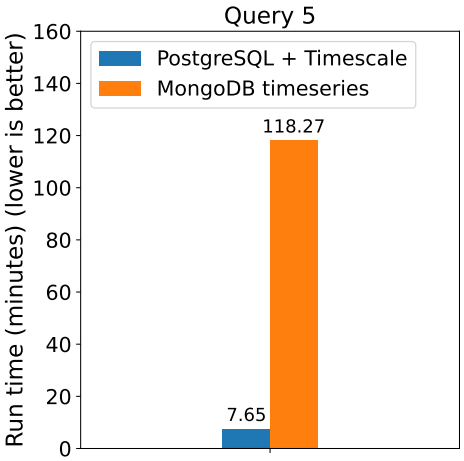

Figure 5.7: Plot of the runtimes of Query 5.

Chapter 6

Analysis and Discussion

We now proceed to analyse the results in the previous chapter. First we discuss the queries one by one in detail (Section 6.1), and then we examine the overall performance and identify common factors observed across the queries (Section 6.2). Additionally, reliability and validity concerns are discussed in Sections 6.3 and 6.4 respectively.

6.1 Query-by-Query Analysis

We can immediately observe considerable variation both in the absolute runtimes of the queries and in the relative results obtained from the two databases. The former vary from a few seconds to several hours, and the latter show difference in performance of more than an order of magnitude in both directions.

The reasons behind the large variation in performance are likely to have distinct causes, which we will discuss in detail in the following subsections.

6.1.1 Query 1

Query 1 is one of the queries that showed the largest difference in performance between the two databases, with MongoDB beating Timescale by a factor of 35. The query requires processing a relatively small subset of all values, as we only need the measurements relative to a few stations. However, since we are interested in every year on record they span the whole temporal range of the dataset.

It is likely that the structure of the Timescale table is not well suited to this type of query (at least as formulated by us), perhaps requiring it to swap

temporal chunks in and out of memory as it traverses the index picking out the `master_ids` corresponding to the selected stations. In contrast, it appears that MongoDB can effectively handle our query implementation, making good use of indexes and finding the relevant documents quickly.

6.1.2 Query 2

Our implementations of Query 2 for the two DBMSs are fairly similar in structure, with the main difference being the attention paid to the order of operations with MongoDB.

Again we are interested in a small subset of all values, but this time the condition we use to filter the data (selecting measurements on the 24th of December) is not amenable to indexing. As such, our query requires a full scan of the values corresponding to minimum temperature measurements, and it seems that Timescale is better at handling such a scan.

In this specific case, the query's performance could be improved in both DBMSs by ad hoc solutions like building indexes over materialised views of the values including the computed properties we use for filtering, but this comes at a cost of increased storage requirements and would not benefit different filtering conditions. A slightly more general optimisation could be to produce a generated collection which incorporates properties used for filtering (*i.e.*, in this case, a list of timestamps corresponding to the Christmas Eve of every year on record) and using it as part of our query. It is possible that this approach would allow the databases to make efficient use of their time series-specific optimisations, but this was not attempted due to time constraints.

6.1.3 Query 3

Similarly to Query 1, the third query requires processing a small subset of all values based on selecting a few measuring stations. In this query, however, we apply further filtering (to only select low temperatures), apply different processing (to pair up the measurements of two separate parameters), and ultimately produce more data. For Timescale, we observe similar performance as for Query 1, but for MongoDB the query takes significantly longer to complete, drastically reducing its advantage and bringing it almost on par with Timescale.

We believe that the reason for this is found mainly in the increased complexity of the pipeline stages of the MongoDB query, which might reduce the database server's ability to optimise the query execution. It is, however,

also possible that MongoDB's performance for this type of query is more sensitive to the size of the result set.

6.1.4 Query 4

We observed the largest relative difference in performance in Query 4, with Timescale surpassing MongoDB by a factor of over 100. The intrinsic structure of this query sharply constrains the opportunities for optimisation, as it simply singles out a narrow time range, selecting all precipitation measurements from it and applying no further processing.

As this seems well suited to Timescale's strengths, its good absolute performance is not surprising. However, it appears that MongoDB's time series solution is not equally adept at taking advantage of the structure of the data. In fact, to our surprise the MongoDB query as originally designed (with the time period–based filtering applied first in the pipeline) performed even worse than the one ultimately used in this comparison (which first selects precipitation measurements and only then filters them by time period).

As this seems such a central use case for time series databases, MongoDB's poor performance is surprising. While it is possible that we have not managed to fully take advantage of MongoDB's capabilities, considerable effort was spent attempting to optimise the query. Even if the performance ideally attainable proved to be superior, the difficulty in achieving it for such a straightforward example would be a significant drawback of adopting MongoDB as DBMS.

We additionally note that the amount of data returned by the query for 2022 is roughly 1.5 times larger than for 1996, which seems to account perfectly for the difference in Timescale's execution time across the two parameters. In contrast, MongoDB's execution time remained approximately the same, suggesting that the bottleneck for its performance is not in the data processing step. Possibly, it could be related to inefficient disk access patterns, resulting in I/O latency dominating the query execution time.

6.1.5 Query 5

In Query 5 we require a pairing of different parameters in resemblance to Query 3, but a notable difference is that it requires processing all measurements instead of being limited to a small subset of stations.

Timescale is able to process the query more than an order of magnitude faster than MongoDB, and in considerably less time than it takes to process

Query 3. This is somewhat puzzling given the larger amount of data processed. Conversely, MongoDB's runtime is about doubled compared to Query 3, which continues the trend of MongoDB's performance being more sensitive to the amount of data processed, although this consideration alone would lead us to expect an even greater slowdown. We hypothesise that this query allows for more efficient disk access, reducing I/O overhead compared to Query 3, giving support to our interpretation of the previously discussed result.

6.2 Combined Analysis

Our benchmarks do not indicate a clear winner between the two DBMSs, with each showing massively superior performance over the other in some cases. Although Timescale was more frequently the faster of the two, and by a larger margin, the huge gap favouring MongoDB in the first query renders any claim of distinct superiority untenable.

It is worth noting that during our testing, we observed that the performance of MongoDB was extremely sensitive to the precise formulation of the query. Some initial attempts resulted in prohibitive execution times even for a test database less than a hundredth of the size of the one used for our benchmarks, and successive tweaking yielded improvements of over an order of magnitude.

This was to some extent anticipated, as MongoDB only performs a limited set of optimisations on its query pipelines[36]. Instead, the user is expected to mindfully design their queries and take advantage of their knowledge of the structure of the dataset, in order to realise the best possible performance.

By contrast, PostgreSQL's query planner is able to deploy a much wider range of optimisations, as the rigid and declarative nature of SQL allows for more transformations to be applied to the submitted query. This is perhaps the reason we found it easier to achieve acceptable performance in our construction of SQL queries.

This leaves open the possibility that with more time at our disposal we might have been able to find better performing formulations, especially in the case of MongoDB, or even a better structure for our data, enabling more efficient querying.

Overall, we were able to achieve much greater performance with Timescale with queries selecting values from a restricted time period, and with MongoDB when picking out a small subset of values (identified by a separate indexed property) across large time spans. With queries that require processing large amount of values from large time spans, or non-indexed subsets, Timescale proved again to be the faster solution.

6.3 Reliability Analysis

We generally observe very tight dispersion across reruns of the same query for both DBMSs — except for Query 5, where two runs of Timescale took considerably longer than the others.

This larger variation is unexpected, and may point to a less stable environment than we initially assumed. This would imply the need for more measurements in order to be confident in the representativeness of the computed averages, but as the difference in runtime between the two DBMSs was in all cases of much greater magnitude, we believe it does not impact our results.

Additionally, the sporadic appearance of these variations suggests that, whatever the cause, it is an uncommon occurrence and localised in time. Thus, the sets of measurements where we observed very consistent performance were likely simply not affected by these anomalies.

6.4 Validity Analysis

Our research questions are concerned with time series data in general, while our implementation was conducted solely on weather data. Weather data is assumed to be a representative sample of time series data, so this in itself should not introduce any biases. However, the queries selected for benchmarking were modelled after miniworld weather data use cases, rather than general time series data use cases.

We found, while analysing the query execution plans, that several of these queries did not end up making use of the timestamp as index. Instead, they resulted in other index scans and sequential scans, and constraints on different attributes.

Our absolute runtimes are broadly comparable to other results found in the literature for complex queries on large amount of data[22, 23], which strengthens the validity of our results.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In our work, we set out to compare the performance of two popular Database Management Systems, MongoDB and PostgreSQL with the Timescale extension, for usage with time series data. We selected query execution time as the performance metric of interest, and SMHI's publicly available meteorological observations as sample dataset. Our supervisors at Zenon suggested representative inquiries to be performed against this data, that we implemented as queries for both DBMSs. We then executed these queries, recording execution time.

From our results, neither of the two systems emerged as consistently superior, but rather our benchmarks showed wide disparities in performance favouring either MongoDB or Timescale depending on the specific query. Given the large range of execution times spanned, we consider it likely that the structure of the query, and the tuning made possible by the properties of the data (*i.e.*, indexes and chunking), played a larger role in determining performance than the intrinsic capabilities and limitations of the DBMS.

While this may suggest that either system is better suited to the queries they performed well on, with the lower performing database unable to efficiently execute the query under consideration, it may also be indicative of a difference in how aptly the queries were implemented for each system. This is especially relevant in the case of MongoDB, where even simple queries can result in abysmal performance if not enough care is taken in their design.

In conclusion, we consider our results insufficient to fully settle the question of which DBMS is better suited for time series data. Nevertheless, they provide insight into the performance characteristics of both systems, and

the considerations to keep into account in order to reach the full potential of these systems.

7.2 Limitations

The main limiting factors of the study were time, in conjunction with computational power, disk storage, and disk speed. The first server used for the test environment was not big enough to store both databases with the complete dataset. As previously discussed, it was very important to use the same test environment for both databases, so this necessitated changing to a server with more storage space. This, together with the low speed of the import methods — partially due to the limited I/O performance of the server's disk storage — reduced the time available for implementation and data collection.

Several of the trial queries also had a substantial execution time. This meant that we were not able to perform as many runs as we would have wished.

Both the constrained storage space and the long runtimes limited our chances of running the queries against subsets of the data of different sizes, which we initially intended to do. Consequently, our results are restricted to a comparison for running the queries against the entire dataset, about one billion rows. Hence, we can only compare the DBMSs for this data amount.

Another limiting factor was the modest experience of the authors. The translation from common language to the respective query languages were likely not optimal, which circumscribes the comparative results between the DBMSs to these circumstances.

We furthermore find that we might not have exhaustively answered the first research question as stated in 1.2.3. While we knew from the beginning that we were conducting a case study, and thus letting each database paradigm be represented by only one DBMS out of many, these choices may have affected the results more than expected. Specifically, the extent of internal optimisations done by different DBMSs seems to vary substantially, likely even within paradigms, thus diminishing the completeness of our response to the more overarching of our two research questions.

When it comes to the second research question, the main drawback of our results are that the queries chosen might not accurately reflect the concept of "common time series data workloads". While chosen to represent miniworld use cases, in the end few of them ended up actually using the timestamp as an index. Benchmarking a larger set of more disperse queries would have been beneficial, but the time to do so was lacking.

7.3 Future Work

There are numerous ways to expand upon the field of DBMS comparison and determining their adequacy for certain applications. Which path to pursue depends on the level of detail one wishes to study. Narrow expansion would include running the same queries on differently sized subsets of our current dataset, or running additional queries on the same dataset. Further augmentation might entail testing the same DBMSs for completely different time series datasets, or other DBMSs on the same dataset, or again different reformulations of the same queries and different configurations for the same DBMSs.

Since this study springs from a desire to aid the selection of an appropriate database system for various time series applications, it would also be of interest to investigate other desired DBMS qualities. Performance could be further analysed in terms of CPU time, memory usage, or throughput, but, naturally, it will not be the only sought-after attribute. Qualities such as storage requirements, scalability, availability, stability, security and ease of use are all examples of other valued qualities. While time series applications might have exceptional need for high performance due to extensive amounts of recorded data, these other characteristics cannot be ignored.

7.4 Closing Remarks

Today, many companies have opted to store and host their data and applications in the cloud. Such solutions can easily quickly become expensive, as it is not uncommon that they are paid by used space or per transaction. Thus, an appropriate choice of DBMS can lead to large economic benefits for companies.

The amount of active servers accounting for cloud storage worldwide is unfathomable. These servers consume a lot of energy, which in turn has effects on the climate [37]. On a broader scale, decreasing storage requirements and CPU usage will save energy and thereby lower an application's ecological footprint.

While not conclusive, our results have been considered informative by our supervisors at Zenon. We hope that they may be further useful to other practitioners in the field, as well as provide a starting point for future research, leading to ever more effective solutions in deploying databases for time series data.

References

[1] P. Beynon-Davies, *Database Systems. 3rd Ed.* 1st Dec. 2003. [Online]. Available: https://www.semanticscholar.org/paper/Database-systems.-3rd-ed.-Beynon-Davies/c1464c8657bf2be54a1642046a390440b5f1f5dc (visited on 12/04/2023).

[2] R. Elmasri and S. Navathe, *Fundamentals of Database Systems*. Addison-Wesley, 2011, 1172 pp., ISBN: 978-0-13-608620-8. Google Books: ZdhAQgAACAAJ.

[3] D. Maier, *The Theory of Relational Databases*. Rockville, Md: Computer Science Press, 1983, 637 pp., ISBN: 978-0-914894-42-1.

[4] L. Bitincka, A. Ganapathi, S. Sorkin and S. Zhang, 'Optimizing Data Analysis with a Semi-structured Time Series Database', presented at the Usenix, 3rd Oct. 2010. [Online]. Available: https://www.semanticscholar.org/paper/Optimizing-Data-Analysis-with-a-Semi-structured-Bitincka-Ganapathi/19d02dfb848ae36e5c243dfa12e21d9cee5da83f (visited on 13/04/2023).

[5] S. Rhea, E. Wang, E. Wong, E. Atkins and N. Storer, 'LittleTable: A Time-Series Database and Its Uses', in *Proceedings of the 2017 ACM International Conference on Management of Data*, ser. SIGMOD '17, New York, NY, USA: Association for Computing Machinery, 9th May 2017, pp. 125–138, ISBN: 978-1-4503-4197-4. DOI: 10.1145/3035918.3056102. [Online]. Available: https://dl.acm.org/doi/10.1145/3035918.3056102 (visited on 05/04/2023).

[6] R. Allard, 'Use of time-series analysis in infectious disease surveillance.', *Bull World Health Organ*, vol. 76, no. 4, pp. 327–333, 1998, ISSN: 0042-9686. pmid: 9803583. [Online]. Available: https://w

ww.ncbi.nlm.nih.gov/pmc/articles/PMC2305771/ (visited on 14/04/2023).

[7] S. M. Focardi and F. J. Fabozzi 3, 'A methodology for index tracking based on time-series clustering', *Quantitative Finance*, vol. 4, no. 4, pp. 417–425, 1st Aug. 2004, ISSN: 1469-7688. DOI: 10.1080/14697680400008668. [Online]. Available: https://doi.org/10.1080/14697680400008668 (visited on 14/04/2023).

[8] Z. Bar-Joseph, G. Gerber, D. K. Gifford, T. S. Jaakkola and I. Simon, 'A new approach to analyzing gene expression time series data', in *Proceedings of the Sixth Annual International Conference on Computational Biology*, ser. RECOMB '02, New York, NY, USA: Association for Computing Machinery, 18th Apr. 2002, pp. 39–48, ISBN: 978-1-58113-498-8. DOI: 10.1145/565196.565202. [Online]. Available: https://dl.acm.org/doi/10.1145/565196.565202 (visited on 14/04/2023).

[9] G. Jain and B. Mallick, 'A Review on Weather Forecasting Techniques', *International Journal of Advanced Research in Computer and Communication Engineering*, vol. 5, no. 12, pp. 177–180, 30th Dec. 2016, ISSN: 22781021. DOI: 10.17148/IJARCCE.2016.51237. [Online]. Available: http://ijarcce.com/upload/2016/december-16/IJARCCE%2037.pdf (visited on 14/04/2023).

[10] D. Parker, T. Legg and C. Folland, 'A new daily Central England Temperature series, 1772–1991', *International Journal of Climatology*, vol. 12, pp. 317–342, 1st May 1992. DOI: 10.1002/joc.3370120402.

[11] 'PostgreSQL: About - www.postgresql.org/'. (), [Online]. Available: https://www.postgresql.org/about/ (visited on 25/04/2023).

[12] *Timescale/timescaledb*, Timescale, 21st Apr. 2023. [Online]. Available: https://github.com/timescale/timescaledb (visited on 21/04/2023).

[13] 'Timescale Documentation | About hypertables'. (), [Online]. Available: https://docs.timescale.com/use-timescale/latest/hypertables/about-hypertables/#hypertables (visited on 12/04/2023).

[14] 'Timescale Documentation | About distributed hypertables'. (), [Online]. Available: https://docs.timescale.com/use-timescale/latest/distributed-hypertables/about-distributed-hypertables/ (visited on 22/05/2023).

[15] *MongoDB README*, mongodb, 25th Apr. 2023. [Online]. Available: https://github.com/mongodb/mongo (visited on 25/04/2023).

[16] 'Time Series — MongoDB Manual'. (), [Online]. Available: https://www.mongodb.com/docs/manual/core/timeseries-collections/ (visited on 22/05/2023).

[17] E. Botoeva, D. Calvanese, B. Cogrel, M. Rezk and G. Xiao, 'OBDA Beyond Relational DBs: A Study for MongoDB',

[18] C. J. Choi, 'A study and comparison of NoSQL databases', M.Sc. thesis, California State University, Northridge, May 2014.

[19] C. Győrödi, R. Győrödi, G. Pecherle and A. Olah, 'A comparative study: MongoDB vs. MySQL', in *2015 13th International Conference on Engineering of Modern Electric Systems (EMES)*, Jun. 2015, pp. 1–6. DOI: 10.1109/EMES.2015.7158433.

[20] B. Jose and S. Abraham, 'Performance analysis of NoSQL and relational databases with MongoDB and MySQL', *Materials Today: Proceedings*, International Multi-conference on Computing, Communication, Electrical & Nanotechnology, I2CN-2K19, 25th & 26th April 2019, vol. 24, pp. 2036–2043, 1st Jan. 2020, ISSN: 2214-7853. DOI: 10.1016/j.matpr.2020.03.634. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2214785320324159 (visited on 11/04/2023).

[21] M.-G. Jung, S.-A. Youn, J. Bae and Y.-L. Choi, '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)*, Nov. 2015, pp. 14–17. DOI: 10.1109/DTA.2015.14.

[22] A. Makris, K. Tserpes, G. Spiliopoulos and D. Anagnostopoulos, 'Performance Evaluation of MongoDB and PostgreSQL for Spatio-temporal Data', presented at the EDBT/ICDT Workshops, 2019. [Online]. Available: https://www.semanticscholar.org/paper/Performance-Evaluation-of-MongoDB-and-

PostgreSQL-Makris-Tserpes/1b3c5f7959c6e1e80c7176134da6be0beb6d6744 (visited on 25/04/2023).

[23] A. Makris, K. Tserpes, G. Spiliopoulos, D. Zissis and D. Anagnostopoulos, 'MongoDB Vs PostgreSQL: A comparative study on performance aspects', *Geoinformatica*, vol. 25, no. 2, pp. 243–268, 1st Apr. 2021, ISSN: 1573-7624. DOI: 10.1007/s10707-020-00407-w. [Online]. Available: https://doi.org/10.1007/s10707-020-00407-w (visited on 25/04/2023).

[24] P. Grzesik and D. Mrozek, 'Comparative Analysis of Time Series Databases in the Context of Edge Computing for Low Power Sensor Networks', in *Computational Science – ICCS 2020*, V. V. Krzhizhanovskaya, G. Závodszky, M. H. Lees, J. J. Dongarra, P. M. A. Sloot, S. Brissos and J. Teixeira, Eds., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2020, pp. 371–383, ISBN: 978-3-030-50426-7. DOI: 10.1007/978-3-030-50426-7_28.

[25] 'SMHI Opendata - opendata-download-metobs.smhi.se/'. (), [Online]. Available: https://opendata-download-metobs.smhi.se/ (visited on 01/05/2023).

[26] 'Villkor för användning | SMHI'. (), [Online]. Available: https://www.smhi.se/data/oppna-data/information-om-oppna-data/villkor-for-anvandning-1.30622 (visited on 18/04/2023).

[27] R. Saavedra and A. Smith, 'Measuring cache and TLB performance and their effect on benchmark runtimes', *IEEE Trans. Comput.*, vol. 44, no. 10, pp. 1223–1235, Oct./1995, ISSN: 00189340. DOI: 10.1109/12.467697. [Online]. Available: http://ieeexplore.ieee.org/document/467697/ (visited on 28/04/2023).

[28] M. Raasveldt, P. Holanda, T. Gubner and H. Mühleisen, 'Fair Benchmarking Considered Difficult: Common Pitfalls In Database Performance Testing', in *Proceedings of the Workshop on Testing Database Systems*, ser. DBTest'18, New York, NY, USA: Association for Computing Machinery, 15th Jun. 2018, pp. 1–6, ISBN: 978-1-4503-5826-2. DOI: 10.1145/3209950.3209955. [Online]. Available: https://dl.acm.org/doi/10.1145/3209950.3209955 (visited on 28/04/2023).

[29] T. Mytkowicz, A. Diwan, M. Hauswirth and P. F. Sweeney, 'Producing wrong data without doing anything obviously wrong!',

[30] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke and T. E. Oliphant, 'Array programming with NumPy', *Nature*, vol. 585, no. 7825, pp. 357–362, 17th Sep. 2020, ISSN: 0028-0836, 1476-4687. DOI: 10.1038/s41586-020-2649-2. [Online]. Available: https://www.nature.com/articles/s41586-020-2649-2 (visited on 01/05/2023).

[31] P. Virtanen *et al.*, 'SciPy 1.0: Fundamental algorithms for scientific computing in Python', *Nat Methods*, vol. 17, no. 3, pp. 261–272, 2nd Mar. 2020, ISSN: 1548-7091, 1548-7105. DOI: 10.1038/s41592-019-0686-2. [Online]. Available: http://www.nature.com/articles/s41592-019-0686-2 (visited on 01/05/2023).

[32] J. D. Hunter, 'Matplotlib: A 2D Graphics Environment', *Comput. Sci. Eng.*, vol. 9, no. 3, pp. 90–95, 2007, ISSN: 1521-9615. DOI: 10.1109/MCSE.2007.55. [Online]. Available: http://ieeexplore.ieee.org/document/4160265/ (visited on 01/05/2023).

[33] D. Peter, *Hyperfine*, version 1.16.1, Mar. 2023. [Online]. Available: https://github.com/sharkdp/hyperfine (visited on 22/05/2023).

[34] 'SMHI Open Data API Docs - Meteorological Observations - opendata.smhi.se/'. (), [Online]. Available: https://opendata.smhi.se/apidocs/metobs/index.html (visited on 22/05/2023).

[35] 'MongoDB Stored Procedures Feature Equivalent', MongoDB. (), [Online]. Available: https://www.mongodb.com/features/stored-procedures (visited on 23/05/2023).

[36] 'Aggregation Pipeline Optimization — MongoDB Manual'. (), [Online]. Available: https://www.mongodb.com/docs/manual/core/aggregation-pipeline-optimization/ (visited on 24/05/2023).

[37] A. Katal, S. Dahiya and T. Choudhury, 'Energy efficiency in cloud computing data centers: A survey on software technologies', *Cluster Comput*, vol. 26, no. 3, pp. 1845–1875, Jun. 2023, ISSN: 1386-7857, 1573-7543. DOI: 10.1007/s10586-022-03713-0. [Online]. Available: https://link.springer.com/10.1007/s10586-022-03713-0 (visited on 25/05/2023).

Appendices

Appendix A

Query Implementation

We present here the full code for all the queries we implemented in the project, both for PostgreSQL and for MongoDB.

A.1 PostgreSQL

```
/*1. Month average for temperatures for every station based on hourly values
↪  (for service desk)*/

CREATE OR REPLACE FUNCTION query1_all()
    RETURNS TABLE
            (
                station_nr INTEGER,
                year       NUMERIC,
                jan        DOUBLE PRECISION,
                feb        DOUBLE PRECISION,
                mar        DOUBLE PRECISION,
                apr        DOUBLE PRECISION,
                may        DOUBLE PRECISION,
                june       DOUBLE PRECISION,
                july       DOUBLE PRECISION,
                aug        DOUBLE PRECISION,
                sep        DOUBLE PRECISION,
                oct        DOUBLE PRECISION,
                nov        DOUBLE PRECISION,
                "dec"        DOUBLE PRECISION
            )
AS
$$
BEGIN
    RETURN QUERY
        SELECT station_id as station_nr,
               EXTRACT(year from tt.timestamp)
               ↪  AS year,
               AVG(value) FILTER (WHERE extract(month from tt.timestamp) = 1)
               ↪  AS jan,
```

```
            AVG(value) FILTER (WHERE extract(month from tt.timestamp) = 2)
            ↪  AS feb,
            AVG(value) FILTER (WHERE extract(month from tt.timestamp) = 3)
            ↪  AS mar,
            AVG(value) FILTER (WHERE extract(month from tt.timestamp) = 4)
            ↪  AS apr,
            AVG(value) FILTER (WHERE extract(month from tt.timestamp) = 5)
            ↪  AS may,
            AVG(value) FILTER (WHERE extract(month from tt.timestamp) = 6)
            ↪  AS june,
            AVG(value) FILTER (WHERE extract(month from tt.timestamp) = 7)
            ↪  AS july,
            AVG(value) FILTER (WHERE extract(month from tt.timestamp) = 8)
            ↪  AS aug,
            AVG(value) FILTER (WHERE extract(month from tt.timestamp) = 9)
            ↪  AS sep,
            AVG(value) FILTER (WHERE extract(month from tt.timestamp) =
            ↪  10) AS oct,
            AVG(value) FILTER (WHERE extract(month from tt.timestamp) =
            ↪  11) AS nov,
            AVG(value) FILTER (WHERE extract(month from tt.timestamp) =
            ↪  12) AS "dec"
        FROM ((SELECT station_id,
                      master_id
               FROM positions
                        JOIN comp_ind_table ON positions.position_id =
                        ↪  comp_ind_table.position_id
               WHERE param_key = 1
                 AND (station_id IN (134110, 162860, 65160, 82260, 97530,
                 ↪  85240, 78400, 97200, 62180, 180940))
               GROUP BY station_id, master_id) t
              JOIN values ON t.master_id = values.master_id) tt
        GROUP BY station_id, year;
END;
$$ LANGUAGE plpgsql;


/*****************************************************

                      NEW QUERY

*****************************************************/

/*2. Min temperatures for every christmas eve*/
CREATE OR REPLACE FUNCTION query2()
    RETURNS TABLE
            (
                "year"      NUMERIC,
                "Minimum temp" DOUBLE PRECISION
            )
AS
$$
BEGIN
    RETURN QUERY
        SELECT year_val   AS year,
               MIN(value) AS "Minimum temp"
        FROM (SELECT master_id FROM comp_ind_table WHERE param_key = 19) t
                 INNER JOIN
```

```
            (SELECT master_id, value, extract(year FROM timestamp) AS
            ↪  year_val
             FROM values
             WHERE EXTRACT(month FROM timestamp) = 12
               AND EXTRACT(day FROM timestamp) = 24) v
            ON v.master_id = t.master_id
        GROUP BY year_val
        ORDER BY year_val;
END;
$$ LANGUAGE plpgsql;

/* In this one, we use generate_series to pick out the years
 */
CREATE OR REPLACE FUNCTION query2_trick()
    RETURNS TABLE
            (
                "year"      NUMERIC,
                "Minimum temp" DOUBLE PRECISION
            )
LANGUAGE plpgsql
AS
$$
DECLARE min_year NUMERIC;
DECLARE max_year NUMERIC;
BEGIN
        SELECT EXTRACT(year FROM MIN(timestamp)), EXTRACT(year FROM
        ↪  MAX(timestamp)) into min_year, max_year from values;
    RETURN QUERY
        SELECT year_val   AS year,
               MIN(value) AS "Minimum temp"
        FROM ((
                        (SELECT generate_series as year_val from
                        ↪  generate_series(min_year, max_year) ) as G
                    INNER JOIN
             (SELECT master_id, value, timestamp  FROM values) as V
                        ON timestamp between make_date(year_val::INT, 12,
                        ↪  24)::TIMESTAMP AND make_date(year_val::INT, 12,
                        ↪  25)::TIMESTAMP
                ) v
                        INNER JOIN (SELECT master_id FROM comp_ind_table
                        ↪  WHERE param_key = 19) t
            ON v.master_id = t.master_id)  as x
        GROUP BY year_val
        ORDER BY year_val;
END;
$$;

/*****************************************************

                        NEW QUERY

*****************************************************/

/* 3. Extreme cooling effects: wind and temperature for alpine stations when
↪  temperature is below -20 degrees*/
CREATE OR REPLACE FUNCTION query3()
    RETURNS TABLE
            (
                station_nr   INTEGER,
```

```
                station_name VARCHAR,
                "Timestamp"  TIMESTAMPTZ,
                temperature  DOUBLE PRECISION,
                wind         DOUBLE PRECISION
            )
AS
$$
BEGIN
    RETURN QUERY
        SELECT station_id                                   AS station_nr,
               station_name_                                AS station_name,
               timestamp                                    AS "Timestamp",
               MIN(value) FILTER ( WHERE param_key = 1 ) AS temperature,
               MIN(value) FILTER ( WHERE param_key = 4 ) AS wind
        FROM positions
                 JOIN
             (SELECT station_id AS alpstation_id, name AS station_name_
              FROM stations
              WHERE station_id IN
                    (167990, 170930, 171790, 172770, 172940, 173900, 177930,
                    ↪  178860, 178970, 179960)) alp_stations
             ON alp_stations.alpstation_id = positions.station_id
                 JOIN comp_ind_table ON comp_ind_table.position_id =
                 ↪  positions.position_id
                 JOIN values v ON comp_ind_table.master_id = v.master_id
        WHERE ((param_key = 1 AND value < -20) OR param_key = 4)
        GROUP BY station_id, station_name_, timestamp
        HAVING COUNT(value) > 1;
END;
$$ LANGUAGE plpgsql;


/*******************************************************

                        NEW QUERY

*******************************************************/

/*4. Hourly precipitation values for one year per station (for graphical
↪  display)*/

CREATE OR REPLACE FUNCTION query4(year INTEGER)
    RETURNS TABLE
            (
                station       VARCHAR,
                "Timestamp"   TIMESTAMPTZ,
                precipitation DOUBLE PRECISION
            )
AS
$$
BEGIN
    RETURN QUERY
        SELECT cit.name AS station, Timestamp , value AS precipitation
        FROM values
                 JOIN
             (SELECT master_id, name, param_key
              FROM comp_ind_table
                       JOIN positions p ON comp_ind_table.position_id =
                       ↪  p.position_id
```

```
                    JOIN stations s ON p.station_id = s.station_id) cit
            ON values.master_id = cit.master_id
        WHERE param_key = 7
          AND timestamp BETWEEN make_date(year, 1, 1)::TIMESTAMP AND
          ↪  make_date(year, 12, 31)::TIMESTAMP;
END;
$$ LANGUAGE plpgsql;


/*******************************************************

                      NEW QUERY

*******************************************************/

/*5. Temperature and wind speed average marked with week number and week day
  (Monday, Tuesday etc)*/
CREATE OR REPLACE FUNCTION query5(from_year INTEGER)
    RETURNS TABLE
            (
                station_nr INTEGER,
                week       NUMERIC,
                day        TEXT,
                avg_temp   DOUBLE PRECISION,
                avg_wind   DOUBLE PRECISION,
                                day_nr           NUMERIC
            )
AS
$$
BEGIN
    RETURN QUERY
        SELECT station_id                                    AS station_nr,
               EXTRACT(week FROM timestamp)                  AS week,
               TO_CHAR(timestamp, 'Dy')                      AS day,
               AVG(value) FILTER (WHERE param_key = 1)       AS avg_temp,
               AVG(value) FILTER (WHERE param_key = 4)       AS avg_wind,
               EXTRACT(dow FROM timestamp)                   AS day_nr
        FROM positions
                 JOIN comp_ind_table cit ON positions.position_id =
                 ↪  cit.position_id
                 JOIN values v ON cit.master_id = v.master_id
        WHERE (param_key = 1 OR param_key = 4)
          AND timestamp >= make_date(from_year, 1, 1)::TIMESTAMP
        GROUP BY station_id, week, day_nr, day;
END;
$$ LANGUAGE plpgsql;
```

A.2 MongoDB

```
/* Query 1. By-year monthly average temperatures for selected stations based
↪  on hourly values */

var checked_stations = [
  122610, 134110, 162860, 65160, 82260, 97530, 85240, 78400, 97200, 62180,
  180940,
```

```
];

var pipeline = [
  // pick the stations we want
  {
    $match: { _id: { $in: checked_stations } },
  },

  // get the positions
  {
    $lookup: {
      from: "positions",
      localField: "_id",
      foreignField: "station_id",
      as: "position",
    },
  },
  {
    $unwind: "$position",
  },

  // get the master ids
  {
    $lookup: {
      from: "comp_ind",
      let: { pos_id: "$position._id" },
      pipeline: [
        {
          $match: {
            $and: [
              { param: 1 }, // temperature
              { $expr: { $eq: ["$position_id", "$$pos_id"] } },
            ],
          },
        },
      ],
      as: "master",
    },
  },
  { $unwind: "$master" },

  // get the corresponding values
  {
    $lookup: {
      from: "values",
      localField: "master._id",
      foreignField: "master_id",
      as: "values",
    },
  },
  {
    $unwind: "$values",
  },

  // compute the averages
  {
    $group: {
      _id: { year: { $year: "$values.timestamp" }, station: "$_id" },
      name: { $first: "$name" },
```

```
jan: {
  $avg: {
    $cond: [
      { $eq: [{ $month: "$values.timestamp" }, 1] },
      "$values.value",
      null,
    ],
  },
},
feb: {
  $avg: {
    $cond: [
      { $eq: [{ $month: "$values.timestamp" }, 2] },
      "$values.value",
      null,
    ],
  },
},
mar: {
  $avg: {
    $cond: [
      { $eq: [{ $month: "$values.timestamp" }, 3] },
      "$values.value",
      null,
    ],
  },
},
apr: {
  $avg: {
    $cond: [
      { $eq: [{ $month: "$values.timestamp" }, 4] },
      "$values.value",
      null,
    ],
  },
},
may: {
  $avg: {
    $cond: [
      { $eq: [{ $month: "$values.timestamp" }, 5] },
      "$values.value",
      null,
    ],
  },
},
june: {
  $avg: {
    $cond: [
      { $eq: [{ $month: "$values.timestamp" }, 6] },
      "$values.value",
      null,
    ],
  },
},
july: {
  $avg: {
    $cond: [
      { $eq: [{ $month: "$values.timestamp" }, 7] },
      "$values.value",
```

```
          null,
        ],
      },
    },
    aug: {
      $avg: {
        $cond: [
          { $eq: [{ $month: "$values.timestamp" }, 8] },
          "$values.value",
          null,
        ],
      },
    },
    sep: {
      $avg: {
        $cond: [
          { $eq: [{ $month: "$values.timestamp" }, 9] },
          "$values.value",
          null,
        ],
      },
    },
    oct: {
      $avg: {
        $cond: [
          { $eq: [{ $month: "$values.timestamp" }, 10] },
          "$values.value",
          null,
        ],
      },
    },
    nov: {
      $avg: {
        $cond: [
          { $eq: [{ $month: "$values.timestamp" }, 11] },
          "$values.value",
          null,
        ],
      },
    },
    dec: {
      $avg: {
        $cond: [
          { $eq: [{ $month: "$values.timestamp" }, 12] },
          "$values.value",
          null,
        ],
      },
    },
  },
},

// pick the properties we care about
{
  $project: {
    _id: 0,
    station_id: "$_id.station",
    year: "$_id.year",
    name: 1,
```

```
      jan: 1,
      feb: 1,
      mar: 1,
      apr: 1,
      may: 1,
      june: 1,
      july: 1,
      aug: 1,
      sep: 1,
      oct: 1,
      nov: 1,
      dec: 1,
    },
  },

  {
    // output to temp collection
    $out: "temp_query_results",
  },
];

db.stations.aggregate(pipeline);

/* Query 2. Minimum temperature on Christmas Eve for every year on record */

db.comp_ind.aggregate([
  // select minimum temperature measurements
  { $match: { param: 19 } },

  // find values
  {
    $lookup: {
      from: "values",
      let: { master_id: "$_id" },
      pipeline: [
        {
          $match: {
            $expr: {
              $and: [
                // filter for Christmas Eve and miimum temperature
                { $eq: ["$master_id", "$$master_id"] },
                { $eq: [{ $month: "$timestamp" }, 12] },
                { $eq: [{ $dayOfMonth: "$timestamp" }, 24] },
              ],
            },
          },
        },
      ],
      as: "value",
    },
  },
  { $unwind: "$value" },

  // get the year and the minimum temperature in case of multiple
  ↪  measurements per day
  {
    $group: {
      _id: { $year: "$value.timestamp" },
      "Minimum temp": { $min: "$value.value" },
```

```
    },
  },

  {
    $sort: {
      _id: 1,
    },
  },
  {
    $project: {
      year: "$_id",
      "Minimum temp": 1,
      _id: 0,
    },
  },
  {
    $out: "temp_query_results",
  },
]);

/* 3. Extreme cooling effects: wind and temperature for alpine stations when
↪  temperature is below -20 degrees*/

var selected_stations = [
  167990, 170930, 171790, 172770, 172940, 173900, 177930, 178860, 178970,
  179960,
];

// in this pipeline we have access to the master_ids and params variables
var values_pipeline = {
  // get the values for the master_ids of the station
  match_master_id: {
    $match: { $expr: { $in: ["$master_id", "$$master_ids"] } },
  },
  // set the param field in the value document, so we know what measure it is
  set_param: {
    $addFields: {
      param: {
        $arrayElemAt: [
          "$$params",
          { $indexOfArray: ["$$master_ids", "$master_id"] },
        ],
      },
    },
  },
  // filter out for low temperatures
  filter_low_temps: {
    $match: {
      // here having saved the param is useful
      $or: [
        { param: 4 }, // we keep wind measures
        { value: { $lt: -20 } }, // if it isn't wind then it's temp, must be
        ↪  low
      ],
    },
  },

  // pair the temp and wind values for same time together
  group_by_timestamp: {
```

```
    $group: {
      _id: { timestamp: "$timestamp" },
      temperature: {
        $min: {
          $cond: [{ $eq: ["$param", 1] }, "$value", null],
        },
      },
      wind: {
        $max: {
          $cond: [{ $eq: ["$param", 4] }, "$value", null],
        },
      },
    },
  },

  // filter out for only the ones that have both (so the temp was <-20 at that
  ↪  moment)
  filter_for_both: {
    $match: {
      temperature: { $ne: null },
      wind: { $ne: null },
    },
  },

  // get the data we care about
  project: {
    $project: {
      _id: 0,
      timestamp: "$_id.timestamp",
      temperature: 1,
      wind: 1,
    },
  },
};

var query_pipeline = {
  // find our stations
  match_stations: {
    $match: {
      _id: { $in: selected_stations },
    },
  },

  // get corresponding master_ids for our stations and our parameters
  lookup_master_id: {
    $lookup: {
      from: "comp_ind",
      let: { pos: "$positions" },
      pipeline: [
        {
          $match: {
            $and: [
              { $expr: { $in: ["$position_id", "$$pos"] } },
              { param: { $in: [1, 4] } },
            ],
          },
        },
      ],
      as: "master",
```

```
    },
  },

  // lookup values for the stations, the pipeline is complex enough to be its
  ↪  own variable
  lookup_values: {
    $lookup: {
      from: "values",
      // store list of master_ids and params for the inner pipeline
      let: {
        master_ids: { $map: { input: "$master", in: "$$this._id" } },
        params: { $map: { input: "$master", in: "$$this.param" } },
      },

      pipeline: [
        values_pipeline.match_master_id,
        values_pipeline.set_param,
        values_pipeline.filter_low_temps,
        values_pipeline.group_by_timestamp,
        values_pipeline.filter_for_both,
        values_pipeline.project,
      ],
      as: "values",
    },
  },

  // get the data we want
  project: {
    $project: {
      _id: 0,
      station_id: "$_id",
      name: 1,
      timestamp: "$values.timestamp",
      temperature: "$values.temperature",
      wind: "$values.wind",
    },
  },
};

full_pipeline = [
  query_pipeline.match_stations,
  query_pipeline.lookup_master_id,
  query_pipeline.lookup_values,
  { $unwind: "$values" },
  query_pipeline.project,
  { $out: "temp_query_results" },
];

db.stations.aggregate(full_pipeline);

/*4. Hourly precipitation values for one year per station (for graphical
↪  display)*/

var year = 2022; // change to desired year

db.comp_ind.aggregate([
  // select precipitation measurements
  { $match: { param: 7 } },
```

```
  // find values
  {
    $lookup: {
      from: "values",
      let: { master_id: "$_id" },
      pipeline: [
        {
          $match: {
            $expr: {
              $and: [
                // filter for precipitation and year of interest
                { $eq: ["$master_id", "$$master_id"] },
                { $gte: ["$timestamp", new Date(year, 0, 1)] },
                { $lte: ["$timestamp", new Date(year, 11, 31)] },
              ],
            },
          },
        },
      ],
      as: "value",
    },
  },
  { $unwind: "$value" },
  {
    // pick out the position_id for the master_id
    $lookup: {
      from: "positions",
      localField: "position_id",
      foreignField: "_id",
      as: "position",
    },
  },
  { $unwind: "$position" },
  {
    // pick out the station for each measurement
    $lookup: {
      from: "stations",
      localField: "position.station_id",
      foreignField: "_id",
      as: "station",
    },
  },
  { $unwind: "$station" },
  {
    $project: {
      _id: 0,
      station_id: "$station._id",
      station_name: "$station.name",
      timestamp: "$value.timestamp",
      precipitation: "$value.value",
    },
  },
  { $out: "temp_query_results" },
]);

/*5. Temperature and wind speed average marked with week number and week day
   (Monday, Tuesday etc)*/
var days = ["Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"];
```

```
db.comp_ind.aggregate([
  // pick out temp and wind speed
  { $match: { param: { $in: [1, 4] } } },
  //get position
  {
    $lookup: {
      from: "positions",
      localField: "position_id",
      foreignField: "_id",
      as: "position",
    },
  },
  { $unwind: "$position" },

  // pick out the station for each measurement
  {
    $lookup: {
      from: "stations",
      localField: "position.station_id",
      foreignField: "_id",
      as: "station",
    },
  },
  { $unwind: "$station" },

  // get the values
  {
    $lookup: {
      from: "values",
      localField: "_id",
      foreignField: "master_id",
      as: "values",
    },
  },
  {
    $unwind: "$values",
  },

  // compute the averages per week and day
  {
    $group: {
      _id: {
        station_id: "$station._id",
        week: { $week: "$values.timestamp" },
        day_nr: { $dayOfWeek: "$values.timestamp" },
      },
      avg_temperature: {
        $avg: {
          $cond: [{ $eq: ["$param", 1] }, "$values.value", null],
        },
      },
      avg_wind: {
        $avg: {
          $cond: [{ $eq: ["$param", 4] }, "$values.value", null],
        },
      },
    },
  },
```

```
  // get the info we care about
  {
    $project: {
      _id: 0,
      station_id: "$_id.station_id",
      week: "$_id.week",
      day: { $arrayElemAt: [days, "$_id.day_nr"] },
      avg_temperature: 1,
      avg_wind: 1,
      day_nr: "$_id.day_nr",
    },
  },
  { $out: "temp_query_results" },
]);
```

Appendix B

Data Retrieval Scripts

B.1 Metadata Download Script

```python
#!/usr/bin/env python3

from utils import *
import os
import json
from functools import wraps
import pathlib

DOWNLOAD_DIR = 'metadata'
PARAM_DIR = f'{DOWNLOAD_DIR}/parameters'
STATION_DIR = f'{DOWNLOAD_DIR}/stations'


def file_or_create(filename: str, func):
    """Read json data from file or create from provided callback and save to
    ↪  file if it doesn't exist"""
    pathlib.Path(filename).parent.mkdir(parents=True, exist_ok=True)
    try:
        with open(filename, 'r') as f:
            data = json.load(f)
    except FileNotFoundError:
        try:
            with open(filename, 'x') as f:
                data = func()
                json.dump(data, f, indent=2, ensure_ascii=False)
        except Exception as e:
            print(
                f'exception occurred trying to produce data for {filename},
                ↪  deleting file'
            )
            os.remove(filename)
            raise e
    return data


def file_cache(filename: str):
```

```
    """Decorator to cache function result in file"""

    def decorator(func):

        @wraps(func)
        def wrapper(*args, **kwargs):
            return file_or_create(filename, lambda: func(*args, **kwargs))

        return wrapper

    return decorator


def main():
    stations = set()
    pairs = 0
    total_hrs = 0
    total_moves = 0

    session = requests.Session()
    api_info = file_or_create(f'{DOWNLOAD_DIR}/api_info.json',
                              lambda: get_api_info(session))
    version_info = file_or_create(
        f'{DOWNLOAD_DIR}/version_info.json',
        lambda: get_version_info(api_info, session)[1])
    parameters = parse_parameters(version_info)

    for param in parameters:
        param_details = file_or_create(f'{PARAM_DIR}/{param.key}.json',
                                       lambda: get_parameter(param, session))
        param_codes = file_or_create(
            f'{PARAM_DIR}/codes/{param.key}.json',
            lambda: get_parameter_codes(param_details, session))
        st_count = len(param_details['station'])
        has_codes = 'has codes' if param_codes is not None else 'no codes'
        print(
            f'Processing {st_count} stations for param {param.key}
            ↪  ({has_codes})',
            f'({param.name}, {param.summary})')
        pairs += st_count
        for station in param_details['station']:
            station_param_details = file_or_create(
                f'{STATION_DIR}/{station["key"]}-{param.key}.json',
                lambda: get_station_param_with_period(station, session))
            if 'corrected-archive' not in station_param_details:
                print(f'No corrected-archive for
                ↪  {station["key"]}-{param.key}')
            if station['key'] not in stations:
                moves = len(station_param_details['position']) - 1
                # print( f'Station {station["key"]} has moved {moves} times')
                total_moves += moves
            stations.add(station['key'])

            for pos in station_param_details['position']:
                # timestamps are ms, we need hrs
                duration = (pos['to'] - pos['from']) // 1000 // 3600
                total_hrs += duration
        print(
```

```
            f'Processed all {st_count} stations for param {param.key},
            ↪  {pairs} pairs total'
        )

    print(
        f'Found {len(stations)} stations, giving {pairs} station-parameter
        ↪  pairs,',
        f'covering {total_hrs} hours of data. Stations have moved
        ↪  {total_moves} times.'
    )


if __name__ == '__main__':
    main()
```

B.2 Metadata SQL Import Script

```
#!/usr/bin/env python3
"""
Script to generate the DB script to populate it with metadata from the API
"""

import json
import os
import requests
from utils import *

DATA_DIR = 'data'
DOWNLOAD_DIR = 'metadata'
PARAM_DIR = f'{DOWNLOAD_DIR}/parameters'
STATION_DIR = f'{DOWNLOAD_DIR}/stations'


def generate_sql(params, position_ids, param_positions, stations):
    param_sql = [
        'INSERT INTO params (param_key, name, has_code, summary, is_interval)
        ↪  VALUES'
    ]
    codes_sql = [
        'INSERT INTO codes (param_key, code_number, description) VALUES'
    ]
    station_sql = [
        'INSERT INTO stations (station_id, updated, owner, is_active,
        ↪  network_type, name) VALUES'
    ]
    positions_sql = [
        'INSERT INTO positions (station_id, latitude, longitude, elevation)
        ↪  VALUES'
    ]
    # TODO: check what we're doing with this table (with the from_date and
    ↪  to_date)
    comp_ind_sql = [
        'INSERT INTO comp_ind_table (param_key, position_id, from_date,
        ↪  to_date) VALUES'
    ]
    station_sql.extend(
```

```
        f"({station.key}, '{station.updated}', '{station.owner}', "
        f"{str(station.active).upper()}, '{station.network_type}',
        ↪  '{station.name}'),"
        for station in stations.values())
    positions_sql.extend(
        f"({pos.station}, {pos.latitude}, {pos.longitude}, {pos.elevation}),"
        for pos in position_ids.keys())
    for param in params:
        param_sql.append(
            f"({param.key}, '{param.name}', {param.codes is not None},
            ↪  '{param.summary}', {str(param.is_interval).upper()}),"
        )
        if param.codes is not None:
            codes_sql.extend(
                f"({param.key}, {code_number}, '{description}'),"
                for code_number, description in param.codes.items())
        for posid, (frm, to) in param_positions[param.key].items():
            # TODO: avoid this quadratic nested loop, another dict would give
            ↪  us O(1) here
            pos: Position = next(p for p, i in position_ids.items()
                                 if i == posid)
            # use the stored procedure to get the position id, creating an
            ↪  entry if necessary
            get_pos_id =
            ↪  f"get_or_create_position({pos.station},{pos.latitude},{pos.longitude},{pos.elevation})"
            ↪  # TODO: implement this stored procedure
            comp_ind_sql.append(
                f"({param.key}, {get_pos_id}, '{frm}', '{to}'),")
    sqls = (param_sql, codes_sql, station_sql, positions_sql, comp_ind_sql)
    for sql in sqls:
        # ensure the last line of every query ends with a semicolon instead
        ↪  of a comma
        sql[-1] = sql[-1][:-1] + ';'
    return sqls


def main():
    params, position_ids, param_positions, stations = read_all_data()
    sqls = generate_sql(params, position_ids, param_positions, stations)
    with open('postgres/generated/populate_db.sql', 'w') as f:
        for sql in sqls:
            f.writelines(intersperse(sql, '\n'))
            f.write('\n\n')


if __name__ == '__main__':
    main()
```

B.3 Data Download Script

```
#!/usr/bin/env python3

import json
import os
import requests
import pathlib
```

```
from utils import *


def download_data(station_file: str,
                  session: requests.Session) -> Optional[int]:
    data = readjson(f'{STATION_DIR}/{station_file}')
    if 'corrected-archive' not in data:
        print(
            f'WARNING: {station_file} ({data["title"]}) has no
            ↪  corrected-archive'
        )
        return None
    print(f'INFO: {station_file} has corrected-archive, downloading')
    total = 0
    for i, info in enumerate(data['corrected-archive']['data']):
        url = info['link'][0]['href']
        r = session.get(url)
        dest_file = f'{DATA_DIR}/{station_file.removesuffix(".json")}-{i}.csv'
        try:
            with open(dest_file, 'xb') as f:
                total += f.write(r.content)
        except FileExistsError:
            print(f'WARNING: {dest_file} already exists, skipping')
            total += os.path.getsize(dest_file)
    return total


def main():
    session = requests.Session()
    total = 0  # bytes downloaded
    count = 0
    ask_incr = 2**30 * 4  # ask every 2GB
    next_ask = ask_incr
    pathlib.Path(DATA_DIR).mkdir(parents=True, exist_ok=True)
    for file in os.listdir(STATION_DIR):
        downloaded = download_data(file, session)
        if downloaded is None:
            continue
        total += downloaded
        count += 1
        print(f'INFO: Downloaded {downloaded} bytes for {file}')
        print(
            f'INFO: total: {total/(2**20)} MB, {count} files downloaded or
            ↪  stored'
        )
        if total > next_ask:
            print(
                f'INFO: Downloaded(/stored) {total/(2**30)} GB, continue?
                ↪  [y/n]'
            )
            if not input().lower().startswith('y'):
                break
            next_ask += ask_incr


if __name__ == '__main__':
    main()
```

B.4 Utility Functions and Classes

```
import json
from typing import Iterable, Iterator, Optional, TypeVar
from collections import namedtuple
import requests
import semver
import datetime as dt
import os
import sys

from dataclasses import dataclass

baseurl = "https://opendata-download-metobs.smhi.se"

DATA_DIR = 'data'
DOWNLOAD_DIR = 'metadata'
PARAM_DIR = f'{DOWNLOAD_DIR}/parameters'
STATION_DIR = f'{DOWNLOAD_DIR}/stations'

T = TypeVar('T')
K = TypeVar('K')
V = TypeVar('V')


def dt_from_timestamp(seconds: float,
                      tz: Optional[dt.tzinfo] = None) -> dt.datetime:
    '''
    Should do the same as dt.datetime.fromtimestamp()

    Workaround required since the datetime one doesn't work with negative
    ↪  timestamps on windows
    (and maybe even with low timestamps)
    '''
    return (dt.datetime.fromtimestamp(1_000_000, tz=tz) +
            dt.timedelta(seconds=seconds - 1_000_000))


def intersperse(iterable: Iterable[T], delimiter: T) -> Iterator[T]:
    """Intersperse a delimiter between elements of an iterable"""
    it = iter(iterable)
    yield next(it)
    for x in it:
        yield delimiter
        yield x


def readjson(filename):
    with open(filename, 'r') as f:
        data = json.load(f)
    return data


def filter_key(it: Iterable[dict[K, V]], key: K,
               value: V) -> Iterator[dict[K, V]]:
    """return an iterator over the elements which have the given value for
    ↪  the given key"""
    return filter(lambda d: d[key] == value, it)
```

```
def get_json_url(links: Iterable[dict[str, str]]) -> str | None:
    link = next(filter_key(links, 'type', 'application/json'), None)
    return link['href'] if link is not None else None


PosValidity = namedtuple('PosValidity', ['pos', 'from_date', 'to_date'])


@dataclass(frozen=True)
class Position:
    station: int
    elevation: float
    latitude: float
    longitude: float

    @classmethod
    def from_info(cls, station: int, position_info: dict) -> PosValidity:
        """
        Create a Position from the position info in a station json file

        Return position, from_date, to_date
        """
        from_ = position_info['from']
        to_ = position_info['to']
        height = position_info['height']
        latitude = position_info['latitude']
        longitude = position_info['longitude']
        pos = cls(
            station=station,
            elevation=height,
            latitude=latitude,
            longitude=longitude,
        )
        from_date = dt_from_timestamp(from_ / 1000, tz=dt.timezone.utc)
        to_date = dt_from_timestamp(to_ / 1000, tz=dt.timezone.utc)
        return PosValidity(pos, from_date, to_date)

    def as_document(self, _id):
        return {
            '_id': _id,
            'elevation': self.elevation,
            'latitude': self.latitude,
            'longitude': self.longitude,
        }


@dataclass
class Station:
    key: int  # station id
    name: str
    active: bool
    network_type: str  # CORE or ADDITIONAL
    owner: str
    updated: dt.datetime

    @classmethod
    def from_info(cls, name: str,
```

```
                station_info: dict) -> tuple['Station', list[PosValidity]]:
        """
        Create a Station from the station info in a station-param json file

        Return station, list of positions with valid time range
        """
        key = int(station_info['key'])
        active = station_info['active']
        network_type = station_info['measuringStations']
        owner = station_info['owner']
        updated_ts: int = station_info['updated']  # ms timestamp
        updated = dt_from_timestamp(updated_ts / 1000, tz=dt.timezone.utc)
        positions = []
        for position_info in station_info['position']:
            pos_from_to = Position.from_info(key, position_info)
            positions.append(pos_from_to)
        station = cls(
            key=key,
            name=name,
            active=active,
            network_type=network_type,
            owner=owner,
            updated=updated,
        )
        return station, positions

    def as_document(self, positions: list[int]) -> dict:
        return {
            '_id': self.key,
            'name': self.name,
            'is_active': self.active,
            'network_type': self.network_type,
            'owner': self.owner,
            'positions': positions,
        }


@dataclass
class Parameter:
    key: int
    name: str
    summary: str
    is_interval: bool
    codes: Optional[dict[int, str]]

    @classmethod
    def from_info(cls, param_info: dict) -> 'Parameter':
        title = param_info['title']
        key = int(param_info['key'])
        summary = param_info['summary']
        value_type: str = param_info['valueType'].upper()
        assert value_type in ('INTERVAL', 'SAMPLING')
        is_interval = value_type == 'INTERVAL'
        has_code = next(filter_key(param_info['link'], 'rel', 'codes'),
                        None) is not None
        if has_code:
            codes_data = readjson(f'{PARAM_DIR}/codes/{key}.json')
            codes = {e['key']: e['value'] for e in codes_data['entry']}
        else:
```

```
            codes = None
        return cls(
            key=key,
            name=title.split(':')[0],
            summary=summary,
            is_interval=is_interval,
            codes=codes,
        )

    def as_document(self) -> dict:
        codes = [{
            "code_number": k,
            "description": v
        } for k, v in self.codes.items()] if self.codes is not None else None
        return {
            '_id': self.key,
            'name': self.name,
            'summary': self.summary,
            'type': 'INTERVAL' if self.is_interval else 'SAMPLING',
            'codes': codes
        }


@dataclass
class RawParameter:
    name: str
    key: str
    href: str
    summary: Optional[str] = None
    has_codes: Optional[bool] = None
    codes_url: Optional[str] = None

    @classmethod
    def from_version_json(cls, param_info: dict) -> 'RawParameter':
        name = param_info['title']
        key = param_info['key']
        href = get_json_url(param_info['link'])
        summary = param_info.get('summary', None)
        if href is None:
            raise ValueError('No url found in parameter', param_info)
        return cls(
            name=name,
            key=key,
            href=href,
            summary=summary,
        )

    def get_info(self, session=None):
        if session is None:
            session = requests.Session()
        r = session.get(self.href)
        info = r.json()
        return info


def get_api_info(session=None):
    if session is None:
        session = requests.Session()
    """Get the api info from the SMHI api"""
```

```
    url = baseurl + "/api.json"
    r = session.get(url)
    # print('request response:', r.apparent_encoding)
    # sys.stdout.buffer.write(r.content)

    return r.json()


def get_version_info(api_info: dict[str, list[dict]], session=None):
    """Get the version info from the SMHI api"""
    if session is None:
        session = requests.Session()
    actual_versions = filter(lambda v: v['key'] != 'latest',
                             api_info['version'])
    sorted_versions = sorted(
        actual_versions,
        key=lambda v: semver.Version.parse(v['key'],
                                           optional_minor_and_patch=True),
    )
    version = sorted_versions[-1]  # use latest named version
    url = get_json_url(version['link'])
    if url is None:
        raise ValueError('No url found in selected version', version)
    r = session.get(url)
    return version['key'], r.json()


def parse_parameters(version_info) -> list[RawParameter]:
    return [
        RawParameter.from_version_json(p) for p in version_info['resource']
    ]


def get_parameter(p: RawParameter, session=None):
    return p.get_info(session)


def get_parameter_metadata(param_info, session=None):
    if session is None:
        session = requests.Session()
    url = get_json_url(filter_key(param_info['link'], 'rel', 'metadata'))
    if url is None:
        raise ValueError('No url found in parameter', param_info)
    r = session.get(url)
    return r.json()


def get_parameter_codes(param_info, session=None):
    url = get_json_url(filter_key(param_info['link'], 'rel', 'codes'))
    if url is None:
        return None
    if session is None:
        session = requests.Session()
    r = session.get(url)
    return r.json()


def get_station_param(station_entry, session=None):
    if session is None:
```

```
        session = requests.Session()
    url = get_json_url(station_entry['link'])
    if url is None:
        raise ValueError('No url found in station', station_entry)
    r = session.get(url)
    return r.json()


def get_period(period_entry, session=None):
    if session is None:
        session = requests.Session()
    url = get_json_url(period_entry['link'])
    if url is None:
        raise ValueError('No url found in period', period_entry)
    r = session.get(url)
    return r.json()


def get_station_param_with_period(station_entry, session=None):
    """Get the station data with the period data for the corrected-archive
    ↪  period"""
    station_data = get_station_param(station_entry, session)
    period_entry = next(
        filter_key(station_data['period'], 'key', 'corrected-archive'), None)
    if period_entry is not None:
        period_data = get_period(period_entry, session)
        station_data['corrected-archive'] = period_data
    return station_data


def read_all_data() -> tuple[list[Parameter], dict[Position, int], dict[
    int, dict[int, tuple[int, int]]], dict[int, Station]]:
    params = []
    param_positions: dict[int, dict] = {}  # key -> {pos_id -> (from, to)}
    station_names: dict[int, str] = {}  # key -> name
    for f in os.listdir(PARAM_DIR):
        if not f.endswith('.json'):
            continue
        param_info = readjson(f'{PARAM_DIR}/{f}')
        param = Parameter.from_info(param_info)
        params.append(param)
        param_positions[param.key] = {}
        for station in param_info['station']:
            sk = int(station['key'])
            if sk not in station_names:
                station_names[sk] = station['name']
            else:
                oldname = station['name']
                newname = station_names[sk]
                if newname in oldname:
                    # newly found name is a substring of old name, so we're
                    ↪  good
                    pass
                elif oldname in newname:
                    # newly found name is more precise, use that
                    station_names[sk] = newname
                else:
                    # newly found name is different, warn and concatenate
                    print(
```

```
                    f'WARNING: new name found for station {sk}
                    ↪  ({oldname}): {station_names[sk]}'
                )
                station_names[sk] = oldname + ' aka ' + newname

stations = {}
position_ids = {}
pos_id = 1
for f in os.listdir(STATION_DIR):
    if not f.endswith('.json'):
        continue
    key, param_key = f.removesuffix('.json').split('-')
    key, param_key = int(key), int(param_key)
    station_info = readjson(f'{STATION_DIR}/{f}')
    name = station_names[int(station_info['key'])]
    station, positions = Station.from_info(name, station_info)
    assert key == station.key
    if station.key not in stations:
        stations[station.key] = station
    for p in positions:
        if p.pos not in position_ids:
            position_ids[p.pos] = pos_id
            pos_id += 1
        param_positions[param_key][position_ids[p.pos]] = (p.from_date,
                                                           p.to_date)
return params, position_ids, param_positions, stations
```

Appendix C

DBMS Setup Scripts

C.1 PostgreSQL Setup Scripts

SQL code to create the database, tables, and indexes.

```
CREATE TYPE NETWORK_TYPE AS ENUM ('CORE','ADDITIONAL');
CREATE TYPE QUALITY AS ENUM ('G','Y','R');


CREATE TABLE params (
 param_key INT NOT NULL,
 name VARCHAR(255) NOT NULL,
 has_code BOOLEAN NOT NULL,
 unit VARCHAR(255),
 summary TEXT NOT NULL,
 is_interval BOOLEAN NOT NULL
);

ALTER TABLE params ADD CONSTRAINT PK_params PRIMARY KEY (param_key);


CREATE TABLE stations (
 station_id INT NOT NULL,
 updated TIMESTAMPTZ(6) NOT NULL,
 owner VARCHAR(255) NOT NULL,
 is_active BOOLEAN NOT NULL,
 network_type NETWORK_TYPE NOT NULL,
 name VARCHAR(255) NOT NULL
);

ALTER TABLE stations ADD CONSTRAINT PK_stations PRIMARY KEY (station_id);


CREATE TABLE codes (
 param_key INT NOT NULL,
 code_number INT NOT NULL,
 description TEXT NOT NULL
);
```

```
ALTER TABLE codes ADD CONSTRAINT PK_codes PRIMARY KEY
↪  (param_key,code_number);


CREATE TABLE positions (
 position_id INT GENERATED ALWAYS AS IDENTITY NOT NULL,
 latitude DOUBLE PRECISION NOT NULL,
 longitude DOUBLE PRECISION NOT NULL,
 elevation DOUBLE PRECISION NOT NULL,
 station_id INT NOT NULL
);

ALTER TABLE positions ADD CONSTRAINT PK_positions PRIMARY KEY (position_id);


CREATE TABLE comp_ind_table (
 master_id INT GENERATED ALWAYS AS IDENTITY NOT NULL,
 param_key INT NOT NULL,
 position_id INT NOT NULL,
 from_date TIMESTAMPTZ(6) NOT NULL,
 to_date TIMESTAMPTZ(6)
);

ALTER TABLE comp_ind_table ADD CONSTRAINT PK_comp_ind_table PRIMARY KEY
↪  (master_id);


CREATE TABLE values (
 timestamp TIMESTAMPTZ(6) NOT NULL,
 master_id INT NOT NULL,
 value DOUBLE PRECISION NOT NULL,
 quality QUALITY NOT NULL
);


ALTER TABLE codes ADD CONSTRAINT FK_codes_0 FOREIGN KEY (param_key)
↪  REFERENCES params (param_key);


ALTER TABLE positions ADD CONSTRAINT FK_positions_0 FOREIGN KEY (station_id)
↪  REFERENCES stations (station_id);


ALTER TABLE comp_ind_table ADD CONSTRAINT FK_comp_ind_table_0 FOREIGN KEY
↪  (param_key) REFERENCES params (param_key);
ALTER TABLE comp_ind_table ADD CONSTRAINT FK_comp_ind_table_1 FOREIGN KEY
↪  (position_id) REFERENCES positions (position_id);


ALTER TABLE values ADD CONSTRAINT FK_values_0 FOREIGN KEY (master_id)
↪  REFERENCES comp_ind_table (master_id);


-- temporary table to read files from csvs before processing,
-- every file should give one of period_start (+period end) or the pair (date,
↪  time)
-- (plus value and quality).
-- param_key and station_key will be added subsequently,
-- the processed flag can be used if we want to do something like take in a
↪  file,
```

```
-- set the param_key and station_key, then take in more files again and set
↪  their keys
-- (without transferring everything to the main values table), so that we
↪  know
-- which values are fresh from the last file and need to be processed
-- (ie, we set the flag to TRUE when we set the keys, and for the newly
↪  ingested rows it will be NULL,
-- so we only set keys for the ones where it isn't TRUE)

CREATE TABLE temp_read_values (
 param_key INT,
 station_key INT,
 value DOUBLE PRECISION,
 quality QUALITY,
 date DATE,
 time TIME,
 period_start TIMESTAMP(6),
 period_end TIMESTAMP(6),
 representative_date VARCHAR(16)
);

-- create a table that's a copy of values in structure
SELECT * into temp_values_noindex from values ;


-- Get a position ID or create a new position (and return its ID) if it
↪  doesn't exist.
CREATE OR REPLACE FUNCTION get_or_create_position(
        p_station_id INTEGER,
        p_latitude DOUBLE PRECISION,
        p_longitude DOUBLE PRECISION,
        p_elevation DOUBLE PRECISION
)
RETURNS INTEGER
LANGUAGE plpgsql
AS $$
DECLARE
        pos_id INTEGER;
BEGIN
        -- BEGIN TRANSACTION;
        SELECT position_id INTO pos_id
                FROM positions
                WHERE station_id = p_station_id AND latitude = p_latitude AND
                ↪  longitude = p_longitude AND elevation = p_elevation
                LIMIT 1;
        IF pos_id IS NULL THEN
                INSERT INTO positions (station_id, latitude, longitude,
                ↪  elevation)
                        VALUES (p_station_id, p_latitude, p_longitude,
                        ↪  p_elevation)
                        RETURNING position_id INTO pos_id;
        END IF;
        -- COMMIT;
        RETURN pos_id;
END;
$$;


CREATE INDEX IF NOT EXISTS pos_station_idx ON positions (station_id);
-- CREATE INDEX IF NOT EXISTS values_master_idx ON values (master_id);
```

```
CREATE INDEX IF NOT EXISTS master_idx ON comp_ind_table
↪  (param_key,position_id);
CREATE INDEX IF NOT EXISTS pos_master_idx ON comp_ind_table (position_id);


SELECT create_hypertable(
        'values',
        'timestamp',
        chunk_time_interval => INTERVAL '2 years', -- one year should be ~1GB
        create_default_indexes => FALSE
);


CREATE OR REPLACE PROCEDURE transfer_values()
LANGUAGE plpgsql
AS $$
BEGIN
        -- transfer values from the temporary table to the values table for
        ↪  samplings
        INSERT INTO values(timestamp, value, master_id, quality)
        SELECT timestamp,
                   value,
                   master_id,
                   quality
        FROM (
                positions INNER JOIN (
                        SELECT
                                COALESCE(period_start, CAST(date AS
                                ↪  timestamp) + CAST(time AS interval)) AS
                                ↪  timestamp,
                                param_key,
                                station_key,
                                value,
                                quality
                        FROM temp_read_values
                ) AS t
                ON t.station_key = positions.station_id
        ) values_positions INNER JOIN comp_ind_table ON
                        (comp_ind_table.param_key =
                        ↪  values_positions.param_key AND
                        ↪  values_positions.position_id =
                        ↪  comp_ind_table.position_id)
        WHERE values_positions.timestamp BETWEEN from_date AND to_date;
END;
$$;

CREATE OR REPLACE PROCEDURE transfer_values_station_param(
        p_station_id INTEGER,
        p_param_key INTEGER
)
LANGUAGE plpgsql
AS $$
BEGIN
        INSERT INTO values(timestamp, value, master_id, quality)
        SELECT  t.timestamp,
                   value,
                   master_id,
                   quality
        FROM
```

```
                (SELECT
                        COALESCE(period_start, CAST(date AS timestamp) +
                        ↪  CAST(time AS interval)) as timestamp,
                        value,
                        quality
                        FROM temp_read_values
                        ORDER BY timestamp
                ) AS t,
                (
                        (SELECT position_id FROM positions WHERE station_id =
                        ↪  p_station_id) AS station_positions
                        INNER JOIN comp_ind_table ON
                        ↪  station_positions.position_id =
                        ↪  comp_ind_table.position_id
                )
        WHERE
                t.timestamp BETWEEN from_date AND to_date
                AND comp_ind_table.param_key = p_param_key;
END;
$$;


CREATE OR REPLACE PROCEDURE transfer_temp_values_station_param(
        p_station_id INTEGER,
        p_param_key INTEGER
)
LANGUAGE plpgsql
AS $$
BEGIN
        INSERT INTO temp_values_noindex (timestamp, value, master_id,
        ↪  quality)
        SELECT  t.timestamp,
                   value,
                   master_id,
                   quality
        FROM
                (SELECT
                        COALESCE(period_start, CAST(date AS timestamp) +
                        ↪  CAST(time AS interval)) as timestamp,
                        value,
                        quality
                        FROM temp_read_values
                        ORDER BY timestamp
                ) AS t,
                (
                        (SELECT position_id FROM positions WHERE station_id =
                        ↪  p_station_id) AS station_positions
                        INNER JOIN comp_ind_table ON
                        ↪  station_positions.position_id =
                        ↪  comp_ind_table.position_id
                )
        WHERE
                t.timestamp BETWEEN from_date AND to_date
                AND comp_ind_table.param_key = p_param_key;
END;
$$;


CREATE OR REPLACE PROCEDURE transfer_values_to_hypertable(batch_size INTEGER)
```

```
LANGUAGE plpgsql
AS $$
BEGIN
        WITH deleted_rows as (
                DELETE
                FROM temp_values_noindex
                WHERE ctid IN (
                                SELECT ctid
                                FROM temp_values_noindex
                                ORDER BY timestamp limit batch_size
                                )
                RETURNING *
        )
        INSERT INTO values(timestamp, value, master_id, quality)
        SELECT timestamp,
                   value,
                   master_id,
                   quality
                FROM deleted_rows ORDER BY timestamp;
END;
$$;
```

C.2 PostgreSQL Data Import Scripts

```
#!/usr/bin/env bash
# import data from our csv into the db's temp table
# usage: ./import_csv.sh <filename> <database> <table (default
↪  temp_read_values)>

filename="$1"
database="$2"
table="${3:-temp_read_values}"
if { head -n 30 "$filename" | grep -q -e '^Från Datum Tid (UTC);Till Datum Tid
↪  (UTC);' ; } ; then
        # interval measurements
        columns="period_start, period_end, representative_date, value,
        ↪  quality"
        header_marker='^Från Datum Tid (UTC)'
elif { head -n 30 "$filename" | grep -q -e '^Datum;Tid (UTC);' ; }; then
        # sampling measurements
        columns="date, time, value, quality"
        header_marker='^Datum;Tid (UTC)'
else
        echo "Header doesn't match sampling nor interval measurements,
        ↪  exiting" >&2
        exit 1
fi

query="COPY $table ($columns) FROM STDIN DELIMITER ';' CSV;"
# sed script explanation:
# - delete lines up to the header line (with Från Datum...)
# - delete everything after ";;" (the "comments" at the end)
#        (limit to first 100 lines for perf?, should be enough since it's just
↪  the lines that have the initial comments)
# - delete empty lines (eg, if there are more comments than data, or trailing
↪  newline maybe)
```

```
#        since psql complains otherwise
preprocess="1,/$header_marker/d; 1,100 s/;;.*\$//; /^\$/d;"

# read all values following the header into the db, removing empty lines (and
↪  the trailing "comments")
# and save return value for later
copy_output=$(sed -e "$preprocess" "$filename" | psql -U weather -d
↪  "$database" -c "$query" 2>&1)
# check exit code
failed=$?
if [ $failed -ne 0 ] ; then
        echo "Failed to import $filename ($failed):" >&2
        printf "%s" "$copy_output" | sed -e 's/^/\t/' >&2
        exit 1
fi

insert_count="${copy_output#COPY }"
echo "$insert_count"
exit 0


#!/usr/bin/env bash

total=0
current=0
filecount=0
database="$1"
currdir=$(dirname "$0")
filelist=$(cat -)
fullfiles=$(realpath $filelist)
default_logfile="$currdir/.postgres-imported-$(date +%Y-%m-%d_%H-%M-%S).log"
logfile="$(realpath "${2:-$default_logfile}")"
pushd "$currdir"
touch "$logfile"
truncate -s 0 "$logfile"
for f in $fullfiles; do
        echo "processing file: $f" >&2
        count=$(../postgres/import_csv.sh "$f" "$database")
        result=$?
        if [ $result -eq 0 ]; then
                basename "$f" | {
                        IFS='-' read -r station param _rest
                        # move everything to final table, clean temp table
                        psql -U weather -d "$database" -c "CALL
                        ↪  transfer_temp_values_station_param($station,
                        ↪  $param); TRUNCATE temp_read_values;"
                        if [ $? -ne 0 ]; then
                                echo "ERROR: failed to transfer values to
                                ↪  final table after file $f" >&2
                                # truncate temp table to avoid broken data
                                psql -U weather -d "$database" -c "TRUNCATE
                                ↪  temp_read_values;"
                        else
                                # save file as already imported just in case
                                echo "$f" » "$logfile"
                        fi
                }
                current=$((current + count))
                filecount=$((filecount + 1))
```

```
                if [ $current -gt 100000 ]; then
                        total=$((total + current))
                        echo "Total imported so far: $total rows from
                        ↪  $filecount files" >&2
                        current=0
                fi
        fi
done
total=$((total + current))
echo "Data import completed, total imported: $total rows from $filecount files"
↪  >&2
popd
```

C.3 MongoDB Setup Scripts

```
db.createCollection("stations", {
  clusteredIndex: {
    key: { _id: 1 },
    unique: true,
    name: "stations_id_idx",
  },
});

db.createCollection("positions", {
  clusteredIndex: {
    key: { _id: 1 },
    unique: true,
    name: "positions_id_idx",
  },
});

db.createCollection("parameters", {
  clusteredIndex: {
    key: { _id: 1 },
    unique: true,
    name: "parameters_id_idx",
  },
});

db.createCollection("comp_ind", {
  clusteredIndex: {
    key: { _id: 1 },
    unique: true,
    name: "comp_ind_id_idx",
  },
});

db.createCollection("values", {
  timeseries: {
    timeField: "timestamp",
    metaField: "master_id",
    granularity: "minutes",
  },
});
```

C.4 MongoDB Migration Script

```
#!/bin/bash
database="weatherdb"
dump_dir="../data/dbdump/"

psql -U weather -d "$database" -c "\\copy (SELECT timestamp, master_id,
↪  value, quality FROM values) TO $dump_dir/all_values.csv DELIMITER ','
↪  CSV;"
psql -U weather -d "$database" -c "\\copy (SELECT station_id, name, owner,
↪  is_active, network_type, updated FROM stations) TO $dump_dir/stations.csv
↪  DELIMITER ',' CSV;"
psql -U weather -d "$database" -c "\\copy (SELECT position_id, latitude,
↪  longitude, elevation, station_id FROM positions) TO
↪  $dump_dir/positions.csv DELIMITER ',' CSV;"
psql -U weather -d "$database" -c "\\copy (SELECT master_id, param_key,
↪  position_id, from_date, to_date FROM comp_ind_table) TO
↪  $dump_dir/comp_ind.csv DELIMITER ',' CSV;"
psql -U weather -d "$database" -c "\\copy (SELECT param_key, name, summary,
↪  is_interval FROM params) TO $dump_dir/params.csv DELIMITER ',' CSV;"

collection="values"
fields="timestamp.date(2006-01-02
↪  15:04:05+00),master_id.int32(),value.double(),quality.string()"
mongoimport --db "$database" --collection "$collection" --type csv  --fields
↪  "$fields" --columnsHaveTypes "$dump_dir/all_values.csv" 2>&1
collection="comp_ind"
fields="_id.int32(),param.int32(),position_id.int32(),from.date(2006-01-02
↪  15:04:05+00),to.date(2006-01-02 15:04:05+00)"
mongoimport --db "$database" --collection "$collection" --type csv  --fields
↪  "$fields" --columnsHaveTypes "$dump_dir/comp_ind.csv" 2>&1
collection="positions"
fields="_id.int32(),latitude.double(),longitude.double(),elevation.double(),station_id.int32()"
mongoimport --db "$database" --collection "$collection" --type csv  --fields
↪  "$fields" --columnsHaveTypes "$dump_dir/positions.csv" 2>&1
mongosh "$database" --eval 'db.stations.updateMany({}, {$set: {positions:
↪  []}})'
mongosh "$database" --eval 'db.position.createIndex({station_id: 1}, {name:
↪  "station_pos_idx"});'
mongosh "$database" --eval 'db.comp_ind.createIndex({param:1, position_id:
↪  1}, {name: "param_pos_idx"});'
mongosh "$database" --eval 'db.comp_ind.createIndex({position_id: 1, param:
↪  1}, {name: "pos_master_idx"});'
mongosh "$database" --eval 'db.values.createIndex({master_id: 1, timestamp:
↪  1}, {name: "master_time_value_idx"});'
mongosh "$database" --eval 'db.values.createIndex({timestamp: 1, master_id:
↪  1}, {name: "time_master_value_idx"});'
```

TRITA-EECS-EX-2023:403