Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

Kristian Salo

Analysis and future development of the sensor data storage of an IoT application

Master's Thesis
Espoo, June 30, 2020

Supervisor: Professor Petri Vuorimaa
Advisor: Olli Laine M.Sc. (Tech.)

Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

ABSTRACT OF MASTER'S THESIS

Author: Kristian Salo

Title:
Analysis and future development of the sensor data storage of an IoT application

Date: June 30, 2020 **Pages:** vii + 70

Major: Computer Science **Code:** SCI3042

Supervisor: Professor Petri Vuorimaa

Advisor: Olli Laine M.Sc. (Tech.)

An increasing number of sources from IoT to monitoring generate time-series data. This thesis examines the data storage of an IoT system, which uses a MySQL database for storage. The system has a history of performance problems for operations on the sensor data, such as some database queries taking tens of seconds to execute. Also, the system is expected to start receiving significantly increased amounts of sensor data in the following years. For these reasons, there is considerable uncertainty in the performance of the current sensor storage in the future. This thesis is undertaken to examine what should be done to the data storage of the system to ensure it performs acceptably in the future.

The objective of this thesis is to obtain a recommendation for the future development of the sensor system. More precisely, a list of suggestions that improve the performance of the system, and a time estimate for when they need to be implemented at latest is proposed. This recommendation is based on testing different data storage solution alternatives with the most essential use cases of the application, on future datasets. Three data storage solutions are tested: two improved MySQL solutions and a PostgreSQL solution with the TimescaleDB extension.

The results demonstrate that the data of the sensor system can be feasibly stored in a modified version of the existing MySQL database at least until 2028, when the size of the dataset is over 700 GB. This is enabled by implementing optimizations such as partitioning the sensor data and pre-calculation of data for performance-intensive queries. The recommendation is to implement these optimizations on the current MySQL database as soon as possible, but at latest at the start of 2021, when the number of slow queries was perceived to start growing rapidly. The PostgreSQL and TimescaleDB solution did not yield further performance gains, and is therefore not recommended for this system.

Keywords: sensor data storage, time-series data storage, relational databases, mysql, postgresql, timescaledb

Language: English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tieto-, tietoliikenne- ja informaatiotekniikan maisteriohjelma

DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Kristian Salo		
Työn nimi:	IoT-sovelluksen sensoridatan tallennusratkaisun analysointi ja jatkokehitys		
Päiväys:	30. kesäkuuta 2020	**Sivumäärä:**	vii + 70
Pääaine:	Tietotekniikka	**Koodi:**	SCI3042
Valvoja:	Professori Petri Vuorimaa		
Ohjaaja:	Diplomi-insinööri Olli Laine		

Suuri määrä erilaisia sovelluksia IoT:sta monitorointiin tuottaa aikasarjadataa. Tämä diplomityö tutkii erään IoT-järjestelmän sensoridatan tallennusta. Tällä hetkellä järjestelmän sensoridata tallennetaan MySQL-tietokantaan, jonka suorituskyky on paikoin ollut epätyydyttävä. Esimerkiksi joidenkin tietokantakyselyiden suoritusaika on ollut kymmeniä sekunteja. Lisäksi järjestelmän sensoreista kerättävän tietomäärän odotetaan kasvavan huomattavasti lähivuosina. Näistä syistä järjestelmän tiedontallennusratkaisun oletetaan olevan riittämätön tulevaisuudessa. Tämä työ tehtiin, jotta voitaisiin selvittää ratkaisuja, joilla järjestelmän tiedon tallennus saataisiin toimimaan riittävän nopeasti tulevaisuudessa.

Tämän työn tavoitteena on antaa suositus tutkittavan järjestelmän sensoridatan tallennuksen jatkokehityksestä. Suositus koostuu muutoksista jotka parantavat järjestelmän suorituskykyä, ja niiden toteuttamisen aika-arviosta. Suositus perustuu työssä tehtäviin testeihin: erilaisia tiedontallennusvaihtoehtoja testataan järjestelmän ennustetuilla tulevaisuuden tietomäärillä, käyttäen mittareina järjestelmän keskeisten käyttötapausten tietokantakyselyiden suoritusaikoja. Testit tehdään kolmelle tallennusvaihtoehdolle: kahdelle eri tavoin parannellulle MySQL-versiolle, ja PostgreSQL-versiolle, jossa on käytössä TimescaleDB-lisäosa.

Testien tulokset näyttävät, että IoT-järjestelmän sensoridata voidaan tallentaa jo käytössä olevaan MySQL-tietokantaan ainakin vuoteen 2028 saakka, jolloin tietomäärä on yli 700 GB. Tietokantaan tehdyt parannukset, kuten sensoridatan partitionti ja hitaiden kyselyiden tuloksien esilaskenta mahdollistavat tämän. Tuloksien perusteella nämä parannukset suositellaan otettavaksi käyttöön mahdollisimman pian, kuitenkin viimeistään vuoden 2021 alkuun mennessä, jolloin hitaiden kyselyiden määrän nähtiin kasvavan huomattavasti. Testattu PostgreSQL ja TimescaleDB -ratkaisu ei tuonut lisäsuorituskykyä MySQL-versiohin verrattuna, joten siihen siirtymistä ei suositella.

Asiasanat:	sensoridatan tallennus, aikasarjadatan tallennus, relaatiotietokannat, mysql, postgresql, timescaledb
Kieli:	Englanti

Acknowledgements

I wish to express my gratitude for everyone involved in the process of writing this thesis. Although it was a one-man effort for the most part, it was only made possible by the help and support of several parties.

First, warm thanks to my supervisor Petri Vuorimaa for providing guidance and devoting time to reading and commenting the thesis.

Thanks to Olli Laine for coming up with the idea for this thesis in an ad hoc coffee table discussion, and also offering to be the instructor for it without a second thought - your creativity and willingness to help made this thesis possible. Also, I am thankful to Vincit and the owner of the sensor system investigated in this thesis for facilitating this work.

I am grateful for the guidance and proofreading provided by Hannu Hartikainen and Mikko Pitkänen.

Finally, and most importantly, thank you to my beloved wife for patiently supporting and believing in me through the ups and downs of the writing process.

Helsinki, June 30, 2020

Kristian Salo

Acronyms

ACID Atomicity, Consistency, Isolation, Durability.

API Application Programming Interface.

GCP Google Cloud Platform.

IoT Internet of things.

JSON JavaScript Object Notation.

LSM-Tree Log-Structured Merge-Tree.

NoSQL Not Only SQL.

OLTP Online Transaction Processing.

ORM Object-Relational Mapper.

RAM Random-Access Memory.

RDBMS Relational Database Management System.

SQL Structured Query Language.

SSTable Sorted-String Table.

TSDB Time Series Database.

XML Extensible Markup Language.

Contents

Chapter 1

Introduction

Time-series data is being generated by an increasing number of sources ranging from Internet of things (IoT) sensor data to application usage data and monitoring data. This data often has special characteristics in comparison to traditional application data: the datasets are hundreds or thousands of GBs, the data is generated and needs to be ingested rapidly, and large amounts of this data are accessed by queries. New storage solutions such as Not Only SQL (NoSQL) databases and dedicated Time Series Databases (TSDBs) are often considered suitable solutions for addressing the special requirements of time-series data. Traditional Relational Database Management Systems (RDBMSes) are regarded unfeasible for this use case. However, according to the DB-engines ranking, RDBMSes are by far the most popular type of databases [9]. As a result, many applications are likely storing time-series data in an RDBMS.

This thesis examines the data storage of an IoT sensor system, which uses a MySQL database for storage. This system gathers data from sensors in buildings, and visualizes the data in different formats in three applications. The system has a history of performance problems related to the sensor data: occasionally inserting the data causes the system to become unresponsive, and some queries to the data have unfeasibly long execution times. Also, the system is expected to start receiving significantly increased volumes of sensor data in the upcoming years. For these reasons, there is considerable uncertainty in the feasibility of the performance of the system in the future.

The main research question of this thesis is: what should be done to the data storage of the system to ensure it performs acceptably in the future? This question can be further divided into several subquestions. First, how long will the performance of the current system remain acceptable? Second, how can the system be changed to improve its performance? Third, how much do these changes improve the performance of the system? And, finally,

what are the costs and benefits of implementing these changes?

The objective of this thesis is to obtain a recommendation for the future development of the sensor storage of the system. This recommendation is based on testing different data storage solution alternatives with the most essential use cases of the application, on estimated future datasets. The results of these tests are evaluated against the costs of migrating the existing system to use the solutions. The recommendation includes a list of storage system changes that improve the performance of the sensor data storage. Furthermore, a time estimate for when these changes should be implemented at latest, i.e., when the system will no longer be feasible to use as is, is presented.

As mentioned, this thesis investigates the methods by which the sensor data of the system can be stored and accessed feasibly in the future. However, the purpose of this work is not to create a finalized working storage system that could be deployed to a production environment. Rather, the most essential parts of such environments are tested to obtain an estimate of their performance. Also, addressing problematic queries or methods of using the sensor data that may be encountered while overviewing the system is not in the scope of this thesis. Instead, the storage performance is assumed to be constrained by the organization of the data, not by, e.g., inefficiently formatted queries or use cases, which are considered to be straightforward to address later.

This thesis is structured into six chapters, starting with this introduction. The second chapter overviews different types of databases, techniques for increasing database performance and aspects related to storing time-series data. The following chapter examines the storage system of this thesis: its architecture, sensor reading storage structure, data usage patterns, problems and future requirements. The fourth chapter outlines the test cases, test data generation, test environment, tested storage solutions and test results. The fifth chapter evaluates and compares the test results, and also analyzes the reliability of the performed tests. Finally, the last chapter presents the recommendation for the future development of the system, and also gives general recommendations for increasing the performance of time-series data storage.

Chapter 2

Background

Data is the most essential part of almost every modern web application. The sources and types of data vary greatly between applications, but the challenges in storing it have many similarities. The storage of almost every web application is handled by a database, the most popular types being relational databases and NoSQL databases. This chapter examines databases in general, and then investigates the storage of a specific type of data, time-series data; these matters are considered to provide sufficient background for analyzing the sensor system of this thesis and implementing optimizations to it. First, an overview of the most prevalent database types is given. Then, approaches to database performance are presented. Finally, time-series data, its characteristics and storage considerations are outlined.

2.1 Different databases

This section outlines the most popular database types: relational databases, NoSQL databases and NewSQL databases.

2.1.1 Relational

The relational model was first proposed by Codd [4] in the 1970s. It lead to the creation of RDBMSes, that enable creating, controlling and maintaining access to the database. They have since remained the most popular choice of database. The relational model applies the concepts of relational algebra to databases: the data is stored in relations that are unordered collections of tuples, each tuple uniquely identifiable. These tuples contain attributes, which are atomic values such as numbers and strings; nested values such as lists or maps cannot be stored in an attribute. An essential idea in the

relational model is hiding the internal representation of the data from users. As a result, the users can focus only on the natural structure of the data.

The relational model states that the operations on the data should be done with a high-level query language. After the first RDBMSes emerged, Structured Query Language (SQL) quickly became the query language used in nearly every commercial RDBMS. It is a declarative language: instead of specifying how an operation should be done, it specifies the pattern of the data that should be the output of the query, leaving the execution details of the query to the database's query optimizer. [16]

The structure of the data in a relational database is enforced by defining the format of the relations (tables in SQL) and attributes of tuples (columns and rows in SQL). This specification is known as the database schema. The schema can include constraints of many types, from the naming of tables and columns of rows, to various integrity constraints. For example, the data type of each column must be specified, columns can be made nullable or non-nullable, and a column can be constrained to always reference a row in another table. These constraints are enforced by the database engine every time data is written to the database. Thus, the data in a relational database is always consistent, i.e., it conforms to the database schema. [16]

Modeling relationships in SQL is usually done by separating the data into multiple tables, and referencing related records in other tables by their unique identifiers with an integrity constraint. This is known as normalization, and was first proposed by Codd as part of his relational model. The purpose of normalization is to reduce data redundancy and improve data integrity. [4] To illustrate, the main purpose of normalization is to ensure that each piece of data is stored only once, and will maintain its integrity through different database operations. Such inconsistencies in data are often referred to as data anomalies; they are eliminated by arranging the columns and tables of the database according to sets of rules referred to as normal forms. A database is often considered normalized if it follows the third normal form. [7]

Figure 2.1 presents an example SQL schema of an apartment rental service. The schema has four tables: `Customer`, `Country`, `Apartment` and `Reservation`. Relationships in the schema are modeled by foreign key references. For example, each row in `Reservation` has foreign keys to both `Customer` and `Apartment`. Also, there is a separate `Country` table to prevent data duplication: if the country data was stored in `Customer` table, the country abbreviations would be duplicated. For example, each customer row with the country named Finland would always contain its abbreviation (FI); in the current scheme the abbreviation is only stored once in the `Country` table, which is referenced from the `Customer` table.

However, normalization can cause related data to be spread across multiple tables, creating a need for multi-table joins when queried. As a result, it may sometimes be beneficial to store data in a denormalized manner, i.e., not according to the normal forms. Storing derived data and pre-joining tables are examples of normalization. [2] For instance, the apartment rental application of Figure 2.1 could denormalize its data model by storing the apartment description in the `Reservation` table (pre-join), and adding an `end_date` column calculated from `start_date` and `duration` columns to the `Reservation` table (derived data). These would make querying reservation data faster, since no joins or calculations would need to be performed. Denormalization is also enabled by later versions of SQL (SQL:1999 and after), which introduce the possibility to store and query structured data types such as Extensible Markup Language (XML) and JavaScript Object Notation (JSON) in a single row. [16] Although this reduces the need for multi-table access, thus often providing better read query performance, it also eliminates the benefits of normalization: data anomalies may occur, writes become slower, additional storage space is needed and application-level code may become more complex. [2]

Transactions have been an important concept in RDBMSes since they first emerged. Transactions are often characterized by the acronym ACID (*Atomicity, Consistency, Isolation, Durability*) [16]:

- *Atomicity* refers to a transaction being a method of grouping several read and write statements into one logical unit. This unit is then executed as a whole: the entire transaction either succeeds (commits), or fails (aborts or rollbacks).

- *Consistency* means that whether the transaction succeeds or fails, the database will be in a consistent state. A consistent state is one where the constraints of the database are satisfied.

- *Isolation* indicates that transactions are isolated from each other: concurrently running transactions will not interfere with each other. Such interference can occur when the data written by one transaction is read or written by other concurrently executing transactions. For example, if one transaction makes multiple writes, either all of them are visible to other transactions, or none are. Different databases support different isolation levels, providing varying isolation guarantees, and ensuring certain types of concurrency errors cannot occur.

- *Durability* is the promise that once a transaction commits, any data written by it is stored durably, and will not be lost, even if a hardware error or power outage takes place.

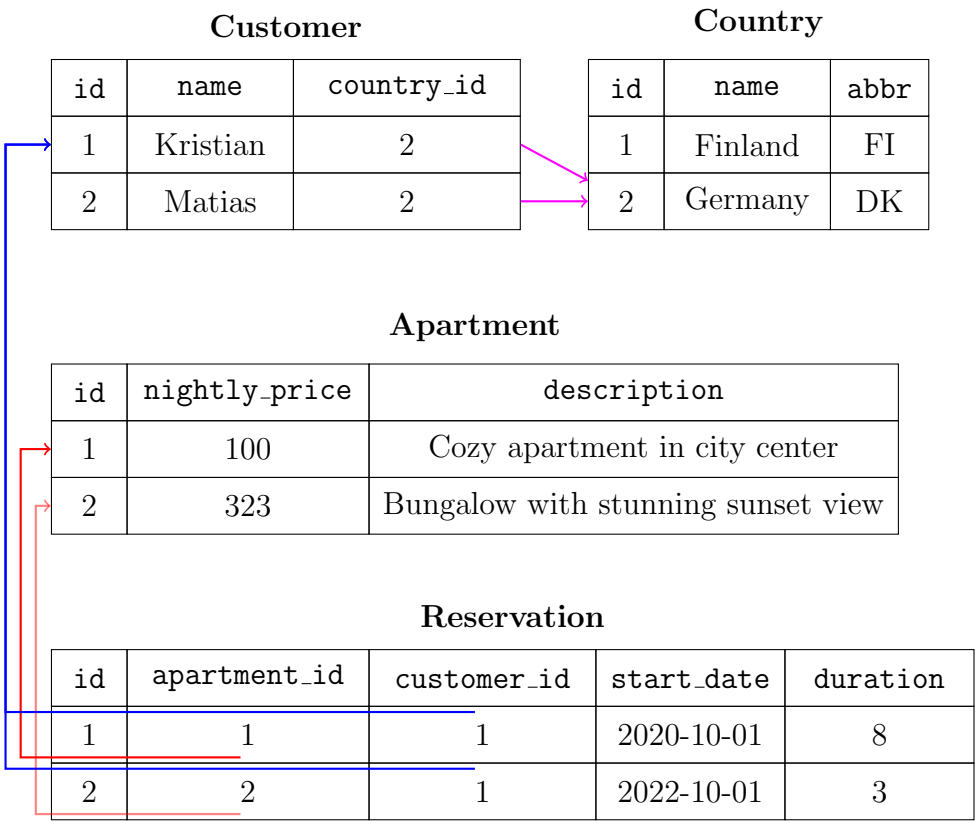

Figure 2.1: An example SQL schema of an apartment rental service. The arrows represent foreign key references. The schema is normalized in the third normal form; it has no redundant data nor possibilities for data anomalies on database operations.

All in all, transactions make error handling in application-level code easier: database errors caused by hardware failures, network downtime or race conditions between concurrently executing client statements need not be handled. Instead, these fault-tolerance mechanisms are implemented by the RDBMS. All major RDBMSes, such as PostgreSQL, MySQL, Oracle and SQL Server, implement ACID transactions. [16]

2.1.2 NoSQL

NoSQL databases originally referred to "open-source, distributed, nonrelational databases" in a 2009 NoSQL meetup. The term NoSQL has since been used to denote various different database properties, and has become difficult to define unambiguously. [26] In this thesis, it is used as an umbrella term to refer to non-relational databases.

Although the NoSQL family of databases includes a large number of different data stores, several common characteristics can be attributed to them:

- No use of SQL. Many NoSQL databases have query languages that have some similarities to SQL, making them easier to adopt. However, none are close to the flexible query capabilities of SQL. [26]
- Complete lack of schema or a very flexible schema. As a result, fields in the database can be added or modified without defining changes in the schema first. This is especially useful for rapid application development and heterogeneous data. [26]
- Built-in support horizontal for scaling in a cluster of multiple machines. The database can be split across multiple machines (partitioning, see 2.2.3), or hosted on multiple machines (replication, see 2.2.2). [14]
- High availability. Many NoSQL databases are used by large companies that cannot afford downtime. As a result, the data stores must be easily replicable and include a failover mechanism to provide continuous service even in case of data center or machine failures. [14]

Due to the umbrella nature of the NoSQL term, not all NoSQL databases focus on all of these properties. For example, graph databases are not primarily designed for running on clusters. However, almost all NoSQL databases include some characteristics of each property. All in all, NoSQL stores can be considered flexible, highly scalable, highly available, non-relational and non-transactional.

The data model of the majority of NoSQL databases differs significantly from that of relational databases. Instead of operating on rows of simple, atomic values, more complex structures are used. These structures can be read from and written to as units. Sadalage and Fowler [26] refer to this as "aggretate orientation", a term they introduced specifically for this common data modeling pattern found in NoSQL stores. This technique is often combined with denormalization, which is also used in relational databases, as explained in Subsection 2.1.1.

When deciding the aggregate boundaries, i.e., the units of data that are stored together, it is important to consider the data access patterns of the application. Data that is frequently read from or written to together should be stored in the same aggregate, because then it is also stored together. [26] Listing 2.1 presents an example data model for a NoSQL apartment rental service. The model has only one aggregate, a customer, which includes the customer's rentals as a nested structure. Also, the country of the customer is stored as a denormalized string in the aggregate. These design choices allow the rental data of a customer to be read and modified quickly: only this one aggregate needs to be accessed.

Aggregate orientation has many consequences to the operation of NoSQL databases. First, it enables running them easily on clusters, since it groups information that is usually accessed and manipulated together into same nodes, reducing the need of multi-node access for a single query. Second, it enables ACID-like modifications of the data, despite NoSQL databases lacking the ACID transactional support of relational databases. To clarify, the majority of NoSQL stores support atomic modifications of a single aggregate. Therefore, if the modifications only span one aggregate, which is often the case for a correctly designed data model, they can be executed atomically. Finally, it makes handling inter-aggregate relationships difficult. For instance, data access and manipulation spanning multiple aggregates need to be handled in application code. Also, satisfying new types of queries may be infeasible because the data model has been designed for certain data access patterns. For this reason, some NoSQL databases provide a method of handling queries that are not feasible on the aggregate structure. These are often called materialized views. They include the pre-computed result of some query that is hard to execute on the data model. [26] For example, obtaining all rentals of an apartment for the data model of Listing 2.1 would require traversing the entire customer dataset, which is infeasible for larger amounts of customers.

```
// customers
{
 "id": 1,
 "name": "Kristian",
 "country": "Finland",
 "countryAbbr": "FI",
 "rentals": [
  {
   "apartmentId": 1,
   "apartmentDescription": "Authentic NYC experience",
   "price": 223,
   "startDate": "2020-10-01",
   "endDate": "2020-18-01"
  },
  {
   "apartmentId": 2,
   "apartmentDescription": "Hideaway by the ocean",
   "price": 149,
   "startDate": "2020-10-01",
   "endDate": "2020-10-03"
  }
 ]
}
```

Listing 2.1: An example NoSQL data model for an apartment rental service. The model enables fast retrieval and modification of a user's rentals. However, for example, fetching all rentals of an apartment would require scanning the entire customer dataset, which is slow or infeasible for larger data sets.

2.1.2.1 Types of NoSQL stores

NoSQL data stores can be categorized based on their data model: key-value, document, graph and column-family (wide-column) stores. This categorization does not include all NoSQL databases, but it represents the most common options, and is a widely used. [26] The following paragraphs examine each of these types in further detail. Table 2.1 summarizes their key features and provides examples of each type.

Key-value stores are the simplest NoSQL stores from an Application Programming Interface (API) perspective. Most of them support three operations: getting the value of a key, putting a new value for a key and deleting the value of a key. As a result, key-value stores are similar to map or dictio-

nary data structures in programming languages: they store blobs of data by a key. Since the value data is opaque to the key-value store, operations on the data are done solely based on the keys, and the structure and type of the values must be handled in application code. Because all key-value store operations are done via a primary key, they generally have excellent performance and scalability. However, due to this simplicity, they prevent modeling any relationships in the stored data. Also, most of the key value stores provide no transactional guarantees for writes. For example, if multiple writes to the same key are done concurrently, the latest write overwrites the preceding writes. For these reasons, key-value stores are especially suitable for storing large volumes of non-critical data: session data, user profile data and shopping cart data are good examples of typical use cases for key-value stores. [26]

Document stores operate on collections of JSON documents, each document uniquely identifiable within its collection. Their structure is somewhat identical to relational databases: a collection corresponds to a table and a document to a row. However, there are significant differences between the two: the documents in a collection can have different shapes, whereas the rows in a table all have the same columns. Also, document databases generally do not support transactions spanning multiple documents. Moreover, due to the aggregate-orientation of documents, document databases do not support ad hoc queries. However, they support a richer set of queries than key-value stores, because they are aware of the structure of the documents. Consequently, they enable queries based on the attributes of the documents, and also fetching only parts of the document if desired. Also, they support more complex query operations such as sorting and grouping. As a result, document stores are a feature-rich tool suitable for a variety of use cases. For example, web analytics and content management systems are well suited for these databases. [26]

Column-family stores enable storing rows of columns. These columns are grouped into column families, which are aggregates of data that are often accessed together, and also stored together by the database. As a result, column-family stores are essentially two-dimensional maps; they can be compared to a relational database view of multiple tables. Each row in a column-family database can have different columns, and the number of columns can be large: for example, Cassandra has a concept of wide rows, which can have thousands of columns. [26] Also, the columns are sparse, i.e., each row only stores the columns it has a value for; in a relational database all rows of the table reserve storage space for all columns, even the ones with NULL values. Therefore, the storage space is used efficiently in column-family stores, making it suitable for storing huge amounts of data with large

Type	Query possibilities	Transactions	Relationships	Example databases
Key-value	Limited	None	No support	Couchbase, Memcached, Redis
Document	Average	Atomic	Some support	CouchDB, MongoDB, OrientDB
Column-family	Some	Atomic	No support	Cassandra, HBase, Bigtable
Graph	Extensive	ACID	Extensive	ArangoDB, Neo4j, GraphDB

Table 2.1: A summary of properties of different types of NoSQL databases and examples of each type.

numbers of varying attributes. The query possibilities for column-family stores are more restricted than those of document stores, but more expressive than key-value stores. They allow some more complex operations such as range queries and logical operators (and, or), but these are only supported on keys and indexed columns. Features such as joins and aggregates are not possible at all. [14] Event logging data is a good example of a suitable data for a column-family store: different types of events would have different columns, and high ingestion rates and data volumes would be easily supported.

Graph databases specialize in storing and managing highly linked data. They store graphs consisting of nodes and edges. Both nodes and edges themselves may include objects of key-value pairs. Graph databases can have a pre-defined schema, which may restrict the types of keys and values included in edges and nodes. Also, more complex restrictions are possible, such as allowing only certain types of edges to exist between certain nodes. [14] Graph databases often have support for ACID transactions. Also, their query languages support complex graph-specific operations such as traversals of arbitrary length and graph shape matching. [26] Consequently, they differ from other NoSQL databases: their data model is based on the properties of the data that is stored instead of aggregates of common query patterns. This is enabled by having a flexible query language. Graph databases are suitable for data with a substantial number of relationships: social networks, routing and recommendation engines are examples of such cases [26].

2.1.3 NewSQL

The term NewSQL refers to a class of relational database management systems with the objective of combining the horizontal scalability and fault tolerance of NoSQL solutions with the ACID guarantees of relational databases. NewSQL stores use SQL as their query language and support the relational data model. However, they are based on different architectures than traditional RDBMSes. They are said to utilize the possibilities offered by

cloud platforms while also offering the benefits of well-established relational databases. Popular NewSQL stores are, for example, Google Spanner, VoltDB, Clustrix and NuoDB. [13]

The data model of NewSQL solutions is relational by definition. For example, VoltDB and Clustrix provide their clients a relational view of the data. Google Spanner, on the other hand, uses a semi-relational model, in which tables are mappings from the primary keys to other columns. Despite the interaction with these databases occurring through the abstraction of tables and columns, the internal representation of data in NewSQL databases may be totally different. For instance, NuoDB can utilize any compatible key-value store as its data storage backend. [13]

Queries in NewSQL solutions are done with SQL. However, none of the NewSQL solutions presented here are fully SQL compliant, and there are considerable differences in their support for SQL features. For example, VoltDB does not support the *having* clause, joining tables with themselves, or joins for tables that are not partitioned on the same value. On the other hand, Clustrix and NuoDB are almost fully SQL compliant, and have only minor deviations from the standard. [13]

Despite the advantages presented above, NewSQL databases are not a silver bullet combining the advantages of relational databases and NoSQL stores. Most of the NewSQL stores are proprietary software, such as all the examples presented in this thesis. Also, many of these stores are specialized to limited scenarios, limiting their wide-spread adoption. [25] Moreover, the majority of NewSQL stores use Random-Access Memory (RAM) for storing their data, which enables them to be fast; however, it also makes them more expensive to use, since RAM is more expensive than disk space which is used by traditional RDBMSes. Finally, despite resembling SQL databases, they can only partially benefit from the extensive tooling of relational databases. [17]

2.1.4 Choosing a database

Different databases are aimed at solving different data storage problems, and are therefore suitable for distinct use cases. Also, the choice of database has significant long-term consequences: if the selected database has performance problems or operational challenges, migrating to a different database is time-consuming and nontrivial, often requiring considerable changes to all code that interfaces with the database. [24] Therefore, the choice of database should be done carefully, and different options assessed thoroughly. This subsection highlights the advantages and disadvantages of relational, NoSQL and NewSQL databases for different use cases, and provides insight for choosing

a database.

NoSQL stores are a recent technology that is often presented as a superior alternative to relational databases. Sadalage and Fowler [26] propose that there are two central reasons for choosing a NoSQL store over a relational database: programmer productivity and data access performance.

Programmer productivity with NoSQL databases can be increased in a few ways. First, the fundamental problem of transforming the aggregate data structures of application level code into rows of tables for relational databases (often referred to as the object-relational mismatch [16]) is removed by the aggregate orientation of NoSQL databases, allowing data to be saved in the format it is used in application-level code. Object-Relational Mapper (ORM) tools mitigate this problem for relational databases, but they are not suitable for all use cases: especially performance-intensive data access often requires workarounds. Second, NoSQL databases are better suited for working with nonuniform data. Their schema-free nature allows ad hoc fields to be saved and queried effortlessly. [26]

There are many reasons why a NoSQL database may perform better than a relational one. First, fetching aggregates from aggregate-oriented NoSQL databases may be fast compared to relational databases, in which the data is spread over multiple tables. This locality of data storage may provide performance benefits for both reads and writes. Second, the built-in ability to scale horizontally by adding more servers into the resource pool enables NoSQL databases to perform better at scale than relational databases. [26] Scaling relational databases horizontally is also possible, but fundamentally they are designed to be scaled vertically, i.e., by adding more processing power to a single server [28].

However, despite the promises of NoSQL, there are many reasons to favor a relational database: in fact, Sadalage and Fowler [26] argue that the majority of applications are best served by a relational database. First, the majority of applications never need to process the tera- or petabytes of data and the millions of users that internet giants, the requirements of which initially started the NoSQL movement, such as Google and Amazon do. As a result, the performance and scalability of SQL databases is more than sufficient for these applications. Second, handling relationships in data is natural with the constraints of relational databases; in NoSQL data stores relationships need to be handled in application-level code, and there are no database-level guarantees of their integrity. Third, the lack of ACID transactions can be a major restriction for mission-critical systems such as applications in healthcare or banking industries. [26] To illustrate, it has been argued that dealing with the performance problems of transaction overuse is preferable to working with the lack of transaction support in NoSQL [6]. Additionally, SQL has

more features than NoSQL query languages, enabling it to handle various data access patterns. Finally, the maturity of relational databases further supports their use: developers are familiar and productive with them, there are many tools for working with them and they are less likely to contain defects than newer solutions. [26]

In any case, it should be emphasized that the arguments in the favor of any database technology are not absolute but rather application-dependent. As a result, the expectations about possible benefits of some database technology should be tested and evaluated for each application separately. [26] For example, if a NoSQL store is selected because of its developer productivity and performance, these should be evaluated. Essential features of the application should be tested on the database to see if they can be developed efficiently and if the aggregate-oriented data model is suitable for them. Also, the performance of the database should be tested against realistic application loads, with realistic data. The database should be selected only if the tests prove that the benefits are analogous to the expected ones.

Separating the data storage of an application into different databases is a viable option, especially when the application stores various types of data. For example, user session data is best suited for a key-value store and transactional order data for a relational database. This concept of utilizing multiple, best suited databases for each type of data is called polyglot persistence by Sadalage and Fowler [26]. Polyglot persistence has the obvious advantage of using the most suitable database solution for each type of data. However, it also incurs considerable programming complexity, as different databases need to be interfaced. Also, the maintenance overhead is increased because multiple databases need to be deployed and managed. Therefore, the advantages of polyglot persistence need to be evaluated against this complexity.

To summarize, selecting a database is a complex and important design decision. Sadalage and Fowler [26] recommend that applications start the database selection process from relational databases. According to them, a NoSQL database should only be selected if it demonstrates significant advantages for the application after evaluating it in practice. Some good reasons for selecting a NoSQL solution are, for example: performance and scalability problems with relational databases, a heterogenous and rapidly changing data model and aggregate-orientation of the stored data. If the scalability and performance properties of NoSQL solutions are desired while also maintaining consistency and full ACID transactional support, NewSQL databases may be a good solution. However, it is worth noticing that they are a relatively new technology, and also have their disadvantages. Finally, it may be a good alternative to utilize multiple different data stores, especially if the applications stores many considerably differing types of data with spe-

cial storage requirements. All in all, different options should be carefully considered and tested before selecting a database.

2.2 Database performance

Database performance is essential for most present-day web applications, and the trend of gathering increasing amounts of data has furthermore emphasized it. Increasing the performance of a database is challenging, because it is dependent on multiple factors: having an appropriate schema, creating proper indexes, distributing the data via replication and partitioning, tuning the database servers, and optimizing the queries. This section highlights the most common aspects that should be considered when working with database performance, regardless of the specific database technology in use.

2.2.1 Indexing

A database index is a data structure that improves the performance of reads, while incurring additional write-time overheads. It functions much like an index of a book: it is used to look up the location of the desired data on disk, preventing the database from having to scan the whole dataset when querying for certain rows only. Different indexes are needed for querying different data, or for querying the same data in different ways. When inserting, updating or deleting data, the index needs to be updated accordingly, causing a write penalty. Therefore, indexes should be created with consideration: if a certain query is frequent in the application, it should be indexed. [16] The following paragraphs examine the two most popular types of of indexes, B-trees and Log-Structured Merge-Trees (LSM-Trees).

B-trees are the default index option in the majority of relational databases, and many NoSQL stores also support them. In essence, B-trees store key-value pairs that are sorted by keys, enabling efficient range queries and lookups. The time complexity for these operations is logarithmic due to the tree structure and data being sorted. B-trees divide the database into fixed-size pages (nodes), usually between 2 and 16 kBs in size, and operate on one page at a time. Each page is identified by an address, which are used to construct B-trees. B-trees consist of a hierarchy of these pages. This hierarchy can be navigated to locate the desired items quickly. [16]

Figure 2.2 illustrates an example B-tree that is used to look up key 325. The lookup process is started from the root of the tree, which is depicted as the upmost page. The page contains several keys and references to child pages. The child pages store continues ranges of keys, and the keys between

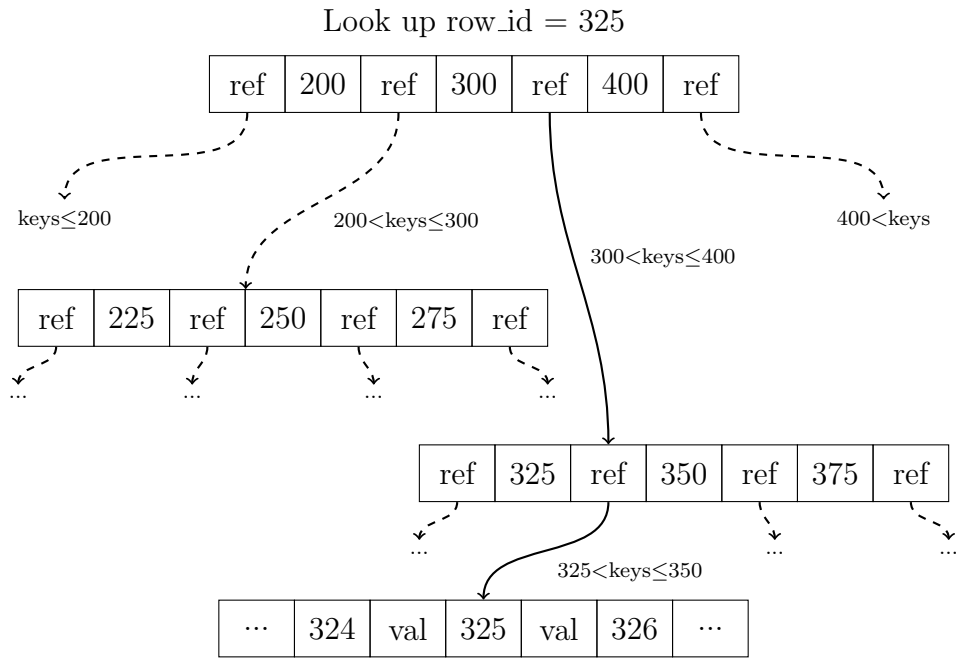

Figure 2.2: Lookup of a value with a B-tree index.

the references of the parent page illustrate the boundaries between these ranges. As a result, a value is looked up by traversing the B-tree into the ranges the key is in until a leaf page is entered. Leaf pages have no references to other pages; they contain a value for each key inline or a reference to the disk page in which the values are stored.

B-trees are an example of a mutable storage structure: writes are applied to existing values in place. For example, when updating the value of an existing B-tree key, the key is searched, the value updated, and the entire page containing the value written to disk. A new key is inserted by finding a range that encompasses that key, and inserting a new key in it. If the page is full, it is split into two half-full pages, and the references from the parent page are updated to point to these new pages. This ensures that B-trees remain balanced, and retain their logarithmic time complexity. [16]

LSM-Trees are an immutable on-disk storage structure that are widely used in modern NoSQL systems. For example, Cassandra, Google Bigtable and HBase use a LSM-Tree indexing. Immutability refers to these structures not changing after they have been created. As a result, all write operations to LSM-Trees are implemented by adding a new entry to the structure: inserts and updates simply add new entries, while deletes add a tombstone entry

indicating the deletion of an entry. The consequence of this immutability is that there may be multiple copies of each entry. Therefore, LSM-Trees require additional work at read time, since multiple locations may need to be accessed to determine the absence or value of an entry. [20]

An LSM-Tree consist of an in-memory component and an on-disk component. All incoming writes are appended to a sorted in-memory component, often referred to as a *memtable*. This memtable may be implemented with any sorted indexing structure: B-trees and skip-lists are widely used options. When the size of the in-memory component exceeds some selected threshold, it is written to disk as a new disk component. These disk components are often implemented using Sorted-String Tables (SSTables). An SSTable contains a list of data blocks and an index block. A data block stores ordered key-value pairs, and the index block stores the key-ranges of all data blocks. [20]

Since a new SSTable is added every time the in-memory component is written to disk, disk components with many entries for a single record accumulate over time. This is addressed by *compaction*, which iterates over several SSTables and merges their contents into a new SSTable, which only has a single entry for each value. The compaction process is efficient even for large SSTables because they are sorted. The tables can be iterated over one entry at a time, similar to the iteration in the *mergesort* algorithm. [16]

The structure of an LSM-Tree is shown in Figure 2.3. There is a memory-resident memtable and multiple (in this case only two, but often numerous) on-disk SSTables. The index of each SSTable is sparse: it consists of only the first entry of each data block which is sufficient since the data blocks are relatively small, and can be searched quickly. This makes the size of the index small, and it can be stored in memory easily, ensuring rapid data accesses. Also, the two SSTables contain duplicate values. For example, the key *eggs* has been deleted, but is still stored in the preceding SSTable. Also, the value of key *bacon* has been updated, and the old value is stored in the old SSTable. These duplicate values would be deleted by the compaction process which would merge these SSTables into one new SSTable.

Figure 2.3 also illustrates the lookup process of a key from a LSM-Tree. First, the in-memory memtable is queried to check if the key is there. Then, the most recent SSTable on disk is queried. The data block of the key *cider* is looked up from the sorted sparse index: since bag $\leq$ cider $<$ coke, the value may be the data block starting from offset 64567. That data block is then searched for the key. Because the key is not found, the second most recent SSTable is searched, with the same process as the first one. This time, the key is found, and its value is returned as the result of the query.

B-trees and LSM-Trees have different advantages. LSM-Trees often per-

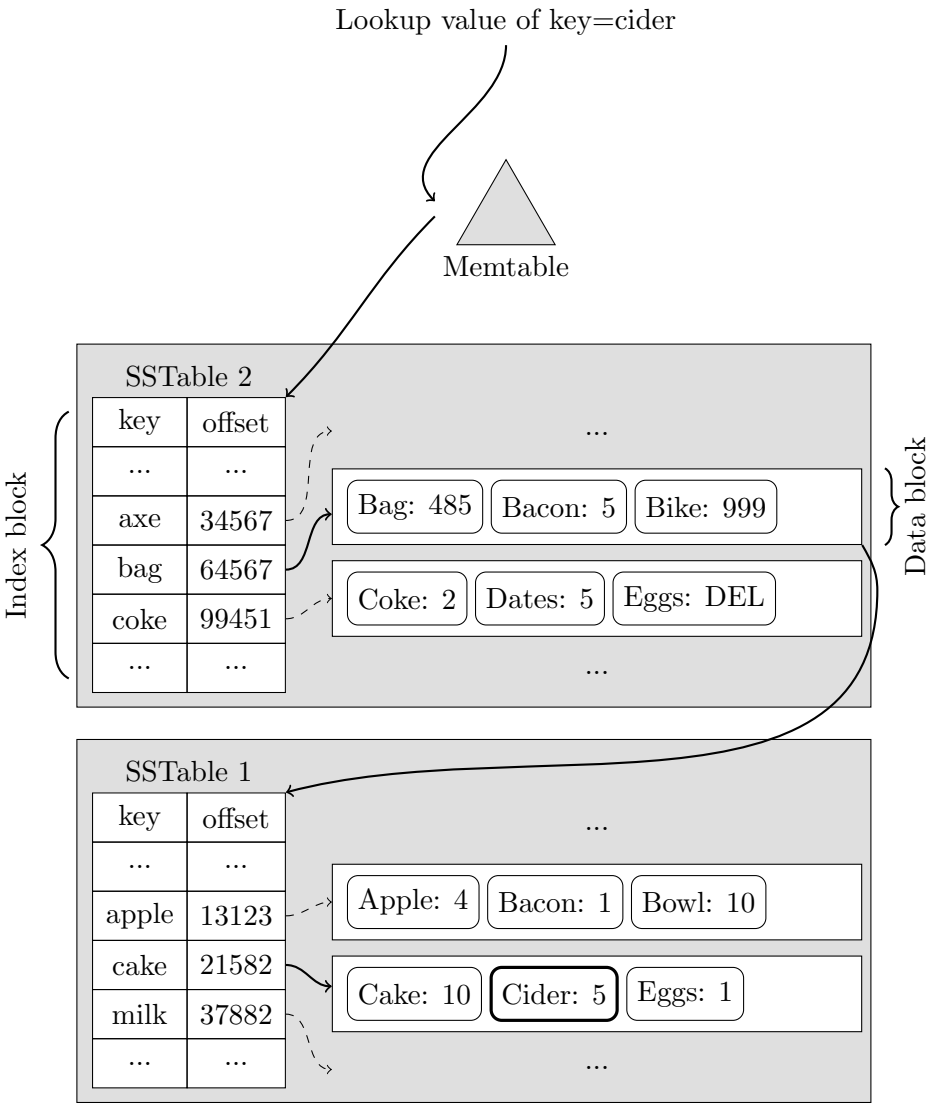

Figure 2.3: Lookup of a value with a LSM-Tree index.

form better in write-intensive applications. This is because they do not have to locate records on disk for writes as B-trees do, but instead always append new records sequentially. This sequential access pattern typically results in better performance than the random access pattern of B-trees. Also, B-trees always have to write an entire page at a time, even if only a single value has changed, whereas writes for LSM-Trees are always entire memtables. However, B-trees may offer more predictable performance than LSM-Trees. The writes caused by the compaction of LSM-Trees can interfere with incoming read and write requests, resulting in long-running queries. This is especially relevant for larger data sets with more perfomance-intensive compaction operations. Also, reads for LSM-Trees may be slower because they potentially need to traverse multiple SSTables to find an entry. These multiple entries for a single key also result in transactional semantics being difficult to implement, whereas B-trees only store each key once, making transactional guarantees easier to implement. All in all, B-trees provide consistently good performance for a wide variety of workloads, making them a good option for many use cases. LSM-Trees, on the other hand, perform better than B-trees in some situations, making them a viable alternative for many applications. [16]

2.2.2 Replication

Replication refers to storing (replicating) the same data on multiple servers. Therefore, it is a way to achieve higher read performance: requests can be distributed evenly on multiple servers instead of being handled by a single machine. In addition, it allows the data to be kept geographically near to the users, reducing access latencies. Furthermore, it increases the availability of the system: in case of a failure of one server, its load can be allocated to other servers. However, despite having many benefits, replication is a complex architecture and is difficult to implement correctly. The key difficulties in replication are related to handling changes in the replicated data. The next paragraphs explore the two most popular algorithms for replicating changes in the data: single-leader and multi-leader replication. Almost all major distributed databases implement one of these approaches. [16]

Single-leader (also known as active/passive or master-slave replication) is the most common replication algorithm. It designates one of the replicas as the leader. The leader is responsible for handling all write requests from clients, which are first written to its local storage. The remaining replicas are designated as followers. Whenever the leader writes to its local storage, it also sends a notification about the data change to all the follower replicas as a replication log or change stream. Then, the followers update their local

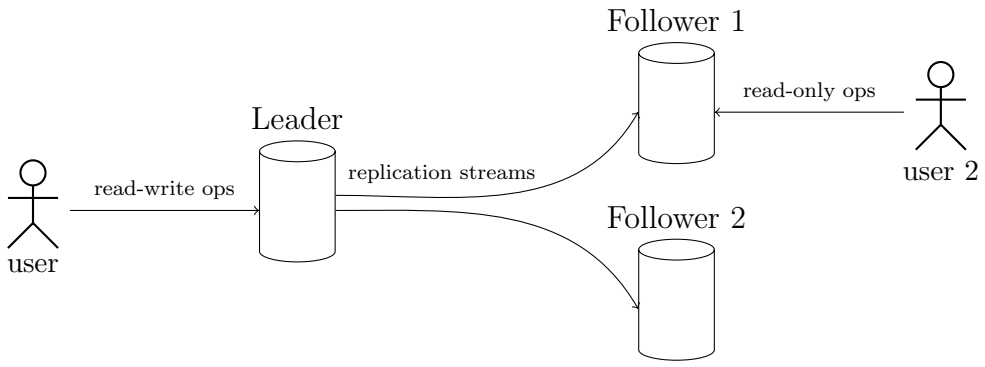

Figure 2.4: Single-leader replication topology.

storages accordingly. From the point of view of the client, reads are possible to all replicas, but writes must be handled by the leader. This type of replication is popular amongst relational databases, including PostgreSQL, MySQL and Oracle. Moreover, it is also used in some NoSQL data stores, such as MongoDB and Redis. [16] Figure 2.4 illustrates the single-leader topology: users can read from both leader and follower nodes, but only the leader accepts writes, which are propagated to the followers.

Multi-leader replication allows multiple nodes to accept writes. Therefore, it eliminates the main problem with single-leader architecture, in which all write request must be handled by the single leader. If there are problems with connecting to the leader, writes to the database cannot be done. Multi-leader replication not only allows writes to be handled even if one leader is down, but also provides scalability for write requests, since multiple nodes can handle them. The only difference between multi-leader and single-leader replication is the number of leader nodes; replication is done similarly by sending a data change for each write to all other nodes. [16] However, the problems caused by allowing concurrent and possibly conflicting writes are generally considered to outweigh the write scalability benefits. Therefore, multi-leader architecture should only be used if it provides additional benefits when compared with the single-leader option. [15] For example, Kleppmann [16] argues that when the data is stored in multiple data centers or when client applications support offline operation, multi-leader architecture is a reasonable choice. Some databases, such as CouchDB support multi-leader replication by default, but it is also often implemented with external tools, such as BDR for PostgreSQL and Tungsten for MySQL [1, 5, 29].

2.2.3 Partitioning

Partitioning (also, sharding) refers to splitting the data into partitions, which are stored on different servers. It provides scalability in three dimensions: read requests, write requests and data storage are all distributed to different partitions. However, the scalability is only achieved if the data is spread evenly across all partitions. Therefore, an essential part of partitioning is choosing the correct partitioning scheme, i.e., a method of deterministically dividing data into partitioned nodes. [16] The following paragraphs examine two popular partitioning schemes: key-range partitioning and hash partitioning. Then, the use of secondary indexes with a partitioned database is investigated.

Key-range partitioning assigns a continuous range of keys (from a given minimum to a given maximum) to each partition. When the key-range boundaries are known, operations on a key become straightforward: the partition of the key can be easily determined, after which the operation can be handed over to the node containing that partition. The keys inside a partition can be kept sorted, which makes range scans fast. Partition boundaries in a key-range scheme should be chosen to reflect the distribution of the data. If the data has a high density of keys in a certain range (i.e., skew), that range should also contain more partitions. Also, the partitions should be chosen to prevent hot spots caused by access patterns. For example, if the database contains sensor measurements, using the measurement timestamp as the partition key would lead to only the last partition being used for inserts. In this case, a better option would be to use a combined key, such as prefixing the timestamp with a sensor identifier, resulting in the inserts being distributed evenly across partitions, while stile being able to perform efficient range queries for each sensor individually. [16]

Hash partitioning splits the data into partitions based on the value of a hash function. Each partition is assigned a range of hash values, and every key whose hash value is within a partition's range is stored in that partition. A good hash function returns evenly distributed values even for skewed values; these values appear to be random even for similar input values. Thus, hash partitioning prevents the problems with skew and hot spots that are inherent in key-range partitioning. However, hash partitioning does not support efficient range queries, because adjacent input keys are placed in different partitions. [16] For example, when using hash partitioning (hash sharding) in MongoDB, range queries have to be sent to all partitions [21].

Using indexes on other fields than the primary key in a partitioned topology requires additional consideration. If the partitioning scheme uses key-range or hash partitioning as described in the previous chapters, it only

supports operations on the primary keys of the records. However, it is often desirable to access the records also based on the values of some other fields, such as the city or price in an apartment rental service. There are two main methods to partitioning a database using secondary indexes: global and local partitioning. In global (also term-based) partitioning, the secondary index itself is partitioned: it is split across different partitions. Alternatively, in local (also document-based) partitioning, each partition separately maintains its own secondary indexes, which only index the records in that partition. These alternatives differ in their read and write performance. Reading from a global index is efficient: only the partition containing that index needs to be queried. On the contrary, reading from a local index requires performing a scatter/gather search, i.e., querying all partitions in parallel, and combining the results. On the other hand, writing to a global index may require updating multiple partitions in different nodes, because the indexed terms may be on different partitions. Conversely, only the local partition needs to be modified when writing to a local index. [16]

2.3 Time-series data

There were over 25 million connected IoT devices at the end of 2019 [27]. These devices create huge amounts of data, which must, in order for it to provide value, be processed, stored and queryable. Despite the devices being used widely in different applications and use cases, they have one thing in common: they all generate time-series data. Additionally, time-series data is generated by many other sources, such as DevOps, web applications and scientific measurements. This section investigates the features of time-series data: its characteristics, storage requirements, and storage options are discussed.

2.3.1 Characteristics

A time series is traditionally defined as a sequence of numbers about a certain quantity, collected over a period of time, at regular intervals. However, this traditional definition only covers a subset of what is considered time-series data today. Kulkarni [18] argues that time-series datasets have three things in common:

1. The data that arrives to the dataset is almost exclusively added as a new item.

2. The arriving data is typically in chronological order.

3. Time is a primary dimension in the data.

This definition generalizes the traditional definition of a time series considerably. The data may not be only numerical, and does not necessarily need to be collected at regular intervals. Instead, the most essential properties are the "append-only" nature of the addition of new data, and the data being time-dependent. Namiot [22] considers time-series data to simply be "a sequence of data points". Therefore, it can be concluded that the concept of time-series data refers more to the way, in which new data is collected, than to the shape or type of the data itself.

Kulkarni [18] further emphasizes that the practice of appending new data to the database is what creates value to time-series datasets. It enables measuring changes in the data: what has historically happened, what is currently occurring, and what are the trends for the future. To summarize, time-series data, at its core, is about measuring changes in the system or process of choice over time.

Time-series datasets have special characteristics that have implications on the storage systems that are used for them. First, since time-series data is often collected from numerous devices with modest to high write rates per device, the utilized storage systems need to be able to process high ingestion rates. Also, due to the "append-only" nature of time-series data, the primary method of ingestion is INSERT statements; UPDATE and DELETE statements may also be used, but they are significantly less frequent than INSERTs. This leads to large volumes of data accumulating rapidly. [31] For example, it has been argued that autonomous vehicles will collect 4TB of data per day [23]. Second, the query requirements for time-series datasets are often complex. These queries may involve JOINs against relational data, aggregations, complex predicates (e.g., varying checks in a WHERE clause) and windowed operations. As a result, the storage systems in these situations should both scale for large masses of data, and support complex query requirements.

2.3.2 Storage options

The storage of time-series data can be implemented with many different systems, the most common being SQL, NoSQL and dedicated TSDB systems. The following chapters examine each of these options, and discuss their advantages and disadvantages.

Storing time-series data in a traditional RDBMS is possible, but has some challenges. Relational databases are a well-established general-purpose solution for storing and querying large amounts of data. However, writes to

relational databases become slower as data sizes increase. The reason for this is that B-tree indexes for large tables can no longer fit in memory, causing multiple disk accesses for each insert in order to update parts of the index. This is especially problematic for time-series datasets because their size grows linearly as time elapses. Also, deleting rows from these datasets can lead to disk fragmentation, requiring performance-intensive defragmentation operations to maintain good performance. Finally, RDBMS solutions rarely support easy scaling across multiple servers, making scaling them to changing data sizes and loads more difficult. [31]

NoSQL data stores are often considered to perform better in the areas where traditional RDBMSes have problems. That is also true for time-series data: many NoSQL stores provide easy horizontal scalability and fast ingestion rates even for large data sets. [19] However, these performance improvements are often enabled by only having limited support for some features such as rich query possibilities and secondary indexing. For example, NoSQL stores may have high latency or not support at all for multi-field queries. Moreover, they lack the familiarity, reliability and extensive ecosystem of the more widely-adopted RDBMSes. [31]

Time series databases are optimized for storing and querying time-series data. They introduce efficiencies for time-series data by treating time as a first-class citizen. These include, for example, higher ingest rates, faster queries for larger data sets, and better data compression. In addition, they often provide usability improvements such as continuous queries, flexible time aggregation functions, and automated data retention policies. [18] As a result, TSDBs provide an improved experience and many built-in tools for working with time-series data. However, due to their purpose-built nature, they are only suitable for storing time-series data. Consequently, when using them in an application with other types of data, the data storage must be split across different systems. This can lead to increased complexity. Also, they are not as widely known as, for example, RDBMSes are, which can be another obstacle in using them.

Chapter 3

Environment

This chapter presents the sensor system that is analyzed in this thesis. The system is examined in the state it was in on October 1st, 2019: all measurements and specifications are from that moment. First, an overview of the system and its components is given. Then, the storage of the system is examined: its structure, data usage patterns, and queries are overviewed. Finally, the problems and the future development of the system are outlined.

3.1 System overview

The sensor system examined in this thesis gives its users the ability to monitor sensor data from buildings. The primary components are sensors that are installed in apartments, and a Java server application that enables monitoring the sensor values. There is also a new version of the web application, and a mobile application, which provide mostly the same monitoring features, but are intended for specific user groups. The system has been in the market for almost 10 years. During that time, it has steadily acquired more users. Figure 3.1 illustrates the number of new users for each year the system has been in production. The growth in customers have been around 500 per year for the first six years. However, the last two years have seen growth of over 2000 new users per year, and the count from 2019 is still missing the developments of the last two months. Currently the system has 4310 active users, i.e., users that have logged in during the last year. The system has been under active development for its entire lifespan. This organic growth has resulted in the system having many features, but also simultaneously becoming relatively complex.

The sensor system consists of IoT devices in buildings, a Java server and a MySQL database. Each building has a few (typically 1 to 3 collection

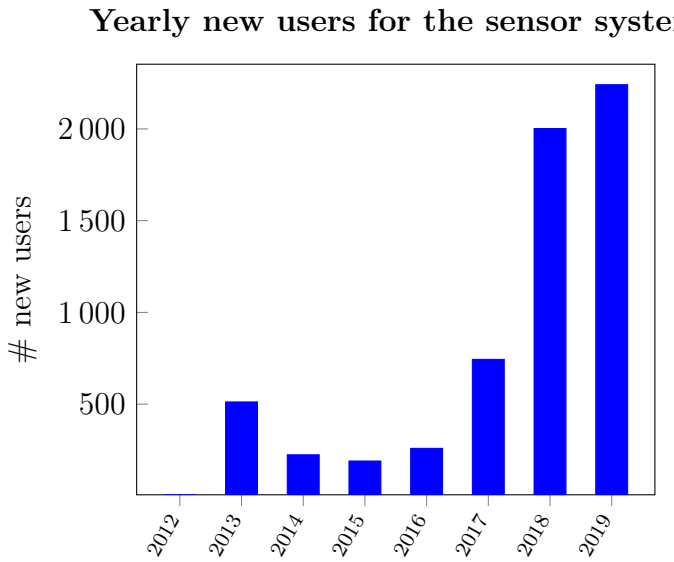

Figure 3.1: The number of new users of the system for each year.

units); each apartment has one apartment unit; and each apartment unit has a few meters (typically 2, but anything from 1 to 4 is possible). There are five generations of these devices, each with different internal specifications. However, the data that they collect and their high-level operation are identical across all generations. The reading data is sent to the Java server by either a collection unit or an apartment unit, depending on the generation. The apartment units read the data from the meters. The Java server and the database are hosted as managed services in Google Cloud Platform (GCP). Several other GCP services are also used, but they are not relevant to the work of this thesis. An overview of the architecture is provided in Figure 3.2. A more detailed look into the storage architecture is given in the next sections.

3.2 Sensor data

The most essential part of the sensor system of this thesis are the sensors and the data collected from them. These sensors are actually meters, and produce a reading once in a fixed time interval. These time intervals vary between buildings, the most common being daily, but hourly and weekly reading frequencies are also possible. Therefore, the main type of data for the sensor system are monotonically increasing meter readings from pre-defined time intervals. This section further investigates this data: its storage, ingestion

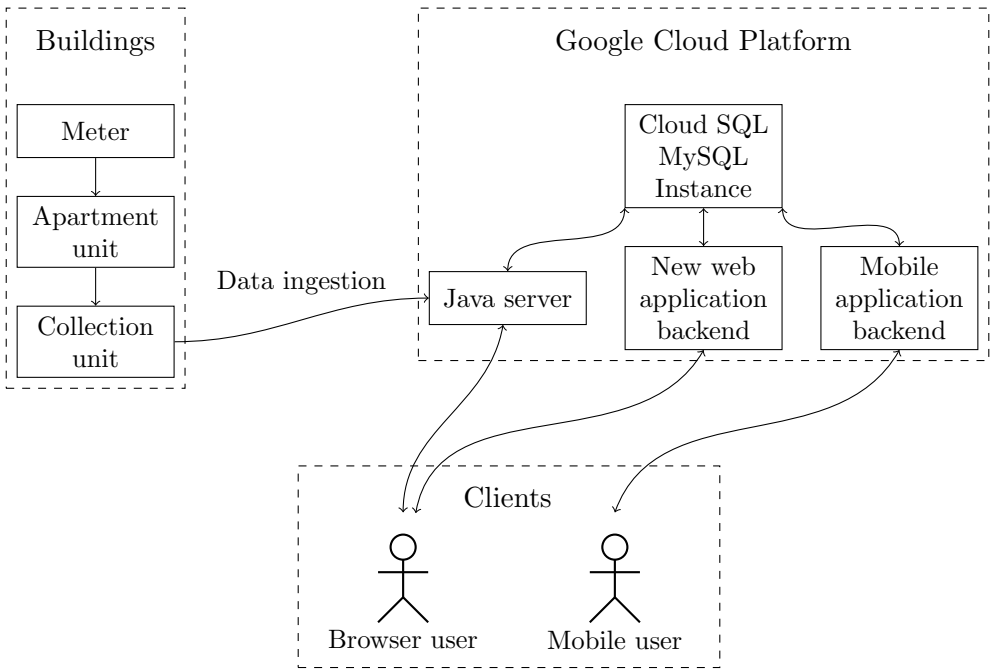

Figure 3.2: The current high-level architecture of the sensor system. The interest in this thesis is to provide a proposition for re-organizing the MySQL database and parts of the applications interacting with it to increase performance and scalability.

and usage are discussed.

3.2.1 Storage solution

The data is stored in a Cloud SQL, which is a managed MySQL service in GCP. The size of the database is 70 GiB and the specifications of the Cloud-SQL instance are: 8 CPUs, 30 GB memory and 141 GB SSD storage. The monthly cost of the instance is around $1000. The instance is configured with such high-end specs to ensure that the majority of the index of the measurement table fits into memory, in order to prevent performance problems as explained in subsection 2.3.2. There is also a GCP managed failover instance.

The database has a large schema of 123 tables. It contains sensor reading data, sensor alarm data, sensor configuration data, other application domain data and application user data. The storage of the sensor readings uses 56 GiB out of the total 70 GiB storage. The data is stored in a table named `MeterReading`. It has 239 725 481 rows, requires 18 GiB of storage, and has

Property name	Type
`id (primary key)`	bigint(20)
`creation_time`	datetime
`importId`	int(11)
`modification_time`	datetime
`readingDate`	datetime
`value`	double
`apartment_id` (foreign key)	bigint(20)
`meter_id` (foreign key)	bigint(20)
Index 1	`id`
Index 2	(`apartment_id`, `readingDate`)
Index 3	(`readingDate`, `meter_id`)
Index 4	`apartment_id`
Index 5	`meter_id`

Table 3.1: The schema of MeterReading table.

37 GiB of indexes.

The schema of the MeterReading table is shown in Table 3.1. `id` is an auto-generated auto-incremented column. `readingDate` and `value` are obtained from the sensors. `creation_time`, `importId` and `modification_time` are auto-generated by the Hibernate database inheritance hierarchy in the Java server. There are two foreign keys: the apartment the sensor is in (`apartment_id`), and the meter the sensor is part of (`meter_id`). Both foreign keys are indexed. Additionally, there are three more indexes: primary key, (apartment_id, readingDate) and (readingDate, meter_id).

3.2.2 Ingestion

The data is currently ingested mainly from 0 to 2 o'clock at night: 90% of the readings in the first week of October were ingested in that timespan, and the ingestion times are not changed for existing meters except in rare cases. This corresponds to an average of 230 000 readings being ingested daily from 0 to 2 o'clock. However, there is also a small but constant amount of collection units that send data hourly: a bit over 1000 readings were received each hour each day during the observation period. These are meters that are read hourly, or meters for which the nightly reading was not successful, and reading is therefore retried. The division of ingested readings is presented in Figure 3.3.

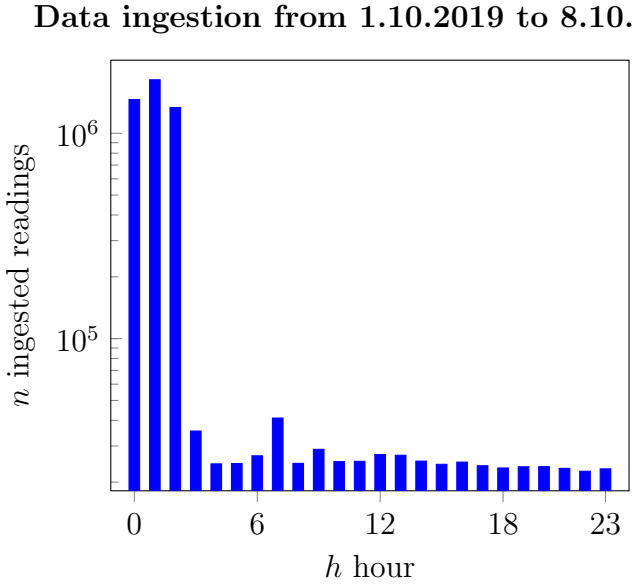

Figure 3.3: The total number of ingested readings for each hour in one week, from 1.10.2019 to 8.10.2019.

In total, 1 809 313 readings were ingested during the first week of October 2019, which corresponds to 258 500 readings each day. Since one row of reading data is approximately 82 bytes, $1\,809\,313 * 82 / 1024 / 1024 / 7 = 20$MiB of sensor data is ingested each day. The increase in storage requirement size is naturally higher, as the indexes also grow when inserting data.

3.2.3 Usage

The data of the sensor system as a whole is used by the Java server, the new web application backend, and the mobile application backend, as demonstrated in Figure 3.2. The SQL queries are hand-written in the new web application and the mobile application; the Java server uses a mixture of hand-written SQL, and queries generated by the Hibernate ORM. The usage can be characterized as Online Transaction Processing (OLTP): the data is fetched and updated with short-lived, transactional queries, and all data points are significant. Table 3.2 summarizes the query operation counts for the whole database in a span of one week. `SELECT` and `INSERT` queries are most common, but `UPDATE` and `DELETE` queries are also used. The relatively high number of `INSERT`s is explained by the ingestion of meter readings, which produces almost 2 000 000 `INSERT`s weekly.

The sensor data stored in the `MeterReading` table is the most essential

Type of query	Count in one week
`SELECT`	8 994 676
`INSERT`	2 647 920
`UPDATE`	234 785
`DELETE`	5997

Table 3.2: Query frequencies for the entire sensor system database.

Purpose	Group	Count
Apartment readings table and chart	Client	13 097
Meter reading ingestion	Device	4129
Reading report for a building	Client	2588
Collection unit readings after a time	Device	2122
Buildings with missing readings	Device	31
Meter reading deletion	Admin	< 10
Buildings with high-reading sensors	Admin	< 10

Table 3.3: Query counts for different `MeterReading` table queries.

data in the old and new web applications, and the mobile application. Its usage can be divided into three groups:

1. Client usage. Read queries for showing meter readings in the applications.

2. Admin usage. Various types of queries for working with raw meter reading data.

3. Device usage. Insert and read queries for adding new meter readings.

Table 3.3 provides an overview of query counts for some queries in each of these groups in one day. It is not an exhaustive list, but provides good examples on the relative counts of queries from different groups. The data in it was obtained by running a MySQL low-level execution monitoring tool called performance schema. The table illustrates that client-facing and device-facing queries are the most frequent groups of operations. Most admin queries are executed very infrequently, some not even daily. The following sections analyze each of these query groups in detail.

3.2.3.1 Client

Client-facing usage is the most frequent and primary method of using meter readings. It consists of SELECT queries for showing meter data for the users in the old, new and mobile applications. This data is displayed in a table or as a time-series chart. However, the primary information in the displayed data is not the raw reading data from the database. Instead, differences of meter readings are the most essential piece of information for end-users. In the terminology of the sensor system, these are called consumptions. These consumptions are conceptually simple to calculate by subtracting a reading from the reading after it: $diff = \Delta r = r_2 - r_1$. However, it is often desirable to show consumptions for a different frequency than the frequency the meters were originally read in. For example, the data from meters that are read daily may be shown in a chart displaying monthly consumptions. This aggregation combined with the handling of possible missing readings causes the conceptually simple consumption calculations to become rather complex. Also, since only the raw meter reading data is stored in the database, these calculations need to be performed separately after each database query.

Figure 3.4 illustrates the difference between the stored data and the displayed data. As can be seen, the stored readings are missing a reading for the last day of October. This needs to be handled somehow; in this case, the consumption for that day is set to zero when daily consumptions are calculated. Other than this problem, the calculation of daily consumptions from the stored readings is rather simple in this case. However, the calculation of monthly consumptions requires summation for the readings of each month, and is therefore more complicated. Also, when calculating the consumptions of a building, the consumptions from meters of the same type need to be added together.

Stored readings

date	value
2020-10-29	2900
2020-10-30	3225
2020-11-01	3323
2020-11-02	3440
2020-11-03	3600

Daily consumptions

date	value
2020-10-30	325
2020-10-31	0
2020-11-01	98
2020-11-02	117
2020-11-03	160

Monthly consumptions

date	value
2020-10-31	325
2020-11-03	375

Figure 3.4: Meter reading data in different formats.

Client-facing queries often fetch the readings of a group of meters for some

timespan. The group of meters consists either of all meters of an apartment or all meters of a building. The timespan varies from one day to several years: typically the three latest years (the current year and the previous two full years) are fetched. Sometimes, even older readings may be queried, but such cases are rare, and only used with the help of admin users. Even though the fundamental purpose of all client-facing queries is essentially the same, i.e., fetching meter consumption data, there are many different queries for this purpose. The differences in these queries exist because the reading data is processed in varying ways. On the one hand, both the mobile backend and new web application backend only require one or two queries to fetch the readings of a building or an apartment. On the other hand, the Java server has more complex data access patterns, often requiring numerous queries to display reading data. For example, displaying the reading charts of a single apartment for the last three years requires over 150 queries. Both cases require application-level processing after the queries to obtain consumptions instead of only raw meter readings. However, this processing is done entirely with the dataset fetched with a few queries in the new web and mobile applications; the Java application executes additional utility queries for this post-processing, resulting in the high number of queries mentioned previously.

Listing 3.1 demonstrates a query for fetching the meter readings of a building, used in the new web application backend. The query does not require complex operations: it only uses a join, simple filtering and ordering by timestamp. That query alone is sufficient for creating all meter-related charts and tables. Listing 3.2, on the other hand, features a query for fetching the readings of an apartment, used in the Java server application. It is considerably more complex, featuring several subqueries and groupings. Moreover, it is also used when displaying the readings of a building: it is executed once for each apartment, and the results are combined. This is a good example on the complexity of the data access patterns of the Java server: the reading data of a building could be fetched with a single query, but instead n queries are used, one for each apartment. For a large building of 200 apartments this results in 200 queries being executed.

```
SELECT mr.readingDate, mr.value, mr.apartment_id
FROM Apartment Ap
INNER JOIN MeterReading Mr
    ON Mr.apartment_id = Ap.id
WHERE
    Ap.building_id = %building_id% AND
    Mr.readingDate >= %oldest_date%;
```

```
ORDER BY
    Mr.readingDate ASC;
```

Listing 3.1: SQL query for fetching the readings of a building, used in the new web application.

```
SELECT
    perDay.meter_id,
    MAX(perDay.value),
    perDay.readingDate
FROM (
    SELECT R.meter_id, R.value, R.readingDate
    FROM MeterReading R
    JOIN Meter N ON R.meter_id = N.id
    JOIN (
        SELECT
            MAX(S.readingDate) AS maxReadingDate,
            M.id AS maxMeterId
        FROM MeterReading S, Meter M
        WHERE
            S.apartment_id = %apartmentId% AND
            M.id = S.meter_id AND
            S.readingDate BETWEEN %start% AND %end%
        GROUP BY (S.readingDate), M.id
    ) AS T1 ON
        R.readingDate = maxReadingDate AND
        R.apartment_id = %apartmentId% AND
        N.id = maxMeterId
) AS perDay
JOIN Meter M
    ON perDay.meter_id = M.id
GROUP BY
    perDay.meter_id,
    perDay.readingDate
ORDER BY
    perDay.readingDate,
    perDay.meter_id;
```

Listing 3.2: Unnecessarily complex SQL query for fetching the readings of an apartment, used in the old Java application.

3.2.3.2 Admin

Admin-facing queries are crucial in maintaining the correct operation of the sensor system. For example, erroneous meter devices occasionally send invalid reading data, which is saved to the database, and needs to be manually corrected or removed. As a result, `UPDATE` and `DELETE` statements on the `MeterReading` table need to be executed. Also, in some cases it may be desirable to insert new readings manually, resulting in `INSERT` statements outside the normal reading ingestion flow. However, these situations are encountered rarely, resulting in these queries being executed infrequently.

There are also some admin-only features in the web applications that require access to meter reading data. For example, the meter readings have a certain maximum value: when it is reached, the meter will stop working. Therefore, admin users need to monitor meters whose reading value is approaching the maximum value, in order to replace them before the maximum value is reached. This monitoring uses a complex query that fetches all the meters that have values over a given threshold. Because the query is necessarily not filtered to the readings of a building or an apartment like other `MeterReading` queries are, it needs to access the readings of all buildings, making it considerably slower. However, this query is used infrequently and only by a small number a users.

3.2.3.3 Device

Device-facing queries are used constantly to ingest new readings. Besides the normal ingestion flow, which is presented in section 3.2.2, there are other types of queries originating from the devices. For example, there are queries that are run daily for finding the buildings that are missing readings for the current day. This data is then used to re-trigger the ingestion for meters with missing readings. Listing 3.3 presents an example of a query for fetching buildings with missing readings for the current day. The query requires using a subquery for finding the collection units that have readings for the current day, which are then used to find the buildings that are missing the readings. There are also queries for checking if there are readings for a single collection unit after a given time, which are then used to determine the time of the next reading for these units. These queries are similar to the subquery in Listing 3.3.

```
SELECT B.id
FROM Building B
INNER JOIN CollectionUnit CU
    ON B.id = CU.building_id
```

```
WHERE
    CU.id NOT IN (
        SELECT DISTINCT CU.id
        FROM CollectionUnit CU
        INNER JOIN Meter M
            ON M.collection_unit_id = CU.id
        INNER JOIN MeterReading MR
            ON MR.meter_id = M.id
        WHERE
            MR.readingDate > %cur_date%
    );
```

Listing 3.3: SQL query for fetching the buildings with missing readings for the current day.

3.2.4 Problems

The most critical problem in the data storage of the sensor system is the uncertainty of its performance. This uncertainty is caused by a history of performance problems, the most severe of which are related to data ingestion. Occasionally, when new reading data is inserted to the database, the database becomes unable to process any other queries. More specifically, some `INSERT` statement takes an extremely long time to complete. This time can be multiple hours, during which no other readings can be ingested and no read queries executed. As a result, the whole system is effectively down for the users and reading data is lost. The specific reason for this long-running `INSERT` statement is not fully known. However, it is assumed to be related to index balancing: if an insert causes an extensive rebalancing of the indexes and the indexes do not fit in memory, this balancing may take a long time, which is naturally unacceptable for a customer-facing sensor system. This problem was temporarily fixed by switching to a database instance with more RAM, but it is not a desirable long-term solution due to the increased costs.

Another cause for the uncertainty in the performance of the sensor system are slow reading queries. There are some large buildings of more than 200 meters, for which the query for the last three years of meter data is unacceptably slow. This is due to a large number of readings that have to first be fetched from the database and then processed into consumptions for display in the user interface. This execution takes at least a few seconds, and in the worst case tens of seconds.

The difference between the stored meter data and the data displayed to the users is another storage problem. As described in Subsection 3.2.3, the

stored data is raw meter readings, and the most essential displayed data is consumptions, i.e., differences of these raw readings. The consumption information needs to be calculated from the queried raw data for each database query. This requires processing the raw measurement data in application-level code, which is problematic because this processing is implemented separately in many codebases (Java server, new web application and mobile application), resulting in slightly different operation for edge cases, and different consumption values depending on the application that is used. Since this is the most essential data provided by the system, it should be consistent.

There are also many smaller data usage related problems in the Java application. First, it contains query patterns that are generally considered to be bad practices, such as performing queries in a loop. For example, fetching the readings of a building is done by iterating over the apartments of the building and executing a reading query for each apartment separately, when a single query could be used for the entire building. This results in considerably slower execution times than the optimal single-query solution. Second, the Java server contains many redundant queries. For instance, when displaying the readings in a chart, separate queries are executed for finding the number of readings per date and the next reading after each date, when these could be calculated from the reading data that is already in memory, which leads to increased runtimes. Finally, since the Java application has grown organically over its entire lifespan of almost 10 years, it has become complex and difficult to understand as a whole. There are hundreds of queries that have been added to the system as it has grown and new requirements have emerged. As a result, it is not clear what these queries fetch, how the fetched data is used, and if the data schema is suitable for them. Furthermore, it is also not clear if they are written smartly to provide good performance.

3.3 Future requirements

The most significant future requirements for the sensor system are related to its data ingestion. As mentioned in Subsection 3.2.2, the amount of ingested data is constantly growing. Currently the growth rate is about 250 000 readings a day, 230 000 of which are ingested within a two-hour timespan. This steady growth is further accelerated by new meters being added to new buildings: in the last year, about 25 000 of new sensors were added to the system. This amount has been similar also in the previous years. As a result, the amount of readings ingested each day has historically increased by about 25 000 every year.

Figure 3.5 illustrates the history and prediction of both meter count and

reading ingestion. Historically, meters in daily reading have accounted for the majority of the ingested readings. However, in the future, the majority of new meters are expected to be read hourly. There will also be meters that are read daily, but they are mostly meters that were added before 2020. Moreover, the number of new meters added to the system is expected to start growing more rapidly than before. Therefore, the number of ingested readings is expected to start to significantly increase in comparison to historical growth. For this reason, the size of the `MeterReading` table will start to grow substantially faster than before.

The increase of meters that are read hourly also causes further uncertainty in the performance of the operations on the meter reading data. As mentioned in section 3.2.4, some of the large buildings that are read daily already have unacceptably slow reading queries. In a few years, there will be buildings of the same size, but with 24 times the data, since they have been read hourly. It can be expected that their performance will be noticeably worse than that of the already slow daily large buildings. Moreover, it is possible that the performance of apartment reading queries will also start to degrade as data is accumulated, and hourly read apartments are queried. Finally, it is not certain if the MySQL database can ingest the growing number of hourly read meters fast enough: if all hourly meters send their readings on the hour, they should be ingested within a few minutes to provide relatively real-time data to the users.

There are also several other requirements for the system in the future. First, the costs of running and maintaining the system should be moderate. As a result, it is desirable to use technical solutions that are competitively priced. Second, the worst-case performance of the system should be acceptable: no client-facing query should take more than a few seconds to execute. Finally, situations where the whole system becomes unresponsive as described in Subsection 3.2.4 should not occur.

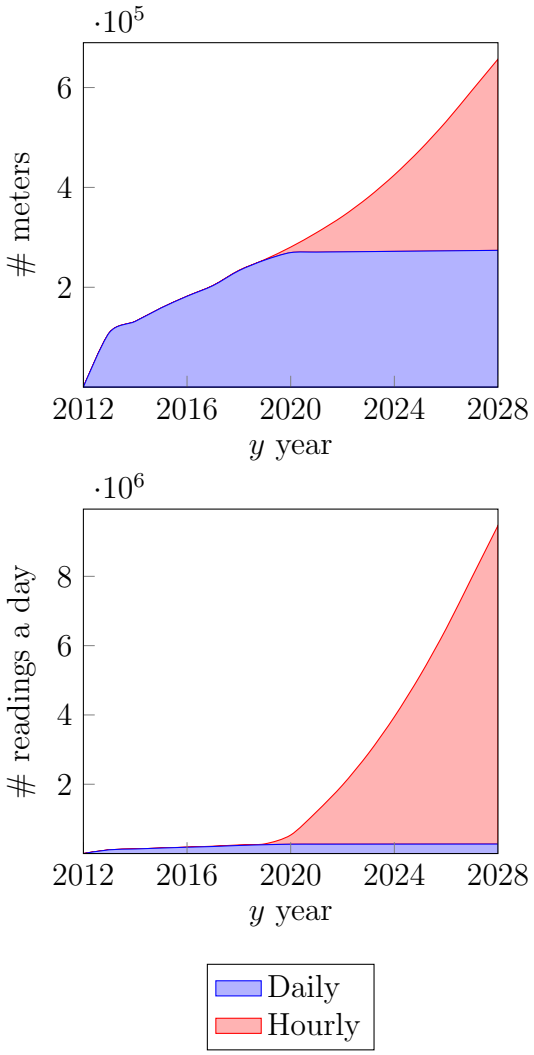

Figure 3.5: History and prediction of the number of meters and ingested readings, separated by reading frequency.

Chapter 4

Experimental

As mentioned in section 3.3, the feasibility of the current data storage solution in the future is uncertain. This uncertainty is mainly due to the volatile performance of the current sensor system combined with increasing amounts of data in the future. For this reason, this thesis performs tests for the data storage of the sensor system. The purpose of these tests is to get an estimate on the lifetime of the current storage system. Also, a few other, more time-series-optimized storage solutions are tested, to obtain knowledge of their relative performance for storing the sensor data of the system.

The testing is done in a way that resembles the existing application and its data usage patterns as closely as possible. For this reason, the tests are performed with application-specific test data and test queries, in an environment that is similar to that of the production environment of the application. Essentially, the tests run a predefined set of test cases for different database sizes, corresponding to the sizes the database is estimated to reach in the future. These tests are performed on the existing storage solution and two other solutions.

This chapter first examines the test cases, test data generation and test environment used in testing the sensor system. Then, the storage solutions the tests are performed on are overviewed. Finally, the obtained test results are presented.

4.1 Test cases

The tests that are run on the different storage solutions consists of two test cases: query tests and ingestion tests. These two cases cover the most essential data usage patterns of the sensor system. Query tests reflect the ways the clients of the application use its data, as presented in section 3.2.3.1;

ingestion tests test the path of data ingestion that is examined in section 3.2.2. These test cases are further examined in the following sections.

These two tests do not encompass all of the data usage patterns of the application. For example, admin usage patterns such as `UPDATEs` and `DELETEs` are not tested. The reason only these two test cases were selected is that they constitute the majority of the sensor data usage, therefore largely defining the performance of the system, as shown in Table 3.3. Also, their results and performance are perceived directly by the clients of the application, adding to their importance. The data usage patterns that are left out of the tests are run on the background, or infrequently by admin users. For this reason, their performance is not as critical.

4.1.1 Query tests

Query tests are designed to encompass the way the clients of the sensor system use the sensor data. They test the time it takes to fetch the last three years (the current + two previous full years) of readings and consumptions for either an apartment or a building. More precisely, the end result is a list of all timestamps in the reading frequency of the meter of the last three years, each containing a reading and a consumption (i.e., the difference to the previous reading) for all available meter types. Currently there are two types of sensors, resulting in each timestamp having two readings and consumptions. This format is used because it can be displayed in a table or a time-series chart without additional processing. Also, is can be processed to a smaller frequency rapidly. To summarize, the data obtained from these queries can be used as is or with little processing in the most of the user interface elements.

There are separate query tests for different sizes of buildings, and for buildings and apartments with different reading frequencies. The test cases are:

- Hourly large (300 meters) building
- Hourly medium (150 meters) building
- Hourly small (50 meters) building
- Daily large (300 meters) building
- Daily medium (150 meters) building
- Daily small (50 meters) building

- Hourly apartment (2 meters)
- Daily apartment (2 meters)

The reason for this separation is that the size of the building and the reading frequency greatly affect the amount of data that is fetched and processed. As a result, the execution times are also expected to vary between these cases. For this reason, it is best to separate the tests. For example, a building with 300 meters read hourly has $3 * 365 * 24 * 300 = 7884000$ data points for three years, whereas a building with 50 meters read daily has $3 * 365 * 50 = 54750$ points.

These query tests are executed once on an database with no additional load. They are executed sequentially, one at a time, to obtain an estimate of their best-case performance. Then, an ingestion load is applied to the database, and the query tests are executed again. This is done to evaluate their performance when the database is also receiving other queries. The ingestion load consists of repeatedly inserting the data of all hourly meters of the system into the database. The insertion is done similarly as described in section 4.1.2.

In total, these query tests are executed 800 times. Each test case is executed 100 times: 50 of these executions come from ten buildings/apartments, the readings of which are fetched 5 times each; the remaining 50 come from 50 distinct buildings/apartments. The same buildings and apartments are used for all tested storage solutions, to ensure the tests are comparable. The reason for fetching the readings of some buildings/apartments multiple times is to simulate actual usage of the service: some buildings/apartments are queried more frequently than others. These repetitions will most likely benefit more from the caching of the database, as is the case for the actual system also.

4.1.2 Ingestion tests

Ingestion tests are designed to test the performance of the ingestion of meter reading data into the system. More precisely, the ingestion is tested after the moment the readings of a collection unit are received by the application running in the cloud. The tests measure the time it takes to insert the readings of a single collection unit into the `MeterReading` table, and update other tables containing denormalized data about the latest reading of a meter. Also, the total time it takes to insert all the readings received during the timespan of an hour is measured.

The ingestion tests are executed by doing concurrent ingestion for the readings of multiple collection units. Ten concurrent threads are run, each

Test #	Time	Hourly devices	Daily devices
1	6.4.2020	1126	257 988
2	1.1.2021	11 050	269 450
3	1.1.2023	71 050	270 950
4	1.1.2025	153 050	271 950
5	1.1.2028	321 050	273 450

Table 4.1: The moments of time and estimated numbers of devices used in testing the storage solutions.

sequentially ingesting new readings. The readings of all hourly meters and the readings of 1/4th of daily meters are ingested. This ingestion is repeated four times, to more accurately estimate the execution times. This amount of readings (all hourly + 1/4th of daily) is a worst-case estimate of the maximum amount of readings that may be ingested into the system during one hour.

4.2 Test data

The tests are based on a dataset obtained from the production database of the system on 6.4.2020. This dataset is then extrapolated with new buildings, devices and their meter readings. The same base data and generated data are used on all tested storage solutions. The generated data is based on the estimate of the amount of new meters presented in Figure 3.5. Table 4.1 presents the five data points the tests are executed on.

Based on these five test points, four test datasets were generated, since the first point is the existing dataset. The generation is based on the amount of new meters required in the next test point. For example, when generating data for point 2, $11050 - 1126 = 9924$ new hourly and $269450 - 257988 = 11462$ daily new meters are required. Then, a sufficient amount of different sizes of buildings are generated, to contain the desired amount of meters. These buildings and meters are created on evenly spaced intervals between the time data points. Finally, meter readings for the created meters are generated. New reading values are generated randomly, and new reading times are also given some random offset. This way, the generated data closely resembles the actual data that is accumulated in the sensor system over time.

4.3 Test environment

The tests are executed in GCP, in an environment that is similar to that of the production sensor system. The instance parameters, locations and cloud services, and the technologies for the test code are the same as in the production sensor system. The tests cases are run in App Engine, which is a serverless web application platform, and also used for the mobile, new, and legacy applications of the sensor system [10]. The storage solutions are hosted on CloudSQL if possible, and on virtual machines in Compute Engine if CloudSQL is not supported for a given solution [12]. The App Engine instances and databases were deployed on the same `europe-west-3a` zone, and their communication is done via external network.

The instances in App Engine are of type B4, i.e., they have 1024 MBs of memory and a 2.4 GHz CPU [3]. They are deployed in the Java 8 standard environment as WAR files. The CloudSQL instances are of type `db-n1-highmem2`, i.e., they have 13 GBs of memory and 2 CPUs; for the last data point in year 2028 an instance of type `db-n1-highmem4` with 26 GBs of memory and 4 CPUs was used, because of the large amounts of data [11].

The tests are written in Clojure. The database is interfaced with next.jdbc, which is a Clojure wrapper for the JDBC database connector. HikariCP is used for database connection pooling. A connection pool of 1 is used for the sequential query tests, and a pool of ten is used for the concurrent ingestion tests. The execution time of the tests is measesured with Clojure's `time` function.

4.4 Storage solutions

The tests are executed on three different storage solutions: an intermediate MySQL solution, an optimized MySQL solution, and a PostgreSQL + TimescaleDB solution. The intermediate MySQL solution is the version that is currently running in production; the optimized MySQL solution is an improved version of the intermediate solution, optimized for good performance with meter reading data; the PostgreSQL + TimescaleDB solution is an alternative storage solution, which is purpose-built for time-series data. These solutions were selected because migrating to them from the existing solution is straightforward, and they were expected to produce sufficient performance gains. A NoSQL solution was not tested because the amounts of data in the system are not too large for a RDBMS (hundreds of GBs), and the migration to such a solution would require considerably more work [8]. The following sections examine each of these solutions in further detail.

Time	Partition interval
Until 2023	3 months
Until 2024	2 months
Until 2025	1 months
Until 2028	2 weeks

Table 4.2: The partitioning intervals for the intermediate and optimized MySQL solutions.

4.4.1 Intermediate MySQL

The intermediate MySQL solution is an improved version of the storage solution presented in section 3.2.1, and is currently running in production. As described in section 3.2.4, the previous version had significant problems in ingestion, sometimes even causing unresponsiveness of the application. For this reason, improvements to the solution were urgently done. It is presumed that after these improvements, the application will not anymore exhibit unresponsiveness in data ingestion. However, the other problems presented in section 3.2.4 will still be present, since these improvements only addressed the must urgent ingestion problem.

There are several improvements in the intermediate MySQL solution. First, the `MeterReading` table was partitioned on the `readingDate` column. This makes the indexes of the table smaller, since they are local to the partitions. Also, since MySQL forbids foreign keys in partitioned tables, the foreign keys of `MeterReading` were removed. The partitioning intervals are presented in Table 4.2. They are selected so that the size of the last partition requires at most 50% of the memory of the instance. Second, the schema of the `MeterReading` table was optimized: `id`, `importId` and `creation_time` columns were removed; `(id)`, `(apartmend_id)` and `(meter_id)` indexes were dropped; and the `(readingDate, meter_id)` index was changed to `(meter_id, readingDate)`, which was also made the primary key of the table. All the dropped indexes and columns were redundant. The new schema is shown in Table 4.3. Finally, a `latestReadingDate` column was added to the `Meter` table, allowing the latest reading of a meter to be fetched quickly.

4.4.2 Optimized MySQL

The optimized MySQL solution is based on the intermediate MySQL solution, with a few optimizations for increasing the performance of the opera-

Property name	Type
modification_time	datetime
readingDate	datetime
value	double
apartment_id	bigint(20)
meter_id	bigint(20)
Index 1 (Primary key)	(meter_id, readingDate)
Index 2	(readingDate)
Index 3	(apartment_id, readingDate)

Table 4.3: The schema of MeterReading table in the intermediate MySQL solution.

tions on meter data. There are four optimizations that separate the optimized version from the intermediate version: pre-calculation of building consumption data, batch inserts for hourly readings, using a trigger to update derived data and the removal of redundant data from the `MeterReading` table. These are examined in more detail in the following paragraphs.

The most important optimization is the pre-calculation of building consumption data. It addresses the slowness of fetching the readings of a building, which is explained in section 3.2.4. After the readings for the hour are ingested, a batch job is executed, fetching a few latest readings for all buildings, calculating consumptions from these readings, and inserting or updating these consumptions for that building if they differ from the last batch execution. These consumptions are stored in a separate table, `BuildingConsumptions`, which is partitioned into chunks of three months. Now, when fetching the consumptions of a building, as is done in the query tests, the `BuildingConsumptions` table can be queried without any additional processing; previously reading data had to be fetched from the `MeterReading` table and post-processed to obtain consumptions.

Batch inserts for hourly readings improve the performance of inserting hourly reading data to the database. The readings of new hourly devices are sent by apartment units. Because each apartment unit often only has two meters, just two readings are added in one `INSERT`. Batch inserts combine the readings of many apartment units, and ingest them in one `INSERT` statement. For these tests, the readings of 500 apartment units (i.e., 1000 meter readings), are inserted per statement. This drastically reduces the amount of queries the database has to handle.

The optimized MySQL solution utilizes an trigger on `INSERT`s to the

Property name	Type
modification_time	datetime
readingDate	datetime
value	double
meter_id	bigint(20)
Index 1 (Primary key)	(meter_id, readingDate)

Table 4.4: The schema of MeterReading table in the optimized MySQL solution.

MeterReading table to update derived reading data. Without this trigger, the application first has to do one INSERT to the MeterReading table, and then a few separate UPDATEs to tables with derived data that needs to be updated. By using a trigger to do these updates when a row is inserted to the MeterReading table, these round-trips for the UPDATE statements are eliminated.

The schema in the optimized version was further optimized from that of the improved MySQL solution. The (readingDate) index and the (apartment_id, readingDate) indexes were dropped, and the apartment_id column was removed. The data in these was redundant and not required by the queries. The reduced schema is presented in Table 4.4. The deletion of this column and indexes reduce the size of the MeterReading table by about 40%.

4.4.3 PostgreSQL and TimescaleDB

The third tested storage solution is PostgreSQL with the TimescaleDB extension. TimescaleDB is a PostgreSQL extension for working with time-series data. It provides efficient ingestion, automated time-interval partitioning, high-performing queries, data compression and retention, and time-series utilities and analytics. These features are available on so-called hypertables, which have a full SQL interface, but utilize the improvements TimescaleDB offers [31]. This solution was deployed on Google Compute Engine, because the CloudSQL environment does not currently support the TimescaleDB extension.

With this solution, the entire dataset of the sensor system was migrated from MySQL to PostgreSQL. The solution is based on the optimized MySQL solution, and uses the same base dataset and optimizations as it. The MeterReading and BuildingConsumptions tables were converted to hyper-

Time	Partition interval
Until 2021	3 months
Until 2023	2 weeks
Until 2028	1 week

Table 4.5: The partitioning intervals for the intermediate and optimized MySQL solutions.

tables and partitioned by time. TimescaleDB recommends configuring the partitioning interval so that the last partitions of all hypertables consume at most 1/4 of the memory of the instance. For this reason, the partitioning of the `MeterReading` table was more frequent than in MySQL; the partitioning intervals are presented in Table 4.5.

The performance of data ingestion in this solution was considerably slower than that of the optimized MySQL solution, when the same data and queries were executed on both databases. This was because the serializable isolation level of PostgreSQL caused reading `INSERT` queries to be aborted due to a serialization error in the trigger of the `MeterReading` table. As a result, these `INSERT` statements had to be retried tens of times. This slowness was only fixed by loosening the isolation level for the PostgreSQL solution from serializable to read committed.

The compression provided by TimescaleDB was utilized in this solution in both the `MeterReading` and `BuildingConsumptions` tables. TimescaleDB enables compressing partitions of data that are older than a given threshold. According to them, compression often significantly decreases the required storage space, and may also improve query performance, because more data can be kept in memory. For the dataset of the sensor system of this thesis, the size of the `MeterReading` table decreased to 15% of its original size, and `BuildingConsumptions` to 7% of its original size. Also, the query performance was consistently 10% to 40% better than without compression.

Some other improvements enabled by using TimescaleDB and PostgreSQL were also tested. PostgreSQL has a concept of statement-level triggers, which enables accessing all the rows of an `INSERT` statement in a single execution of a trigger function. This would most likely increase ingestion performance, because the current row-level trigger must be executed separately for each row, i.e., 1000 times when doing an `INSERT` with 1000 rows. However, TimescaleDB does not yet support statement-level triggers on hypertables. For this reason, statement-level triggers were not used in the solution. Another possible improvement that was not used in the tested solution was to utilize the extensive aggregation functions provided by TimescaleDB to do

the processing from readings to consumptions in a database query. However, this query was not faster than a separate post-processing step in application-level code, and was therefore not used.

4.5 Results

This section examines the results obtained from executing the two test cases, query and ingestion, on the three storage solutions, with the data points 1 to 5. First, the methods that are used in analyzing and visualizing the results are examined. Then, the sizes of the storage solutions for the different data points are highlighted. Finally, the test results are presented.

There are no results for the intermediate solution for datasets #4 and #5 because the solution performed unfeasibly slow for dataset #3. For this reason, this solution was excluded from the tests for the last two datasets, and there are is no data in any of the result plots for these datasets for the intermediate solution.

4.5.1 Analysis

As mentioned in section 3.3, the worst-case performance of operations on the sensor data should be acceptable. For this reason, the test results are visualized as box plots: the lower and upper whisker of a plot represent the minimum and maximum values; the lower and upper part of the box represent the 10th and 90th percentiles; and the middle of the box represents the median. This visualization highlights the distribution of the data, and also presents the best-case and worst-case performance. For these reasons, it was preferred over showing only average values, which would hide the actual distribution of the data.

4.5.2 Storage solution sizes

Table 4.1 presents the sizes of different storage solutions for the test datasets. The size of the optimized MySQL solution is consistently about 40% smaller than the intermediate solution. The size of the uncompressed PostgreSQL + TimescaleDB dataset is consistently about 25% larger than the optimized MySQL dataset. However, the compression of `MeterReading` and `BuildingConsumptions` tables significantly decreases the amount of required storage. After compression, the dataset requires about 20% of the space of the uncompressed dataset for the larger dataset, where the sizes of these compressed tables largely dominate the storage requirements.

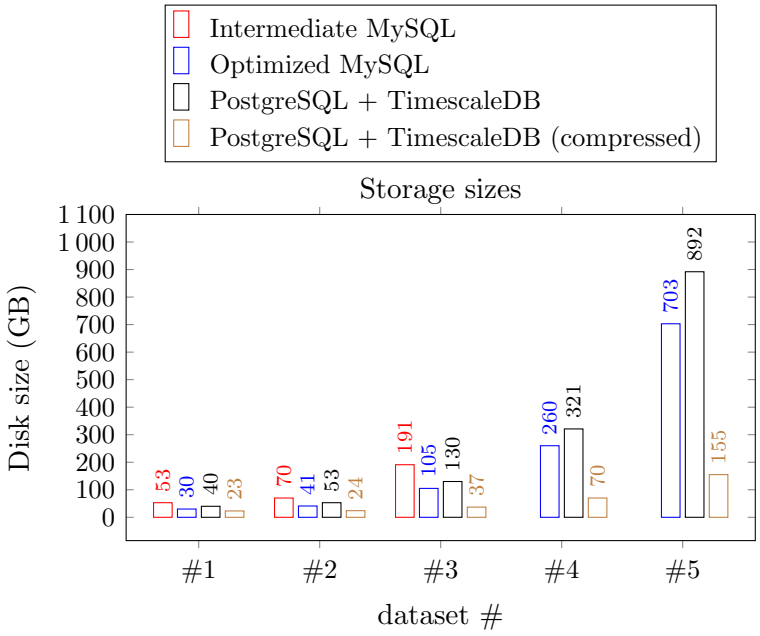

Figure 4.1: The sizes of the tested storage solutions at estimated future times.

4.5.3 Building queries

Figure 4.2 presents the results of building query tests on a database without load. There are no results for the hourly queries for dataset #1 because that dataset does not have enough hourly meters to do repeatable tests. As Table 4.1 demonstrates, there are only about 1000 devices in dataset #1. Also, since the majority of hourly devices are created after 2020.04.06, they have at most eight months of data when testing dataset #2, and two years and eight months of data for dataset #3. Only when testing datasets #4 and #5 the meters have full three years of data to be fetched by the query tests.

All plots demonstrate that building queries on the intermediate MySQL are considerable slower than on the other two solutions. The hourly queries are unfeasibly slow, especially on dataset #3: fetching the consumptions of a large hourly building takes more than 20 seconds even in the best case, and also the median of a small hourly building is over ten seconds. The queries for large and medium buildings are also slow for dataset #2: both may take over ten seconds. The daily queries for the intermediate solution are also slower that the other solutions, but their performance is considerably better than that of the hourly queries. For example, daily queries for dataset #3

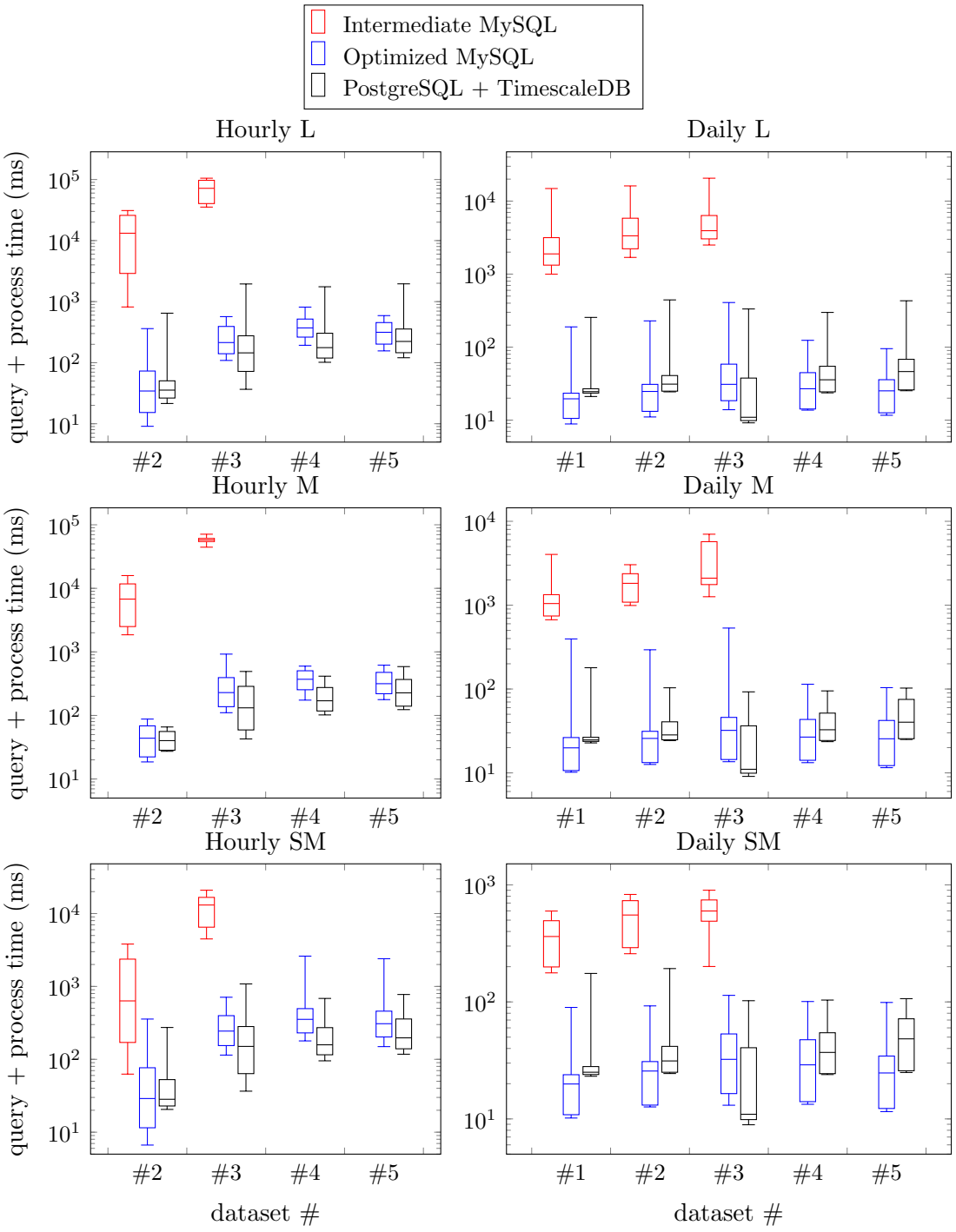

Figure 4.2: Results of query tests without load.

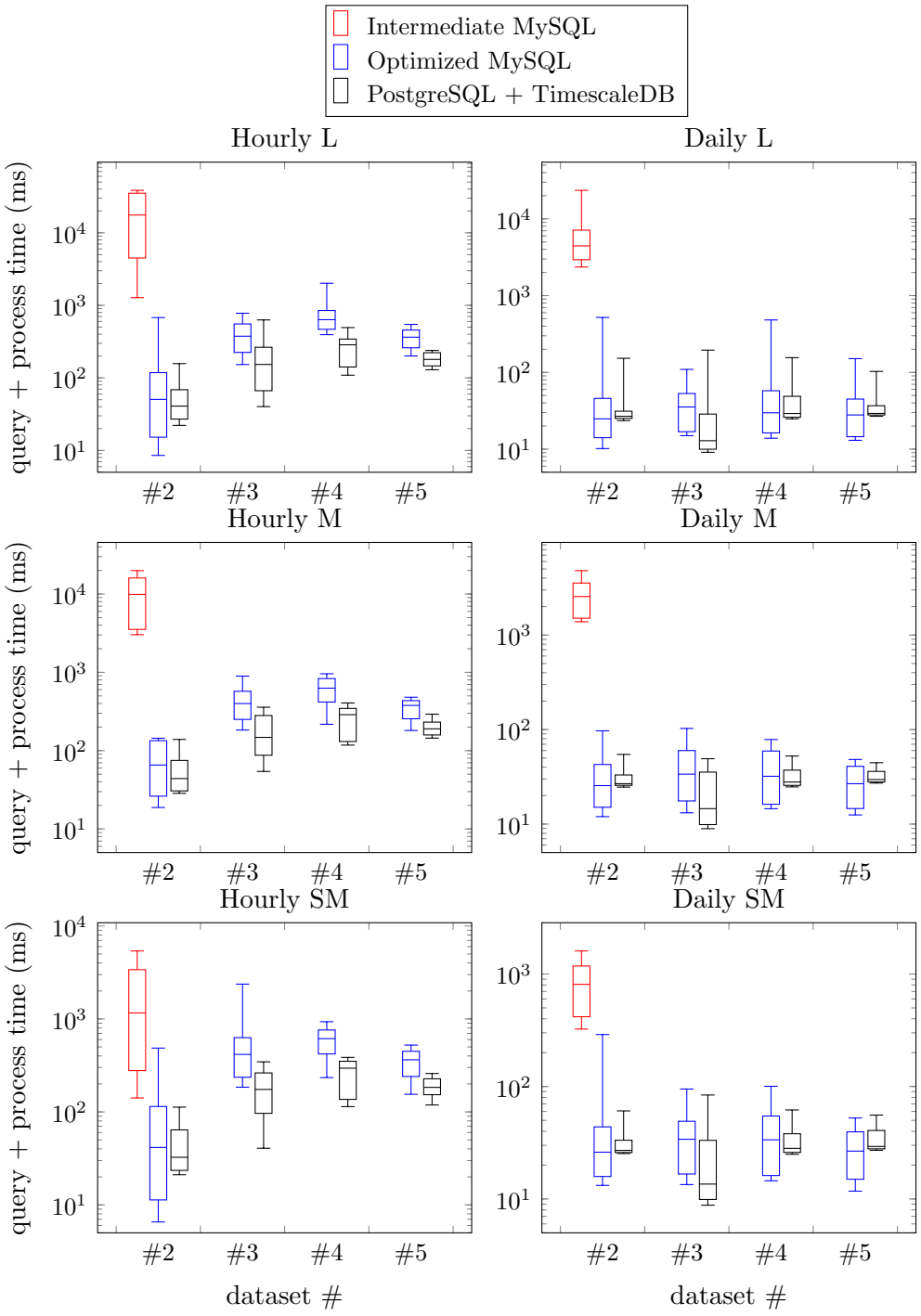

Figure 4.3: Results of query tests with ingest load.

may in the worst case take over ten seconds, but they are mostly faster than 5 seconds.

The performances of the optimized MySQL solution and the PostgreSQL + TimescaleDB solution are in the same order of magnitude. The PostgreSQL-based solution is consistently faster for hourly buildings, whereas for daily buildings it is mostly slower. All in all, the performance of these two solutions is rather identical. They return consumptions for different sizes of hourly buildings in less than one second, the median query times being around 200 ms. For daily building the query times are consistently less than 100 ms. The query times do not increase much as dataset sizes do: the difference in query times for dataset #3 and #5 is less than 100 ms for most test cases.

Figure 4.3 presents building query test results when the database is handling an ingest load. Dataset #1 is not included in any of the plots because the ingest load was implemented by inserting hourly readings, and there were not enough hourly meters for dataset #1 to implement the load. Also, the improved MySQL solution has no data for dataset #3, because the results without the load were already unfeasibly slow.

The execution times for the building queries with an ingestion load are rather identical to those without the load. The queries with load are in most cases some 100 ms slower, as is expected. The different solutions perform similarly with respect to each other, with or without load: the intermediate MySQL solution is clearly the slowest, optimized MySQL and PostgreSQL perform rather similarly, and the execution times do not slow down significantly as more data is stored in the database.

4.5.4 Apartment queries

Figure 4.4 illustrates the apartment query test results. The tests without load are missing datasets #4 and #5 for the intermediate MySQL solution as in Figure 4.2, and the tests with load are missing dataset #1 for all solutions and dataset #3 for the intermediate MySQL solution as in Figure 4.3.

The results of all storage solutions are relatively similar for all datasets, with and without load. The only major difference is the significantly slower performance of the PostgreSQL solution for daily apartments for datasets #4 and #5. Other than that, the performance is consistently good for all solutions: hourly queries take at most a few seconds for the last datasets, and daily queries repeatedly execute in only a few hundred milliseconds or less.

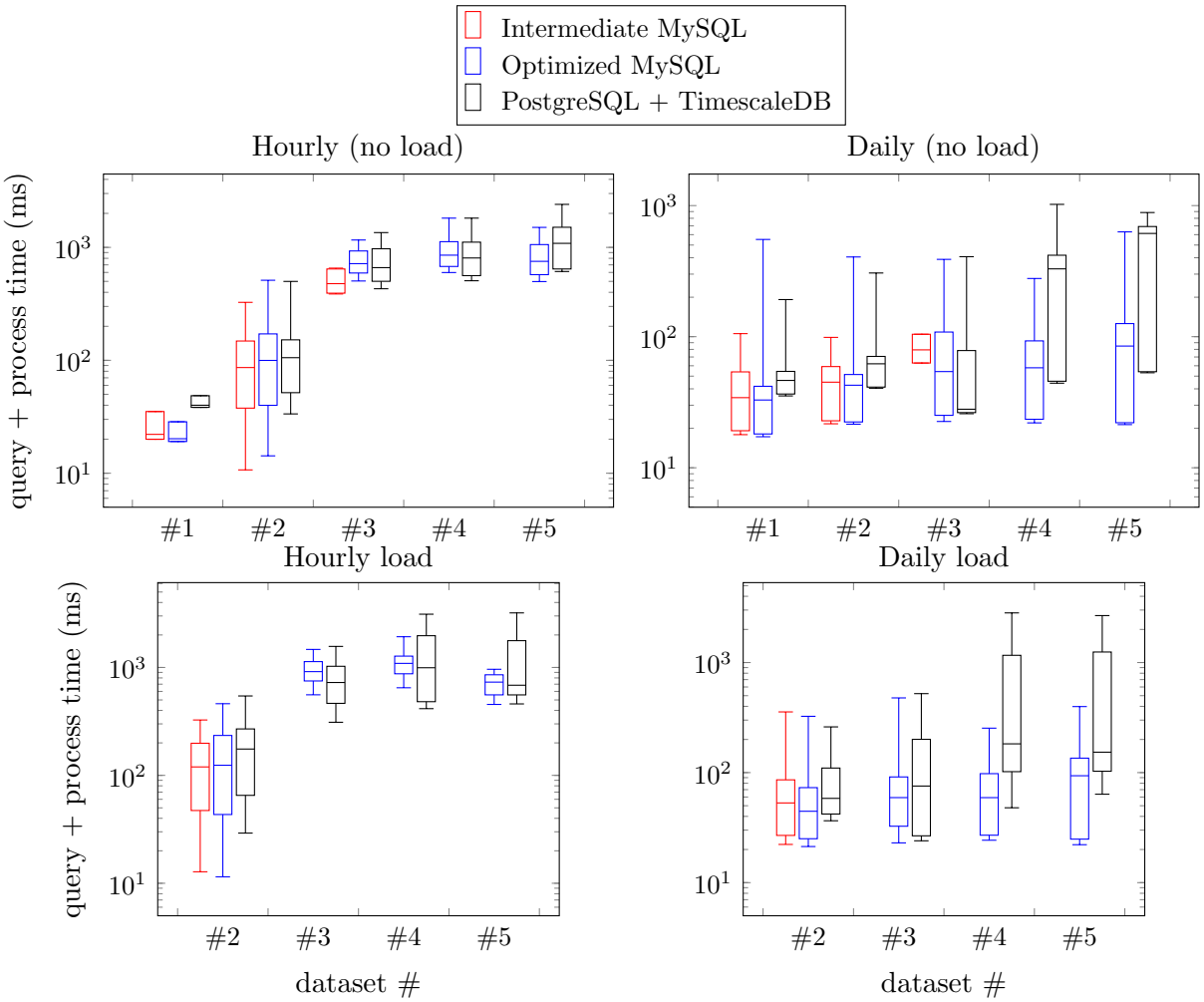

Figure 4.4: Results of apartment query tests with and without load.

4.5.5 Ingestion

Figure 4.5 presents the results of ingestion tests. Again, the tests are missing datasets #4 and #5 for the intermediate MySQL solution as in Figure 4.2. The figures present the time it takes to insert the readings of one device: for hourly readings the device is an apartment unit and for daily readings a collection unit. A collection unit sends tens or even hundreds of readings, and an apartment unit sends two readings. For this reason the ingestion times for hourly readings are faster.

There are differences in how the intermediate solution compares to the other two solutions between hourly and daily devices. For hourly devices, there is a significant difference between the performance of the intermediate solution and the others. Also, the performance of the optimized MySQL solution is slightly slower than that of the PostgreSQL solution for hourly devices. For daily devices, the performances of all solutions are largely identical.

There are considerable differences in the all devices figure, which depicts the total time it takes to insert all hourly readings and 1/4th of daily readings, as explained in section 4.1.2. For the optimized MySQL solution and the PostgreSQL solution this also includes the time it takes to execute the batch job for calculating building consumptions. The intermediate MySQL solution is the fastest in datasets #1, #2 and #3. For all datasets, the PostgreSQL solution is faster than the optimized MySQL solution.

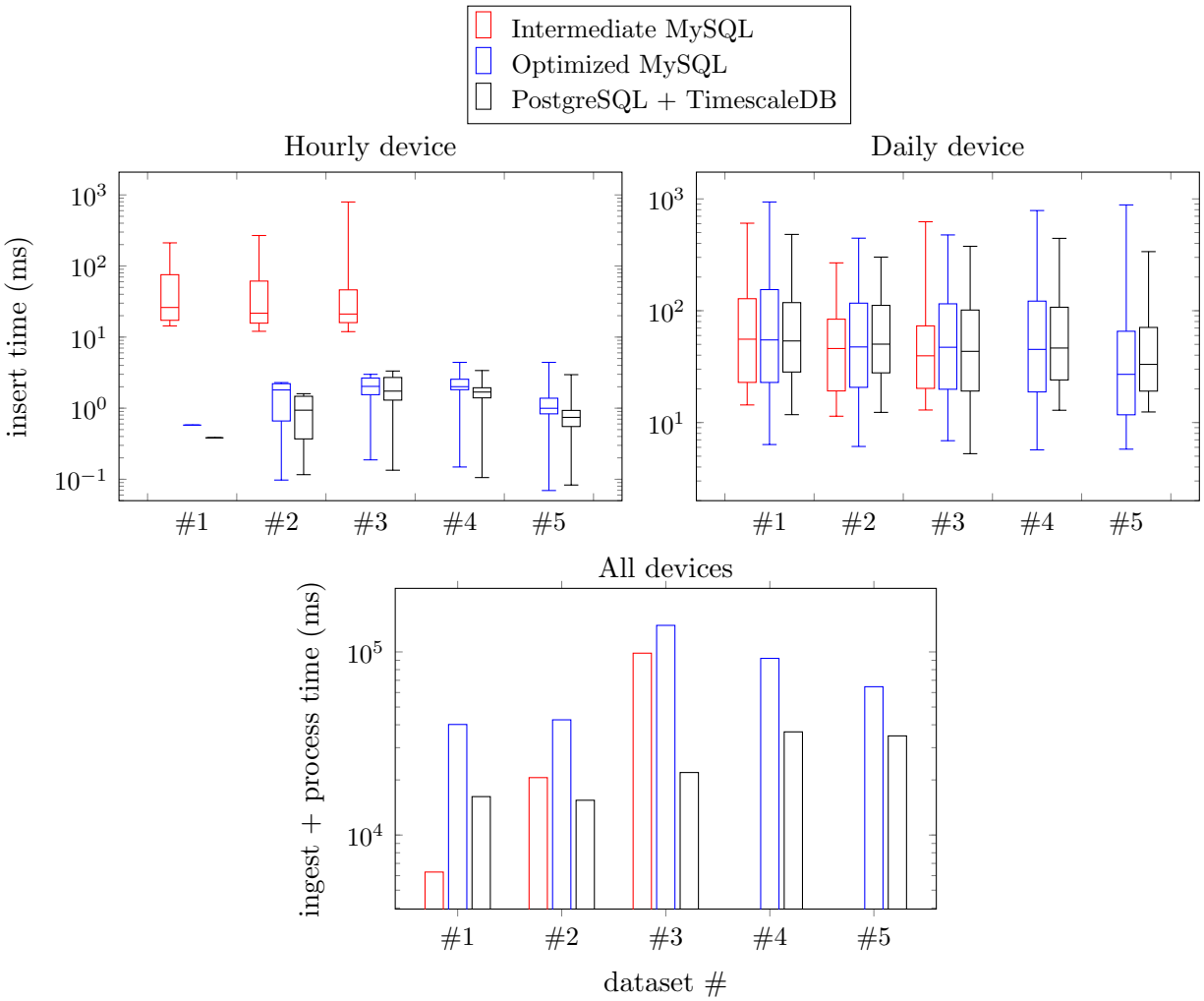

Figure 4.5: Results of ingestion tests.

Chapter 5

Discussion

This chapter discusses the results presented in chapter 4. First, the performance of each of the storage solutions is examined and analyzed. Then, the reliability of the obtained test results is investigated.

5.1 Performance of tested storage solutions

This section evaluates the performance of each of the tested storage solutions.

5.1.1 Intermediate MySQL solution

The intermediate MySQL solution was designed to address to problem of the database occasionally freezing due to a long-running meter reading `INSERT` query. Based on the tests, this issue was fixed by the changes implemented in the intermediate solution. It is presumed that the partitioning of the `MeterReading` table is the main reason the problem is solved: due to the partitioning, the data and indexes of the last partition of the `MeterReading` table always fit in memory, making `INSERT`s to the table fast. Previously one `INSERT` could have, in the worst case, caused rebalancing of an index spanning the entire table, with a size of tens of GBs. Because this amount of data would not have fitted in memory, extensive disk accesses would have been required.

However, as mentioned in section 4.4.1, the intermediate solution did not address the other problems present in the sensor system. One of the most significant of these problems are the slow building consumption queries. As illustrated by Figure 4.2, the performance of hourly readings will become unacceptably slow by the end of 2020, if the estimates for the growth of the system are valid. Also, daily queries, especially for large buildings, are al-

ready occasionally slow in the 2020/04 base dataset, as has also been observed by the users of the application.

The intermediate solution performs well outside the unfeasibly slow performance of building query tests. It was not tested with datasets after year 2023 (dataset #3) due to the slowness of the building queries, but until that dataset the other tests had acceptable execution times. In apartment query tests, the consumptions of an apartment were always fetched in less than a second. In ingestion tests, the total time it took to ingest all readings in the test was less than 2 minutes, which is within the bounds of the real-time requirements mentioned in section 3.3. All in all, the intermediate solution exhibited reasonably good performance for all datasets it was tested on. For this reason, it seems to be a feasible solution at least until 2023 for all other areas than building consumption queries.

5.1.2 Optimized MySQL solution

The optimized MySQL solution is an improved version of the intermediate solution, so it was expected to perform similarly or better in all of the tests. The most significant increase in performance was seen in building query tests, which are faster by at least an order of magnitude in all cases. This difference is a consequence of the pre-calculation of building consumptions. Because of it, building consumption data is fetched by only querying the `BuildingConsumptions` table, whereas the improved solution needs to query the larger `MeterReading` table and process the data to obtain consumptions. As a result, the tests indicate that this optimization solves the issue of slow building consumptions, which is one of the major problems in the system. Moreover, the tests indicate that the performance of building consumption queries is sufficiently fast at least until 2028.

The optimized solution yielded acceptable performance for all apartment query cases until the 2028 dataset. However, its performance was rather identical to that of the intermediate solution: in some cases it was slightly faster, and in the others slower. This was expected, because none of the optimizations directly address these queries. The removal of the redundant `apartment_id` column results in the returned dataset of these queries being smaller by 20%, but it did not have a noticeable performance impact on these tests.

The execution times of ingestion tests for the optimized solution were better than those of the intermediate solution. This is in line with the implemented optimizations: batching of hourly readings and using a trigger for updates of derived data should both provide improvements for ingestion. The performance of hourly device ingestion is clearly faster for this solution,

which is largely due to the batching of hourly inserts: the amount of distinct queries the database has to handle is decreased by a factor of 500 by using larger single `INSERT` statements. However, the performance of daily device ingestion is similar to the intermediate solution, indicating that the use of a trigger does not result in noticeable performance benefits. The ingestion times for all devices are at first slower than for the intermediate solution because they also include the pre-calculation of building consumptions. However, they are still always faster than two minutes, and remain feasible at least until 2028.

The tests indicate that the optimized MySQL solution yields sufficient performance at least until 2028 in all test cases. However, not all of the optimizations used in the tests are necessary, especially when the cost of implementing them is contrasted with the benefits they provide. More precisely, the pre-calculation of building consumptions is definitely required, as it solves the problem of unfeasibly slow building consumption queries. The batching of hourly inserts does provide significant improvements for the ingestion of hourly devices, the number of which will grow in the future. As a result, it will help in rapidly ingesting the hourly readings, and therefore provide real-time operability for the system. For this reason, implementing it is recommended. The two remaining optimizations, using a trigger for updating derived data and further optimization of the `MeterReading` schema, do not produce noticeable decreases in execution times. Therefore, implementing them is not suggested.

5.1.3 PostgreSQL and TimescaleDB solution

Because the PostgreSQL + TimescaleDB is purpose-built for storing time-series data and has the same optimizations as the optimized MySQL solution, the results of the PostgreSQL solution are expected to be similar or better. Moreover, the solution uses a looser read committed isolation level and compression, which should increase performance. However, the only test cases where the PostgreSQL solution performs consistently better than the MySQL solution are hourly building queries, hourly device ingestion and the ingestion of all devices. Furthermore, the performance increase is small in both hourly queries and hourly device ingestion; only the ingestion of all devices performs significantly better. This better performance is likely caused by the lower isolation level: it enables the concurrently executed `INSERT`s and also the concurrently executed building consumption batch job to be executed faster.

In all other tests cases the performance of the PostgreSQL solution is similar or worse than that of the optimized MySQL solution. As a result,

the use of TimescaleDB and its hypertables appears to not yield performance benefits for the essential data operations of this sensor system. The only observed performance benefits were due to the loosened isolation level: these benefits would most likely also have been perceived in the optimized MySQL solution had the same isolation level been used in it. In fact, the PostgreSQL solution would have had considerably lower ingest performance if it had been tested with the same serializable isolation level than the MySQL solution, as mentioned in section 4.4.3.

All in all, the use of TimescaleDB does not appear to provide performance improvements in comparison to the optimized MySQL solution. However, it does save on data storage costs: the compressed dataset is only about 20% of the size of the uncompressed one, as demonstrated by Table 4.1. It also provides many time-series-related utilities such as data retention, query helpers and analytics, which are not useful currently, but may appear to be helpful in the future. On the other hand, the migration to another database would require a considerable amount of work. Also, the solution must currently be hosted on Google Compute Engine, which requires substantially more maintenance work when compared with the hosted CloudSQL option. For these reasons and the similar or lower test performance, the migration to the PostgreSQL + TimescaleDB solution is not endorsed.

5.2 Reliability

This section analyzes the reliability of the obtained test results. First, factors that were not considered in testing but might have an impact on the results are discussed. Second, aspects that possibly decrease the reliability of the tests are overviewed. Then, the comparability of the storage solutions is examined. Finally, the applicability of the results is analyzed.

There are a few aspects that were not considered in these tests even though they have an effect on database performance. This was done to ensure that the tests do not contain unnecessarily many variables. Also, these aspects were considered to be easily testable on their own, if they are to be optimized. First, the database instances were not tuned manually: the CloudSQL MySQL instances used the default system settings also used in the production instance of the system, and the PostgreSQL instance was automatically configured with a command-line tool called timescaledb-tune [30]. Second, the size of the connection pool of the databases was not optimized. Instead, a default size of ten was used for the ingestion tests and the query ingestion load, where concurrent requests were made. Other tests used a pool of one connection, because the operations were sequential. Fi-

nally, the partitioning interval size was not optimized. Instead, the size was selected so that at most half of the memory is required by the latest active partition for MySQL solutions, and a quarter of memory for PostgreSQL + TimescaleDB. There were no recommendations for this parameter for MySQL; for TimescaleDB a recommendation of 1/4th of the instance memory was given, and therefore also used in these tests.

There are also some factors that may decrease the reliability of the obtained results. First, the test datasets are generated based on the estimated growth of the system. Therefore, if the estimate is incorrect, the tested dataset will not resemble that actual dataset the storage system will use. However, this is not a major problem, because the main purpose of these tests is to ensure that the system works with significantly larger datasets: the small details of the tested datasets are not relevant. Second, the state the database is in the tests is not quite similar to the state it will be in after years of actual use. For example, in the tests, the query planner does not have as good of an estimate of the amount and organization of data because the data was just batch inserted, instead of being added organically over the years. Also, the caches of the database contain different data in the tests than after the database being in actual use. However, these factors are inherent limitations in testing any future database states, and were not considered problematic. Finally, the tests were mostly done in a database without any additional load except for the query tests with a simulated ingestion load. For this reason, the results are presumably slightly faster than they would be on a database with additional load. The tests were not done with an additional load because creating a load that would accurately resemble that of the production system would have been challenging, and added unnecessary complexity. Moreover, as the query load tests demonstrate, the difference between execution times on a database without load and with load is not substantial.

The results between the different tested storage solutions are considered comparable. The tests on all solutions are executed in the same order, ensuring that the databases are in identical states. Also, all solutions use the same sets of generated data. As a result, the methods of executing the tests are not considered to introduce bias. However, there are some differences on the setup of the storage solutions that may introduce some bias. First, the queries used in testing both the MySQL solutions and PostgreSQL solutions are obtained from the running MySQL production instances. Therefore they are primarily optimized for MySQL. Thus, further optimization for PostgreSQL could, in theory, yield further performance gains for the PostgreSQL solution. However, this method of testing was used, because it was considered to give a realistic estimate of the performance after migrating from MySQL

to PostgreSQL + TimescaleDB. Second, as mentioned in section 5.1.3, PostgreSQL uses a looser isolation level, read committed, than MySQL, which uses serializable isolation. This does make the comparison skewed, and favors the PostgreSQL solution. However, it was not planned beforehand, but was rather done because the ingest performance of PostgreSQL was unfeasibly slow without it. As a result, it should be acknowledged when comparing the solutions.

The results are considered reliable and applicable to the sensor system of this thesis. The test setup largely resembles the actual setup of the current storage system running in production. The factors of unreliability overviewed in this section are acknowledged, and are not considered to skew the results. However, the results should not be used to obtain conclusions about the performance of the tested database or technologies in general. The performed tests are custom-built for this sensor system, which limits the applicability of the results to this system. Yet, general conclusions about database structure and query patterns that are potentially useful in improving the performance of time-series data storage can be obtained.

Chapter 6

Conclusions

The purpose of this thesis was to address the uncertainty in the future feasibility of the sensor system, which is caused by a history of performance problems and further increased by the expected rapid growth of sensor data. Two concrete cases of slowness were identified: occasional long-running meter reading `INSERT` statements and slow queries for calculating the consumptions of large buildings. These problems were addressed by testing the performance of three storage solutions, which were designed to improve the performance of the data usage patterns of this system: an intermediate MySQL solution, an optimized MySQL solution, and a PostgreSQL + TimescaleDB solution. These storage solutions were tested with the core use cases of the system, on estimated future datasets until 2028.

Based on the obtained results, the problem of uncertainty in the performance of the system is considered solved, at least until 2028. The results of both the optimized MySQL solution and the PostgreSQL + TimescaleDB solution were feasible for all tested use cases until the last dataset, for year 2028. They also provided acceptable performance for both concrete cases of slowness: no long-running `INSERT`s occurred, and all building consumptions queries had fast execution times of at most a second.

The recommendation for the future development of the data storage of the sensor system is to implement the optimizations of the optimized MySQL solution. Because the performance of this optimized solution is identical to the PostgreSQL + TimescaleDB solution, and migrating from the existing system to the PostgreSQL system requires considerably more work than implementing the optimizations of the MySQL solution, the MySQL solution is considered the better of the two. Based on the test results, there are already queries in the currently-running intermediate MySQL solution that take over 10 seconds. At the start of year 2021, the amount of such queries will increase, because the number of hourly read buildings will increase, and these

buildings will accumulate data. For these reasons, the recommendation is to start implementing the optimizations as soon as possible, at latest in the start of 2021. This will ensure that the problems are fixed before they are encountered widely by the clients of the system.

The implementation of these optimizations will require changes to the application-level code that interfaces with the database. For the new web application and mobile application these changes are relatively small, because they only contain a few queries that access data from the `MeterReading` table. However, the Java application contains many such queries, some of which are not written smartly to provide good performance, as explained in section 3.2.3.1. For this reason, fixing the queries of the Java application will require more work. However, it is necessary if the application is desired to remain usable in the future. Especially, the queries that fetch the consumptions of buildings, many of which are already slow, should be prioritized to be modified to use the optimized `BuildingConsumptions` table.

Based on the tests, it was concluded that the main problem causing the unfeasible slowness in the system is not the amount of stored data, but rather the inefficient queries for accessing the data. The occasional long-running inserts were likely caused by the amount of stored data, but they were addressed by partitioning the data into partitions that are smaller than the amount of available memory. After this change, the sheer amount of data in the `MeterReading` table has not appeared to be a problem. This is demonstrated by the apartment query tests, which access a large amount of data (the last three years) from the `MeterReading` table. As Figure 4.4 presents, the execution times of these queries remain very identical for datasets #3 to #5 even though the size of the database grows from 105 GB to 703 GB, and the change in size is caused almost entirely by the growth of the `MeterReading` table. Based on this observation, it is assumed that the database will also be usable after 2028, as long as the partition time interval is scaled to prevent single partitions from becoming larger than the memory of the database instance. This further decreases the uncertainty in the feasibility of the MySQL data storage solution, as the problems in it appear to be addressable by using suitable methods of organizing and querying the data.

Another observation obtained from the test results is that larger result set sizes fetched from the `MeterReading` table correspond to slower query performance. This is demonstrated by comparing the performance of building and apartment queries for the intermediate MySQL solution: they both access the same time span, and therefore the same partitions of data, but the performance of building queries is considerably slower. This is assumed to be because buildings fetch the readings of numerous apartments, typically from 25 to 150, and therefore the result set is this many times larger. This

result set size is further increased by increasing number of buildings in hourly reading, for which the result set is 24 times larger to those in daily reading. This problem was addressed by pre-calculating the building consumptions in the optimized MySQL solution, reducing the size of the returned dataset by the amount of apartments in a building, i.e., by a factor of 25 to 150. Because large result sets seem to cause slower query times, consideration should be given in the future to prevent the result sets from becoming too large.

The work performed in this thesis also suggests some other possible future developments for the system. First, it seems that if further performance benefits for fetching reading data are desired, another storage solution may need to be explored. This is because the main use case of the meter data includes fetching the last three years of meter readings for a certain group of meters. Because the data is partitioned by reading time, this query always requires accessing tens, and in the future hundreds, of GBs of data, more than can fit into the memory of the database. As a result, these queries always require reading data from the disk, which slows them down. It is also a possible that the disks accesses set a fundamental performance limitation for these queries, possibly explaining why the PostgreSQL + TimescaleDB did not perform better for these queries, even though it is optimized for storing time-series data. For this reason, it seems that if even faster execution times for these queries are required, the data needs to be organized differently on disk. For example, if the data was partitioned by `(meter_id, readingDate)`, the readings of a single apartment and single building would often be stored in one partition, assuming their `meter_id`s are adjacent, which they often are. In this case, only one disk seek would need to be done per query, likely increasing query performance. This type of partitioning is not possible in MySQL or PostgreSQL with TimescaleDB, so another storage solution would need to be used. For example, Cassandra and many other NoSQL stores support this type of partitioning. This type of solution would require splitting the application data into multiple services, and bring with it the problems associated with such solutions, as explained in 2.1.4. Nevertheless, this alternative may be worth exploring, if additional performance gains are desired.

Another suggestion for the future development of the application is to consider the use of a service to encapsulate the access to meter reading data, as proposed by Sadalage and Fowler [26]. As mentioned in section 3.2.4, the logic for handling readings is written separately in many codebases, and may therefore function slightly differently, especially on edge cases. If a reading service was used, this logic could be implemented once in it, and then used from all different codebases, ensuring the results are consistent in all applications. Moreover, if the reading data access was encapsulated in a

service, the underlaying storage method for reading data could be updated or even swapped entirely without having to update the applications using the data.

Despite the results of this thesis being specific to this sensor system, some general recommendations for storing time-series data in a relational database may be obtained based on the performed work:

- Test with application-specific data and queries
- Try partitioning the data by timestamps
- Remove redundant indexes from the data
- Use a less strict isolation level
- Use batching for inserts
- Use triggers for updating derived data
- Review queries on the data
- Compare methods for handling derived data

Firstly, and most importantly, it is recommended to always test any changes with application-specific data and queries. This ensures that the benefits that are thought to be obtainable by using a solution or technology are actually realized on the application they are applied to. Second, partitioning the time-series data by the timestamp value may be helpful. For this system it was helpful because it decreased the size of the indexes that need to be updated when inserting data, thus making ingestion faster and more reliable. For some other applications it may help because only a limited timespan of values are queried, and thus only a few partitions may need to be accessed.

Removing redundant indexes from the data may also increase performance. Indexes are a tradeoff between increased query performance and decreased insert performance: if some index is not required by the queries or it is possible to change the schema to combine indexes, removing an index should be considered. Also, a less strict isolation level may help increase performance, especially if a lot of concurrent queries that modify data are executed. However, it also results in encountering more effects of concurrency. As a result, it needs to be analyzed based on the requirements of the application.

The performance of data ingestion can potentially be further improved by using batching and triggers. Batching refers to grouping multiple inserted

rows into one `INSERT` statement, instead of executing them in separate statements. It may decrease execution times by reducing the amount of separate statements the database has to handle. For this system, it considerably increased the speed of `INSERT` statements, as shown in Figure 4.5. Using triggers reduces the amount of round trips that are required between the client and the database by using a database trigger to do certain operations that would otherwise be performed separately by the client.

It is also important to analyze the queries that access the time-series data. Because these queries are often targeting a dataset of hundreds of GBs, it is important that they are optimized for performance. For example, the Java application in the sensor system of this thesis has many query patterns that are slow, but can be made performant by simple modifications.

Finally, different methods for handling derived data should be compared. There are many options: calculating derived data after queries on application-level code, calculating derived data in queries with database functions and SQL, pre-calculating derived data into a separate database table using a batch job or even using a materialized view for storing the data. Depending on the stored data and its query patterns different solutions may be appropriate. For the sensor data of the sensor system of this thesis, calculations after querying and during queries were considerably slower than using a batch job to pre-calculate the data.

It should be noted that even though the list of recommendations does not include changing to an entirely different database, that option is not considered inferior. Sometimes it is the only feasible solution: in these cases the solution should be tested thoroughly before performing the migration. However, the migration to another database often requires a considerable amount of work, and also includes high risks. In many cases the amounts of stored data are not too large for a relational database, and simply optimizing the relational database and the data access patterns may be sufficient. For the sensor system of this thesis, such optimizations were found to significantly increase the performance of the system, making it a feasible solution for the storage of the growing future sensor dataset.

Bibliography

[1] 2ndQuadrant. BDR. URL https://www.2ndquadrant.com/en/resources/postgres-bdr-2ndquadrant/.

[2] Gleb Bryksin and Alex B. A Detailed Guide to Database Denormalization with Examples in 2020, jan 2020. URL https://rubygarage.org/blog/database-denormalization-with-examples.

[3] Cloud Google. The App Engine Standard Environment. URL https://cloud.google.com/appengine/docs/standard.

[4] Edgar F. Codd. *A Relational Model of Data for Large Shared Data Banks*, pages 263–294. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002. doi: 10.1007/978-3-642-59412-0_16.

[5] Continuent Ltd. Tungsten Replicator – All You Need To Know. URL https://www.continuent.com/solutions/tungsten-replicator-all-you-need-to-know/.

[6] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J J Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, and Et al. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.*, 31(3), aug 2013. doi: 10.1145/2491245.

[7] C. J. Date. *An introduction to database systems*. Addison-Wesley, 2000. ISBN 0201385902.

[8] Muntazir Fadhel, Emil Sekerinski, and Shucai Yao. A comparison of time series databases for storing water quality data. In *Advances in Intelligent Systems and Computing*, volume 909, pages 302–313. Springer Verlag, oct 2019. ISBN 9783030114336. doi: 10.1007/978-3-030-11434-3_33.

[9] Matthias Gelbmann and Paul Andlinger. MySQL is the DBMS of the Year 2019, jan 201. URL https://db-engines.com/en/blog{_}post/83.

[10] Google Cloud. App Engine, . URL `https://cloud.google.com/appengine`.

[11] Google Cloud. Instance settings — Cloud SQL for MySQL, . URL `https://cloud.google.com/sql/docs/mysql/instance-settings`.

[12] Google Cloud. Compute Engine: Virtual Machines (VMs), . URL `https://cloud.google.com/compute`.

[13] Katarina Grolinger, Wilson A Higashino, Abhinav Tiwari, and Miriam A.M. Capretz. Data management in cloud environments: NoSQL and NewSQL data stores. *Journal of Cloud Computing*, 2 (1), 2013. ISSN 2192113X. doi: 10.1186/2192-113X-2-22. URL `http://www.journalofcloudcomputing.com/content/2/1/22`.

[14] Robin Hecht and Stefan Jablonski. NoSQL evaluation: A use case oriented survey. In *2011 International Conference on Cloud and Service Computing*, pages 336–341. IEEE, dec 2011. ISBN 978-1-4577-1637-9.

[15] Robert Hodges. If You *Must* Deploy Multi-Master Replication, Read This First, 2012. URL `http://scale-out-blog.blogspot.com/2012/04/if-you-must-deploy-multi-master.html`.

[16] Martin Kleppmann. *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems*. 2017. ISBN 9781491903063.

[17] Vika Klochkova. SQL vs. NoSQL vs. NewSQL: Comparative Advantages and Disadvantages - DZone Database, nov 2016. URL `https://dzone.com/articles/sql-nosql-newsql-comparative-advantages-and-disadvantages`.

[18] Ajay Kulkarni. What the heck is time-series data (and why do I need a time-series database)?, 2018. URL `https://blog.timescale.com/blog/what-the-heck-is-time-series-data-and-why-do-i-need-a-time-series-database-dcf3b1`

[19] Chen Li, Jianxin Li, Jinghui Si, and Yangyang Zhang. FluteDB: An efficient and dependable time-series database storage engine. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 10658 LNCS, pages 446–456. Springer Verlag, dec 2017. ISBN 9783319723945. doi: 10.1007/978-3-319-72395-2_41.

[20] Chen Luo and Michael J. Carey. LSM-based storage techniques: a survey. In *VLDB Journal*, volume 29, pages 393–418. Springer, dec 2020. doi: 10.1007/s00778-019-00555-y. URL `http://arxiv.org/abs/1812.07527http://dx.doi.org/10.1007/s00778-019-00555-y`.

[21] MongoDB Inc. New Hash-based Sharding Feature in MongoDB 2.4, 2013. URL `https://www.mongodb.com/blog/post/new-hash-based-sharding-feature-in-mongodb-24`.

[22] Dmitry Namiot. Time Series Databases. In *XVII International Conference Data Analytics and Management in Data Intensive Domains (DAMDID/RCDL)*, 2015.

[23] Nelson Patrick. Just one autonomous car will use 4,000 GB of data per day, 2016. URL `https://www.networkworld.com/article/3147892/one-autonomous-car-will-use-4000-gb-of-dataday.html`.

[24] Alex Petrov and O'Reilly for Higher Education (Firm). *Database Internals: A Deep Dive into How Distributed Data Systems Work*. 2019. ISBN 9781492040347. URL `https://learning.oreilly.com/library/view/database-internals/9781492040330/?ar`.

[25] John Piekos. SQL vs. NoSQL vs. NewSQL: Finding the Right Solution - Dataconomy, aug 2015. URL `https://dataconomy.com/2015/08/sql-vs-nosql-vs-newsql-finding-the-right-solution/`.

[26] Pramod J. Sadalage and Martin Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley, 2012. ISBN 9780321826626.

[27] Statista. IoT: number of connected devices worldwide 2012-2025, 2020. URL `https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/`.

[28] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural Era (It's time for a complete rewrite). In *33rd International Conference on Very Large Data Bases, VLDB 2007 - Conference Proceedings*, pages 1150–1160. Association for Computing Machinery, Inc, 2007. ISBN 9781595936493. doi: 10.1145/3226595.3226637.

[29] The Apache Software Foundation. 2. Replication — Apache CouchDB® 2.3 Documentation. URL `https://docs.couchdb.org/en/stable/replication/`.

[30] Timescale Inc. timescaledb-tune: A tool for tuning TimescaleDB for better performance by adjusting settings to match your system's CPU and memory resources. URL `https://github.com/timescale/timescaledb-tune`.

[31] Timescale Inc. TimescaleDB: SQL made scalable for time-series data. Technical report, 2017. URL `https://pdfs.semanticscholar.org/049a/af11fa98525b663da18f39d5dcc5d345eb9a.pdf`.