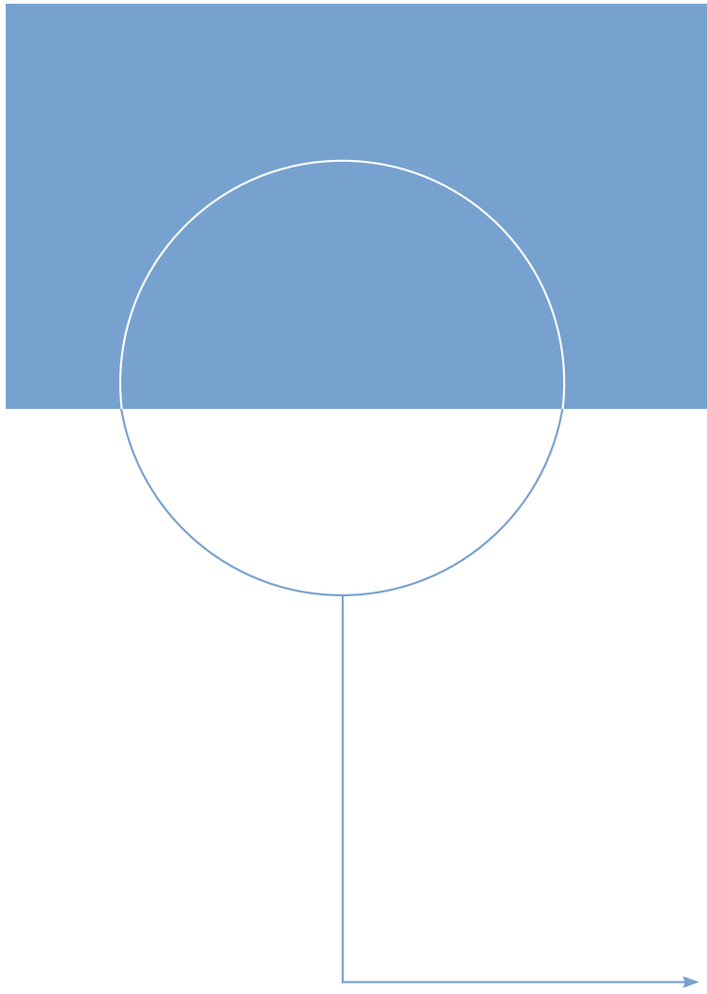

Master's thesis

2020

Mari Fredriksen

NTNU
Norwegian University of
Science and Technology
Faculty of Information Technology and Electrical
Engineering
Department of Computer Science

Mari Fredriksen

Database Management Systems in Smart Cities: Requirements for IoT and Time-Series Data

June 2020

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

NTNU

Norwegian University of
Science and Technology

Database Management Systems in Smart Cities: Requirements for IoT and Time-Series Data

Mari Fredriksen

Computer Science
Submission date: June 2020
Supervisor: Svein Erik Bratsberg

Norwegian University of Science and Technology
Department of Computer Science

Abstract

The emerge of the Internet of Things (IoT) has opened endless opportunities as well as challenges. Smart city is a domain utilizing IoT and has been a phenomenon for almost a decade. More cities across the world are becoming ”smart” every year. Smart cities collect data that can make cities smarter both in the prospect of infrastructure and life quality, but also sustainability. To achieve this vast amounts of data are collected throughout the city and are processed, stored, and analyzed. In this thesis, three database systems, one relational database, MySQL, and two non-relational databases, Cassandra, and MongoDB are presented and reviewed in light of the requirements for IoT data management found in literature. Through experiments, and evaluation of theory based on a literature review, this thesis shows the potential of the three database systems of being suited for smart city data in a time-series format. MongoDB has promising results as it performs almost perfectly in terms of theoretical requirements. Cassandra shows great potential in terms of time-series storage because of its architecture, but have some limitations when handling aggregations. MySQL, have some large drawbacks based on the requirements but performed equally well as MongoDB for execution times, which shows that on limited data amounts, MySQL can work with IoT data. It is concluded that, as expected, none of the studied database systems are perfect for IoT and each has limitations and possibilities.

Sammendrag

Fremveksten av ”Internet of Things” (IoT) de siste årene har åpnet opp endeløse muligheter, men kommer også med en del utfordringer. Smarte byer har vært et fenomen i snart et tiår og stadig flere byer rundt om i verden blir ”smarte”. Smarte byer samler inn data som kan brukes for å gjøre byer ”smartere” både med tanke på infrastruktur og livskvalitet, men også med tanke på bærekraft. For å oppnå dette må store mengder data samles inn, prosesseres, lagres og analyseres. I denne masteroppgaven presenteres tre databasesystemer, MySQL, MongoDB og Cassandra. De blir studert med tanke på krav til håndtering av IoT data og tidsseriedata, som er funnet i et innledende litteraturstudie. Gjennom evaluering av eksperimenter og teori funnet i litteraturstudiet, viser dette prosjektet hvilke databasesystemer som kan passe for IoT data på et tidsserie format. MongoDB har et stort potensiale for dette ved å dekke nesten alle kravene for datahåndtering og databasesystemer funnet i teorien. Cassandra har gode resultater for spørringer hvor et tidsintervall hentes ut av databasen, men har problemer med aggregering av data. MySQL har noen store mangler med tanke på krav til datahåndtering, men resultatene av eksperimentene viser at MySQL gjør det omtrent like bra som MongoDB. Dette antyder at databasen kan bli brukt i IoT dersom mengden data er begrenset. Det konkluderes med at ingen av databasene som er studert i denne oppgaven er perfekte til bruk for IoT tidsseriedata i smarte byer, men alle har sine muligheter og begrensninger.

Acknowledgements

First, I would like to thank Svein Erik Bratsberg for being the supervisor of this thesis. Also, I would like to thank my partner for being the best friend and support-team throughout this semester.

Table of Contents

List of Tables

List of Figures

Abbreviations

IoT	=	Internet of Things
GHG	=	Green House Gas
ICT	=	Information and Communication Technology
ZEN	=	Zero Emission Neighbourhoods
NTNU	=	Norwegian University of Science and Technology
IT	=	Information Technology
D2C	=	Distributed-to-Centralized
F2C	=	Fog-to-Cloud
QoS	=	Quality of Service
WSN	=	Wireless Sensor Networks
RDBMS	=	Relational Database Management System
SQL	=	Structured Query Language
NoSQL	=	Non-Relation Database
TSDB	=	Time Series Database System
OLTP	=	Online Transaction Processing
XML	=	Extensible Markup Language
JSON	=	JavaScript Object Notation
CQL	=	Cassandra Query Language

Chapter 1

Introduction

According to the United Nations Population Fund, there were approximately 7.7 billion people in the world in 2019. By 2050, the population is expected to approach 10 billion and an estimated 70 percent of these people will be living in urban areas [65]. Cities consume between 60 and 80 percent of energy worldwide and are responsible for large shares of the Green House Gas (GHG) emissions [4]. To meet new demands of sustainability, but at the same time maintain quality of life in cities, new systems to manage and build smart cities is required. The term ”smart city” is an approach to handle these new challenges by making use of Information and Communication Technology (ICT).

At the same time as the population is growing, increasing amounts of people can access the internet. At the end of 2019, about 60 percent of the world's population were internet users. In Europe, the number is heading towards 90 percent. Besides, the number of devices each person is connecting to the internet is growing. With the emerge of IoT, not only computers and smartphones are connected to the internet, watches, smart home devices, and even refrigerators or coffee makers are connected to the internet. Also, sensors and monitoring devices can be found everywhere for instance in locks, parking, or traffic lights. According to [37], in 2020, 24 billion ”things” are on the internet.

Both increased population, increase in people accessing the internet and increased number of devices each person is connecting to the internet, the amounts of data, is dramatically increasing. How these huge amounts of data are handled will be an important factor in the coming years, and will also be an important success factor as to how valuable this data proves to be. The time-series format of much of the data generated by smart cities and IoT creates some additional challenges as storing this data needs some new considerations. Choosing an appropriate database management system to store and manage the data generated by a smart city is important because it can have a large impact on the efficiency and intelligence of smart city platforms.

1.1 Background and motivation

Smart cities are cities utilizing ICT to be smarter and to achieve higher quality of living and sustainability. The Zero Emissions Neighbourhood (ZEN) [1] center at NTNU aims to create Zero Emission Neighbourhoods contributing to a more sustainable society facing climate change. In a research project in collaboration with ZEN researchers, in the fall of 2019, the author of this thesis investigated ICT architecture in smart cities with a focus on data management [36]. Wanting to continue within the same research area, this thesis extends the research in [36] by looking further into database systems that can be suited for the data gathered by smart cities. IoT has emerged as the most important technology within smart cities and thousands of sensors within cities are every minute and second gathering enormous amounts of heterogeneous data, often in a time-series format. This data needs to be stored and handled efficiently to produce value. Researching ways to handle this has gained popularity over the recent years as the handling of these amounts of data has shown to be complex. With a theoretical background from database systems and with a background on smart cities and IoT, there is a hope that this master thesis can be a positive contributor to the research field of smart cities and IoT, contributing to the database system domain.

1.2 Research questions

The research questions defined below will be answered in Chapter 6. The questions have been motivated by the background described in the section above and will work as motivation and guidelines for the research in this thesis.

RQ1 What database systems are researched in the literature about IoT data in smart cities and time-series data?

RQ2 What are the requirements for IoT and time-series data management in database systems?

RQ3 What databases are suited for IoT and time-series data based on the requirements found in literature, in **RQ2**?

RQ4 How do the databases from **RQ3** perform in experiments testing the requirements for data management in IoT related to smart city use, compared to the expected performance from literature?

1.3 Research method and research design

To answer the research questions they will be followed from the top down. First, there will be conducted a literature review to gain an understanding of the theory which is relevant

[1] https://fmezen.no/

for the thesis, trying to answer research questions **RQ1** and **RQ2**. Secondly, a study to understand the capabilities and architecture of three popular open-source database systems is conducted related to the findings in the literature about the requirements of IoT data management in **RQ3**. Finally, a scientific experiment investigating the fitness of the three databases in light of IoT requirements will be performed and analyzed.

1.3.1 Theory and Related Work

It is important to gain a deeper understanding of several aspects to be able to make some recommendations about database systems suited for smart cities. Firstly, knowledge about smart cites, how they operate, and characteristics are needed. Second, an understanding of IoT and what types of data that is handled in IoT is important to take on the next part which is database systems. Database systems and some important concepts will be discussed to know which best fits the characteristics of IoT data in smart cities. A literature review of databases will be conducted to find the most relevant databases and storage systems in theory. This will form the basis for the next phase of the project.

1.3.2 Implementation and Experiments

In this phase, there will be important to do experiments relevant to the research found in the first phase of the project. Testing relevant queries to find which database performs best under the given requirements. It is important to find a dataset that is similar to the one that will be found in smart city scenario so that the experiments will have real value. The research is limited to finding out what database performs better in the case that the data is inserted into the database as they are with limited use of manipulation and handling of the data. MySQL [2] and the two NoSQL databases, Apache Cassandra [3] and MongoDB [4] is tested in the experiments. By conducting the experiments the hope is to answer research question **RQ4**.

1.4 Research scope

The main scope of this thesis is to gain an understanding of what database system is suitable for storing IoT data from smart cities, which typically is in the format of time-series. More specifically, this research concentrate on the centralized data management part of the ICT architecture, where all the city data will be stored. Firstly a literature study is done to (1) Find relevant research in literature about database systems used in smart cities and IoT. (2) Requirements for managing data in IoT and time-series data are research in theory in the relevant literature. Three popular open-source database systems, MySQL, MongoDB, and Cassandra, are evaluated in detail on the requirement found in the literature review. Finally, experiments evaluating the performance of the three database systems in terms of execution times are presented. The experiments are limited to the case that minimal processing and managing of the data is done to the data before entering the database.

[2] https://www.mysql.com/
[3] http://cassandra.apache.org/
[4] https://www.mongodb.com/

1.5 Limitations

One limitation of the research in this thesis is the number of databases researched in detail. Before the research, there was already an idea of wanting to investigate MongoDB in particular, as well as Cassandra. As a result of this, the literature review has been dragged in that direction. A broader analysis of all relevant databases could be a solution to overcome this limitation. This is mentioned as a recommendation for future work. Another limitation that is related to the previous point, is the execution of the research strategy ”literature review”. Though search terms were developed before doing the research and the studied articles were saved, no technique for conducting a literature review was followed in detail. As a result, the process might have been both more time consuming for the author, but also some interesting papers might not have been studied.

1.6 Disposition

In Chapter 2, the background information and literature review of relevant information is presented. Section 2.1 contains definitions of some relevant theory and ICT architecture related to smart cities. In Section 2.2, IoT and time-series data is defined. Also, the requirements of data management in IoT are presented. Storage systems are presented and discussed related to IoT and time-series data storage in MySQL, MongoDB and Cassandra in Section 2.3. The data management requirements found in Section 2.2.3 is discussed in light of the three database systems MySQL, MongoDB, and Cassandra in Section 2.3.4. Finally, Chapter 2 ends with a review of related work. In Chapter 3, the methodology and research strategies used in the thesis is presented. Furthermore, in Chapter 4 the experimental setup is explained for the performed experiments. Chapter 5 contains an evaluation of the work and the goals and expected results are discussed up against the literature. Finally, Chapter 6 answers the research questions, shows the conclusions made in the thesis, and discusses some future work prospects.

Chapter 2

Background

2.1 Smart Cities

A single definition of the term ”Smart City” has not yet been agreed upon, despite being popular among researchers the recent years. The properties that must be fulfilled for a city to be considered ”smart” is developing as new technologies and opportunities arise each year. Several surveys are trying to make a common understanding of the term and to find a single definition for the expression [4, 38], jet no clear and consistent definition of a smart city among different stakeholders exists to this day. The term smart city was first introduced in the 1990s and at that time the focus was on the significance of new ICT concerning modern infrastructures within cities [4]. A common understanding of the term smart city was in 2012 stated by the European Commission: ”to use diverse technologies to help in achieving sustainability in smart cities and a general goal of smart cities is to improve sustainability with help of technologies,” [2].

Because the population on earth and especially urban populations are growing fast, cities have a huge impact on the environment. In 2020 about 80 percent of the population of the world is living in urban areas [4] and cities together consume between 60 and 80 percent of energy worldwide and are responsible for large shares of the Green House Gas (GHG) emissions. Pollution and sustainable living have become a large problem in many large cities and the need for systems regulating all aspects of city living is growing. Smart cities aim to tackle these problems with the use of ICT. A short description of possible domains within a smart city is listed below, from [78]:

- **Smart Parking:** Monitoring of available parking in a city.
- **Structural Health:** Monitoring of material conditions in buildings, bridges, and historical monuments.
- **Noise Urban Maps:** Monitoring of noise from bar areas or traffic.
- **Traffic Congestion:** Monitoring of vehicles and public transport to optimize driving, bicycling, and walking routes.

- **Smart Lightning:** Intelligent and weather adaptive street lights.
- **Waste Management:** Detection of rubbish levels in containers to optimize trash collection routes.
- **Intelligent Transportation System:** Smart roads and intelligent highways with warning messages and diversions according to climate conditions and unexpected events like accidents or traffic jams.

Smart city initiatives are being developed in cities all over the world, many of them still being in an early pilot stage. The ZEN center at NTNU and SmartSantander, in the city of Santander in Germany, are two examples of smart city pilot platforms.

2.1.1 ICT Architecture in Smart Cities

Information and communications technology (ICT) architecture is an extended term of information technology (IT). ICT involves hardware, software, network devices as well as any product that will store, retrieve, manipulate, transmit, or receive information electronically or digitally. ICT architecture involves the description, coordination, and structuring of an enterprise's ICT systems. "The goal of using ICT is to improve existing systems and functionalities by making them more efficient, user-friendly, or in general more citizen-centric," [36]. In smart cities, ICT architecture is a popular research field because of the importance of well structured, and efficient architecture to handle the complex management of the technology in cities generating vasts amounts of data. In the preceding subsections, popular ICT architecture used in state-of-the-art smart city platforms are explained.

Centralized computing

Centralized computing, also referred to as cloud computing, is a schema where all the computing hardware is located in one same geographical location. Based on NIST [1] definition "cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources such as networks, servers, storage, applications, and services, that can be rapidly provisioned and released with minimal management effort or service provider interaction," [51]. Having this large computational resource is cheap and leads to the shared utilization of equipment. It also opens up huge opportunities for utilizing big data. The main drawback of cloud computing is the high latency which may occur between IoT devices and a cloud that is located far away from each other. The cloud could be located in another county, or even in another continent. Other concerns in cloud computing are safety. Because all the resources are located at the same spot it could be easier for an attacker to compromise a large volume of data. However, most cloud providers have a high focus on security and offer built-in security services for the user's applications.

[1] U.S. National Institute of Standards and Technology

Distributed-to-centralized computing

In an initial work on database management systems and ICT architecture in smart cities, [36], the following was explained about distributed-to-centralized ICT architecture.
”Distributed-to-centralized (D2C) computing also called fog-to-cloud (F2C) computing is a relatively new concept that can be seen as an extension of cloud computing. Cloud computing lacks efficient support for the development of IoT services with strong requirements in latency, security while minimizing the traffic load in the network” [48]. In D2C architecture the computation is moved closer to the edge of the network. A cloud is still included in the architecture, but data that requires low latency are moved to the edge of the network. At the same time, the processing time of the cloud is reduced because the fog relieves the cloud. In smart cities, the use of IoT is a must these days. Utilizing fog architecture in IoT applications can lead to several advantages:

- Provide low latency services.
- Offers location-aware services.
- Provides better scalability which supports a widely geographically distributed application.
- Offers better Quality of Service (QoS). Fog nodes can support the QoS requirements of services locally.
- Provides more efficient communication with other systems either through the cloud or other fogs.
- Supports better mobility and access control for different types of mobile devices as they travel around the city. The ”traveling” mobile device can connect to more nodes around the city.

2.1.2 Zero Emission Neighborhoods

The ZEN Research Center is a smart city initiative at NTNU, which aims to reduce greenhouse gas emissions of neighborhoods towards zero within its life cycle. Four points have been developed as keys to achieve this goal:

1. Plan, design and operate buildings and associated infrastructure components toward zero life cycle GHG emissions.
2. Become highly energy-efficient and powered by a high share of renewable energy in the neighborhood energy supply system.
3. Manage energy flows and exchanges with the surrounding energy system smartly and flexibly.
4. Promote sustainable transport patterns and smart mobility systems.

Recently a high focus of the ZEN center has been on D2C computing. Traditionally, data management in smart cities has focused on centralized facilities based on cloud computing technologies, but to benefit from the advantages discussed in the section above, the D2C architecture is being developed in the ZEN pilot. Managing the data in this type of architecture is highly complex and the ZEN center still lacks research on databases that are suited for the data that is gathered and stored from cities. In [36], preliminary research was initiated on databases that are suited for ZEN center: ”What is important when selecting the database is that it must be able to handle large amounts of data from the city. It needs to be scalable as cities are growing and there is an increased use of IoT devices which again increases the volumes of data. The database must handle multiple data types [77, 18], coming from several different IoT devices generating heterogeneous data [31], it must be able to handle both historical and real-time data [31, 64, 7, 18]. Also, the database must be able to integrate with some data processing platform that can provide even more analytic processing power to handle big data management, like Spark or similar systems [77, 19, 18].”

2.2 Internet of Things

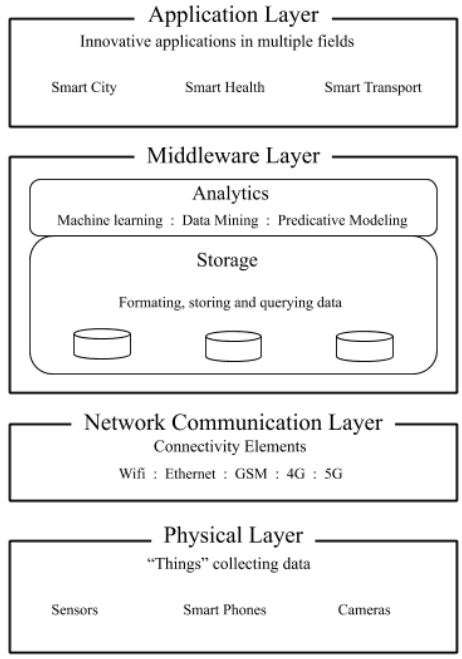

Figure 2.1: Illustration of IoT architecture.

In 2014, The Gartner Group predicted that by this year, 26 billion things are connected to the internet [67]. These days, there are not only computers and laptops that are con-

nected to the internet. Also, smartphones, smartwatches, home lighting, or even coffee makers or fridges can be connected to the internet. The same goes for devices on a communal level, such as traffic lights, surveillance cameras, parking spots, etc. The sum of all these devices results in a continuous stream of large amounts of data. This raises the challenge of storing and processing the data efficiently.

The applications of IoT are endless. In [78], an overview of the applications of IoT is presented. They divide the applications into 14 domains: Transportation, Smart Home, Smart City, Lifestyle, Retail, Agriculture, Smart Factory, Supply chain, Emergency, Health care, User interaction, Culture and Tourism, Environment, and Energy. Among them, we can find smart homes, and smart cities which were described in more detail in Section 2.1.

In Figure 2.1, the architecture for IoT is described. In the bottom layer, the ”things” are collecting data, such as sensors or cameras, or even data from social media platforms such as Twitter or Facebook. The second layer is the network communication layer connecting the devices to the above layer that is the middleware layer containing the storage and analytic power. In the top layer, is the application layer which can be applied to several different domains which are discussed below, for instance, smart city.

Because of all the different sources of data in IoT, there are a huge number of different types of data. Some data is discrete, some are continuous, some data is automatically generated and some might be generated by humans. In [20], IoT data is classified in the following areas:

- **Radio Frequency Identification (RFID):** This is identification and tracking using radio waves where RFID tags can be inserted into objects and used to transmit and receive information.

- **Addresses/Unique Identifiers:** Objects in IoT need to be uniquely identified with IP addresses. As the number of IoT devices grows, so does the number of IP addresses that are needed.

- **Descriptive data about objects, processes, and systems:** Metadata is data about data and is essential to enable users to find and access the appropriate data. Metadata is not just collected about objects, but also about processes and systems.

- **Positional Data and Pervasive Environmental Data:** Provides the location of particular objects with GPS or a local positioning system.

- **Sensor data - Multidimensional Time-Series Data:** Much of the data in IoT is collected from sensors and enters through wireless sensor networks (WSNs). Sensor data is most often at a format of time-series because data about the environmental status at a location is captured at a specific time. Often data from sensors are also wanted to be queried related to a certain time interval.

- **Historical Data:** As time passes data captured from the sensors becomes historical. Volumes become a challenge and therefore it is important to decide which data should be kept in the systems. Data can be archived in data warehouses if it is needed frequently and stored in less accessible structures if data is rarely accessed.

- **Physics Models - Models that are templates for reality:** Physics models will need to be represented so that they can be accessed and used in algorithms as needed by the applications.

2.2.1 Time-Series Data

Time-series data is data that is collected over a time interval as points over a sequence of time. Often the time-series data is stored in the database with indexing on the timestamp rather than an id. Though time-series data comes in many different formats like expressed earlier, some properties are true for all times series [44]:

- The time-series data have no relations to each other. Though a lot of the time-series data can highly correlate.
- The data points are immutable. This means that as the data is generated and stored, there is no need for updating or modifying of the data points.
- Data points typically arrive in a timely order, so that the functionalities of the storage mechanisms only needs an append functionality.
- Time is the dominant and primary index, and the time interval between the data points is usually in a regular and fixed interval.

One of the most dominant sources of data in IoT is sensors. Sensor data enters IoT through WSN [20]. WSNs are easy to set up for monitoring all sorts of events and standards have been developed to support the setup of these.

Some sensors continuously monitor the status of some environmental phenomena, for example, temperature or air conditions. In this case, decisions have to be made on how frequently the data should be captured, at every measurement, or at certain time intervals. In some cases, it might be useful to obtain data only when the sensor is queried.

2.2.2 Big Data and IoT

Big data became a popular term a few years ago, as more applications are continuously generating more data. A set of data is considered ”big” when it meets the “three Vs” requirements: Volume, Variety, Velocity [33]. Two other characteristics have been added to the V’s, Veracity, and Value. The five Vs of big data is explained below:

- **Volume:** The volume of data refers to the size of data managed by the system. Data that is generated automatically, like data from IoT devices or sensors are typically voluminous. Also, data generated from other fields withing smart cities can be considered voluminous, such as traffic congestion with monitoring of vehicles or public transport.
- **Variety:** Big data includes structured, semi-structured, and unstructured data. The sources of data are coming in from all types of places, such as click-streams on social media, location data, image data from satellites, e-mails, or videos to name some. In smart cities, this is most definitely the case as already explained in Section 2.1.

- **Velocity:** Velocity refers to the speed at which data is created, accumulated, ingested, and processed. For the data to be considered as big data, the velocity of the generated data must be large and the time intervals at which new data is generated is small.
- **Veracity:** Veracity has two features: the credibility of the source, and the suitability of data for its target audience. The data coming in from numerous sensors and devices have a variety of trustworthiness. The data must go through some degree of quality testing and credibility analysis. Many sources can be uncertain, incomplete, and inaccurate, this goes for especially sensors, which often generate low quality and untrustworthy data.
- **Value:** The data that are generated and analyzed by the system must be able to generate some value for the organization which gathered the data to be considered as big data. In a world where data is gathered everywhere, the question if the data can generate value might only be dependent on the techniques of processing and analyzing the data, though the credibility of data sources can decrease its value.

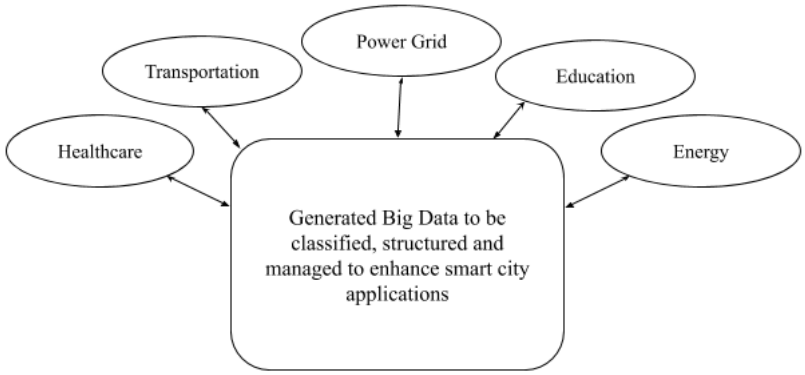

Figure 2.2: The relationship between big data and smart city.

IoT may be classified as big data as all the first three properties described above is present. Big data is an important domain within smart cities as well. In Figure 2.2 the relationships between big data systems and smart city applications, such as smart transportation, smart grid, and energy is demonstrated. Though big data have almost endless capabilities, it comes with a set of issues and challenges. In the next section, requirements for the management of IoT data are presented and discussed.

2.2.3 Data Management Requirements

As IoT is a complex research field, many properties must be present to manage the IoT data. In this section, requirements for the management of the IoT data relevant to database systems are presented. Seven requirements for data management in IoT applications were considered in [6].

1. **Data heterogeneity:** As already mentioned in the section on IoT the data coming from IoT devices is highly heterogeneous. The database systems that are used in IoT applications must be able to handle this.

2. **Semantic interoperability:** Different devices, agents, and applications in IoT must be able to exchange data and knowledge. In IoT, the same data can be used in many applications. Hence, the database should be easily understood by the user to that they should have a uniform format with modeling languages, query languages, and so on.

3. **Scalability:** A scalable data management system is a system that can execute large requests with low response times and redistribute data on the new hardware if necessary.

4. **Real-Time Processing:** IoT applications are often reliable for processing real-time data. The data will need to be stored and processed in a highly responsive manner, which means that performance is important.

5. **Security:** Privacy and security are important in IoT data management because the data which is gathered by IoT devices might contain private and sensitive information.

6. **Spatial data handling:** A lot of the data in IoT are data generated by moving devices, such as mobile phones. Hence there is a need for being able to handle geospatial data describing the devices with relation to the geographic location in a spatial referencing system (e.g GPS).

7. **Data aggregation:** Aggregation of data coming from multiple sources are required to both analyze the data and generate valuable information, and to eliminate redundancy so that only the most critical and useful data is stored.

In this thesis, the handling of spatial data is not the focus and the will not be investigated any further than to chart the capabilities of the databases to handle this. Furthermore, some other characteristics have been established in [82]. The following properties of IoT data must be treated by the database system to improve the efficiency of the storage:

1. **Massive data:** This point has already been mentioned in the above requirements.

2. **Data is ordered:** This point is closely related to time-series data, as data from for instance sensors often are marked with a timestamp. It is natural to order this data and insert it in the correct order in the database.

3. **Time based data retrieval:** Typically queries on IoT data are related to time and the database system should be able to handle queries based on time intervals.

4. **Data rarely changes:** After a sensor or other IoT devices have read data and is inserted into the database, there is rarely a need to change it. This means that consistency is typically not an issue, because the user will very unlikely want to retrieve old data instead of new data.

5. **IoT data expires:** As previously mentioned, IoT data is most often used to monitor and detect specific events. When this is the case old data is not useful and can be deleted or aggregated.

Another research examining the functionalities required by an IoT database system is found in [10].

1. **Simultaneous users support:** The applications which contain the IoT data, often require to be used by a large number of users simultaneously. The database system must be of a type that can handle high workloads and multiple requests at the same time.

2. **Clustering, management tools:** A cluster management tool is a software program that helps manage a group of clusters through a graphical user interface or by accessing a command line. With this tool, it is possible to monitor nodes in the cluster, configure services, and administer the entire cluster server [3]. In IoT, this is important because of the vast amount of data and the need to monitor nodes in the cluster to make sure that the data is managed properly.

3. **Asynchronous notifications:** In IoT database system asynchronous notifications are important so that a server is not blocked for long periods, waiting for receivers to complete notification handling.

4. **Triggers and Stored procedures:** A trigger in a database system is a stored procedure that runs automatically when various events happen, such as an update, insert, or delete. Stored procedures are a defined set of SQL statements stored in a relational database management system so that it can be used by multiple programs.

5. **Transactions and transaction rollbacks:** Rollbacks are important for a database system to be able to recover from a crash. By rolling back the database can the restored to a consistent state.

6. **JSON data types:** There are several advantages of utilizing JSON data types. The syntax is widely known and easy to use, in addition to providing fast responses. It has a wide range of supported browser compatibility and the applications made with the coding of JSON does not require much effort to make it all browser compatible. Furthermore, JSON is a well-suited tool for sharing data of any size, even videos, and audio, which proves to be well suitable for a smart city domain with highly heterogeneous data.

7. **Aggregation functions:** Are already discussed in previous requirements.

2.3 Storage Systems

In this section, different types of storage systems are introduced. First, the relational database MySQL is presented. IoT in relational databases is discussed before NoSQL databases are defined. The two NoSQL databases Cassandra and MongoDB are presented before providing a short section about Time-Series Database Systems (TSDB). The IoT

data management requirements, and requirements for time-series data, found in Section 2.2.3 and Section 2.3.3 are summarized in Table 2.2 and 2.3 and discussed concerning MySQL, MongoDB and Cassandra. Furthermore, database systems found in smart city literature is presented. Finally, IoT and cloud computing is discussed and some popular cloud platforms are mentioned.

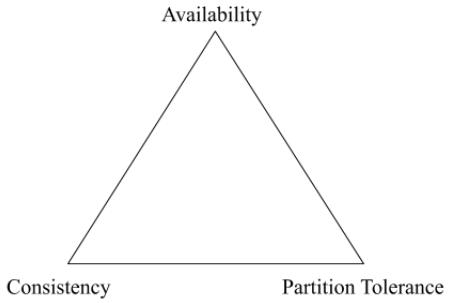

Figure 2.3: CAP theorem illustration.

CAP Theorem CAP is used to describe some desired properties of databases with replication. Each letter refers to one desired property, Consistency (among replicated copies), Availability (in the system for reading and write operations) and partition tolerance (in the face of the nodes in the system being partitioned by a network fault)[32, p. 889]. The CAP theorem states that it is not possible to achieve all three of the desirable properties - Consistency, Availability, and Partition tolerance, in Figure 2.3 - at the same time in a distributed system with data replications [32, p. 889]. In a system with data replication as the ones in NoSQL systems, concurrency control is much more complex and thereby also keeping up the ACID properties of transactions that are running concurrently. In NoSQL systems, one would, therefore, need to choose two of the properties which are the most important for your application. Many NoSQL systems choose to exclude the consistency property and rely on the system being eventually consistent. Recall in Section 2.2.3, it was mentioned that for applications relying on time series data, the consistency was not seen as an important factor.

2.3.1 RDBMS

Relational databases are widely known as SQL databases, named after the query language used in relational databases [45]. The main construct of representing data in the relational model is a relation. A relation consists of a relation schema and a relation instance. The relation instance is a table, and the relation schema describes the column heads for the table [70].

MySQL

MySQL is one of the most popular Relational Database Management Systems (RDBMS). It is open-source and is used by many big companies across the globe. MySQL enables users to deliver high-performance and scalable Online Transaction Processing (OLTP) applications. ”MySQL is an ACID-compliant database and aims to deliver reliability, performance, and ease of use,” [22].

Indexing is an important feature in MySQL. Indexes are used to find rows with specific column values quickly. Without an index, MySQL must begin with the first row and then read through the entire table to find the relevant rows. The larger the table, the more these costs [24]. Most MySQL indexes are stored in B-trees, except indexes on spatial data types, that use R-trees. Spatial data in relational databases can be saved according to dedicated projects and needs, even by adding additional columns with coordinates describing the facts gathered. In [68], standards are explored to minimize problems that might occur when using this method in databases. These problems can involve difficulties in exchanging or transferring data. Most relational databases have extensions for handling spatial data, such as Oracle Spatial. MySQL uses MySQL Spatial extension, which implements a standard OpenGIS and only provides 2D dimensions without reference sets [68]. New in MySQL is an Enterprise Edition that enables users to interact with the database through Document Store, making the database able to handle both SQL and NoSQL [23]. This function will not be investigated further in this thesis as open-source software is the focus. But it is worth mentioning because this new addition to the MySQL system could have a big potential within IoT data management.

Storing time-series in MySQL

Whenever new data is added to a MySQL or relational database, a new record is created. For time-series data this new record will have the timestamp as a key. This leads to increased cardinality of the table whenever new data is added [28]. The more data that is stored in the tables the bigger the table gets, which again increases the cardinality as each update to the database is an update of a new time-series entering the storage. One approach that has proven to be reliable for handling high cardinality in time-series data, is using a B-tree structure for indexing of data.

IoT in relational databases

Though NoSQL databases are widely accepted as the norm for IoT data management, relational databases have been studied for IoT applications by many researchers. In Section 2.3.5 about database management systems in smart cities, some smart city platforms have implemented their pilots with relational databases such as MySQL. [31] even found that the performance of MySQL and MongoDB on their platform performed equally well.

In IoT, an important factor, which already have been established is the importance of having scalable applications. The database management system needs to scale well as the data entering IoT applications are increasing. In relational databases, vertical scalability is supported. Vertical refers to the ability to increase the performance of a single node by adding resources such as memory or processors to the already existing node. The

main advantage of vertical scalability is that it consumes less power compared to running on multiple servers and reduces administrative efforts as we need to handle and manage only on system. ”Moreover, the implementation is easier, reduces software costs and application comparability is retained,” [37].

Another property that is important to handle within IoT is faster data retrieval. In relational databases, tables are linked together. To retrieve queries, join operations have to be made, creating views. This process is time-consuming, unlike NoSQL databases which often store data in the form of objects that are retrieved with all related data, eliminating the time-consuming join process.

Many IoT applications are gathering and responsible for storing sensitive data that need to be protected. Properties like security, authentication, and integrity are important for the database management system to handle so that sensitive data are not compromised. As relational databases are experienced and mature, most of these issues are already taken care of in SQL. Such security concerns may not be addressed in many NoSQL systems, according to [72].

2.3.2 NoSQL

Non-relational databases have grown in popularity the recent years and are generally referred to as Not Only SQL. Most NoSQL systems are distributed storage systems and have a high focus on performance, availability, data replication, and scalability as opposed to an emphasis on immediate data consistency, powerful query languages, and structured data storage [32, p.883], like relational systems as described in the section above.

Below, characteristics of NoSQL systems from [32] are defined and discussed whether they are relevant to IoT data in this thesis:

- **Scalability:** Two types of scalability exist for distributed systems. Vertical and horizontal scalability. In NoSQL systems, horizontal scalability is used and when the system needs expanding, more nodes are added to expand data storage as the volume grows. On the other hand, vertical scalability refers to utilizing the same number of nodes to expand. This was already explained in Section 2.3.1. In IoT, some applications might find that the amounts of data at some point, will exceed the capacity of the fixed number of nodes and there is a need for horizontal scalability.

- **Availability:** Many systems using NoSQL databases are reliable for being highly available. To meet these requirements, data is replicated over two or more nodes to make sure that if one node fails, data is still available on other nodes. Replicating data over several nodes can also improve read performance, but on the other hand, write performance might be compromised because each update must be applied to multiple nodes. This can be solved by not requiring serializable consistency, so eventual consistency can be used.

- **Replication Models:** In NoSQL systems either master-slave or master-master replication is used. Each technique has its advantages and disadvantages which can affect the consistency of the database similar to the previous point.

- **Sharding of Files:** Sharding, also known as horizontal partitioning is used combined with replicating the shards to improve load balancing. Besides, it can improve data availability. In many NoSQL applications, files can have millions of records and these records can be accessed concurrently by thousands of users. To offset the load on one single node, the records are partitioned on several nodes.

- **High-Performance Data Access:** To achieve higher efficiency of finding individual records among millions of data records in a file, either range partitioning or hashing on object keys is used.

- **Not requiring a schema:** In most NoSQL systems a semi-structured, self-describing data is used to provide higher flexibility as opposed to relational systems. In IoT, this point is a very important as data from IoT devices is highly heterogeneous.

- **Less powerful query language:** Many applications using NoSQL systems do not require powerful queries. Many only require the CRUD (Create, Read, Update, Delete) operations, and having the rich query language such as MySQL for relational databases might not be necessary for IoT.

- **Versioning:** Some NoSQL systems require functionality to store the timestamps of when the data was created. In IoT, especially time-series data, a timestamp is already existing related to the data being created when the data is recorded by, for instance, a measuring device.

In Table 2.1, an overview of relational and non-relational databases is shown. It demonstrates what types of systems the database systems are most suited for, scenarios in which they are suited for, how they scale, as well as different data models, which are already mentioned in the above sections. Note that IoT applications are listed in the scenarios of use of NoSQL applications, whereas for SQL the use case scenarios are more centered around management systems.

Categories of NoSQL systems

Though there are some common properties for all NoSQL databases as demonstrated above in Table 2.1, four main categories of NoSQL systems exist. Each of these has different properties and are suited to serve different applications:

- **Document-based NoSQL systems:** In document-based NoSQL systems data is stored as collections of similar documents. There is no requirement to specify schema, the documents are specified as self-describing data. The documents can have different data elements and they can be stored in various formats, such as XML (Extensible Markup Language) or JSON (JavaScript Object Notation) [32, p.890]. MongoDB or CouchDB [2] are examples of document-based NoSQL systems. Because of the flexibility of the schema of document-based NoSQL systems, they can are initially seen as a good fit for highly complex IoT data.

[2] https://couchdb.apache.org/

Table 2.1: Overview of relational (SQL) and non-relational (NoSQL) database, as presented in [53].

	NoSQL or non-relational	SQL or relational
BEST FOR	• Handling large, unrelated, indeterminate, or rapidly changing data. • Schema-agnostic data or schema dictated by the app. • Apps where performance and availability are more important than strong consistency. • Always-on apps that serve users around the world.	• Handling data that is relational and has logical and discrete requirements that can be identified in advance. • Schema that must be maintained and kept in sync between the app and database. • Legacy systems built for relational structures. • Apps requiring complex querying or multi-row transactions.
SCENARIOS	• Mobile apps. • Real-time analytics. • Content management. • Penalization. • IoT applications. • Database migration.	• Accounting, finance, and banking systems. • Inventory management systems. • Transaction management systems.
SCALE	• Scales data horizontally by sharding across servers.	• Scales data vertically by increasing server load.
DATA MODEL	• Database types: key:value, document, column, and graph databases. • Stores data depending on database type.	• Database type: tables of rows, grouped into relations. • Uses Structured Query Language (SQL). • Stores data as rows in tables; related data stored separately and joined for complex queries.

- **NoSQL key-value stores:** The idea behind key-value stores is relatively simple, the key is a unique identifier associated with a data item and is used to locate this data item rapidly. The value is the data and can have a different format for each database system. In many key-value stores, there is no query language but rather a set of operations that can be used by the application programmers [32, p.896]. Examples of key-value data stores are DynamoDB [3] and Voldemort [4].

- **Column-based or wide column NoSQL systems:** The basic idea behind column-oriented databases is that one attribute of a set of datasets is stored in one unit (in columns), as opposed to row-oriented store (like in SQL) where the attributes are

[3] https://aws.amazon.com/dynamodb/
[4] https://www.project-voldemort.com/voldemort/

stored in one unit [50]. An example of column-based NoSQL systems is Hbase [5].

- **Graph-based NoSQL systems:** In graph databases, the data is represented as a graph. A graph is a collection of nodes and edges representing the types of entities and relationships they represent. Neo4j [6] is an example of a graph-based NoSQL system [32, p.904].

In the preceding section, the focus is on two open-source NoSQL databases, which often are researched when studying IoT data management, Cassandra and MongoDB. Some important architectural features are explained as well as a discussion related to IoT and time-series data.

Apache Cassandra

Cassandra is a database that is hard to categorize into one of the four categories of NoSQL systems mentioned above. It is an open-source distributed database that is written in Java. Cassandra can fit both structured and unstructured data because of its ability to scale elastically as well as linearly [71]. Cassandra Query Language (CQL) is the query language of the Cassandra database. The syntax of CQL closely resembles the syntax of MySQL. The performance of Cassandra increases as the number of nodes in the cluster increases. One of the main strengths of Cassandra is the fast write speed while not sacrificing read efficiency [30].

Cassandra is one of the databases that are derived from BigTable among HBase and RocksDB etc. [29]. The storage structure that is used in these systems is called Memtables. In Memtables recently inserted data stays in memory. The Memtables are not flushed to disk until it is either full, reached the maximum age, or the user specifies to do so. After the flush, the data is put into Sorted-String Tables (SSTables) on disk. This structure provides Cassandra with high write performance. Compaction of SSTables is the operation of merging two or more SSTables. It is primarily necessary so that when a read operation is performed, there is no need to seek multiple SSTables. Because the SSTables are already sorted, the operation is I/O bound. That being said, if compaction is performed frequently, it becomes much too I/O intensive which might affect the system performance.

Another important feature of Cassandra is indexing. An index makes it possible to access data in Cassandra using attributes other than the partition key. Using indexes provides benefits such as fast, efficient lookup of data matching a given condition [27]. The indexes provide fast retrieval of data when queried by the row key without the need of creating explicit indexes. In Cassandra, each node maintains all indexes of tables it manages. Besides, each node knows the range of keys that are managed by the other nodes. This way requested rows are located using only relevant nodes. The indexes are located in a separate table from the data in which they belong to. Another important thing to know is that additional indexes can be made over different fields [1].

Storing Time-Series Data in Cassandra Cassandra has several patterns for storing time-series data and is often mentioned as one of the primary choices when using NoSQL

[5] https://hbase.apache.org/
[6] https://neo4j.com/

databases to store time-series data in the literature. Data in Cassandra is written sequentially to disk. The simplest model for storing time-series data is creating a wide row of data for each measurement [63]. By storing data this way, in a partition, cells are by default naturally ordered by the cell's name. So the time-series data will get data sorted "for free" [66]. Here each new timestamp and measurements get its column. E.g.:

```
SensorID, {timestamp1, value1}, {timestamp2, value2}
... {timestampN, valueN}
```

A Figure demonstrating how this is stored in Cassandra is shown in Figure 2.4.

Cassandra has a feature that enables the possibility of limiting the row size. This is useful if the time interval at which the data is stored into the database is very small. If this is the case it can be difficult to store the entire data in one single row, because the number of columns would be endless as time goes. To handle this problem it is possible to split a row into multiple rows, e.g. rows with the same ID, but a new row for each date to be added to the primary key, in case the number of measurements each day is large. An illustration of how this is done is shown in Figure 2.5 and below:

```
SensorID, Date {timestamp1, value1}, {timestamp2, value2}
... {timestamp24, valueN}
```

Another useful feature when handling IoT data is to be able to remove no longer needed data. In Cassandra, a feature is offered to handle this automatically [71]. In [63], this functionality is referred to as the roll-back function. This can be useful in the case that the storage is limited. In Cassandra, this feature is implemented with a Time To Live (TTL) feature. The TTL can be created upon data insertion. When the TTL is up, the data will be deleted from the database.

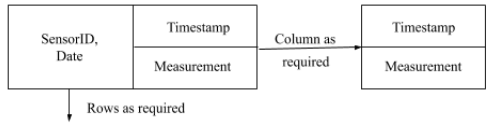

Figure 2.4: Model of storing data for a single measurement per row in Cassandra.

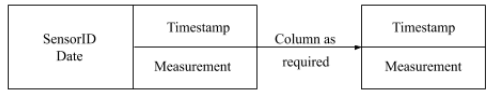

Figure 2.5: Model of storing data per date and associated measurements in rows in Cassandra.

Limitations Cassandra is known for its fast writes, but some problems are still related to the write performance of Cassandra. As insert/append operations perform extremely well, updates are conceptually missing in Cassandra, though an update function exists. When

a value needs to be updated, in reality, a new entry is added with a younger timestamp. Having this done many times over will take up much space. Also, it can affect read performance as Cassandra might have to read through a lot a data on a single key to check for the newest replica. That being said, compaction is performed to merge such data and free up space. Also, Cassandra has some problems regarding reads. Querying data that is not a partition key creates problems. The problem is that secondary indexes and SSTable Attached Secondary Index (SASIs) don't contain the partition key, which means there's no way to know what node stores the indexed data. It leads to searching for the data on all nodes in the cluster, which is neither cheap nor quick. Another possible problem of reads in Cassandra is the use of bloom filters. Though this storage structure can help retrieve IoT queries faster, it can also lead to a waste of time and resources while searching in the wrong places because of their probabilistic nature. Finally, Cassandra has a problem with reading tables with many columns. If there are thousands of columns stored, the reads will be very slow or might even not be possible to be performed at all [14].

MongoDB

Like previously mentioned MongoDB is a document-based NoSQL database that is highly scalable and available. Documents are stored in collections and the documents provide high flexibility of storage. Also, MongoDB has a rich query language compared to other NoSQL databases, it implements many features of relational databases, such as sorting, secondary indexing, range queries, and nested document querying [47].

When new files are created in MongoDB everything is flushed to disc, releasing memory. To increase the performance of MongoDB indexing of documents, is used. All documents are indexed automatically, but additionally, the user can specify indexes. All indexes uses a B-tree structure [1]. Although indexing is important to achieve highly efficient reads, it may harm insert performance. Different types of indexes are offered in MongoDB. Single field index, compound indexes, multikey index, hash index, and text indexing.

The two most important capabilities of MongoDB is durability and concurrency. Durability is enabled by the creation of replicas. Master-slave replication is the replication strategy used in MongoDB. The master can read or write, while the slave serves as a backup. If the master node goes down, a slave with more recent data is promoted to master. The replication of data from the master to slave is asynchronous, which means that all updates are not done immediately.

Writes in MongoDB can either be done by INSERT, DELETE, or UPDATE operation. For INSERT and DELETE, MongoDB either inserts or removes the corresponding document keys from each index in the target collection. An UPDATE operation may result in updates to a subset of the index on the collection, depending on the keys affected by the update [62].

In MongoDB, like in Cassandra a feature making it possible to remove data that is no longer useful is implemented. The model to do so is implemented by adding a "Capped" feature which is a parameter based on the number of records. This feature is specified during the creation of a collection in contrast to Cassandra. Capped is a Boolean feature that can be viewed as a fixed-sized collection that supports high-throughput operations. Once the fixed-size collection is filled up, the oldest documents can be overwritten by

newer documents.

Storing time-series Data in MongoDB There are some different strategies when storing time-series data in MongoDB. We already know that in MongoDB, data is stored in documents. One strategy that would most resemble the model utilized by RDBMS is storing one document per event.

```
{ timestamp : "2020-02-12 22:04:23",
  type : "Temperature",
  value : 21.1},
{ timestamp : "2020-02-12 22:04:24",
  type : "Temperature",
  value : 20.9},
{ timestamp : "2020-02-12 22:04:25",
  type : "Temperature",
  value : 20.8}
```

Using this strategy would lead to complications when reading data from the database. Reading one minute of data would take 3600 seconds which would cause a lot of reading latency if using a sensor transmitting data every second like in the example above [49].

Another strategy, taking advantage of embedding within a document is to store one document per minute. An example of how this can be done is shown below. When using this strategy, the number of reads would be eliminated drastically and latency would decrease. Besides, it is optimized for storage because writes will be faster as: for updates (one per second) than for inserts (one per minute). Because instead of allocating a new insert, a smaller update using the update() method in MongoDB, of the existing document will take place.

```
{ timestamp_minute : "2020-02-12 22:04:23",
  type : "Temperature",
  values :  {0:21.1, 1:20.9, 2:20.8, ... , 59:21.0}}
```

Two strategies can be used if data needs to be stored in a more compact format, with one document per hour. In the first strategy, seconds are stored from 0 to 3599 for an hour. With this approach, there is an extra workload during update operations. Another approach is storing data at an hourly level in documents, but with nesting documents for each minute. This approach is shown in an example below. It requires much fewer steps for updates that the first approach.

```
{ timestamp_hour : "2020-02-12 22:04:23",
  type : "Temperature",
  values :  {
    0 : {0:21.1, 1:20.9, 2:20.8, ... , 59:21.0},
    ... ,
    58 : {0:20.1, 1:20.2, 2:20.3, ... , 59:20.0},
    59 : {0:20.0, 1:20.1, 2:20.3, ... , 59:21.0}
    }
}
```

Other advantages of using this compound method are not just the read latency, but the collection size. The size of the collections is according to [57] up to ten times the size if you store 28 days of time-series data in seconds as opposed to minutes. Also, the size of indexes leads to poor scalability if time-series data are stored per second, where the storing of minutes is 60 times smaller than per second.

Limitations According to [43], MongoDB reports scalability constraints as the amount of data reaches hundreds of GigaBytes (GB). For the case of IoT and smart city time-series data, this could show to be a problem. Another limitation of MongoDB is the memory consumption of MongoDB due to the setup of MongoDB which stores the key name along with every document. Besides, unlike in relational databases, joins are not possible in MongoDB so in the case that joins are important to the application, this could be a huge drawback. That being said, in IoT applications, like already discussed, queries are often not too complex, and joins might not even be required.

2.3.3 Time-Series Database Systems

Time-series data is generated in IoT by devices at a large scale from billions of devices all over the world. Some properties are special when handling time-series workloads compared to typical database online transaction processing (OLTP) workloads. The writes in time-series data are typically insets and not updates. Hence, the writes are insert-heavy and are related to recent time ranges. Reads are typically on continuous time-ranges, not random, and usually happen independently of writes and rarely in the same transaction. Also, time-series insert volumes tend to be huge and are accumulated more quickly than OLTP [80]. These properties, together with the characteristics presented in Section 2.2.1, make the handling of time-series data different than OLTP. Because of this, the database system might need some other characteristics than traditional databases. In [34], a list of requirements for a database that stores data recorded as time-series has been made. The requirements are listed below:

1. **In-memory for value alerting:** As the time-series data arrive in the system, the data have to be compared to a trigger immediately to ensure that any threshold number is not met. An example is a temperature measurement device, which could trigger an alarm if the temperature reaches over 30 degrees.

2. **In-memory for trend alerting:** Data arriving at the database could also be compared to previous values to detect trends. In the case of the previous example, an alarm could be triggered in the case that a temperature sensor reports more that 10 degrees increase in a small amount of time.

3. **In-memory for applications and dashboards:** Applications and dashboards need live data in memory to support rapid and continual display updates.

4. **Fast access for real-time analytic, machine learning and AI:** Business intelligence programs, machine learning algorithms and AI programs need fast responsiveness from the data store. This may require data to be in-memory, heavily cached, or efficiently accessed from a combination of memory and disk.

5. **High concurrent for real-time analytic:** A wide range of people need to be able to access the latest readings of time-series data (which might be the most valuable data) at the same time without the limitation of how many queries that arrive at once.
6. **High capacity:** The database storing the time-series needs to be both fast and scalable to accommodate huge amounts of data.
7. **Standard SQL functions:** The SQL standards are well defined and the authors of this article argue that the SQL-standard has a wide and optimized performance that is shared across companies. High performance for these SQL standards are thereby a key asset for a time-series database.
8. **Custom time-series functions:** In addition to the SQL standards, time-series databases can benefit from having custom time-series functions that are not supported by SQL. Examples include functions to toss incoming data where sensor reading has not changed significantly, to save space, or to only return the records with the lowest and the highest reading from large datasets of records to limit the amount of data that needs to be fetched from the database.

Some of the above-mentioned points can be argued upon, depending on the need for the applications that will be using the time-series data. Point number three could vary, as some applications will need consciously update, while others may only need to take out a report every hour or once every 24 hours etc.

Also, some additional points that can be made are the possibility of querying the sensors creating the data directly, and getting reports with the status of the devices only on query demand.

The emerge of IoT has lead to the need for database systems to handle time-series data. The recent years, databases dedicated to handling time-series data have become popular. Below some time-series database systems are mentioned. OpenTSDB was one of the first technologies to address the need to store time-series data on a very large scale. The database is schema-free and built on Apache HBase. According to [41], writes have milliseconds precision and scales to millions of writes per second. It can increase capacity by adding nodes. Another time-series database that is worth mentioning is InfluxDB [11]. InfluxDB is also open-source and is optimized for heavy writing loads. It is schema-free and built upon NoSQL principles.

Amazon has developed a fast, scalable, and fully managed time-series database for IoT applications that make it easy to analyze and store data. Built-in the application is analytic functions such as smoothing, approximation, and interpolation. According to [5], Timestream gives scale and speed to process trillions of events a day, with up to 1000 times faster query performance at 1/10th the cost of relational databases. The data is organized by time intervals, unlike relational databases which reduce the amount of data that needs to be scanned to answer a query. To improve performance, inserts and queries are executed in separate processing tiers which eliminates resource contention. Other technologies to handle time-series data have also emerged the recent years, such as the Chronos Software, an in-memory background-based time database for key-value pairs [79].

Another database system built to handle time-series data is TimescaleDB. ”TimescaleDB is an open-source database built for analyzing time-series data with the power and convenience of SQL — on-premise, at the edge, or in the cloud,” [81]. It is implemented as

an extension of PostgreSQL. The TimescaleDB allows the database to take advantage of many of the attributes of PostgreSQL such as reliability, security, and connectivity to a wide range of third-party tools.

2.3.4 IoT requirements on database systems

Table 2.2: Overview of requirements for IoT data management related to the database systems MySQL, MongoDB and Cassandra.

Requirements/ Database Systems	MySQL	MongoDB	Cassandra
Data heterogeneity		✓	✓
Semantic interoperability	✓	✓	✓
Scalability	Vertical	Horizontal	Horizontal
Real time processing		✓	
Security	✓	✓	✓
Spatial Data Handling	✓	✓	
Data aggregation	✓	✓	✓
Data is ordered (based on time stamp)			Partially
Time based data retrieval	✓	✓	✓

Table 2.3: Overview of requirements for data management of time-series data, related to the database systems MySQL, MongoDB and Cassandra.

Requirements/ Database Systems	MySQL	MongoDB	Cassandra
Expired data should be deleted	✓	✓	✓
Compare to threshold	✓	✓	✓
Compare with recent data	✓	✓	✓
Possibility of viewing and dashboards		✓	✓
Analytic services	✓	✓	✓
Many users access at the same time	✓	✓	✓
Built in time-series functionality			

In this section, a summary of the requirements for database systems to handle IoT and time-series data is presented. A table listing the requirements found in the literature review is given for the database systems MongoDB, Cassandra, and MySQL. They are compared in light of the requirements in Table 2.2 and Table 2.3. See Section 2.2.3 to see a detailed definition on each of the points in the tables.

Data heterogeneity is important within IoT because of the need to handle different types of data. MySQL and relational databases have a less flexible format because of tabular relations. On the other hand, the main idea behind NoSQL database systems is scalability and flexibility. For non-relational databases, different categories have different properties. For instance, document-based databases, such as MongoDB, shown in Table 2.2, or Key-Value stores, have high support for heterogeneous data by providing schema-less structure. However, the NoSQL database Cassandra, also shown with a check-mark

in Table 2.2, have less flexible schema and have lower support for heterogeneous data because of the column-oriented structure of the system.

An issue closely related to data heterogeneity is semantic interoperability. Data heterogeneity can create problems with interoperability and integration of different systems. ”Interoperability among components of large-scale, distributed systems is the ability to exchange services and data with one another. It is based on agreements between requesters and providers on, for example, message passing protocols, procedure names, error codes, and argument types. Semantic interoperability ensures that these exchanges make sense—that the requester and the provider have a common understanding of the “meanings” of the requested services and data,” [40]. As seen in Table 2.2 all three database systems are capable of providing semantic interoperability. MySQL has completed interoperability requirements, and is by nature semantic, as data is must be stored in predefined columns and which provides unambiguous meaning. In all NoSQL databases, semantic interoperability can be achieved by using for instance ontologies [6].

As previously explained, MySQL, and NoSQL databases offer different types of scalability. Though MySQL offers vertical scalability, we argue that horizontal scalability is ultimately required in IoT as the amount of data generated is continuously growing, and adding resources such as hardware or additional storage can only get you so far. Though MongoDB is considered a highly scalable database system, in [1], researchers found that when the number of records in storage escalates, Cassandra outperforms MongoDB in terms of execution time. That being said, experiments done in the above-mentioned research, was not performed using time-series data, which might change the outcome.

MySQL has no support for processing real-time data coming in a large scale. MongoDB has support for real-time analytics and writes on its web page that the tool provides ”Lightweight, low-latency analytics. Integrated into your operational database. In real-time.” [60]. To this day Cassandra does not support real-time analytics, but, can easily provide this functionality by using tools such as Spark [16]. Spark paired together with Cassandra will be able to offer functionalities that are not provided either by Spark or Cassandra alone [16].

Security is an increasingly important factor within databases. Especially within the IoT domain, security and privacy mechanisms is important because of sensitive data. In [72], authors argue that in terms of security, relational databases and SQL are better off than NoSQL database systems because of the system maturity of RDBMS. This article was written in 2015 and since then NoSQL database systems have matured. Cassandra reports having functionalities for TLS/SSL encryption clients and inter-node communication, client authentication, and authorization [35]. The same functionalities can be found in MongoDB [61].

As already mentioned in the section about MySQL, this database system, among other relational database systems, has extensions that make them able to handle spatial data. Most NoSQL databases, including Cassandra are according to [15], missing a feature to handle spatial data indexing and retrieval. Also, the Cassandra Query Language (CQL) is missing a spatial query feature. This makes the handling of spatial data in Cassandra highly inconvenient. In [15], a framework is designed and implemented to extend the CQL with spatial queries. A CQL-like syntax is defined to enable spatial functions while keeping the native CQL query syntax. In MongoDB, spatial data is handled by storing data as

a format named GeoJSON. GeoJSON is a format for encoding a variety of geographical data structures [47]. By following the GeoJSON format, MongoDB is enabled to compute a geospatial index on the geographic information by computing a geo-hash for the coordinate pairs.

Aggregation operations process data records and return completed results. In MySQL, aggregation functions are provided by the operations MIN, MAX, COUNT, SUM and AVG. The same aggregation functions can be found in Cassandra (CQL). However, in Cassandra there are limited possibilities of what can be aggregated. For a column to be aggregated, it has to be contained in the partitioning key of the table. This functionality ensures that Cassandra maintains high performance in terms of execution times of queries, but limits the possibilities of aggregation functions. MongoDB provides much more powerful data aggregation functions than Cassandra. In MongoDB, values are grouped from multiple documents and a variety of operations on the grouped data can be performed to return a single result. There are three ways to perform aggregations in MongoDB [58]. The aggregation pipeline, the map-reduce function, and single-purpose aggregation methods. In the aggregation, pipeline documents enter a multi-stage pipeline that transforms the documents into an aggregated result. Map-reduce operations have two phases. The map phase processes each document and emits one or more objects for each input document. The reduce phase that combines the output of the map operation. Finally single-purpose operation aggregate documents from a single collection.

Time-based data retrieval is possible for all three databases. That being said, for all the systems, this has the possibility of being a time-consuming task. However, Cassandra has one property that provides the database with an advantage compared to the other two databases. In Section 2.3.2, it was mentioned that the records in Cassandra where naturally ordered by the timestamp. This might give Cassandra an advantage of querying time-based data over MongoDB and MySQL. Closely related to this property is the possibility of the database system to order the rows or documents in the natural order of the time-series being created. It has already been mentioned that Cassandra has this property, but a functionality like this, is not known to the author for MongoDB and MySQL.

The first requirement in Table 2.3, the possibility of the database systems to delete data from the system, is achieved by all three systems. The second point is easy for any programmer or developer to check the data toward a threshold before insertion in the database. The third point in the table, on comparing the newest data point with recent data, might be a bit more complex. Now the most recent data must have the possibility of being kept in memory or cache of the systems so that the system does not have to scan through all time-series to find the data with the most recent timestamps. In MySQL, one technique of making this more efficient is by implementing a sharding policy and a script that can move older data from the active list to the archive node and updating the timestamps as the data is moved, as described in [75, p. 554]. In Cassandra, most recent updates are kept in memory, in a Memtable until its flushed to disk. When comparing the new measurement towards the most recent measurement, Cassandra can read from memory, without having to utilize I/O. One solution to handling this in MongoDB, is by using change streams like described in [55].

Viewings and dashboards are important in terms of time-series data, to be able to see the development and monitoring of time-series. MongoDB has developed a service, Mon-

goDB Atlas [59], which is a cloud MongoDB service. ”Dashboards are a collection of charts assembled to create a single unified display of data. Each chart shows data from a single MongoDB collection or view, so dashboards are essential to attain insight into multiple focal points of data in a single display,” [59]. Some of these properties can be accessed using the explain() operator. Cassandra has a command-line tool to that can access similar properties and characteristics. Also editing the configuration files of Cassandra can modify the configurations and clusters of the database. To the knowledge of the author of this thesis, MySQL has no viewing or dashboard functionality built-in. However, there might exist other external services making viewings and dashboards without having to program them.

Because of the nature of IoT systems, several intuitions, users or systems have to be able to connect to the database simultaneously. In MySQL, the maximum allowed simultaneous connections is 100000 [25]. Cassandra, has the availability property of the CAP theorem explained in Section 2.3 and Figure 2.3. Cassandra achieves this by replication sets. MongoDB also has replica sets and though it is not known for having the availability property of the CAP theorem, rather the focus is on consistency and partition tolerance. This might lead to less support for availability in MongoDB than for Cassandra.

The final requirement listed in Table 2.3, is having built-in functions for time-series. MySQL provides a set of functions for manipulating dates and timestamps, such as generating the hour based on a timestamp. However, it misses support for taking advantage of ordered data. Though, trough the provided functionalities, MySQL can obtain simple queries, on larger datasets, the execution times will likely be slow. The same functionalities are available in MongoDB, but Cassandra lacks support for efficient access to the attributes of timestamps. Further functionalities are missing in the two NoSQL databases as well. Being able to handle time-series data in database systems that are not specifically dedicated to this purpose, such as TSDB, is more about cleverly structuring the data so that the time-series can be retrieved efficiently.

2.3.5 Database systems in smart cities

In this section, database systems in existing smart city platforms are reviewed. In the previous project [36], though investigating a set of smart city platforms from several different locations across the world, the description of database systems being used in the pilots is missing for most smart city platforms. Much focus in the smart city research community has been concentrated on the ICT architecture, with the newer research focusing on technology management in D2C architecture. Below related platforms are presented.

The SmartSantander project in the city of Santander in Spain [74], initially created their platform with a relational database. As the project matured the researchers have replaced the relational database system with a non-relational database system. Their chosen database was MongoDB to handle heterogeneous data more flexibly as well as the need for better performance.

An IoT-Fog-Cloud based architecture for Smart City is discussed in [31]. The authors describe a prototype in the case for a smart building scenario where the cloud solution has been implemented with both a relational and non-relational database. MySQL and MongoDB are chosen from each of the two types of database systems and through experiments using the IoT-Fog-Cloud architecture it was found that the performance of both

SQL database and MongoDB performed equally well.

Another proposed smart city architecture using the F2C architecture can be found in [12]. The storage must store and stack up the data from IoT devices after the processing which are used by a decision-making server later. Data storing and processing play a vital role in the comprehension of a smart city. Hence, the proposed architecture of [12] makes use of several techniques, HDFS [7], HBase, HIVE [8], to make the data storing and processing easily.

In [17], a big data platform for smart cities is initiated. The big data platform is made on top of the existing smart city platform Smart Santander,[74], described above. After finding that the Santander platform did not provide the capability of storing, analyzing and processing generated data from the sensors collected from the city of Santander. The author implies that this is a research field that is lacking in most smart city platforms even though it is an important subject. The research that does exist tends to concentrate on presenting a high-level platform architecture design. Sensor data collected from IoT agents are in this platform saved in a NoSQL database as JSON documents. After investigating different NoSQL databases, the authors were left with the following candidates for the chosen database: CouchDB, CouchBase [9], MongoDB, and HBase. A document-based database was chosen because all the data from the SmartSantander testbed is wrapped up as JSON objects [17]. The following reasons behind in the final version selecting CouchDB was given by the authors: ”CouchDB can support incremental map-reduce for real-time processing. This means that view results can be updated incrementally as the database changes. This is a feature that is missing in MongoDB, there, MapReduce results are written to a collection on disk and or resulted in the query and not updated unless the associated jobs get executed again. Second, external documents can be notified with changed documents real-time, given a special feature in CouchDB called changes notifications.” Both these features are listed as important to the architecture design of the system and have therefore been important when selecting a database for the system. An additional layer, for even more intensive and scalable data processing can be added to the architecture.

A new software, OpenFog, proposed in [8], from 2019, mentions the use of Microsoft database services for handling the IoT data from sensors. Azure SQL Database Edge [10] is a tool that aims to move the analytic power closer to the edge of the network, providing AI capabilities and enables the system to process data at the edge before forwarding it to the data center and cloud storage to optimize both network bandwidth and cost [52].

Another solution is chosen in [13]. For the database on the edge some type of relational database system is preferred to provide ACID properties. For the central database system, *new SQL databases*, should be considered because of the huge amounts of data generated across many data centers and its aggregation will require big data technologies for management. To offer fast and efficient retrieval of data, it provides across zone services. New SQL databases are preferred over traditional relational databases and even NoSQL databases. Among this class of database systems we find databases such as Google Span-

[7] http://hadoop.apache.org/

[8] https://hive.apache.org/

[9] https://www.couchbase.com/

[10] https://azure.microsoft.com/en-us/services/sql-database-edge/

ner [11] or ClustrixDB [12].

Like initiated at the beginning of this section, there is not a whole lot of literature about the specifics about storage systems in smart city research. One reason for this might be that many smart city platforms are still in an early stage, a lot a the platforms are early-stage pilots that have not jet generated the amount of data that makes distributed storage systems necessary. In Table 2.4, an overview of the database systems in the smart city platforms above, is presented to give an overview of databases used in smart cities.

Table 2.4: Overview of storage systems used in different smart city platforms.

Author	**Database System**
L.Sanchez et al., 2014	Went from using SQL to using NoSQL
J. Dutta, S. Roy., 2017	SQL and NoSQL performs equally well
M. Babar, F. Arif., 2017	HBase, HDFS, HIVE
B. Cheng et al., 2015	NoSQL Document based - using CouchDB
M. Antonini et al., 2019	Microsoft database services
N. Z. Bawany, J. A. Shamsi., 2015	Google Spanner, ClustrixDB

2.3.6 Cloud

The recent years it has become apparent that cloud computing is a crucial component of IoT and that the cloud can provide valuable application-specific services in many application domains. ”Cloud computing has in many ways been one of the most significant shifts in modern ICT and service for enterprise applications,” [39]. According to [73], ”Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management effort or service provider interaction.” Several cloud providers are entering the market to offer application-specific IoT based services in their clouds [73]. In 2016, [73], claims that there are at least 49 different IoT cloud platforms in the global market. In the previous four years, one can only assume that this number has risen. The same author has surveyed several popular IoT clouds. They are discussed in light of solving several service domains that are relevant in the IoT domain, such as application development, device management, system management, heterogeneity management, data management, tools for analysis, deployment, monitoring, visualization, and research.

But why is the cloud suitable for IoT? As the number of IoT devices is growing each day and the amounts of data are increasing as a result of that, there is a demand for increased storage and processing power. Companies drifting IoT applications and platforms must rely on scalable IT infrastructure in which they initially had to develop, build, and support in-house. Nowadays, it is increasingly popular for companies to buy these services from companies with expertise in cloud computing to ensure quality, availability, and maintenance of this IT infrastructure. ”Cloud computing provides fundamental sup-

[11] https://cloud.google.com/spanner

[12] https://mariadb.com/products/clustrixdb/

port to address the challenges with shared computing resources including computing, storage, networking, and analytical software; the application of these resources has fostered impressive big data advancements", [83]. Cloud-based technology has to cope with this new environment because dealing with big data for concurrent processing has become increasingly complicated. MapReduce is an example of processing in a cloud environment, it allows for the processing of many datasets stored in parallel in a cluster [39]. Cloud storage services offer virtually unlimited storage with high fault tolerance which provides potential solutions to address big data storage challenges. However, hosting big bata on the cloud is expensive given the size of data volume [83], because the cloud providers often make the companies pay for the data storage and processing power that is used.

Some popular databases are developed by big technology companies and some of these are also the most well know. Companies such as Microsoft have developed Azure which has almost unlimited services, also for IoT with their IoT Hub [13] which is said to fit no matter the industry or size of the organization. Azure IoT offers devices, tools, data analytic as well as security capabilities. Also, Azure has developed more specific services to deal with the new and complex IoT paradigm like the Azure SQL Database Edge [52] or Azure IoT Edge [14]. The first was briefly mentioned in Section 2.3.5. These services are at the time of writing this only available as a preview edition, but it illustrates that the industry is realizing the need for technologies specific to IoT related issues and domains. Another popular cloud provider is Amazon Web Services (AWS)[15] which claims to be compatible whether you are looking for computing power, database storage, content delivery, or further functionality, and can help build applications with increased flexibility, scalability, and reliability. Other technology companies with popular IoT cloud platforms include IBM Watson IoT Platform [16], Google Cloud Platform [17], Oracle IoT Cloud Service [18] and Cisco IoT Cloud Connect [19]. Some of these cloud providers have built-in databases to function seamlessly. As an example, Amazon has their DynamoDB which is a managed NoSQL database, or Amazon key-space which is optimized for Apache Cassandra. Azure has developed their CosmosDB [20] which have APIs to integrate easily with SQL, MongoDB, and Cassandra among others.

2.4 Related work

In this section, some related work on experiments and performance testing of databases used in IoT, smart cities and time-series data is presented. In [71], Cassandra and MongoDB are compared to provide efficient storage of discrete time-series data. The authors conclude that Cassandra would be the best choice for collecting and analyzing large volumes of time-series data in sequence as the data is inserted into the database sequentially. This property provides fast retrieval of data when queried by the row key without the need

[13] https://azure.microsoft.com/en-in/services/iot-hub/
[14] https://azure.microsoft.com/en-us/services/iot-edge/
[15] https://aws.amazon.com/
[16] https://www.ibm.com/internet-of-things/solutions/iot-platform/watson-iot-platform
[17] https://cloud.google.com/solutions/iot/
[18] https://docs.oracle.com/en/cloud/paas/iot-cloud/index.html
[19] https://www.cisco.com/c/en/us/solutions/service-provider/iot-cloud-connect/index.html
[20] https://docs.microsoft.com/en-us/azure/cosmos-db

for creating explicit indexes. Cassandra is a good fit in case of the time interval between data gathering is very low and the amount of data to be stored is high because the database system provides easy scalability. On the other hand, the authors mention that MongoDB by its rich query language, is also a good choice for storing time-series data by providing good scalability which is an important requirement. In this research the conclusions where drawn based on theoretical backgrounds for how to store discrete time-series data in MongoDB and Cassandra, and no experiments where provided.

A database system performance evaluation for IoT applications is done in [10]. This research focuses on open-source databases using the three database systems MySQL, MongoDB, and PostgreSQL. MongoDB showed promising results for handling IoT data as opposed to MySQL. According to experiments provided by authors of [10], for a small number of records inserted, PostgreSQL outperforms MySQL as well as MongoDB. However, for a large number of inserted records, MongoDB does better than the other two. For insert queries, PostgreSQL performs best for inserts of a large number of records with the lowest execution time. The authors conclude that MongoDB is not a good option for aggregation function execution of IoT data.

Authors of [72], have conducted experiments on MongoDB and MySQL to compare them for IoT applications. The study is based on the time to execute Select and Insert queries against a varying number of records and threads. The research finds that in some scenarios MongoDB required less response time compared to MySQL, but in other cases MySQL responses where stable compared to MongoDB. This is why the authors conclude that choosing a database for IoT depends on which query is mostly used and the requirements of the application.

Another paper researching performance of NoSQL and relational database management systems for large IoT data are [42]. The authors have selected MongoDB and MySQL for performance testing. Experiments of inserting of data showed that MongoDB has almost four times better throughput than MySQL. MongoDB also performs better in terms of reads. The NoSQL system MongoDB provided better performance for both storing and processing the data is concluded by authors.

Which NoSQL database that is best suited for IoT applications is investigated in [6] by comparing five of the most popular NoSQL databases; Redis [21], Cassandra, MongoDB, Couchbase, and Neo4j. They find that all the five database systems mentioned above had covered many of the data management requirements. This comparison shows that Couchbase, followed by MongoDB and Neo4j provides the best capabilities, as Redis has issues with scalability and Cassandra, some issues with handling spatial data.

Performance evaluation of MongoDB, Cassandra, and HBase for heterogeneous IoT data storage is done in [69]. The results showed that in MongoDB structured data types have superior run-time performance and throughput in comparison to the two other databases. Cassandra showed better run-time and throughput for the unstructured data. In terms of resource use, MongoDB showed the best performance with both data types. According to [69], about 80 percent of data generated by IoT devices in the form of unstructured data. This means that it cannot be stored in relational, SQL format. Unstructured data can be anything from text, images, video, or emails. JSON format is considered to be structured data which in the conducted experiments was generated by a sensor measuring air quality.

[21] https://redis.io/

MongoDB and Cassandra are compared and evaluated by a set of different experiments in [1]. The characteristics that were analyzed are data loading, only reads, reads, and update mix and read-modify-write, and only updates. The results of the research showed that with an increase in data size MongoDB started to reduce its performance. Meanwhile, Cassandra got faster as the data increased. After running experiments on read/update performance, the author concludes that Cassandra performs better than MongoDB in terms of updates, providing lower execution times independently of the database used in the evaluation. MongoDB fell short with increasing amounts of records, while Cassandra seemed to perform better and the authors conclude that Cassandra performed better in almost all scenarios tested.

In [30], the main focus of the work was to investigate if Cassandra could provide good performance in an IoT system. Queries of an IoT real-time environment is used to test the querying processing time by comparing two different types of architectures in Cassandra. The first keeping all the data in one Cassandra table, and the second, keeping multiple tables for each specific application that sends events. From a theoretical viewpoint, we know that the best way of organizing the data is through the creation of one table per application because the table will have much less data per table and less data will have to be filtered and the memory will not be full. The results of the research showed exactly what the researchers had assumed, for all three types of queries tested, the strategy of storing data in separate tables outperformed the other strategy. The authors write in their conclusion of their study that: ” we can conclude that Cassandra can be used on an IoT platform as the main database system because it contains the necessary characteristics to handle the overall requirements of these platforms” [30].

In another study of Cassandra for IoT workloads, Cassandra was compared with ”the latest generation re-design of Cassandra, ScyllaDB, meant to deliver bleeding-edge performance on modern multi-core machines”, [46]. The authors found through their experiments that ScyllaDB has a 10 times improvement in throughput over Cassandra. However, given more write-intensive IoT workloads, Cassandra may be more amenable to IoT applications as results of experiments showed that both the number of writes per second and the write latency, was better in Cassandra than ScyllaDB.

Table 2.5 summarizes the findings from the related work on IoT and time-series data from the literature.

Table 2.5: Summary of database systems from literature that have been evaluated for IoT usage. Databases that where considered for usage is marked with X and the selected database system in the paper is marked with V.

Author	**MySQL**	**PostgreSQL**	**MongoDB**	**Cassandra**	**Redis**	**CouchDB**	**Neo4j**	**HBase**
C. Asiminidis et al., 2018	X	X	V					
S. Rautmare, D. Bhalerao, 2016	V		V					
G .Kiraz, C. Togay	X		V					
S. Amghar et al.			2.	X	X	1.	3.	
E. S. Pramukantoro et al. 2019			V	X				X
D. Ramesh, 2016			X	V				

Chapter 3

Methodology

In this chapter, the methodology and the research strategies used in this project will be presented. An approach from [67] is used to define the research methodology in this thesis.

3.1 Research Strategies

In this research, 4 research questions have been defined. To answer each one of the research questions, different research strategies are used. In the coming sections, the research strategies used in this thesis are explained. Figure 3.1, shows the model of approaches for the overall research process with the research strategies used in this thesis outlined in red.

3.1.1 Literature Review

The first part of a literature review is to research ideas and to discover relevant material about possible research topics within the selected research area. In this thesis, initial literature was researched to find a more specific field within smart cities and IoT and to gain an understanding of what databases should be reviewed in more detail. ”The second part of a literature review is to gather and present evidence to support the claim that there have been created some new knowledge,” [67]. Some objectives of a literature review are placing the research in the thesis in a context of research that has already been published. Also, objectives include, pointing to strengths and weaknesses in work that have already been published, point to gaps that have not previously been identified or address by researchers[67].

3.1.2 Experiments

”An experiment is a strategy that investigates cause and effect relationships, seeking to prove or disapprove a causal link between a factor and an observed outcome,” [67]. The

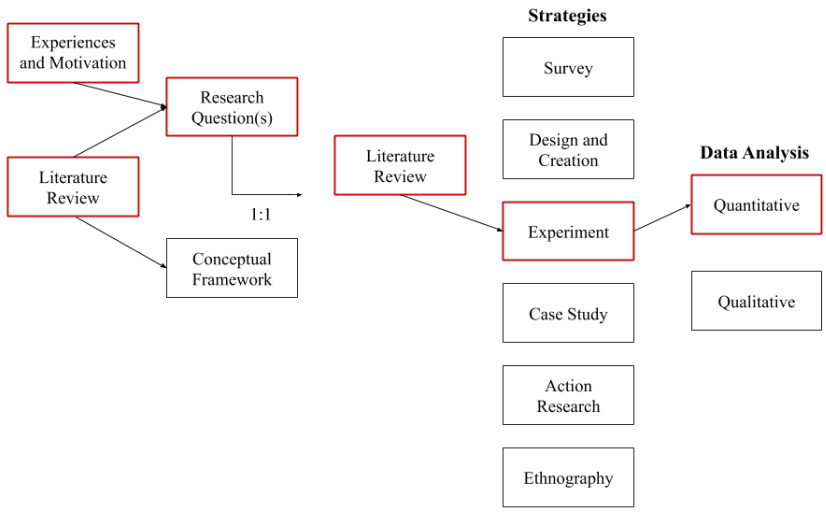

Figure 3.1: Model of approaches for a research process, from [67]. The red outline indicates the strategy used in this master thesis.

experiments in this thesis are based on previous work and experiments found in the literature review. Also, the experiments are based on the requirements of database systems within IoT. The research strategy involves the following steps:

1. **Observation and measurement:** precise measurements of the execution times of queries are recorded.
2. **Manipulation of circumstances:** four different queries are tested to obtain broader measurements of the database system performance.
3. **Repetition:** the experiments are repeated many times to be certain that the measured outcomes are not caused by some arbitrary factors.
4. **Explanation and prediction:** the results can be explained from the theory from which the hypothesis was derived from.

3.1.3 Data Analysis

To be able to draw some conclusions based on the generated data, the data must be analyzed. The idea behind data analysis is to look for patterns in the data and draw conclusions. In the case of this study, only quantitative data is generated. Hence, quantitative data analysis will be performed.

Quantitative Data Analysis

Quantitative data are data that are gathered typically from experiments or surveys, in this project's case, experiments, and is data based on numbers. "A typical and simple way to

analyze quantitative data is to use tables, charts, or graphs. This makes it easier for the reader to observe patterns in the data, [67]."

More patterns can be found by providing simple statistical techniques, such as finding the average value of a set of measurements. The average value will be obtained in this work, from the repeating of the same experiments several times to exclude arbitrary factors from the final results, as explained in the previous section.

Chapter 4

Experiments and experimental set up

In this chapter, the experimental set up will be described in detail. A description of the dataset used in the experiments is explained. Also, the experiments and the queries are explained for both MongoDB, Cassandra, and MySQL.

4.1 Experimental set up

The experimental setup was created with the following characteristics: (1) The operating system was macOS Catalina Version 10.15.3; (2) The machine had a 1.6 GHz dual-core Intel Core i5 with 3MB shared L3 cache, and 4GB of RAM. (4) The dataset will be accounted for in the next section.

To set up the experiments, Python has been used. To integrate the databases with Python, Cassandra-driver, Pymongo, and MySQL cursor for Pyhton was setup. See Table 4.1 for the version used for Pyhton and each of the three databases.

Table 4.1: Versions used in the experiments.

Database	Version
Pyhton	3.6.3
MySQL	8.0.19
MongoDB	4.2.3
Cassandra	3.11.6

4.1.1 Dataset Overview

When finding a dataset to use in the experiments, it was important that is was on the form of time-series data and had a large number of records. In this experiment, a dataset

provided by the NYC Taxi and Limousine Commission (TLC) has been used. The data in the dataset was collected by technology providers authorized under Taxicab & Liberty Passenger Enhancement Programs (TPEP/LPEP) [1].

The dataset on the webpage includes monthly records of all the yellow cab rides from 2009-2019. Each month having approximately seven million records. In the experiments, records from January 2019 are downloaded and used. To simulate the properties of time-series data defined in Section 2.2.1, duplicate records containing the same time-stamps have been removed, so inserted into the database is 2 million records. A screenshot to illustrate the format of the dataset is illustrated in 4.1.

```
1,2019-01-01 00:59:47,2019-01-01 01:18:59,1,2.60,1,N,239,246,1,14,0.5,0.5,1,0,0.3,16.3,
2,2018-12-21 13:48:30,2018-12-21 13:52:40,3,.00,1,N,236,236,1,4.5,0.5,0.5,0,0,0.3,5.8,
2,2018-11-28 15:52:25,2018-11-28 15:55:45,5,.00,1,N,193,193,2,3.5,0.5,0.5,0,0,0.3,7.55,
2,2018-11-28 15:56:57,2018-11-28 15:58:33,5,.00,2,N,193,193,2,52,0,0.5,0,0,0.3,55.55,
2,2018-11-28 16:25:49,2018-11-28 16:28:26,5,.00,1,N,193,193,2,3.5,0.5,0.5,0,5.76,0.3,13.31,
2,2018-11-28 16:29:37,2018-11-28 16:33:43,5,.00,2,N,193,193,2,52,0,0.5,0,0,0.3,55.55,
1,2019-01-01 00:21:28,2019-01-01 00:28:37,1,1.30,1,N,163,229,1,6.5,0.5,0.5,1.25,0,0.3,9.05,
1,2019-01-01 00:32:01,2019-01-01 00:45:39,1,3.70,1,N,229,7,1,13.5,0.5,0.5,3.7,0,0.3,18.5,
1,2019-01-01 00:57:32,2019-01-01 01:09:32,2,2.10,1,N,141,234,1,10,0.5,0.5,1.7,0,0.3,13,
1,2019-01-01 00:24:04,2019-01-01 00:47:06,2,2.80,1,N,246,162,1,15,0.5,0.5,3.25,0,0.3,19.55,
1,2019-01-01 00:21:59,2019-01-01 00:28:24,1,.70,1,N,238,151,1,5.5,0.5,0.5,1.7,0,0.3,8.5,
1,2019-01-01 00:45:21,2019-01-01 01:31:05,1,8.70,1,N,163,25,1,34.5,0.5,0.5,7.15,0,0.3,42.95,
1,2019-01-01 00:43:19,2019-01-01 01:07:42,1,6.30,1,N,224,25,1,21.5,0.5,0.5,5.7,0,0.3,28.5,
1,2019-01-01 00:58:24,2019-01-01 01:15:18,1,2.70,1,N,141,234,1,13,0.5,0.5,1,0,0.3,15.3,
2,2019-01-01 00:23:14,2019-01-01 00:25:40,1,.38,1,N,170,170,2,3.5,0.5,0.5,0,0,0.3,4.8,
2,2019-01-01 00:39:51,2019-01-01 00:48:02,1,.55,1,N,170,170,1,6.5,0.5,0.5,1.95,0,0.3,9.75,
2,2019-01-01 00:46:00,2019-01-01 00:49:07,1,.30,1,N,107,107,1,4,0.5,0.5,1.06,0,0.3,6.36,
```

Figure 4.1: A fraction of the csv file downloaded from NYC TLC.

The dataset has a total of 18 properties, shown below.

- VendorID
- tpep_pickup_datetime
- tpep_dropoff_datetime
- passanger_count
- trip_distance
- RatecodeID
- store_and_fwd_flag
- PULocationID
- DOLocationID
- payment_type
- fare_amount
- extra

[1] NYC Taxi and Limousine Commission - Trip Record Data. 2020 Retrieved from https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page

- mta_tax
- tip_amount
- tolls_amount
- improvement_surcharge
- total_amount
- congestion_surcharge

4.1.2 Repetition of Experiments

Experiments are typically repeated many times to be certain that the measured outcomes are not caused by some arbitrary factor [67]. In [10], 10 repetitions are applied to the experiments, and [42], reports having done "more than three repetitions". Other experimental works researched in the literature in this thesis, have either not repeated the experiments or not mentioned the number of repetitions. Performing too many repetitions can rarely be a disadvantage other than being time-consuming for time-consuming queries. The experiments in this research are repeated 30 times to make sure that eventual irregularities are minimized.

When doing experiments with database testing and the experiments are repeated several times, it is important to avoid that the results from previous experiments are cached in memory. In the case that previous results are cached, the run times for the experiments might end up being smaller. Several steps have been done to try to eliminate the caching results, demonstrated below. Also, between each repetition of the queries, ten records are inserted into the database to avoid that the last handled data in the database is the data read by the previous repetition.

Caching

In MongoDB, this is a functionality that is enabled to retrieve data quicker. To make sure the experiments give the correct results we make sure that none of the previous results are cached. MongoDB keeps the most recently used data in RAM. If indexes are created and the working dataset fits in RAM, MongoDB serves all queries from memory. MongoDB does not cache the query results to return the cached results for identical queries [54]. Besides, MongoDB might cache the query plan. This can be disabled by clearing the execution cache plan of all collections by the following query:

```
>db.colllction_name.getPlanCache().clear()
```

Cassandra includes integrated caching and distributes cache data around the cluster. In Cassandra, one can use CQL to enable or disable caching by configuring the caching table property. Configuring caching in Cassandra determines how much space in memory Cassandra allocates to store rows from the most frequently read partitions of the table. The cache option can be disabled in Cassandra config file, as well [26].

```
WITH caching = {
    'keys' : 'NONE',
    'rows_per_partition' : 'NONE'
}
```

In the cassandra.yaml configuration file, the caching settings is set to 0 to disable the caching of rows.

```
row_cache_size_in_mb = 0
```

In MySQL, the query cache stores the text of a SELECT statement together with the corresponding result that was sent to the client. If an identical statement is received later, the server retrieves the results from the query cache rather than parsing and executing the statement again. The query cache is shared among sessions, so a result set generated by one client can be sent in response to the same query issued by another client. The query cache can be useful in an environment where tables that do not change very often and for which the server receives many identical queries. That being said, according to [21], the query cache is removed from MySQL version 8.0.

Data Analyzis

Because the experiments are repeated, the formulas below are used to calculate the mean (Equation 4.1), variance (Equation 4.2), and standard deviation (Equation 4.3) to analyze the experiments.

Formula for calculating the mean of a set of measurements, where N is the total set of measurements and μ is the mean of the N measurements.

$$\mu = \frac{1}{N}\sum_{i=1}^{N} x_i \tag{4.1}$$

Formula for calculating variance (σ^2). How spread out the measurements are from the mean value.

$$\sigma^2 = \sum_{i=1}^{N} \frac{(x_i - \mu)}{N} \tag{4.2}$$

Standard deviation (σ) is the square root of the variance.

$$\sigma = \sqrt{\sum_{i=1}^{N} \frac{(x_i - \mu)}{N}} \tag{4.3}$$

4.1.3 Recording Execution Time

To measure the performance of the queries in the experiments, the execution times are recorded. To retrieve the execution times the following have been done:

MySQL

The below code demonstrates that the execution time in MySQL is obtained by using the profiling property.

```
mysql_session.execute("Set profiling = 1")
mysql_session.execute("SHOW PROFILES")
```

MongoDB

In MongoDB, the explain method is used to obtain the execution time of the queries. The following is used to reterieve the execution time in milliseconds.

```
mongodb_time = mongo_cursor.explain()
    ["executionStats"]["executionTimeMillis"]
```

Cassandra

The execution time in Cassandra, is obtained by enabling the TRACE property, as showed below.

```
cassandra_time = cassandra_session.
    execute(query, trace=True)
trace = q.get_query_trace()
cassandra_time = trace.duration.total_seconds()
```

Some webpages argue that the use of the TRACE property in Cassandra can increase the execution time. Because a lack of better possibilities, the TRACING property has still been used in the experiments.

4.2 Experiments

The experiments in this project consist of four queries, related to the theory in the literature. The queries are explained below and are listed in the Appendixes. Appendix A demonstrates how records are inserted in MySQL, Appendix B shows MongoDB queries and Appendix C lists the insertion and queries in Cassandra.

4.2.1 Query 1

In Section 2.2, it was mentioned that often data from sensors are wanted to be queried over a certain time interval. Also, in Table 2.2, time-based data retrieval is listed as a requirement for IoT data management. Query 1 aims to test this requirement on the three databases by querying a time interval of seven days from the dataset described above.

4.2.2 Query 2

Query 2, is testing the database's ability to perform aggregation functions over a large set of records. Data aggregation is often listed as a requirement in the literature on IoT data management, and is therefore included in the experiments in this project. Query 2 finds the sum of the attribute $tipamount$ from all records in the dataset.

4.2.3 Query 3

When analyzing data, it is interesting to see the average or a sum for a specific time interval. In Query 3, we want to find out if the time interval together with aggregation functions could affect the performances of the databases. The aggregation functions are tested, but unlike Query 2, fewer records are aggregated. The average of the attribute $tripddistance$ over the same time interval from Query 1 is retrieved.

4.2.4 Query 4

Finally, Query 4, performs a grouping operation, as well as an aggregation function on the grouped data. A typical query in a smart city could be to group data from each sensor together. That is why, in this experiment, the vendors with the same VendorID, which could resemble a sensor ID in a sensor dataset, are grouped and aggregated on the $tipamount$ attribute over all records in the dataset.

Chapter 5

Evaluation

This chapter contains a description of the goals and expected results for the experiments in this project. The results of each query are presented with a column diagram demonstrating the mean execution times. Furthermore, a graph representing the execution time of every repetition of the experiments is shown to be able to observe irregularities and evaluate the results. Also, tables showing the maximum and minimum execution times, as well as the variance and standard deviation for each database in each of the queries. Finally, the results are discussed in light of the expected results and literature from Chapter 2.

5.1 Goals and expected results

The overall goal of the experiments in this thesis is to gain an understanding of what database can be suited for IoT data. In this project, the focus is on the management of time-series data and testing the three databases MySQL, Cassandra, and MongoDB for typical queries that can be relevant for a smart city use case, and also testing the requirements for data management of IoT data. Below the expected results of the databases are discussed.

5.1.1 MySQL

For Query 1, MySQL is not expected to perform particularly well. In MySQL, the primary keys are not ordered by default. This means that to find the queried time interval, MySQL has to scan through all rows in the table. The same is true for the other queries as well, where MySQL has to scan through all the two million records to select the rows for aggregation, which is a time-consuming task in itself. Also, according to [21], the MySQL version used in the experiments in this project does not use caching of recent results. Hence, the execution times are expected to remain poor.

5.1.2 MongoDB

Similar for all the four queries in MongoDB is that it is expected that it will not live up to its full potential in terms of performance, due to not using indexes. However, MongoDB keeps the most recently used query in memory. Because of this, MongoDB is expected to perform relatively well, because it does not have to scan through all the documents to retrieve the query results. To try to diminish this advantage, as mentioned earlier, some additional documents are inserted into the databases between each repetition of the experiment, to put some other results in this memory. Looking at the literature in this thesis, in Section 2.4, MongoDB has performed relatively well for similar types and amounts of data. This could be an indicator that MongoDB will have adequate execution times in the experiments in this project as well.

5.1.3 Cassandra

Cassandra has the property of the cells by default being naturally ordered by the cell's name. This means that the time-series data will get data sorted "for free" [66], as mentioned in Section 2.3.2. Hence, Cassandra is expected to perform well for the queries retrieving the time intervals (Query 1 and Query3). However, Query 3 contains an aggregation of data, along with Query 2, and because there is no partition key or clustering key on either properties to be aggregated, the performance of the aggregations is expected to be slow. For Query 4, to be able to perform the GROUP BY operation, in Casandra, $VendorID$ was set to be a clustering key. This can be seen in Appendix C. Because of this Cassandra might perform better as the clustering key provides the ordering of data on the clustering key.

5.2 Results

5.2.1 Query 1

The results of Query 1 shows that the mean execution time for Cassandra when retrieving time intervals only, outperforms the other two databases. Figure 5.1, illustrates the mean execution time after 30 repetitions of Query 1, with a mean execution time of 0.100 seconds for Cassandra, while MongoDB has a mean value of 3.005 seconds, and MySQL 5.829 seconds. In Figure 5.2, it can be seen that the execution time of MySQL makes some large jumps and from Table 5.1 it is apparent that there is a large difference between the minimum and maximum execution times. These might have come from some unknown factor, and in Table 5.2, the variance, and standard deviation are demonstrated and the standard deviation for MySQL is quite high when seen in relation with the mean execution time.

5.2.2 Query 2

An interesting shift of what database that performs the best in terms of execution time is shown in this experiment. Query 2 finds the sum over all rows or documents in the dataset. Interestingly, the execution times are the opposite than the ones in the previous query.

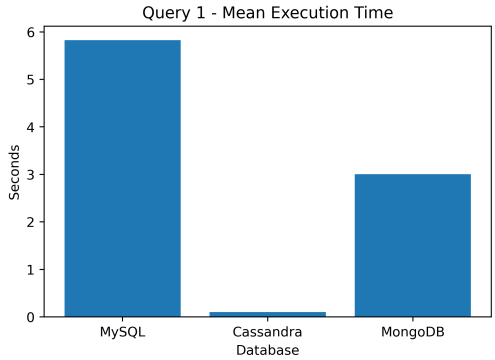

Figure 5.1: Mean execution time of 30 repetitions of Query 1.

Figure 5.2: Execution time for each repetition of Query 1.

Table 5.1: The maximum and minimum execution time of Query 1 in MySQL, Cassandra and MongoDB in seconds.

	MySQL	Cassandra	MongoDB
Max execution time	10.118	0.267	4.909
Min execution time	4.944	0.050	2.528

Table 5.2: The variance and standard deviation of Query 1 in MySQL, Cassandra and MongoDB.

	MySQL	Cassandra	MongoDB
Variance	1.240	0.002	0.248
Standard deviation	1.114	0.050	0.498

Cassandra now has the larges mean execution time with 18.748 seconds, while MySQL, executes the queries with an average of 2.345 seconds. From Figure 5.3, it is clear that MongoDB execution time is better than Cassandra and is closer to MySQL execution time

with a mean of 3.171 seconds. Notice that even the best execution time of Cassandra is more than double the worst execution times of MySQL and MongoDB. In Figure 5.4, it can be seen that in some repetitions MongoDB even performs better than MySQL. Though the execution times of Cassandra varies a lot in each repetition, because all of them do, the standard deviation in Table 5.4, is not too high. Table 5.3 shows that for all the queries the difference between the lowest and highest execution times is large.

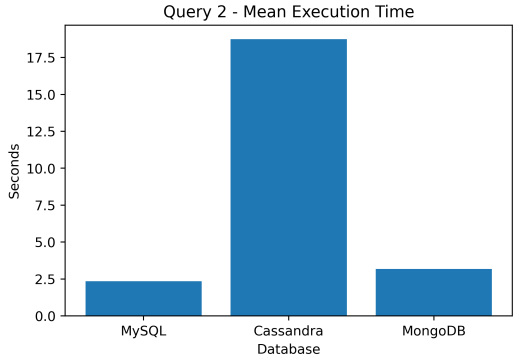

Figure 5.3: Mean execution time of 30 repetitions of Query 2.

Figure 5.4: Execution time for each repetition of Query 2.

Table 5.3: The maximum and minimum execution time of Query 2 in MySQL, Cassandra and MongoDB in seconds.

	MySQL	Cassandra	MongoDB
Max execution time	7.562	29.371	6.383
Min execution time	0.744	13.988	2.589

Table 5.4: The variance and standard deviation of Query 2 in MySQL, Cassandra and MongoDB.

	MySQL	Cassandra	MongoDB
Variance	3.164	17.613	0.589
Standard Deviation	1.779	4.197	0.768

5.2.3 Query 3

In Figure 5.5, the mean execution time of Query 3 is shown for MySQL, Cassandra and MongoDB. The chart shows the same tendency from Query 2, where MySQL has the best performance in terms of execution time with 1.211 seconds, followed by MongoDB at a mean execution time of 1.944 seconds, and finally Cassandra, 3.976 seconds. Notice that the execution time of all databases is less than the previous query, but the one that stands out is Cassandra, with an improvement of 14.772 seconds compared to Query 2. MySQL have a large drop in execution time after around seven repetitions, in Figure 5.6. This might be because of some caching mechanism in MySQL, which might have lead to improved execution time in MySQL when compared to MongoDB and Cassandra. In Table 5.5, this can also be observed with the difference of 3.599 seconds in maximum and minimum execution time in MySQL. As a result of these irregularities in the measurements, the standard deviation of MySQL measurements is high, Table 5.6. This could indicate that the results of this particular experiment might be considered to be repeated or at least looked at with a critical eye.

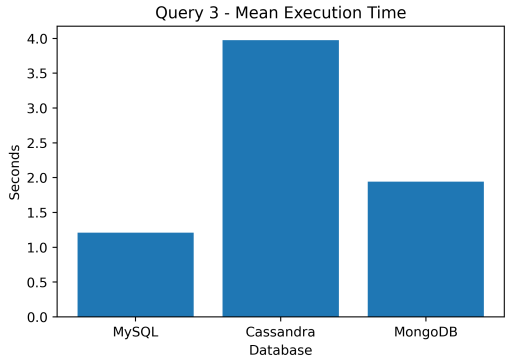

Figure 5.5: Mean execution time of 30 repetitions of Query 3.

Table 5.5: The maximum and minimum execution time of Query 3 in MySQL, Cassandra and MongoDB in seconds.

	MySQL	Cassandra	MongoDB
Max execution time	4.055	6.404	3.732
Min execution time	0.456	3.186	1.656

Figure 5.6: Execution time for each repetition of Query 3.

Table 5.6: Variance and Standard Deviation of Query 3 in MySQL, Cassandra and MongoDB.

	MySQL	Cassandra	MongoDB
Variance	1.061	0.841	0.226
Standard Deviation	1.030	0.917	0.476

5.2.4 Query 4

In Figure 5.7, it can be seen that the mean execution times of the three database systems are 2.951 seconds for MongoDB, which have the best performance for this query, followed by MySQL with a mean of 3.645 seconds and Cassandra execution time five times worse than MongoDB, with 14.809 seconds. From Figure 5.8, MySQL execution time stabilised after about 15 repetitions. This is why MySQL, like in the previous query has a relatively high standard deviation, which can be seen in Table 5.8. In Table 5.7 we see that execution times of MongoDB remain more stable than the other two database systems. MySQL has a difference of almost 9 seconds between the maximum and minimum execution times, while MongoDB, only has a difference a little above 2 seconds.

Table 5.7: Variance and Standard Deviation of Query 4 in MySQL, Cassandra and MongoDB.

	MySQL	Cassandra	MongoDB
Max execution time	10.548	19.083	4.901
Min execution time	1.352	13.342	2.760

Table 5.8: The maximum and minimum execution time of Query 4 in MySQL, Cassandra and MongoDB in seconds.

	MySQL	Cassandra	MongoDB
Variance	8.889	1.631	0.178
Standard Deviation	2.981	1.277	0.422

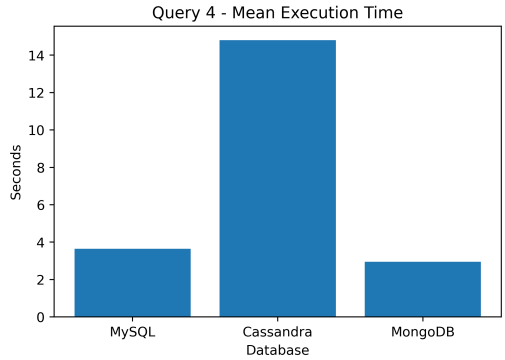

Figure 5.7: Mean execution time of 30 repetitions of Query 4.

Figure 5.8: Execution time for each repetition of Query 4.

5.3 Discussion

Recall at the beginning of this chapter it was stated that the overall goal in this thesis was to gain an understanding of what database can be suited for IoT data. More specifically, for time-series data which is a typical format the data in smart cities and IoT generates. To do so, literature on IoT and time-series data management was investigated, related to the database systems, MySQL, MongoDB, and Cassandra. Also, literature about related work on smart cities and IoT was reviewed to see what other researchers have been investigating.

First, Table 2.4 in Section 2.3.5, summarize the database systems used in literature about smart cities. Firstly, during the literature review, a lack of research on this specific field was noticed. Some of the articles shortly discuss the usage of databases, but it is poorly documented as well as few conducted experiments. However, one thing that seems to be common throughout the literature that does exist, is the need for scalable and flexible storage, and therefore it seems that the research community is headed towards using NoSQL systems, or at least, distributed database systems. This is in line with the trends in

the related work on IoT and database systems performance in Section 2.4, where NoSQL systems were highly represented. Another point to notice is that MongoDB was considered in all the IoT related work in Table 2.5. Though it did not always show the best performance under the conditions set up by the authors, it is interesting to notice that it is always up for consideration and shows that there is some acceptance in the community that MongoDB is a top competitor when it comes to IoT data management.

Furthermore, to summarize the requirements of IoT data management and time-series data, Table 2.2, and Table 2.3 where presented in Section 2.3.4. All three database systems show some positive and negative traits. MySQL is the only relational database among the three, and checks of fewer requirements than the other two systems. Also, the missing requirements, are properties that are of high importance for the database systems' ability to handle IoT data workloads. MySQL disability of handling heterogeneous data is a huge drawback, as well as not being able to scale horizontally.

MongoDB checks off on almost all requirements in the discussed tables, and shows great promise as a storage system being able to handle IoT requirements. The one point that is missing, is the property that data is naturally ordered on timestamp. To compensate for this, MongoDB indexing can provide efficient retrieval of data anyways, if the indexes are designed carefully. Cassandra, having some additional limitations when compared to MongoDB in terms of Table 2.2 and Table 2.3, still shows potential for being a database system that can work with IoT and time-series data. The limitations such as missing real-time processing can easily be provided seamlessly together with external systems, to fully cover the IoT requirements discussed in this work.

Table 5.9: Summary of mean execution times in seconds for each query in the experiments.

	MySQL	Cassandra	MongoDB
Query 1	5.829	0.100	3.005
Query 2	2.345	18.748	3.171
Query 3	1.211	3.976	1.944
Query 4	3.645	14.809	2.951

Now that a theoretical background about the databases has been established, experiments on queries related to the requirements on IoT and time-series data management is conducted. The results were presented in the previous section and in Table 5.9, the mean execution times for each query in the experiments are shown. In Section 4.1.3, it was mentioned that the TRACING property can contribute to higher execution times in Cassandra. Taking this into consideration, one might think that one factor of why the results of Query 2 and Query 4 are high is because of this. However, we argue that this is not the case, as Query 1 execution times are low, also when having the TRACING property in the query. Furthermore, as expected, for Query 1, Cassandra performed well. Cassandra outperformed the two other database systems with a mean execution time of 0.1 seconds. This implies that as expected, the retrieval of the time intervals in Cassandra is, indeed ordered, and the database does not have to scan through the entire table. The same tendency is seen in Query 3, also as expected, but with an increase of execution time, presumably, because of the aggregation function is a time-consuming task in Cassandra without the aggregated attribute as a primary key. The results imply that in terms of storing time-series, Cassandra

could be a promising candidate.

MongoDB, shows quite stable results for all the queries. However, MongoDB was expected to perform better for the execution times after the literature review, and also, it was expected to perform better than MySQL. For Query 1 and 4, the mean execution times were better than MySQL, and for query 3 and 4, though MySQL ended up with a lower mean execution time, from Table 5.4 and Table 5.6, it can be seen that the execution times of each repetition varied among which has the lowest execution time. One explanation of why MongoDB performs worse than expected is that indexes on time-stamps are not utilized and therefore does not live up to its full potential.

MySQL surprised positively in terms of execution times. Above, it was mentioned that MongoDB was expected to perform better than MySQL for all the queries. Similar to MongoDB, the performance showed to be quite stable throughout the queries, though execution times were generally not very low. Another thing that can be mentioned about the results of the MySQL experiments is that the execution times varied a lot. In Figure 5.6 and Figure 5.8, observe that execution times dropped drastically after approximately 8-9 and 15 repetitions. This could be an indicator that MySQL has a cache mechanism that is not accounted for in this project. Though the experiments in this thesis show promising results, MySQL has many theoretical drawbacks that have not fully been exploited through the experiments in this thesis.

5.3.1 Experiences

Through working with both the literature and the experiments, some observations have been made along the way. First, when searching for literature about the use cases of IoT data both in general and more specific to smart cities, there were not a lot of research to find. It seems as the literature and research diminish the user aspects of IoT while focusing on techniques, algorithms, and architecture to make the applications or platforms more efficient. A recommendation for the research community of this field is therefore to study or survey the actual use cases for the data that the systems are built upon, before deciding on design principals etc. about the IoT platforms.

Second, when working with the datasets to set up experiments, it was generally a more user-friendly experience working with MongoDB than the other two systems. The insertion of data into the database was faster, easier, and takes less coding. This conforms with the literature about MongoDB being more flexible, while Cassandra and MySQL take more effort with setup and restrictions when inserting and working with the databases in general. However, the SQL syntax is known to most people in the programming community and can be easier for most people to use.

Chapter 6

Conclusion

In this section, the conclusion of the research in this thesis is presented. First, the research questions are answered to the best extent. Second, the conclusion of the work is presented. Finally, a section mentioning some future work directions is discussed.

6.1 Research Questions

In Chapter 1, four research questions were presented. Throughout this thesis, answering these questions has worked as guidelines for the research. In this section, a summary of the answers to the research questions is presented.

RQ1 What database systems are researched in the literature about IoT data in smart cities and time-series data?

Smart cities lack research about database management in general. The research tends to focus more on ICT architecture and data management of the D2C architecture. In newer studies, a popular research filed is how cloud computing can be used with big data. More generally, in research on IoT, NoSQL databases are largely represented and it seems as document-based systems, often MongoDB, are popular based on their flexible nature, as well as Cassandra because of its write-performance and high scalability. However, many researchers often compare different NoSQL databases with relational databases, most often MySQL, but also database systems such as PostgreSQL. In terms of time-series data, though some literature mention MongoDB with indexing on timestamp rather than ID, Cassandra is heavily represented and mentioned as a good choice. Different patterns for storing time-series in Cassandra have been studied as well as comparing Cassandra towards popular TSDM systems.

RQ2 What are the requirements for IoT and time-series data management in database systems?

Many studies have investigated the requirements of data management of IoT. These requirements are described in Chapter 2, in Section 2.2.3. Some properties that are mentioned several times are scalability, data heterogeneity, and data aggregation. In Section 2.3.3, requirements for databases that store data recorded as time-series are listed. In Section 6.3, some future work direction regarding the requirements for IoT data management is presented.

RQ3 What databases are suited for IoT and time-series data based on the requirements found in literature, in **RQ2**?

Based on the databases considered in the literature, two NoSQL database systems where selected, one document-based, MongoDB, and Apache Cassandra, as well as one relational database, MySQL. These three databases were investigated in detail and are summarized in Section 2.3.4in terms of the requirements found in **RQ2**. MongoDB showed to satisfy all the requirements except one, and thereby showing a great foundation for being suitable for IoT. Cassandra, having some shortcomings, the missing properties are properties that can be provided by using external systems for properties such as real-time processing and analytics. Finally, MySQL showed less potential for being suitable for IoT by missing several properties that is considered important such as horizontal scalability and support for data heterogeneity as mentioned in **RQ2**.

RQ4 How do the databases from **RQ3** perform in experiments testing the requirements for data management in IoT related to smart city use, compared to the expected performance from literature?

The results of the experiments are summarized in Chapter 5. MongoDB and MySQL execution times where quite similar for all four queries. MongoDB was expected to perform better despite not using indexing because recent results are kept in memory in MongoDB. MySQL was on the other hand expected to have slower execution times, than the results recorded in the experiments. Cassandra's execution times of aggregation functions performed extremely bad, however, as expected, Cassandra has great results for retrieving time-interval only, outperforming the other two databases.

6.2 Conclusion

The need for database management systems to handle IoT data is demonstrated through an example of smart city usage in this thesis. Through a literature review, it was found that the research community tends to lean towards the standard of using a NoSQL database. MongoDB is heavily represented among this research, but when time-series data is discussed, Cassandra is often mentioned as a top competitor. However, MySQL was represented in

a lot of the literature as well, and this is why it was decided to look deeper into these three systems in terms of IoT and time-series requirements for data management. Again, MongoDB is a top contender in terms of the requirements by satisfying almost all desired properties reviewed in this work. Cassandra, which also had many important properties, had some limitations when compared with MongoDB. MySQL was also missing some important properties such as heterogeneity and horizontal scalability.

The results shown in Chapter 5, validates what has been shown in literature, that Cassandra is a good choice when handling time-series, and MongoDB giving promising results, even without using indexes. Besides, the results of experiments on MySQL shows that MySQL can be considered for IoT data for smaller domains or organizations.

6.3 Future work

Throughout this work, several aspects have been notices to be potential future work. In this section, future research areas are explained.

6.3.1 A Benchmark test for IoT data management requirements

In Chapter 2, Section 2.2.3, data management requirements for IoT was explored. Further work on IoT requirements on data management, could include continued work on these requirements to include all applications in the IoT domain and work as a standardized benchmark test for all IoT applications. In literature, many benchmarks have been developed in the IoT domain. A benchmark for streaming IoT applications have been developed [76]. IoTAbench is a benchmarking toolkit for IoT Big Data scenarios, which can be expanded to multiple IoT use cases [9]. However, few of these focus on the data management requirements to help support the evaluating of database systems. This is why there has been identified a need for a benchmark to be used as a standard for any IoT applications to evaluate any database system.

6.3.2 Expanding the researched databases

First, in this thesis, only the databases, MySQL, MongoDB, and Cassandra are researched in detail and compared to IoT and time-series requirements. Hence, only those three databases were researched in the experiments. Future work would be to include several other NoSQL databases in the comparisons of requirements and experiments. Even TSDB could be included to test the performance of these when compared with NoSQL systems both theoretical and with experiments.

6.3.3 Experiments with indexes

As the experiments in this thesis were conducted with data inserted as close as possible to their original format, future research should include testing whether the same results of execution times would appear with the use of indexes. Indexing can drastically improve performance. ”Without indexes, MongoDB must perform a collection scan, i.e. scan every document in a collection, to select those documents that match the query statement,” [56].

The same goes for MySQL. In Cassandra, the data model should be designed after the queries that are necessary for the applications to achieve high performance.

Because the experiments in this thesis are executed using minimal setups such as keys, indexes, and caches, they can be used as a basis for a benchmark for the performance in the future work mentioned above.

6.3.4 Cloud

Finally, in Chapter 2 and Section 2.3.6, cloud computing was discussed related to IoT. In the future, using cloud technologies will be a must for many organizations in IoT. Many cloud providers have emerged the recent years and many big companies have popular cloud platforms. Testing the IoT requirements and experiments on the cloud providers database technologies to test their performance with IoT data could be interesting future research. Apart from providing highly scalable options, cloud providers can open up additional opportunities of utilizing for instance big data and other systems together with the database to further satisfy the requirements of IoT data management.

Bibliography

[1] V. Abramova and J. Bernardino. Nosql databases: Mongodb vs cassandra. In *Proceedings of the international C* conference on computer science and software engineering*, pages 14–22, 2013.

[2] H. Ahvenniemi, A. Huovila, I. Pinto-Seppä, and M. Airaksinen. What are the differences between sustainable and smart cities? *Cities*, 60:234–245, 2017.

[3] A. Akela. 4 cluster management tools to compare, 2016. URL `https://dzone.com/articles/4-cluster-management-tools-to-compare`. Last accessed 10 June 2020.

[4] V. Albino, U. Berardi, and R. M. Dangelico. Smart cities: Definitions, dimensions, performance, and initiatives. *Journal of urban Researching information systems and computingtechnology*, 22(1):3–21, 2015.

[5] Amazon. Amazon timestream, 2020. URL `https://aws.amazon.com/timestream/`. Last accessed 4 May 2020.

[6] S. Amghar, S. Cherdal, and S. Mouline. Which nosql database for iot applications? In *2018 International Conference on Selected Topics in Mobile and Wireless Networking (MoWNeT)*, pages 131–137. IEEE, 2018.

[7] G. Anastasi, M. Antonelli, A. Bechini, S. Brienza, E. D'Andrea, D. De Guglielmo, P. Ducange, B. Lazzerini, F. Marcelloni, and A. Segatori. Urban and social sensing for sustainable mobility in smart cities. In *2013 Sustainable Internet and ICT for Sustainability (SustainIT)*, pages 1–4. IEEE, 2013.

[8] M. Antonini, M. Vecchio, and F. Antonelli. Fog computing architectures: A reference for practitioners. *IEEE Internet of Things Magazine*, 2(3):19–25, 2019.

[9] M. Arlitt, M. Marwah, G. Bellala, A. Shah, J. Healey, and B. Vandiver. Iotabench: an internet of things analytics benchmark. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, pages 133–144, 2015.

[10] C. Asiminidis, G. Kokkonis, and S. Kontogiannis. Database systems performance evaluation for iot applications. *International Journal of Database Management Systems (IJDMS) Vol*, 10, 2018.

[11] T. O. Authors. Real-time visibility into stacks, sensors and systems, 2020. URL `https://www.influxdata.com/`. Last accessed 14 May 2020.

[12] M. Babar and F. Arif. Smart urban planning using big data analytics to contend with the interoperability in internet of things. *Future Generation Computer Systems*, 77: 65–76, 2017.

[13] N. Z. Bawany and J. A. Shamsi. Smart city architecture: Vision and challenges. *International Journal of Advanced Computer Science and Applications*, 6(11), 2015.

[14] A. Bekker. Cassandra performance: The most comprehensive overview you'll ever see, 2018. URL `https://www.scnsoft.com/blog/cassandra-performance`. Last accessed 10 June 2020.

[15] M. B. Brahim, W. Drira, F. Filali, and N. Hamdi. Spatial data extension for cassandra nosql database. *Journal of Big Data*, 3(1):11, 2016.

[16] A. A. Chaudhari and P. Mulay. Scsi: real-time data analysis with cassandra and spark. In *Big Data Processing Using Spark in Cloud*, pages 237–264. Springer, 2019.

[17] B. Cheng, S. Longo, F. Cirillo, M. Bauer, and E. Kovacs. Building a big data platform for smart cities: Experience and lessons from santander. In *2015 IEEE International Congress on Big Data*, pages 592–599. IEEE, 2015.

[18] B. Cheng, S. Longo, F. Cirillo, M. Bauer, and E. Kovacs. Building a big data platform for smart cities: Experience and lessons from santander. In *2015 IEEE International Congress on Big Data*, pages 592–599. IEEE, 2015.

[19] A. Clemm. *Network management fundamentals*. Cisco Press, 2006.

[20] J. Cooper and A. James. Challenges for database management in the internet of things. *IETE Technical Review*, 26(5):320–329, 2009.

[21] O. Corporation. The mysql query cache, 2020. URL `https://dev.mysql.com/doc/refman/5.7/en/query-cache.html`. Last accessed 1 June 2020.

[22] O. Corporation. Mysql standard edition, 2020. URL `https://www.mysql.com/products/standard/`. Last accessed 23 March 2020.

[23] O. Corporation. Mysql enterprise edition, 2020. URL `https://www.mysql.com/products/enterprise/`. Last accessed 23 March 2020.

[24] O. Corporation. How mysql uses indexes, 2020. URL `https://dev.mysql.com/doc/refman/8.0/en/mysql-indexes.html`. Last accessed 7 May 2020.

[25] O. Corporation. 5.1.8 server system variables, 2020. URL https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_max_connections. Last accessed 15 June 2020.

[26] DataStax. Data caching, 2020. URL https://docs.datastax.com/en/cassandra-oss/3.0/cassandra/operations/opsDataCachingTOC.html. Last accessed 1 June 2020.

[27] I. DataStax. Indexing, 2020. URL https://docs.datastax.com/en/cql-oss/3.3/cql/cql_using/usePrimaryIndex.html. Last accessed 7 May 2020.

[28] L. Deri, S. Mainardi, and F. Fusco. tsdb: A compressed database for time series. In *International Workshop on Traffic Monitoring and Analysis*, pages 143–156. Springer, 2012.

[29] L. B. Dias, M. Holanda, R. C. Huacarpuma, and R. T. de Sousa Jr. Nosql database performance tuning for iot data. *Proceedings of the 3rd International Conference on Internet of Things, Big Data and Security (IoTBDS 2018)*, pages 277–284, 2018.

[30] A. Duarte and J. Bernardino. Cassandra for internet of things: An experimental evaluation. In *IoTBD*, pages 49–56, 2016.

[31] J. Dutta and S. Roy. Iot-fog-cloud based architecture for smart city: Prototype of a smart building. In *2017 7th International Conference on Cloud Computing, Data Science & Engineering-Confluence*, pages 237–242. IEEE, 2017.

[32] R. Elmasri and S. Navathe. Fundamentals of database systems, chapter 24, 1992.

[33] R. Elmasri and S. Navathe. Fundamentals of database systems, chapter 25, 1992.

[34] M. I. Floyd Smith. Time series - choosing a time series database, 2020. URL https://www.memsql.com/blog/choosing-a-time-series-database/. Last accessed 8 May 2020.

[35] T. A. S. Foundation. Security, 2020. URL https://cassandra.apache.org/doc/latest/operating/security.html. Last accessed 14 May 2020.

[36] M. Fredriksen. Ict architecture in large scale smart cities. Project report in TDT4501, Department of Computer Science Technology, NTNU – Norwegian University of Science and Technology, Dec. 2019.

[37] A. Gupta, R. Christie, and P. Manjula. Scalability in internet of things: features, techniques and research challenges. *Int. J. Comput. Intell. Res*, 13(7):1617–1627, 2017.

[38] C. Harrison, B. Eckman, R. Hamilton, P. Hartswick, J. Kalagnanam, J. Paraszczak, and P. Williams. Foundations for smarter cities. *IBM Journal of research and development*, 54(4):1–16, 2010.

[39] I. A. T. Hashem, I. Yaqoob, N. B. Anuar, S. Mokhtar, A. Gani, and S. U. Khan. The rise of "big data" on cloud computing: Review and open research issues. *Information systems*, 47:98–115, 2015.

[40] S. Heiler. Semantic interoperability. *ACM Computing Surveys (CSUR)*, 27(2):271–273, 1995.

[41] I. Inc. The scalable time series database, 2020. URL `http://opentsdb.net/`. Last accessed 14 May 2020.

[42] G. Kiraz and C. Toğay. Iot data storage: Relational & non-relational database management systems performance comparison. *A. Yazici & C. Turhan (Eds.)*, 34:48–52, 2017.

[43] R. Kumar, S. Charu, and S. Bansal. Effective way to handling big data problems using nosql database (mongodb). *J. Adv. Database Manag. Syst*, 2(2):42–48, 2015.

[44] K. K. Larsen. Timetable: Dynamic time series data store utilizing managed cloud services. Master thesis, Department of Computer Science Technology, NTNU – Norwegian University of Science and Technology, June 2018.

[45] Y. Li and S. Manoharan. A performance comparison of sql and nosql databases. In *2013 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*, pages 15–19. IEEE, 2013.

[46] A. Mahgoub, S. Ganesh, F. Meyer, A. Grama, and S. Chaterji. Suitability of nosql systems—cassandra and scylladb—for iot workloads. In *2017 9th International Conference on Communication Systems and Networks (COMSNETS)*, pages 476–479. IEEE, 2017.

[47] A. Makris, K. Tserpes, G. Spiliopoulos, and D. Anagnostopoulos. Performance evaluation of mongodb and postgresql for spatio-temporal data. In *EDBT/ICDT Workshops*, 2019.

[48] X. Masip-Bruin, E. Marín-Tordera, A. Alonso, and J. Garcia. Fog-to-cloud computing (f2c): The key technology enabler for dependable e-health services deployment. In *2016 Mediterranean ad hoc networking workshop (Med-Hoc-Net)*, pages 1–5. IEEE, 2016.

[49] N. Q. Mehmood, R. Culmone, and L. Mostarda. Modeling temporal aspects of sensor data for mongodb nosql database. *Journal of Big Data*, 4(1):8, 2017.

[50] A. Meier and M. Kaufmann. Nosql databases. In *SQL & NoSQL Databases*, pages 201–218. Springer, 2019.

[51] P. Mell, T. Grance, et al. The nist definition of cloud computing. *NIST Special Publication 800-145*, 2011.

[52] Microsoft. Azure sql database edge, 2020. URL `https://azure.microsoft.com/en-us/services/sql-database-edge/`. Last accessed 4 May 2020.

[53] Microsoft. Nosql databases: An overview for getting started, 2020. URL https://azure.microsoft.com/en-us/overview/nosql-database/. Last accessed 19 March 2020.

[54] I. MongoDB. Mongodb manual, 2008. URL https://docs.mongodb.com/manual/reference/explain-results/#explain.executionStats. Last accessed 1 June 2020.

[55] I. MongoDB. Change streams, 2008. URL https://docs.mongodb.com/manual/changeStreams/. Last accessed 15 June 2020.

[56] I. MongoDB. Indexes, 2008. URL https://docs.mongodb.com/manual/indexes/. Last accessed 25 June 2020.

[57] I. MongoDB. Time series data and mongodb, 2019. URL https://www.mongodb.com/blog/post/time-series-data-and-mongodb-part-2-schema-design-best-practices/. Last accessed 25 May 2020.

[58] I. MongoDB. Aggregation, 2020. URL https://docs.mongodb.com/manual/aggregation/. Last accessed 13 May 2020.

[59] I. MongoDB. Mongodb atlas, 2020. URL https://www.mongodb.com/cloud/atlas. Last accessed 15 June 2020.

[60] I. MongoDB. Real-time analytics, 2020. URL https://www.mongodb.com/use-cases/real-time-analytics. Last accessed 13 May 2020.

[61] I. MongoDB. Security, 2020. URL https://docs.mongodb.com/manual/security/. Last accessed 14 May 2020.

[62] I. MongoDB. Write operation performance, 2020. URL https://docs.mongodb.com/v3.4/core/write-performance/. Last accessed 14 May 2020.

[63] D. Namiot. Time series databases. *DAMDID/RCDL*, 1536:132–137, 2015.

[64] P. G. V. Naranjo, Z. Pooranian, M. Shojafar, M. Conti, and R. Buyya. Focan: A fog-supported smart city network architecture for management of applications in the internet of everything environments. *Journal of Parallel and Distributed Computing*, 132:274–283, 2019.

[65] U. Nations. News - world population projected to reach 9.8 billion in 2050, and 11.2 billion in 2100, 2017. URL https://www.un.org/development/desa/en/news/population/world-population-prospects-2017.html. Last accessed 12 May 2020.

[66] N. Neeraj. *Mastering Apache Cassandra*. Packt Publishing Ltd, 2013.

[67] B. J. Oates. *Researching information systems and computing*. Sage, 2005.

[68] A. Piórkowski et al. Mysql spatial and postgis–implementations of spatial data standards. *Electronic journal of Polish agricultural universities*, 14(1):1–8, 2011.

[69] E. S. Pramukantoro, D. P. Kartikasari, and R. A. Siregar. Performance evaluation of mongodb, cassandra, and hbase for heterogenous iot data storage. In *2019 2nd International Conference on Applied Information Technology and Innovation (ICAITI)*, pages 203–206. IEEE, 2019.

[70] R. Ramakrishnan and J. Gehrke. *Database management systems*. McGraw Hill, 2000.

[71] D. Ramesh, A. Sinha, and S. Singh. Data modelling for discrete time series data using cassandra and mongodb. In *2016 3rd international conference on recent advances in information technology (RAIT)*, pages 598–601. IEEE, 2016.

[72] S. Rautmare and D. Bhalerao. Mysql and nosql database comparison for iot application. In *2016 IEEE International Conference on Advances in Computer Applications (ICACA)*, pages 235–238. IEEE, 2016.

[73] P. P. Ray. A survey of iot cloud platforms. *Future Computing and Informatics Journal*, 1(1-2):35–46, 2016.

[74] L. Sanchez, L. Muñoz, J. A. Galache, P. Sotres, J. R. Santana, V. Gutierrez, R. Ramdhany, A. Gluhak, S. Krco, E. Theodoridis, et al. Smartsantander: Iot experimentation over a smart city testbed. *Computer Networks*, 61:217–238, 2014.

[75] B. Schwartz, P. Zaitsev, and V. Tkachenko. *High performance MySQL: optimization, backups, and replication*. " O'Reilly Media, Inc.", 2012.

[76] A. Shukla and Y. Simmhan. Benchmarking distributed stream processing platforms for iot applications. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 90–106. Springer, 2016.

[77] A. Sinaeepourfard, J. Krogstie, S. A. Petersen, and D. Ahlers. F2c2c-dm: a fog-to-cloudlet-to-cloud data management architecture in smart city. In *2019 IEEE 5th World Forum on Internet of Things (WF-IoT)*, pages 590–595. IEEE, 2019.

[78] I. G. Smith. *The Internet of things 2012: new horizons*. CASAGRAS2, 2012.

[79] M. Tahmassebpour. A new method for time-series big data effective storage. *Ieee Access*, 5:10694–10699, 2017.

[80] I. Timescale. Building a distributed time-series database on postgresql, 2019. URL `https://blog.timescale.com/blog/building-a-distributed-time-series-database-on-postgresql/`. Last accessed 14 May 2020.

[81] I. Timescale. All of your time-series data, instantly accessible, 2020. URL `https://www.timescale.com/`. Last accessed 14 May 2020.

[82] D. G. Waddington and C. Lin. A fast lightweight time-series store for iot data. *arXiv preprint arXiv:1605.01435*, 2016.

[83] C. Yang, Q. Huang, Z. Li, K. Liu, and F. Hu. Big data and cloud computing: innovation opportunities and challenges. *International Journal of Digital Earth*, 10(1): 13–53, 2017.

Appendix A

```
import datetime
from mysql.connector import connect

DB = 'master'
COLL = 'sensor'
mysql_db = connect(host='localhost', user='root',
password='')

mysql_cursor.execute(f"""
CREATE TABLE IF NOT EXISTS {COLL} (
    VendorID int,
    tpep_pickup_datetime timestamp PRIMARY KEY,
    tpep_dropoff_datetime timestamp,
    passenger_count int,
    trip_distance double,
    RatecodeID int,
    store_and_fwd_flag text,
    PULocationID int,
    DOLocationID int,
    payment_type int,
    fare_amount double,
    extra double,
    mta_tax double,
    tip_amount double,
    tolls_amount double,
    improvement_surcharge double,
    total_amount double,
    congestion_surcharge double
    )
""")

def query1_mysql():
    mysql_session = mysql_db.cursor()
    mysql_session.execute(f"USE {DB}")
    mysql_session.execute("Set profiling = 1")
    query = f"""
    SELECT * FROM {COLL}
    WHERE `tpep_pickup_datetime` >= "{start_time}" and
```

```
    `tpep_pickup_datetime` < "{end_time}"
    """
    mysql_session.execute(query)

def query2_mysql():
    mysql_session = mysql_db.cursor()
    mysql_session.execute(f"USE {DB}")
    mysql_session.execute("Set profiling = 1")
    query = f"""
    SELECT SUM(tip_amount) FROM {COLL}
    """
    mysql_session.execute(query)

def query3_mysql():
    mysql_session = mysql_db.cursor()
    mysql_session.execute(f"USE {DB}")
    mysql_session.execute("Set profiling = 1")
    query = f"""
    SELECT AVG(trip_distance) FROM {COLL}
    WHERE tpep_pickup_datetime >= "{start_time}" and
    tpep_pickup_datetime < "{end_time}"
    """
    mysql_session.execute(query)

def query4_mysql():
    mysql_session = mysql_db.cursor()
    mysql_session.execute(f"USE {DB}")
    mysql_session.execute("Set profiling = 1")
    query = f"""
    SELECT SUM(tip_amount) as tips_sum FROM {COLL}
    GROUP BY VendorID
    """
    mysql_session.execute(query)
```

Appendix B

The data in MongoDB is inserted as presented in the dataset.

```
def query1_mongodb():
    mongo_cursor = mongo_db.find(
        {"tpep_pickup_datetime":
        {"$gte": start_time, "$lt": end_time}}
    )

def query2_mongodb():
    pipeline = [
        {"$match": {}},
        {"$group": {"_id": None, "total":
        {"$sum": "$tip_amount"}}}
    ]
    agg_cmd = SON(
        [("aggregate", COLL), ("pipeline", pipeline),
        ("cursor", {})]
    )
    result = mongo_db.aggregate(pipeline)

def query3_mongodb():
    pipeline = [
        {"$match": {"tpep_pickup_datetime" :
        {"$gte": start_time, "$lt": end_time}}},
        {"$group": {"_id": None, "avrage":
        {"$avg": "$trip_distance"}}}
    ]
    agg_cmd = SON(
        [("aggregate", COLL), ("pipeline", pipeline),
        ("cursor", {})]
    )
    result = mongo_db.aggregate(pipeline)

def query4_mongodb():
    pipeline = [
        {"$match": {}},
        {"$group": {"_id": "$VendorID", "sum":
        {"$sum": "$tip_amount"}}}
    ]
```

```
agg_cmd = SON(
    [("aggregate", COLL), ("pipeline", pipeline),
    ("cursor", {})]
)

result = mongo_db.aggregate(pipeline)
```

Appendix C

```
import datetime
from cassandra.cluster
import Cluster, BatchStatement, ExecutionProfile

DB = 'master'
COLL = 'sensor'

cassandra_db = Cluster()
cassandra_db.default_timeout = 60

cassandra_session.execute(
    f"""
    CREATE KEYSPACE IF NOT EXISTS {DB}
    WITH replication = {{
        'class': 'SimpleStrategy', 'replication_factor': '3'
    }}
    """
)

cassandra_session.set_keyspace(DB)
cassandra_session.execute(
    f"""
    CREATE TABLE IF NOT EXISTS {COLL} (
        VendorID int,
        tpep_pickup_datetime timestamp,
        tpep_dropoff_datetime timestamp,
        passenger_count int,
        trip_distance double,
        RatecodeID int,
        store_and_fwd_flag text,
        PULocationID int,
        DOLocationID int,
        payment_type int,
        fare_amount double,
        extra double,
        mta_tax double,
        tip_amount double,
        tolls_amount double,
```

```
        improvement_surcharge double,
        total_amount double,
        congestion_surcharge double,
        PRIMARY KEY ((VendorID), tpep_pickup_datetime)
    )
    WITH caching = {{
     'keys' : 'NONE',
     'rows_per_partition' : 'NONE' }}

    """
)

def query1_cassandra():
    cassandra_session = cassandra_db.connect()
    cassandra_session.set_keyspace(DB)
    query = f"""
    SELECT * FROM {COLL}
    WHERE "tpep_pickup_datetime" >= '{start_time}' and
    "tpep_pickup_datetime" < '{end_time}'
    ALLOW FILTERING
    """
    q = cassandra_session.execute(query, trace=True)

def query2_cassandra():
    cassandra_session = cassandra_db.connect()
    cassandra_session.default_timeout = 60
    cassandra_session.set_keyspace(DB)
    name = "r_sum"
    query = f"""
    SELECT SUM(tip_amount) AS {name} FROM {COLL}
    """
    q = cassandra_session.execute(query, trace=True,
    timeout=60)

def query3_cassandra():
    cassandra_session = cassandra_db.connect()
    cassandra_session.default_timeout = 60
    cassandra_session.set_keyspace(DB)
    avrage = "avr"
    query = f"""
    SELECT AVG(trip_distance) AS {avrage} FROM {COLL}
    WHERE tpep_pickup_datetime >= '{start_time}' and
    tpep_pickup_datetime < '{end_time}'
    ALLOW FILTERING
    """
    q = cassandra_session.execute(query, trace=True)
```

```
def query4_cassandra():
    cassandra_session = cassandra_db.connect()
    cassandra_session.default_timeout = 60
    cassandra_session.set_keyspace(DB)
    tipsum = "sum"
    query = f"""
    SELECT SUM(tip_amount) AS {tipsum} FROM {COLL}
    GROUP BY VendorID
    """
    q = cassandra_session.execute(query, trace=True)
```