MASTER'S THESIS 2022

Bitemporal modeling: implementations, performance and use cases

Anthony Bui

ISSN 1650-2884
LU-CS-EX: 2022-61

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2022-61

Bitemporal modeling: implementations, performance and use cases

Bitemporal modellering: implementationer, prestanda och användarfall

Anthony Bui

Bitemporal modeling: implementations, performance and use cases

Anthony Bui
an7407bu-s@student.lu.se

December 16, 2022

Master's thesis work carried out at Parkster AB.

Supervisors: Adam Nilsson, adam.nilsson@parkster.se
Sergio Rico, sergio.rico@cs.lth.se

Examiner: Roger Henriksson, roger.henriksson@cs.lth.se

Abstract

Database management systems have recently introduced support for managing both valid time and transaction time periods, defining a *bitemporal* database.

Our aim is to investigate the state-of-the-art of bitemporal modeling and some of its implementations in database tools available today. First, literature research was conducted. Based on this, we chose qualitative factors to be used in evaluations. Among these were bitemporal properties featured in the SQL:2011 standard. In benchmarks, we used some queries featured in TPC-BiH, a bitemporal benchmark suite. Then, four tools with differing database models were chosen. Two of these had native bitemporal support, whereas the remaining tools required modeling and implementation.

In conclusion, our results show that bitemporal implementations in database tools are lacking compared to state-of-the-art bitemporal modeling. Benchmarks did not yield satisfying results, although we suspect the reason to be the amount of data used.

Keywords: Temporal modeling, database, benchmark, nosql, sql

Acknowledgements

This thesis was made possible due to Sergio Rico, my supervisor who invested his valuable time on multiple meetings and giving his insight on multiple areas of this work. I am grateful for his patience, as well as his positive energy that helped me push through the harder periods.

I would like to thank Parkster AB for giving me the opportunity and honor of being their first Master's thesis test subject, as well as for their extremely warm welcome and support on-site. In addition, I would like to thank my assigned supervisor, Adam Nilsson, and senior developer, Johan Haleby, for their expertise and patience during the practical parts of this thesis.

Last but not least, I would like to thank my wife for tolerating me during this hectic period.

Contents

1 Introduction **9**
- 1.1 Outline . . . 10
- 1.2 Case Problem . . . 10
- 1.3 Research Questions . . . 10
- 1.4 Scientific Contribution . . . 11

2 Theory & Related Work **13**
- 2.1 Definitions . . . 13
  - 2.1.1 Database and DBMS . . . 13
  - 2.1.2 Data Model . . . 14
  - 2.1.3 Valid Time . . . 14
  - 2.1.4 Transaction Time . . . 15
  - 2.1.5 Timestamping . . . 15
  - 2.1.6 Snapshot Database . . . 16
  - 2.1.7 Temporal Database . . . 16
  - 2.1.8 Joins and Temporal Joins . . . 16
- 2.2 History of DBMS . . . 17
  - 2.2.1 Early Electronic Information Storage . . . 17
  - 2.2.2 Introduction of Hard Disk Drives . . . 17
  - 2.2.3 The IDS . . . 18
- 2.3 Database Models . . . 18
  - 2.3.1 The Relational Model . . . 19
  - 2.3.2 The Object-Oriented Model . . . 20
  - 2.3.3 The Object-Relational Model . . . 21
  - 2.3.4 NoSQL . . . 22
  - 2.3.5 Temporal Models . . . 25
- 2.4 Implementing Temporal Databases . . . 26
  - 2.4.1 SQL-based Implementations . . . 27
  - 2.4.2 NoSQL Implementations . . . 28
- 2.5 The SQL:2011 Standard . . . 29

2.6 Indexing . . . 29
2.6.1 Hash Index . . . 30
2.6.2 B-tree . . . 30
2.6.3 GiST . . . 30
2.7 Query Planning & Optimization . . . 30
2.8 ACID & CAP . . . 31
2.9 Benchmarks & Comparisons . . . 31
2.9.1 SQL-based . . . 32
2.9.2 NoSQL . . . 33
2.9.3 SQL vs. NoSQL . . . 34
2.9.4 Temporal Benchmarks . . . 35
2.10 Survey of Tools . . . 35
2.10.1 Relational DBMSs . . . 36
2.10.2 Object-Relational DBMSs . . . 36
2.10.3 Object-Oriented DBMSs . . . 36
2.10.4 NoSQL . . . 37
2.11 Summary . . . 38

3 Methodology **39**
3.1 Process . . . 39
3.2 Selection of Tools . . . 40
3.3 Measurements & Comparisons . . . 41
3.3.1 Qualitative . . . 41
3.3.2 Quantitative . . . 42
3.4 Retroactivity . . . 43

4 Execution & Results **45**
4.1 MariaDB . . . 45
4.1.1 Setup . . . 45
4.1.2 Modeling & Implementation . . . 46
4.1.3 Ingestion . . . 47
4.1.4 Retroactivity . . . 48
4.2 PostgreSQL . . . 48
4.2.1 Setup . . . 48
4.2.2 Modeling & Implementation . . . 49
4.2.3 Ingestion . . . 50
4.2.4 Retroactivity . . . 50
4.3 XTDB . . . 51
4.3.1 Setup . . . 51
4.3.2 Modeling & Implementation . . . 51
4.3.3 Ingestion . . . 53
4.3.4 Retroactivity . . . 53
4.4 Neo4j . . . 54
4.4.1 Setup . . . 54
4.4.2 Modeling & Implementation . . . 54
4.4.3 Ingestion . . . 54
4.4.4 Retroactivity . . . 56

4.5 Qualitative 56
4.5.1 Bitemporal Support 56
4.5.2 Configuration 57
4.5.3 Scalability 58
4.5.4 Availability 59
4.5.5 Ease of Use 59
4.6 Quantitative 59
4.6.1 Ingestion 59
4.6.2 Point-to-Point 60
4.6.3 Slicing 62
4.6.4 Pure-key audit 63
4.6.5 Bitemporal Queries 63
4.6.6 Summary 64

5 Discussion **67**
5.1 Qualitative Results 67
5.1.1 Bitemporal Support 67
5.1.2 Configuration 68
5.1.3 Scalability 68
5.1.4 Availability 68
5.1.5 Ease of use 68
5.1.6 Takeaways 68
5.2 Quantitative Results 69
5.2.1 Ingestion 69
5.2.2 Query Results 70
5.2.3 Takeaways 72
5.3 Validity Threats 73
5.3.1 Internal Validity 73
5.3.2 External Validity 73
5.4 Reflection 73
5.5 Future Work 74

6 Conclusion **75**

References **77**

Chapter 1
Introduction

The need for documentation has long been an integral part of human history and progression. Examples range from managing laws and cataloging citizens to recording birthdays and sharing food recipes. The amount of information stored increased over time, and with them, the related questions: where do we store our data, and how?

Even before the age of digitization, methods of organizing information were of great importance in order to, for example, minimize lookup times. We have seen sorting in alphabetical order in libraries or document archives, believed to have been used already in the Great Library of Alexandria [52]. When computers were introduced in the 1950s, the same problem had to be solved. Simply sorting by some kind of order, alphabetic or otherwise, was not enough as the amount and complexity of information grew. By 1964, the first database management system (DBMS) was introduced [39]. In the following years, more models on how to structure the data were proposed. One of the core models that suggested storing the data in tables with rows and columns was the "relational model" coined in 1970 [21], a model which is still widely used today.

Despite many proposals on modeling, the aspect that seems to have been neglected is the aspect of time. Usually, a database contains a "snapshot" of reality, which in most cases is the current time. For example, suppose we have a table storing currently active parkings in a parking lot and information about the user and the vehicle's registration number. If the user wishes to extend his parking time, or correct an erroneous registration number, the database overrides the old value and does not keep track of past values. Furthermore, a database usually does not keep track of *when* a change in information occurs.

A benefit to tracking time in databases is the ability to analyze data; for example, if we track the history of parkings, comparisons can be made against parking lots, companies, and industries. If we instead track *when* changes occurred in the database, we can get snapshots of the database at different points in time. This allows us to see what we knew in the database when, for example, a user received a parking ticket when they in fact correctly paid for their parking time. This is a powerful tool and sometimes even legally required for auditing purposes.

To summarize, we can track the time when events happened in reality and when information was registered in the database. In other words, *what* we knew and *when* we knew it. Tracking only one of these would define a *temporal* database, whereas tracking both would define a *bitemporal* database [47]. The aim of this thesis is to investigate the ways to model and implement bitemporality, as well as the takeaways in terms of functionality and performance.

1.1 Outline

In this first chapter, we present the company, the case problem, the desired outcome, and the research questions associated with this work. The second chapter contains a theory section where definitions of temporality, database management systems and related terms are introduced along with their progression since the 1950s. The third chapter introduces the methodology of this thesis and sample data given by Parkster. The fourth chapter will present the gathered experimental data through tables and diagrams. These results will then be discussed in the fifth chapter, along with limitations and suggestions for improving the data collection for future work. Finally, the last chapter summarizes and concludes the thesis as a whole.

1.2 Case Problem

Parkster is a tech company founded in 2010 that offers services related to parking through digital means, such as mobile apps. They have seen rapid growth over the years, working on international expansion and having taken the opportunity to develop new services over time. One of their ideas was a so-called "Parking map", a service that would show parking density in one or several areas over time. This data could help end consumers plan their parking during the day, as well as help larger customers such as event planners and municipals to see the density of areas and plan accordingly.

Parkster started experimenting with applying their data to a bitemporal DBMS, however, due to time constraints, this exploration had to be put on hold. Since then, new alternatives with different models and implementations of bitemporality have emerged. This thesis will continue that work, as well as investigate which alternative best fits their use case.

1.3 Research Questions

The described problem then gives us the following research questions:

RQ 1: What is the state-of-the-art of bitemporal modeling and its implementations?
With this first research question, we want to explore the academic research that has been made on bitemporality so far, starting from the first academic proposal regarding time in databases back in 1956 [14]. We will learn about some of the different temporal models that have been proposed, their implementations as well as eventual problems brought up by academia. Using this knowledge, we wish to survey current tools used in the industry. The state-of-the-art and a survey of tool candidates are presented in section 2.10.

RQ 2: What are the benefits and drawbacks of the different ways to model bitemporality, when applied to Parkster's parking map?
The different ways to model bitemporal databases naturally have their pros and cons, some of which may change depending on the use case. With this RQ, we want to investigate which these are when applied to Parkter's parking map, which in turn will help us learn more about the differences between models and their implementations.

RQ 3: What are the takeaways of having implemented bitemporality in different ways?
Finally, we wish to reflect on the general positives and negatives we encountered after attempting to implement a bitemporal database. These could be factors regarding our methodology, choice of modeling or the reasons behind differences in performance.

1.4 Scientific Contribution

This thesis will summarize most of the research on bitemporal modeling since temporality was first introduced. With the gathered information, this thesis also aims to provide a real-life example of bitemporal modeling, something which seems to be missing in this environment due to the lack of open-source code.

The thesis could also extend on current research by providing an up-to-date overview of the possibilities and performance of different implementations. The qualitative variables, derived from research and related work in the area, may contribute to future work as they demonstrate the requirements for use cases similar to ours. With this, users in similar situations may compare a range of tested tools and implementations or perhaps convince users to introduce bitemporal modeling in the future due to its potential benefits.

Chapter 2

Theory & Related Work

Research around time and temporal databases has been conducted since the 1950s, with a set of recurring scholars who have led the field. The definitions used in this thesis mostly come from these academic leaders' papers. The history section covers a different selection of papers from the 1950s until now. These cover some definitions, the history of temporal databases, the currently used data types, models, examples of implementing said models and comparisons between the different database categories.

2.1 Definitions

A paper proposing consensus definitions on temporal database concepts was released in 1992 [47], where four of the five authors would later become members of the TSQL2 committee dedicated to research on temporal databases. The committee was formed in July 1993 and comprised 18 individuals [104] who later that year published TSQL2, a temporal query language based on SQL. The most relevant temporal definitions are listed below, and other related terms not mentioned in the consensus paper.

2.1.1 Database and DBMS

The archives of documents and information in earlier days could technically be called a database. However, with the introduction of computers, a database nowadays refers to a collection of organized data stored electronically [87]. A database management system, or DBMS, controls a database by implementing the actions that are used on a database. These could be creating, deleting, or searching for data.

2.1.2 Data Model

A data model defines the constructs and formalisms available to define, modify and access data, as well as integrity constraints to express a so-called consistent database. Among these constructs are the data structures and logical concepts such as objects, properties, and interrelationships. A DBMS follows this abstract data model when implementing the actions that can be applied to the database. Thus, the data model acts as a bridge to the physical computer storage [105].

Furthermore, data models can be split into conceptual, logical, and physical data models [44]. If one works in the same order as they are listed, one could see the conceptual model as giving an overview of what the system will contain, how it will be organized, and possibly their relationships. For example, with our parking scenario, this could be parkings and parking zones. Next, the logical model is less abstract and will explain the categories in more detail, now containing attributes such starting time of the parking. On the other hand, parking zones could contain information on location and owner. The physical model adds to this information by giving concrete data types and introducing primary and foreign keys in the case of relational databases discussed later.

One of the most well-known conceptual models is that of the Entity-Relationship (ER), developed by Peter Chen in 1976 [19]. This model can show the relationships between entity sets in a database on a higher level, with symbols representing cardinality and attributes. Figure 2.1 shows an example ER-diagram for our case of parkings. The symbols represent a one-to-many relationship where a single parking zone may have multiple parkings.

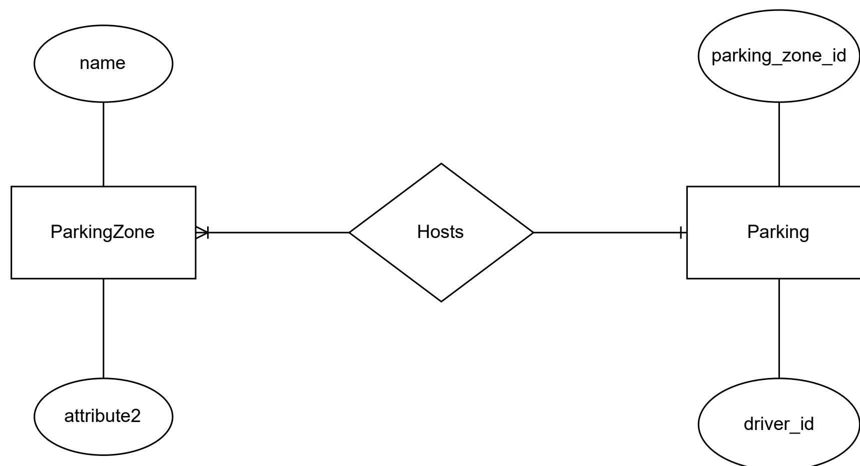


Figure 2.1: Parking example of the ER-diagram

2.1.3 Valid Time

The valid time represents the time when a fact is true in our modeled reality [47]. In our case, this can be modeled as the time when parking took place. There are no restrictions on the valid time since we can set times in the past, the present, and the future if we know or schedule something in advance. An alternative name has been "real-world time," which further emphasizes that we are talking about time in reality.

2.1.4 Transaction Time

The transaction time is the time when a fact was registered and stored (in a database). As opposed to valid time, the transaction time can never be greater than the current time in the real world by definition since it represents the actual time a fact was registered [47]. For example, we can register a parking to last into the future, whereas transaction time in most cases will be the current time. The end date of a transaction time represents the time at which said value is no longer valid in our database.

A property associated with transaction time only allows the addition of data, that is, never allowing deletions or updates of data [47]. Whenever we realize that some data has been proven incorrect, for example, the start time of parking was 10 minutes earlier than we thought, then the old value would be replaced. If transaction times are used, we would instead *logically* delete this value by truncating it with the new, correct value. In most cases, this is done by setting an end date to the transaction time of the now incorrect row, representing an end to its validity in our database. However, this old value remains in the database "physically" as a row that can be queried or looked up. We will provide examples of this later in this chapter.

2.1.5 Timestamping

A timestamp is a time value of some sort associated with an object [47]. For example, when discussing parkings, we can attach the time when the parking started or ended. There are mainly two ways to timestamp: tuple timestamping and attribute timestamping. The stamping itself can also be done in several ways, often restricted to whether tuple or attribute timestamping is used [105].

Tuple Timestamping

This way of timestamping usually applies to relational databases that store their data as *tuples*, which we will introduce in section 2.2. Tuples, more commonly seen as rows containing information on parkings, may have a timestamp applied for its valid time. In this case, stamping an interval with start and end times would be appropriate. The end time is left "open" if the parking duration is unknown. Previous parkings are "closed" periods, assuming we know the validity dates. This form of timestamping works the same if the database tracks transaction times.

A shortcoming of this method is the introduction of data redundancy. For example, a change in parking times would require a new, duplicated row where all information, except parking times, is repeated [105]. This is also known as coalescing and is demonstrated in figure 2.2 where the second and third rows have unnecessary overlapping periods. A model is so-called coalesced if this form of overlapping is disallowed [49].

Attribute Timestamping

Attribute timestamping apply timestamps to each attribute rather than the whole tuple, meaning that we do not need to repeat the unchanged data. Thus, we overcome the data redundancy introduced in tuple timestamping. This is demonstrated in figure 2.2.

parkingZoneID	name	vt_start	vt_end
123	Lund High School	2019-01-01	2019-12-31
123	Lund College	2020-01-01	2020-01-31
123	Lund College	2020-02-01	2038-01-01

parkingZoneID	name
123	{<[2019-01-01, 2019-12-31], Lund High School> <[2020-01-01, 2038-01-01], Lund College>}

Figure 2.2: Parking example of tuple and attribute timestamping

2.1.6 Snapshot Database

A database that does not support or apply either valid time or transaction time is seen as a snapshot database. If time is not tracked, then the stored information is current "as of now" [47]. In the case of parkings, we could for example have a table that stores active parkings. When a parking has been finished, it is deleted from the table. This would be a "snapshot" table where only the information that is currently known as true exists in the database.

2.1.7 Temporal Database

A temporal database supports either valid time or transaction time. If both are supported, it is instead a bitemporal database [47]. In the case of parkings, the database would be temporal, on valid time axis, if we save the time periods of parkings. If we instead save the time at which a change is performed, such as deleting or adding a parking to our "active parkings" table, that would be a temporal database on the transaction time axis. If we save both the active parking periods and the times when these changes are made in the database, that would give us a *bitemporal* database.

2.1.8 Joins and Temporal Joins

Although possible with non-SQL-based databases, discussed more later in this chapter, a join mainly refers to the operation of combining rows of data from different tables. This operation is based on the relational model and its mathematical foundation of using set theory [21]. The combining of rows is done on some condition, or predicates, such as rows sharing the same key or having matching values from different columns.

A temporal join works similarly. However, it requires the overlap of time intervals of the joined tuples [34]. The time intervals can be either valid or transaction time intervals. Using one of these intervals refers to a temporal join, whereas both refer to a bitemporal join.

2.2 History of DBMS

This subsection summarizes the history of databases up until the relational database model before introducing the concept of bitemporality. This section will cover early information storage, the first DBMS, and briefly discuss the earlier models of databases.

2.2.1 Early Electronic Information Storage

Before the dawn of DBMSs, computers were essentially large calculators and had no proper way of storing information due to the low amounts of memory. As the availability of computers increased, so did their usage in the industry. As a result, the tasks shifted toward business administration applications, mostly handling payrolls and general accounting [38]. However, it took longer than expected to get the systems up and running useful work. In addition, the managers processing the data complained about the programming complexity. In contrast, another problem was the lack of standard packages at this time: much of the code had to be written specifically for each company [39].

The data itself was stored as a sequence of codes on magnetic tape. This technique was derived from earlier punch cards during the 1930s. Inherently, processing was inefficient unless the tape was read from the beginning. Even if that was the case, the computers had small internal memory, resulting in the need for several programs stacked after one another should one wish to compute a complex task. There were additional problems, such as read errors and synchronization errors, which later led to the idea of standardized, reusable subroutines [39].

In 1957, the most influential report-generation software was designed by General Electric (GE). Here, staff provided the programs with descriptions of the desired output and the organization of the data inside the relevant master files. The following year, sorting capabilities were added for more complex calculations [39].

However, GE was one of many companies that independently tried producing their systems. It prompted a panel at the IBM Share user group the same year where hope for future collaboration was expressed. That hope came to fruition by the end of the year with the formation of the Share Data Processing Committee, where members attempted to agree on consensus terms and formats. The committee successfully launched cooperative projects, of which many practices are used in open-source projects today. The collaboration between many soon paved the way for some modeling: the creation of *hierarchical relationships* and associations to parent records - a significant step forward in terms of complexity [39]. A simple example applied to parking is demonstrated in figure 2.3.

2.2.2 Introduction of Hard Disk Drives

The first computer that used disk storage was Ramac in 1957. However, disk drives were only widely introduced around 1962. With drives, new technological possibilities emerged. For example, the file management systems that used magnetic tape could be efficient if read from the start in sequence. Disk drives, however, allowed for business data to be placed, checked, retrieved, and updated by different application programs in a short amount of time without having to be read "in sequence" as with tapes.

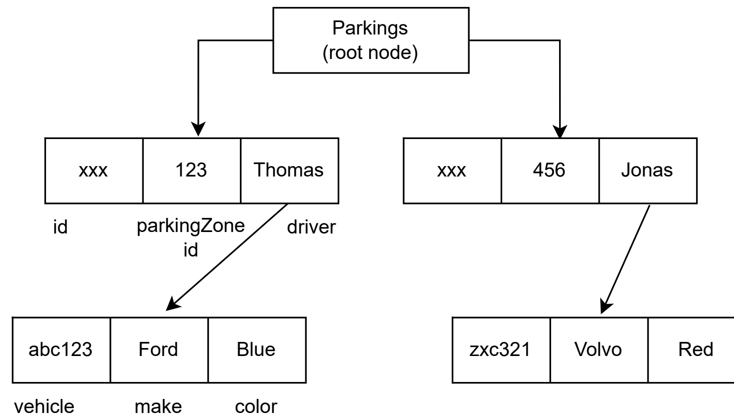


Figure 2.3: Parking example for the Hierarchical Model

However, there was a slight problem: while the information in specific locations of the disk could easily be retrieved, it was much harder to, for example, find the salary of a particular employee. With this, the development of indexing began, which was not a trivial task with problems of formatting, backups, and allocation of disk arrays to different programs [39].

2.2.3 The IDS

The evolution of information storage described in the previous section can arguably be seen as having culminated in the first DBMS in 1964: the Integrated Data Store. The IDS was a breakthrough; it had many features that had been proposed and discussed for some time but were never implemented. The IDS was a navigational database that introduced the so-called "network data model" where relationships between different records were connected through a *graph*, as opposed to the previously used hierarchical data model with magnetic tapes. This, however, meant that the user would have to cycle through chains of records until an end-of-set condition was detected. Additionally, records followed a sort of schema in order to be placed in a specific chain [39], as seen in figure 2.4.

A great benefit of the IDS was allowing programmers to code applications in a high-level language rather than assembly while also enabling this kind of support for disk drives. The IDS was, after all, designed with disk drives in mind, allowing for maximization of performance. The IDS acted as a bridge between the physical disk storage files and application programs, a core feature of DBMSs today, however lacked some other features such as access rules or the drawback of having to write querying programs with explicit IDS calls [40]. Nevertheless, the IDS paved the way for the upcoming relational model as well as the modern DBMS.

2.3 Database Models

Many models have been proposed since the emergence of the IDS and its navigational database model, where some have gained more traction than others. In this section, we give a short introduction to the most well-known models.

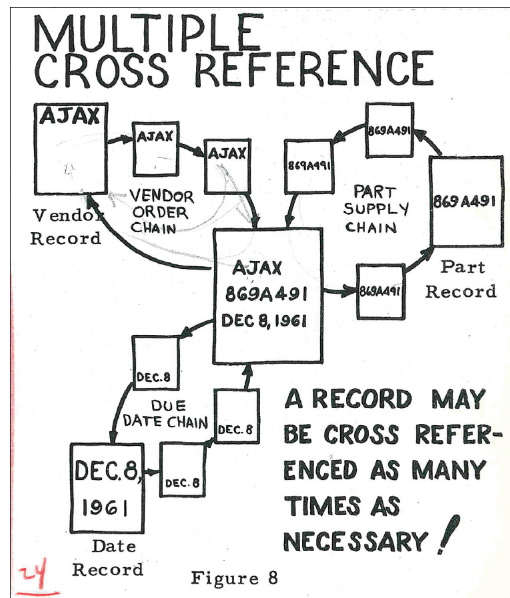


Figure 2.4: "Bachman Diagram" from the 1962 presentation on the IDS [40]

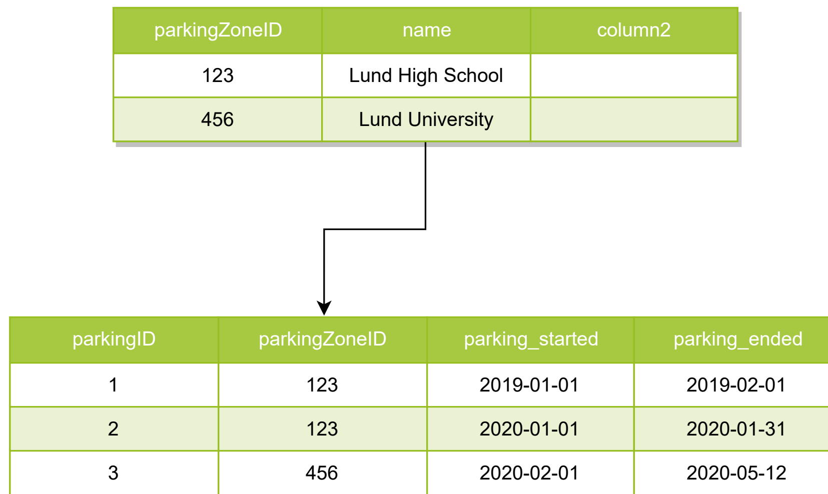
2.3.1 The Relational Model

Feeling that the DBMSs could be improved, Edgar F. Codd published his paper on the relational data model in 1970 [21]. The general idea was to abstract the DBMS to a higher level so that the internal representation of the stored data would be hidden. Users who work at terminals, as well as application programs should not be affected whenever the underlying data or its structure is changed. To achieve this, Codd pointed out three main dependencies of the DBMSs during that time that needed to be addressed: ordering, indexing, and access path dependence.

With the DBMSs that had been released, one could either choose an unordered setting or have each element be ordered to single or multiple orderings associated with the hardware addresses. Codd brings up an example drawback: if mechanical parts are stored in ascending order by their serial number, the DBMS will allow application programs to "assume" this ordering. If this were to change, the application programs would likely not function correctly.

Codd also pointed out some of the flaws of systems that used the hierarchical model during that time. Regarding indexing, Codd mentioned that the time-shared data management system (TDMS) unconditionally provided indexing on all attributes. In contrast, IBM's IMS allowed the user to choose between the classic hierarchical approach or indexing on the primary key only. In addition, the IDS introduced chains in order to incorporate indexes into the file structure. However, application programs using this needed to refer to the chains by name, which, similarly to the ordering dependency, would become a problem should the structure of the chains change.

Lastly, access paths suffer from the same problem. The DBMSs usually allowed for tree-structured files or network models of data, which results in the same kind of error for application programs whenever the structures are changed. A solution would be to program the application to test for what kind of tree structure is used, but it is generally unpractical and may not hold for when new structures are introduced.



parkingZoneID	name	column2
123	Lund High School	
456	Lund University	

parkingID	parkingZoneID	parking_started	parking_ended
1	123	2019-01-01	2019-02-01
2	123	2020-01-01	2020-01-31
3	456	2020-02-01	2020-05-12

Figure 2.5: Parking example for the Relational Model

Codd then formally describes the relational model, the solution to the above problems. The model is built upon set theory and thus reduces redundant data. Some concepts introduced are still in use today; primary key, foreign key, and the normal form. These, together with sets and predicate logic, build the foundation of the relational model. A simple example is demonstrated in figure 2.5 where the primary key "parkingZoneID" in the ParkingZones table may be used as a foreign key in the Parkings table in order to lookup parkings that belong to a specific parking zone. Again, these data models acts on a higher, abstract level and does not describe how these operations would translate into machine code or how the structured data would be stored physically on disk.

It took a few years until the model gained traction, primarily due to costly competing alternatives developed in parallel to the paper's release. In 1974, IBM System R, started development. The system was also the first to implement SQL and was ready for testing around 1978 [17]. Oracle released their first version the same year and their second version the next year, which according to them was the first commercial SQL RDBMS [88]. The first public release of Ingres occurred in 1975 [93].

Customers with a genuine need for relational capabilities adopted truly relational systems, but the performance advantages of network and hierarchical systems for well-optimized transaction processing ensured that they retained a significant niche within the mainframe DBMS market [39].

2.3.2 The Object-Oriented Model

Object-oriented programming began with the release of the Simula language in 1962. Naturally, discussions on how the relational model lacked in areas from an object-oriented point of view took place, but it seems these only gained momentum by the early 1980s. The development of an OODBMS had started from multiple directions since its first mention in 1985: the first major project Orion Research Project and the commercial projects Servio-Logic (later Gemstone), Graphael, O2, V-base (later ONTOS) and Objectivity/DB [3].

A manifesto on OODBS, first published by the end of 1998 and later updated twice, listed requirements of an OODBS, as well as suggestions on how it could be implemented as

an extension to the relational model by allowing custom user-defined types [24]. Around the same time, efforts on standardizing OODBMSs were made cooperatively by five prominent OODBMS vendors, resulting in a standard object definition, query, and manipulation language by 1993 [16]. Even though the joint formation of the OODBMSs vendors was disbanded later on, their work influenced the scene such that support for object-oriented languages, such as C++ and Java, as well as seeing SQL:1999 add partial support for object-oriented features [3] [75].

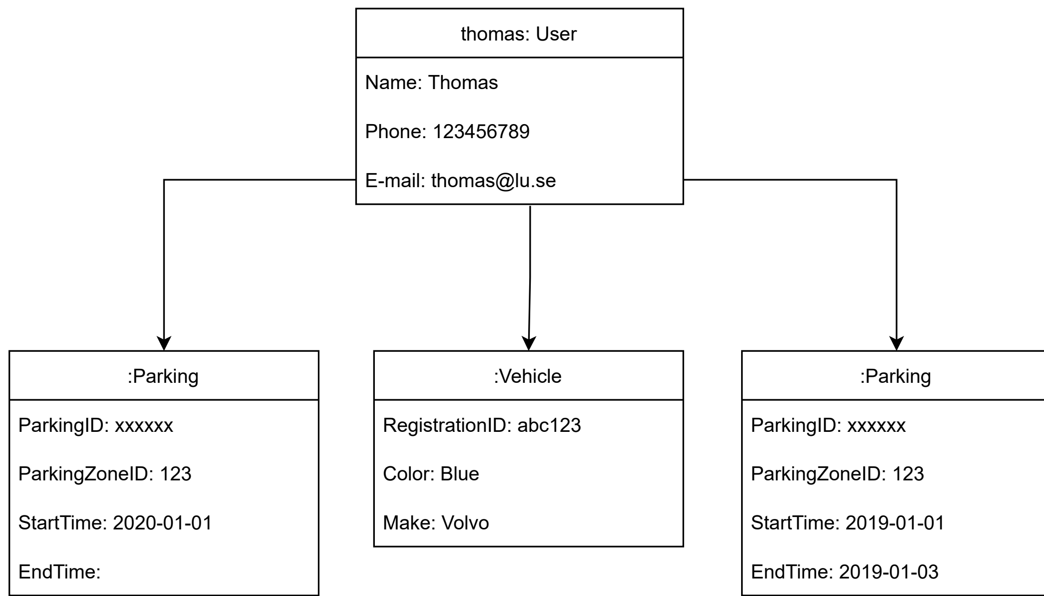


Figure 2.6: Parking example for the Object-Oriented Model

Unsurprisingly, an OODBMS stores its data as objects. This enables many other OOP practices such as encapsulation, complex objects, inheritance, polymorphism, and extensibility [3]. One of the drawbacks of relational databases was (or is) the difficulty of introducing more complex structures. For example, the data in relational databases are stored in tables and rows with primitive data types. Further drawbacks when compared to the OODBMSs are the fixed set of operations on tuples with SQL, problems associated with concurrency, and schema changes [102]. This led to the exploration of introducing object-oriented features, both as a standalone solution and as an extension to the existing relational model.

Figure 2.6 showcases a simple example of the object-oriented model when applied to our parking data. Here, we can represent a user, his parkings, and his vehicles registered on his account. If represented with the relational model, it would require a table for each "type" and fetching this information would require the use of foreign keys.

2.3.3 The Object-Relational Model

As the name might suggest, the object-relational model incorporates some object-oriented features into the relational model. This was done by the RDBMS community in response to the growing popularity of OODBMSs. Here, the users may define, manipulate and query both relational data and objects using common interfaces such as SQL. Custom data types could be introduced through user-defined "abstract data types" (ADT). However, there were some

implementation challenges. For example, the user-defined ADTs needed efficient indexes and operators on structured objects. Additionally, the layout of data on disk storage had to be considered when storing large ADT and structured type objects [102].

Vendors eventually overcame these and published their finished products, along with papers describing their architecture. A comprehensive paper was published in 1996 with an overview of the architectural differences between OODBMSs and ORDBMSs. Even within categories, the products used different data models, different storage models and different object models. However, they all had in common the support for relational tables as well as objects [18]. In 1999, the SQL:1999 standard was released, which included support for user-defined types [27].

2.3.4 NoSQL

The term itself was first used in 1998 when Carlo Strozzi named his open-source database solution that did not use SQL yet was still relational. Carlo later pointed out that the general usage of "NoSQL" from 2009 and onwards had a misleading term as the community instead used the term for solutions that dropped the relational part of databases completely. In contrast, his solution was still a relational database [15].

Regardless of its origins, it seemed like NoSQL would be a strong contender to RDBMSs. With the continued spread of the internet, the amount of data needed to be processed spiked worldwide. Data warehouses and online transaction processes that ran on RDBMSs, at least by 2009, could on many cases be beaten by a factor of 50 when compared to NoSQL databases. Text applications and their billions of messages had long ago experimented with non-relational solutions [107]. On the same note, the cost of disk storage decreased over the years that followed, further adding to the idea of leaving the relational DBMS behind.

The main advantage of NoSQL comes from its definition: by not conforming to the relational model, strict schemas are not used. This benefits flexibility and scaling over time since schemas can be changed on the fly. Additionally, with a relational database, many modeling scenarios would likely require joins to extract all of the information we want. In contrast, NoSQL can achieve this without joins [100]. Currently, four major branches of the umbrella term NoSQL exist.

Document Stores

A database set up as a document store contains documents, which are entities that contain named string fields against their object data values. These documents are mostly formatted as JSON. Because of this, the documents can contain multiple categories or even sub-collections of nested information (containing IDs of other documents), something which would require several tables and joins in RDBMSs. The fields within documents are exposed to the storage management system, allowing applications to query and filter data by the fields [76].

Documents are usually retrieved by a hashed key for even distribution, although some applications allow attributes to serve as keys. Some applications also implement indexing methods on one or several fields for faster lookups. Lastly, some applications allow for in-place updates where only relevant fields are updated rather than updating the whole document [76].

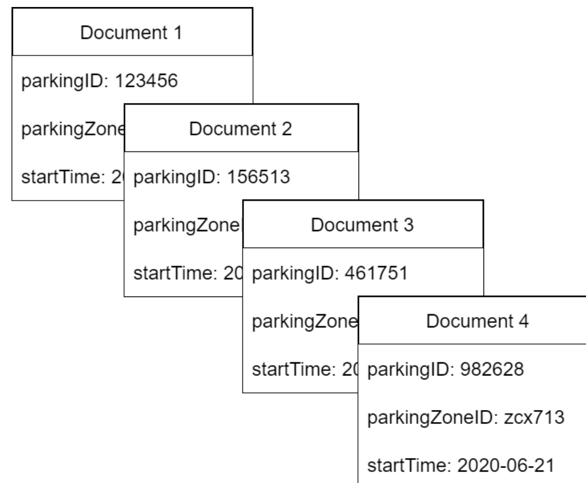


Figure 2.7: Parking example of Document-Stores

Figure 2.7 shows a small example of using documents when representing parkings and parking zones. Note that the documents are independent of each other - shared attributes, in our case, the common parking zone IDs, are what "links" the documents together during queries.

Graph-Data Stores

Also known as a graph database, this model holds nodes or entities that contain our data and the edges that connect these nodes [76]. The graph concept is similar to that of the network model seen with the IDS. However, like with the document store, there are no schemas that the nodes need to follow. This model is great for drawing up graphs with hierarchies, for example showing employees, their departments, and the connection between different levels of entities.

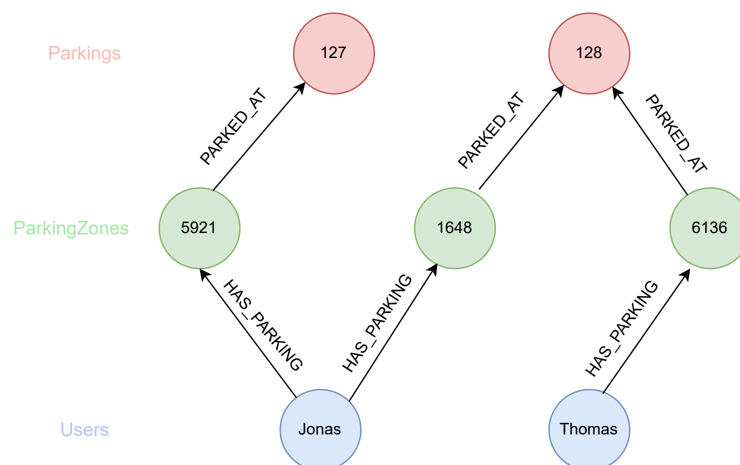


Figure 2.8: Parking example of Graph-Stores with red and green nodes representing parking zones and parkings, respectively

Figure 2.8 showcases a small graph with parking zones, the parkings parked at those zones, and the user that has the parkings. Edges can be directed or undirected, and the edges may

also contain data like the start time of a parking or what vehicle ID is registered on that parking. As with the document store, the graph store is schemaless. This means that the user has more freedom in how to model their data and use case to the graph.

Column-Data Stores

Since a relational database typically stores its data in tables with rows and columns, a column-data store might at first sight seem familiar. This model has similar benefits to the other NoSQL models, that is, not having to conform to a specific schema. The columns are divided into column-families, which in turn may hold a set of columns that are logically related and usually queried as a unit. Columns may be added dynamically, and a row does not need to have a value for each associated column.

Another benefit is that unlike a document store or a key-value store, most column-data stores physically store data by key in order rather than using ID derived from a hash. All columns within a family are stored in the same file on disk, allowing for much faster retrieval by reducing the amount of data to be read from disk when few columns are queried [76].

ParkingID	Driver
001	DriverID: 6163 Name: Thomas Phone: 123456789
002	DriverID: 3617 Name: Jonas Phone: 987654321 E-mail: jonas@lu.se
003	DriverID: 7592

ParkingID	Times
001	StartTime: 2020-01-01 EndTime: 2020-02-01
002	StartTime: 2020-05-01
003	StartTime: 2020-03-01

Figure 2.9: Parking example for Column-Data Stores

Figure 2.9 demonstrates our parking data represented with the column-data store model. In this particular example, as with most column-stores, the parkingID is sequential (not hashed) and used as the primary key.

Key-Value Stores

This model can be described as a large hash table. In that sense, it might not be a good fit for those wishing to migrate from RDBMSs since lookup capabilities by value are limited, and writes to large values are slow. Most implementations only allow simple queries, insertions, and delete operations. Furthermore, when updating values, the entire value must be overwritten as opposed to only modifying the target data.

The benefits of this model are optimized, simple lookups by key as well as scalability since data can easily be distributed across separate machines [76]. Figure 2.10 shows parkings and parking zones represented in key-value stores. Note that while this example shows values containing multiple attributes, querying for these will be slow and is not the recommended usage for this model.

Parkings

Key	Value
parking1	{driverID: 3617, name: Jonas}
parking2	{driverID: 7172, name: Tommy}
parking3	{driverID: 1792}

ParkingZones

Key	Value
parkingZone1	{name: Lund University}
parkingZone2	{name: Lund College, address: Storgatan 3}
parkingZone3	{name: Lund High School}

Figure 2.10: Parking example for Key-Value-stores

2.3.5 Temporal Models

The research on how to best implement temporality was parallel to the development of the database models. The first suggestion on temporality within databases was published in 1956, long before the relational model or the release of the IDS [49]. However, it was mostly after the release of the relational model that the research gained traction. Many proposals came over the years that followed, each with their different takes and concepts on how to model time in databases. The proposals have been divided into valid time, transaction time and bitemporal models. These have been published parallel to each other, not in consecutive order.

Snodgrass et al. [49] published an overview of temporal models that had been proposed up until 1995, which included their differences and strengths. Not all models will be included here since some, for example, differ only in treating periods as either open-ended or closed.

Valid Time Models

The first model using valid time was proposed in 1975 by Wiederhold, shortly after the paper on relational models [117]. The model used an array storing dates, such as the visiting dates of a patient. A patient's attributes could then refer back to the indexes of this array. For example, the temperature of the patient could be recorded for the first two visits as (index, temperature) translating into (1, 37.0), (2, 37.2) [49]. Multiple measurements of multiple types could then be saved for a single date, with time intervals supported as well.

The model LEGOL 2.0 was introduced in 1979 by Jones and proposed adding two implicit attributes to the relational model, "start" and "stop". This model was the first to define a time-oriented algebra [51]. Models that shortly followed proposed different ideas on whether or not objects are snapshot relations (Ariav, 1986 [5]), valid time relations (Sadeghi, 1987 [95]) or both (Navathe, 1987 [80]), as well as slightly different takes on start and end time attributes versus a single attribute, "Period" (Sarda, 1990 [98]).

Many valid time models that followed used attribute timestamping, sometimes with multiple values per attribute. One model also proposed timestamping not only individual attributes of a tuple, but also the tuple itself [20]. Another model used the set operators union, difference, and complementation with timestamps rather than using period intervals [33].

Transaction Time Models

The first transaction time model was proposed in 1978 by Kimball, shortly before the second valid time model [59]. The model did not show how tuples were timestamped, or how times

and facts are associated [49]. In other words, the query language and display of facts did not show any transaction time. The next model published in 1986 by Stonebraker was similar, although differed in that a sequence of state-tuples could be displayed [108]. The last model published in 1991 by Jensen similarly did not allow display or access of timestamps of facts. However, it had a separate log that contained the full timestamped history of tuples along with its operations (insert, delete or update) [50].

Something common with all transaction time models is the property of append-only timestamping, as mentioned in section 2.1.4. It emphasizes the logic that transaction times need to persist in order to truly "replay" what was known throughout time.

Bitemporal Models

As seen earlier, valid time and transaction time are defined as the time period at which something was true in modeled reality and when this was true in our database, respectively. A bitemporal model tracks both of these simultaneously, with the first model published in 1982 by Ben-Zvi [9]. This model has two columns for valid time; one for its start time and one for its end time. Two columns for transaction time were also introduced, where the start time is the timestamp at which we registered the valid time start time, and the end timestamp is the time at which we registered the valid time end time. A fifth column was also present and contained the timestamp of when a tuple had been logically deleted.

As we recall from section 2.1.4, a logical delete refers to ending the validity of information while still allowing it to exist in our database. This follows the fact that we wish to append-only when managing transaction times. In this model, we can see tuples that do not have a timestamp value in the fifth column to be current, whereas tuples that do are historical values that were true in the past.

McKenzie proposed a model in 1988 that used attribute timestamping, although restricted attributes to be single valued. This would allow for the ability to produce Cartesian products more easily than with set-valued attributes due to implementation difficulties [74]. However, this also meant that multiple tuples for the same key would be introduced whenever there were multiple attributes to track for any key. The benefit of reducing redundant data with attribute timestamping would then be lost, as well as introducing the risk of coalescing.

Thompson et al. proposed in 1991 a model with four timestamps that roughly corresponded to valid time and transaction time. An additional column held a boolean value that would represent whether or not attribute values could change historically [110].

Snodgrass et al. also compared all the models against each other and gathered what they thought would be the best properties. This resulted in them proposing their own model: the bitemporal conceptual data model. This model was unique compared to all the previously mentioned models since it timestamped tuples with bitemporal elements (valid time - transaction time pair) in an attribute-timestamping fashion, making it a coalesced model.

2.4 Implementing Temporal Databases

The temporal models are logical models that need to be physically implemented. The models in section 2.3.5 have mostly been designed around the relational model since it was the

prevailing model at the time. Unfortunately, source code for existing DBMS at the time was commercial and proprietary, making it hard to implement bitemporal properties in a "native" manner. Despite this, suggestions for implementing some models have been proposed and made easier with recent years.

2.4.1 SQL-based Implementations

In 1998, Steiner [105] discusses some ways to implement a temporal database. These approaches are very similar to implementation proposals of later years, with minor differences such as at what stage modifications are done to the DBMS.

Layered Implementation

In Steiner's first example, a front-end to Oracle DBMS was developed that essentially translated temporal queries into standard SQL statements. This way of using a software-bridge between the user-application and the DBMS is also known as the *layered approach* [48].

Integrated Implementation

The second example is to extend relation schemas, allowing them to store time-varying data. For instance, we could directly implement some of the models discussed earlier by adding valid time and transaction time through one or several columns in an RDBMS. This method, modifying internal modules of the DBMS, is also known as the *integrated approach* [48].

Steiner then presents integration examples of the Temporal Object Model, a temporal generalization of the object model [85]. One approach involves implementing it from scratch with a new query language, resulting in the DBMS "TOMS". As with most other DBMSs, the data persists in files and supports transactions.

Another approach involves defining an abstract data type for time. In his example, this was done in the object-oriented DBMS "O_2". In short, a root-class supporting time attributes is created and is accompanied by custom methods operating on the objects. This class would then be an abstract data type that objects could inherit if temporal properties are to be used.

Constraints

In all of the above cases, the matter of *constraints* needs to be considered. Briefly mentioned in section 2.1.2, constraints are rules that dictate what states of the database are legal. These are a part of the data model; for those mentioned in section 2.3.5, they are expressed as mathematical rules with set theory since the models are based on the relational model. When extending Oracle with the front-end solution, constraints were handled in separate "meta tables" that stored information such as table names, referencing columns, their corresponding referenced columns, and assertions that handle temporal assertions.

With integrated implementations, methods and functions still needed to be manually written in order to verify constraints during operations. For example, when implementing TOM from scratch, algebraic operations on collections were used to confirm constraints, similar to those used with relational models.

The layered approach is generally faster as the DBMS itself is unmodified and uses available indexes, whereas the system will have difficulty adapting to custom implementations [105] [48].

2.4.2 NoSQL Implementations

Document-Stores

Unsurprisingly, a suggestion on how to implement bitemporality with document-stores is introducing timestamps for valid time and transaction time on the documents as separate fields [29]. This was modeled in figure 2.7, and implemented in figure 4.4. Because a document can contain other documents, the author also discusses a constraint that should be applied to child documents: the timestamps of the parent document must "cover" those of its children so that no child is valid before or after its parent [10]. This constraint was also mentioned by Steiner when implementing the TOM from scratch [105]. For example, assume we have a document for a parking zone. This parking zone contains a list of other document IDs that represent parkings that are parked at this zone. Then, it would not make sense for a parking (child) to have a validity time that precedes that of the parking zone (parent).

Column-Data Stores

Eshtay et al. demonstrated how one could model and implement bitemporality into Cassandra, a Column-Data store [29]. This was done by adding the columns valid time start, valid time end, and transaction time as a single timestamp. When adding new periods, the values are added to the column while still corresponding only to one key, as seen in figure 2.11. When implementing this with code, the authors used a map with the transaction timestamp as a key, and its corresponding value would contain an entry with the salary and its valid times.

ParkingID	Cost
001	(10, [2020-01-01, 2020-01-02]), (20, [2020-01-02, 2020-01-03])
002	(20, [2020-01-07, 2020-01-09])
003	(15, [2020-01-07, 2020-01-08]), (30, [2020-01-09, 2020-01-12])

Figure 2.11: Implementation example for Column-stores

Key-Value Stores

Eshtay et al. also demonstrated the implementation for Redis, a Key-Value store. Here, they proposed adding timestamps to the key-value tuple as additional attributes. A more complex solution with a composite key was also presented. Here, a key consists of a major key, which can be seen as the regular ID, combined with a time-key that "represents the interval of the valid time" [29]. This may be a slightly unclear description since the example demonstrated in figure 2.12 seems to use transaction time as the time key. Unfortunately, we were unsuccessful in contacting the authors for clarification.

Key	Value
parkingZone1:20220101	{name: Lund University, vtStart: 20220101, vtEnd: 99999999}
parkingZone1:20220314	{name: Lund University, vtStart: 20220101, vtEnd: 20220331}
parkingZone1:20220329	{name: Lund College, vtStart: 20220401, vtEnd: 99999999}

Figure 2.12: Implementation example for Key-Value-stores

Graph Databases

One thought would be to apply the same tactic with document-stores and add timestamps to each node alongside its field. However, taking advantage of the fact that graph databases allow attributes on relationship edges, Lju Lazarevic thought of letting each entity have its own set of state nodes, where the relationship to these are timestamped on both time axes [62]. Our implementation of this is shown in figure 2.8.

2.5 The SQL:2011 Standard

Despite the many years of research, it would take until the end of 2011 before the SQL standard with temporal features was published [61]. The standard included much of what has been discussed in section 2.3.5, such as the introduction of the PERIOD data type we have seen in section 2.3.5, application time tables, system-versioned tables, temporal primary keys, and temporal predicates (e.g. overlaps, contains). We note that temporal joins and coalescing are not mentioned in the standard. Another important thing to note is that SQL is "merely" the query language and that the new standard still has to be implemented by developers and vendors.

A paper released in 2018 surveyed the state of databases regarding how well SQL:2011 had been adopted by major vendors [11]. The products listed with bitemporal support were IBM DB2, Oracle, and Teradata. SQL Server only supported system time, and PostgreSQL did not have any native support at all. The paper also briefly discussed the implementations of temporality, such as IBM using a current table and a history table, or that PostgreSQL uses the GiST index to query efficiently over range predicates.

2.6 Indexing

In the introduction, we briefly discussed the need to sort data in order to optimize lookups and mentioned that the relational model arrived in 1970. The art of sorting, or indexing, was its own category and had been researched long before any DBMS, although modern indexing referred to the kind of lookup tables we see in the back of books. In the context of databases, an index refers to the strategy or data structure used to simplify the search for data. As briefly mentioned in section 2.2.2, the introduction of hard disk drives introduced new problems in terms of indexing such as disk space, how to manage operations on data with indexes, and the fact that one index would not fit all use cases [39].

2.6.1 Hash Index

In 1953, Hans Peter Luhn, a senior research engineer at IBM, proposed internally the idea of putting information in "buckets" to speed up searching. An example was applied to the phone number 314-159-2652: group the digits into pairs of 2, giving us 31, 41, 59, 26, 52. Each pair is then added together, and the result of each pair would be a single digit, or if the result were two digits, you would select the last digit. The result would be 45487, the bucket ID into which to insert this phone number. When it was time to look up this number, you would quickly calculate the bucket ID and search through the bucket, which would be much faster than searching through the complete list. In other words, it was the birth of the early hash algorithm [106].

2.6.2 B-tree

The workings of the well-known binary search tree, or BST, were researched by several individuals independently. One of the earliest versions used [118] was also one applied to the magnetic tapes discussed in section 2.2.2. The B-tree is a generalization of said BST [8] and, in short, involves nodes with a child node on the left and a child node on the right. The value of the left child node is less than that of the current node, whereas the value of the right node is greater than that of the current node. Thus, a divide-and-conquer strategy is used. This structure is self-balanced, meaning it keeps its height small to minimize search time, and is also sorted. This is a much-needed property as it allows for operators beyond equality, such as less-than or greater-than.

2.6.3 GiST

The Generalized Search Tree index is, as its name suggests, a generalization of the search tree [41]. GiST is extensible because it supports data types to index, as well as the queries related to the types. Additionally, the GiST can implement B-trees and R-trees, and others as extensions which results in a single code base for indexing multiple applications. Today, we see usage in PostgreSQL and Informix Universal Server.

2.7 Query Planning & Optimization

A query sent to the database describes what data we wish to extract. The receiving DBMS then needs to compute *how* to retrieve the results by generating a so-called query execution plan. The DBMS considers multiple execution alternatives through one or several optimization phases. During the early years of relational DBMSs, these were syntax and semantics analysis, physical query optimization, and query execution [32].

Concrete examples include considering whether or not to use indexes, the join order of several tables, and what algorithms to use when doing so. Over the years that followed, the complexity of queries increased, which in turn gave rise to better query optimization with, for example, logical query optimization or query rewrites.

Query planners and optimizers now exist in most DBMSs. Many also include the functionality to show how a given query is planned, as well as a breakdown of planning time and

actual execution time. This provides the user with more insight into their queries and an idea of how the DBMS traverses the user's schemas. It is possible to override the query planner in rare cases where the user is able to provide a more efficient strategy.

Query caching is another optimization method that might become relevant for us when benchmarking. MariaDB supports this feature, although it is disabled by default. If enabled, results of SELECT queries are stored. During consecutive queries, the current query is matched against stored queries. If a match is successful, the cached result is retrieved. The cached results are cleared whenever data has been written to it [71].

Finally, we mention the ability to access tables using the indexes mentioned above. The query optimizer may opt to skip the use of indexes in some instances, with the most important one being if the query requires most of the table rows [43]

2.8 ACID & CAP

In 1981, Gray discussed the properties atomicity (A), durability (D), and consistency (C) in the context of databases. He meant that transactions in databases should fulfill these requirements. These were all or nothing (A), effects survive failures (D), and a correct transformation (C) [36]. Two years later, Reuter and Härder would complement with isolation: "events within a transaction must be hidden from other concurrent transactions", resulting in the ACID we know today [37].

CAP stands for consistency, availability, and partition tolerance. We note here that consistency in CAP refers to cluster-wide consistency as opposed to ACID, which refers to an individual server or node. Availability refers to the property of always returning a response, even if that response may not be the expected one due to heavy load. Partition tolerance assumes that since communication between nodes is unreliable, the system should be tolerant against communication breaks [35]. The theorem itself was presented by Brewer in 2000 and simply stated that any shared-data system may only fulfill at most two of these properties [12] In other words, it is not possible for a networked system (which is subjected to communication errors) to implement atomic reads and writes while also ensuring a response to every request [35].

Brewer himself and others have raised concerns about how the theorem has been interpreted and used in academia [13]. Nowadays, some operations can be tied to full "ACID consistency" through tuning in NoSQL tools, which argues that it would be simplistic to categorize a system as just covering two of the three properties in CAP [67].

2.9 Benchmarks & Comparisons

The competition was extremely fierce during the early years of DBMSs, not only between vendors but also between the models discussed in section 2.3. As benchmarks could potentially be misleading and biased, the need for a standardized suite rose. One of the first greater attempts was proposed by Gray et al. in 1985 with an article that outlined a test for online transaction processing [30]. Unfortunately, the problems seem to have persisted either way and lasted until 1988, when eight companies formed the Transaction Processing Performance Committee (TPC). Their first benchmark, TPC-A, was released in 1989 and was based on

the then widely used DebitCredit benchmark [112]. The benchmarks would stay the industry standard for many years, releasing updated versions as the tests became obsolete [65] [64]. Despite this, papers that used the early versions TPC-A and TPC-B seem to be scarce.

Additional papers have been published along our timeline of databases, with papers comparing both quantitative and qualitative attributes of products. These have been done between SQL-based models, within NoSQL categories, and between SQL and NoSQL. This section lists some of the relevant papers that includes factors which will be used in our paper as well.

2.9.1 SQL-based

As discussed in section 2.3, object-oriented databases and object-relational databases were introduced as a response to some of the drawbacks in traditional relational databases. Benchmarks and papers discussing qualitative advantages were then published in order to persuade the community.

Qualitative

A comparative analysis between relational and object-oriented databases, published in 1990, argued that OODBMSs were superior. Generalization allows for more structure and user-defined types, schemas are smaller because of inheritance and, in their specific case of using the OODBMS Iris, the GUI was easier to use [57]. In general, this seemed to have been the main argument driving the use of object-oriented databases forward [103] [7] as well as in later years [91], even though commercial options seem scarce.

In 1996, a paper compared many of the commercially available object-relational and object-oriented databases. The main takeaway was that although they share the same category and model, they differed substantially [18]. Some factors were availability, scalability, tool support, and to what extent configuration is possible. With the ORDBMSs, an essential factor discussed was the support for object-oriented concepts.

Quantitative

In 1988, a benchmark between OODBMS VBase and three other relational DBMS was conducted [25] using the "Sun benchmark" [94]. A sample data set of persons, documents, and authors was used. The benchmark measured the results of attribute lookups and insertions. Although the benchmark used was simple (it had not been fully developed at the time), the results showed that OODBMSs could perform as well as, or sometimes better than RDBMSs.

A thesis published in 2007 compared an RDBMS to an ORDBMS and found that the object-relational tests were marginally slower for selects, deletes, inserts and updates. However, object tables with one object reference were 52% faster than a relational join [84].

In 2013, a study with a performance test was conducted between the RDBMS SQL Server and the OODBMS DB4O. The results were that SQL Server performed better with selects, inserts, and updates when the number of objects and object complexity was low, but the opposite was true when these were increased [99].

2.9.2 NoSQL

Non-relational tools as we know are relatively new; however, that has not stopped their growth and popularity. Unlike the SQL-based models, which offer three distinct categories, NoSQL mainly offers four, resulting in a larger pool of tools to evaluate. Many benchmarks have been tested using YSCB, an open-sourced benchmarking suite often applied to NoSQL databases. Average latency, runtime, and average throughput are measured [22].

Qualitative

In 2015, a qualitative evaluation of seven NoSQL tools was published [67]. Of these, two were Column-Data stores, two Key-Value stores, and three Document-stores. Some attributes evaluated included its position in the CAP theorem, configurability, maintainability, scalability, querying possibilities and mechanisms, concurrency control, and partitioning. The authors point out that while CAP is used, most tools allow for tuning of consistency and availability on a per-query basis. The final evaluation is a comprehensive one where each tool is evaluated on each point with a score ranging from "bad" to "great."

Another qualitative study released in 2019 investigated NoSQL storage solutions for big data systems [58]. The authors first mentioned the requirements of a large data storage system: scalability, availability, security, and integration. Since NoSQL is schemaless, developers have more flexibility when sudden changes to structure occur. Other features among the 80 chosen tools solutions include product status (free vs. commercial), product type, support for full-text searches, data compression, languages supported, HTTP, and more.

Additional benchmarking and evaluation papers also mention similar factors [67] [101] [4]. In most of these papers, the main takeaway is that many categories of NoSQL solutions exist with great differences in model, usage, and performance. What to choose ultimately depends on the user's use case and scenario.

Quantitative

One of the earliest papers on NoSQL was published in 2011 and compared the features of Cassandra and HBase, both being Column-Data stores [114]. When comparing both read and write latency in a write-intensive environment, for around 120 million small records, HBase performed much better since commits were done to memory rather than to disk, as done with Cassandra.

In 2015, a paper compared the performance of Cassandra, MongoDB, and Riak; a Key-Value store [60]. The tests were run on two server configurations: one with a single node and the other being a setup of nine nodes simulating a production environment. Tests included read-only, write-only, and hybrid workloads. As mentioned earlier, some tools allow for tuning its stance within CAP; here, the authors ran tests on both strong and eventual consistency, where strong consistency prioritizes equal data across nodes but sacrifices performance [66]. The results showed Cassandra had better throughput but the highest latency. This was due to the spread of request and storage load, indexing, and coordination of read/write operations across nodes. Additionally, setting a strong consistency reduced throughput by 10-25%.

The next year, another paper compared Cassandra and MongoDB while also adding HBase to the tests. [1]. The tests varied both the amount of data nodes and client nodes, with the

largest test involving a total of 4096 threads and 480 million records. For all tests, Cassandra performed the best.

In 2021, Cassandra and MongoDB were once again put through a benchmark, this time with the in-memory Key-Value store Redis [101]. The data set consisted of 800 000 records and multiple runs of the 100 000 operations workload. The general results were that Redis performed better with reads and updates, whereas MongoDB performed better with writes.

2.9.3 SQL vs. NoSQL

Last but not least are the papers that compare and evaluate the relational model against non-relational models. The qualitative differences may often align with the quantitative ones; for example, the degree of consistency or ACID properties usually affect the performance of operations.

Qualitative

The same early paper that benchmarked Cassandra and HBase also included MySQL as a reference point [114]. Besides the core difference between non-relational and relational models, MySQL was listed as having full ACID support, whereas Cassandra and HBase were eventually and locally consistent, respectively.

In 2018, a qualitative paper used the factors we have seen before: scalability, flexibility, and performance [96]. However, this paper also included security as a factor during evaluation. The paper concludes that if data integrity and consistency are the top priorities, relational databases should be used. Examples given are those with sensitive data, such as banking or accounting records. On the other hand, if scalability, availability, and performance are priorities, NoSQL is the preferred option.

In one of the later papers, a general comparison of the relational model against non-relation models was made [77]. Released in 2020, the paper first discusses ACID and it being a relational property, suited for systems with structured data and low growth. NoSQL is linked to the CAP theorem and is best suited for systems with un/semi-structured data and larger growth. The conclusion, as with the previous paper, was that the relational and non-relational models are not mutually exclusive. Instead, they are complementary, have their advantages and disadvantages, and each fit different scenarios.

Quantitative

When MySQL was introduced in the comparison between Cassandra and HBase, it was found that within the write-intensive environment (7000 writes per second), the read and write latency of MySQL exceeded the values of a "real life application." In the read-intensive environment, however, it bested both Cassandra and HBase [114]. However, the authors note that they do not believe this will hold as the size of the data increases.

Another early paper was published in 2013 and compared the document-based MongoDB to the relational SQL Server [90]. The comparison was made through benchmarks with select, insert, and update queries. The results showed that MongoDB performed better with simple queries, however, SQL Server performed better with aggregate queries and when updating and querying non-key attributes.

In 2016, MySQL, MongoDB, and HBase were benchmarked under four different workloads, where each workload varies the number of operations and thread count [122]. It was found that HBase suits applications with high update and insert operations, whereas MySQL suits applications with reads as its most common operation. MongoDB performed nearly as well as HBase and MySQL in terms of updates and reads, respectively. However, MongoDB performed moderately well regarding runtime and throughput. Its high memory utilization was also mentioned as a possible drawback.

A thesis from 2019 benchmarked Cassandra, MongoDB, and PostgreSQL with similar workloads to the previously described papers [86]. The results showed that PostgreSQL performed best overall, except in the case of updates, where MongoDB performed better if the number of updates was less than a million. Interestingly, Cassandra overall had the worst performance. The authors noted that this was not in line with other benchmarking papers. They explained that one possible reason would be the use of a single thread rather than having benchmarked concurrently, as well as using a single-node setup. In other words, as we have discussed, the use case and configuration heavily affects performance.

2.9.4 Temporal Benchmarks

Alongside the research of data models and temporal modeling in general, work on temporal benchmarks had also started [56]. In 1993, the TSQL2 committee mentioned in section 2.1 published papers on a test suite aimed at evaluating the user-friendliness of temporal query languages [46] [45]. Later that year, a similar paper presented temporal queries that could be translated into most of the temporal models present at the time [54]. In 1995, Dunham et al. provided a framework to benchmark indexing structures and algorithms for temporal DBMSs and also proposed some ways to generate application-independent test data [26]. In 1998, Werstein commented on the existing temporal benchmarks and concluded that they were insufficient. Instead, he proposed his list of unique queries in the sense that they require more temporal processing power and storage capabilities evaluating the ability of the database to handle three-dimensional data [116].

TPC-H & TPC-BiH

TPC-H is a decision support benchmark released in 1999, still active as of today [113]. The data simulates a business that holds parts, suppliers, customers, and orders. The volume of data can be chosen, and concurrent queries are supported.

In 2012, a paper proposed adding temporal columns to TPC-H [2]. However, the paper mostly outlines the possibilities rather than listing the data to test and queries. A year later, another paper complemented this idea by extending the queries and update scenarios of TPC-H with a broader range of cases [56].

2.10 Survey of Tools

Today, a wide range of solutions exists in terms of DBMSs, of which many offer some temporal support. Here, we briefly list some of the most popular options and their leading features.

	Temporality	OS-support	Scale	License	Transaction
Oracle	Bitemporal	Extensive	Enterprise	Commercial	ACID
MySQL	None	Extensive	Mixed	GPL	ACID
MariaDB	Bitemporal	Extensive	Mixed	GPL	ACID
SQL Server	Transaction	Linux, Microsoft	Enterprise	Commercial	ACID
DB2	Bitemporal	Extensive	Enterprise	Commercial	ACID
SQLite	None	Any	Mixed	Open-source	ACID
Teradata	Bitemporal	Linux	Enterprise	Commercial	ACID

Table 2.1: Some of the most popular RDBMSs and product status

Many products offer multi-model solutions. However, we have chosen only to include tools that specialize in one category.

2.10.1 Relational DBMSs

Most of the vendors mentioned in section 2.2 are still active today as the most popular choices for relational databases. These include MySQL, Microsoft SQL Server, and IBM DB2. In 2000, SQLite was released as a lightweight, open-source alternative. Additionally, we have Oracle released in 1979, Teradata first released in 1984 and MariaDB released in 2009, all of which support bitemporal tables.

Since these tools share the same data model, their differences lie in technical features. Some of these have been listed in table 2.1 along with their eventual support for temporal tables. The tools differ in many functional areas, such as minor differences in query syntax, triggers, parallel execution, and ability to perform rollbacks. However, discussing them for all of these tools is out of the scope of this paper.

2.10.2 Object-Relational DBMSs

It can be argued that any DBMS that supports custom data types can be called an object-relational DBMS. We have, however, chosen to only list tools which were initially designed as being object-relational: PostgreSQL and Informix. PostgreSQL was first formally released in 1997 but has its roots in the Ingres project in 1975 [93]. PostgreSQL is released under its own "PostgreSQL License", a form of open-source license. It has extensive OS support, may be used at a smaller scale or at enterprise level, however has no native temporal support.

Coincidentally, Illustra also came from the Ingres research project, now named IBM Informix, after first having been acquired by Informix and later IBM. Informix is a commercial product and is similar to PostgreSQL in its OS support and ability to be used at different scales. Unfortunately, there is no native support for temporal tables.

2.10.3 Object-Oriented DBMSs

Object-oriented databases seem to have lagged during the later years, perhaps due to the rising popularity of NoSQL. Table 2.2 shows some of the most successful options. Although Db4O was a popular open-source alternative, it ended official support in 2014. None of the

	Temporality	OS-support	Scale	License	Transaction
IRIS	None	Extensive	Enterprise	Commercial	ACID
Db4O	None	Extensive	Mixed	GPL	ACID
GemStone/S	None	Extensive	Enterprise	Commercial	ACID
ObjectDB	None	JavaVM	Enterprise	Commercial	ACID

Table 2.2: Some of the most popular OODBMSs and product status

	Temporality	OS-support	Scale	License	Transaction
MongoDB	Time Series	Extensive	Mixed	SSPL	CP
XTDB	Bitemporal	JavaVM	Mixed	MIT	ACID
MarkLogic	Bitemporal	Extensive	Enterprise	Commercial	ACID
CouchDB	None	Extensive	Enterprise	Open-source	Eventual C

Table 2.3: Some of the most popular Document-stores and product status

options in the table are listed as having temporal support. However, as we recall from section 2.3.2, OODBMSs have the innate ability to construct user-defined data types, which allows for the implementation of an abstract data type that encompasses objects with temporal features.

2.10.4 NoSQL

Tools within the NoSQL differ to such an extent that they are categorized separately. Most tools emerged after the mid-2000s with the non-relational movement.

Document-Stores

The most popular document-store is MongoDB, first released in 2009. MongoDB stores documents in collections [78]. Collections are analogous to tables in relational databases, which is in contrast to XTDB that stores its data only as documents. They describe themselves as a hybrid between document-store and graph database since they support Datalog queries that traverse graph relationships across documents [121]. A selection of document-stores and their properties can be seen in table 2.3, where we note that they differ in the transaction property as opposed to the SQL-based products.

As with XTDB, MarkLogic is a document-store based database with native bitemporal support. This is done by timestamping each document. However, unlike XTDB, collections are also used to distinguish between temporal and non-temporal data [73]. CouchDB currently has no temporal support.

Graph Databases

One of the earliest graph databases, and also one of the earliest NoSQL tools and most popular graph database today is Neo4j, released in 2007. None of the options listed natively supports bitemporal data, although OrientDB and TigerGraph support time-series [89] [111].

	Temporality	OS-support	Scale	License	Transaction
Neo4j	None	Extensive	Mixed	Open-source	ACID
OrientDB	Time Series	Extensive	Mixed	Open-source	ACID
JanusGraph	None	Extensive	Mixed	Open-source	ACID
TigerGraph	Time Series	Linux	Enterprise	Commercial	ACID

Table 2.4: Some of the most popular Graph-stores and their properties

	Temporality	OS-support	Scale	License	Transaction
Redis	None	Extensive	Mixed	Open-source	Varying CAP
Memcached	None	Extensive	Community	Open-source	CA
Hazelcast	None	JavaVM	Mixed	Open-source	Im/Ev C
etcd	Bitemporal?	Not OS X	Mixed	Open-source	Immediate C

Table 2.5: Some of the most popular Key-value stores and their properties

Column-Data Stores

Most options for column-data stores are only available as cloud services. The first and second most popular tools, Cassandra and HBase, have been extensively benchmarked, as seen in section 2.9. Both were released in 2008, are open-sourced, support immediate or eventual consistency, and can be run on almost any OS, although HBase cannot easily be installed on OS X. HBase supports ACID for singular rows. Unfortunately, neither option natively supports temporal data.

Key-Value Stores

The only key-value store tool seen in earlier benchmarks is Redis, released in 2009. Redis is the latest released tool out of those in table 2.5, with Memcached released in 2003 and Hazelcast in 2008. Like with most other tools, native temporal support does not exist. However, etcd mentions in their docs as of v3.5 [31] that previous versions of key-value pairs are available for "time travel queries". This means that valid time is supported. The key-value store is also immutable, giving us a bitemporal model if transaction times are saved as well.

2.11 Summary

We have introduced terms and concepts related to temporal databases. We have listed definitions, of which the two most important ones are valid time and transaction time. Valid time is the time in our modeled reality, such as the parking period. Transaction time is the time at which this information was known to be true in our database. We have briefly mentioned the history of DBMSs where the birth of database models, the evolution of indexing methods, and transaction concepts have been discussed. We have seen how one can implement the proposed temporal models and surveyed some of the tools available today in terms of their qualitative and quantitative performance, as well as their native temporal support.

Chapter 3
Methodology

In this chapter, we describe the process and steps required to fulfill the goals of this thesis. We have two main goals. The first is to provide a general overview of bitemporal modeling. The second is to implement the models from section 2.3 across different tools in order to evaluate the advantages and drawbacks of each. Therefore, the nature of our work is explorative. Inspiration on how to conduct our research and set up our measurements have been taken from section 2.9 and literature on thesis works [42] [119].

3.1 Process

Figure 3.1 shows an overview of the steps followed in this thesis to address the research questions. We have started our work by studying literature on temporal research, most of which have been research papers. Other materials include manuals, documentation, thesis papers, conference presentations, and historical summaries. The result of this step is included in chapter 2.

An important part of our literature study was gathering the qualitative and quantitative factors to consider during evaluation. These were listed in section 2.9, and the chosen factors for our evaluation are listed in section 3.3.

In section 2.10, we discussed some of the most popular tools available today. Together with the wishes of Parkster AB, the tools selected to implement bitemporal modeling on are MariaDB, PostgreSQL, XTDB, and Neo4j. More details about the reasoning behind this selection are described in section 3.2. For each tool, we describe in chapter 4 how the bitemporal models were implemented.

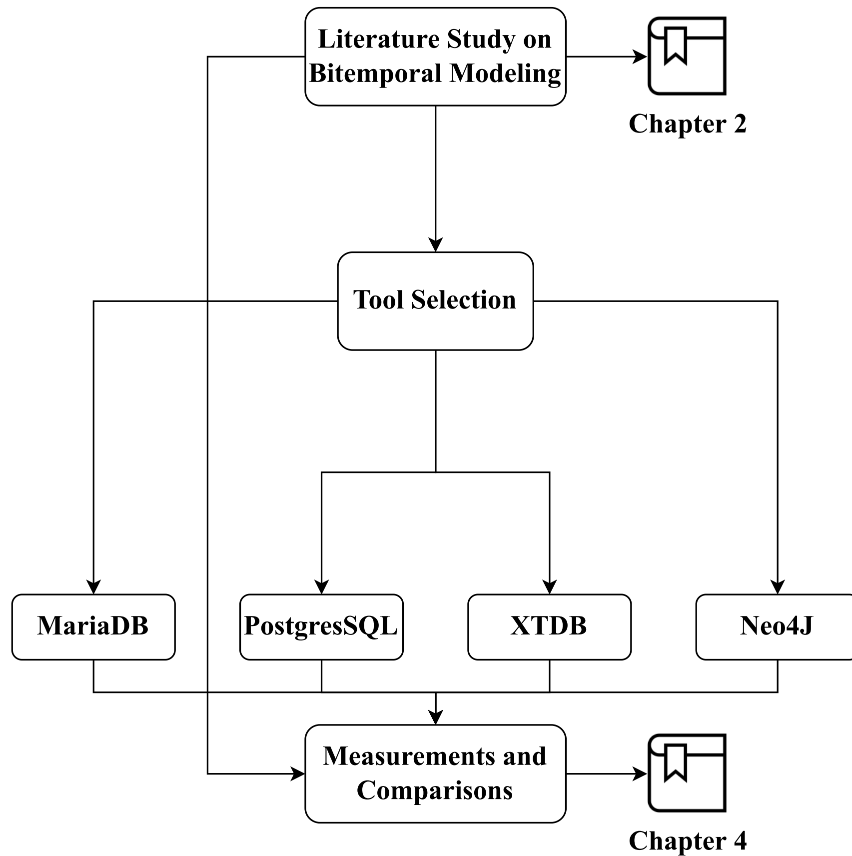


Figure 3.1: Overview of methodology for this thesis

	MariaDB	PostgreSQL	XTDB	Neo4j
Category	Relational	Object-Relational	Document	Graph
Bitemporal Support	Yes	No	Yes	No
License	GPL	Open-source	MIT	GPLv3

Table 3.1: Selected tools to be used in evaluation & benchmarks

3.2 Selection of Tools

In this section, we discuss and motivate our selected tools. The selection should aid us in our investigation of RQ1, where we survey current tools used in the industry. We primarily aim to cover a range of tools among the different database categories so that we may demonstrate and apply different models.

Our first choice was MariaDB, a relational DBMS with native support for bitemporal tables. Next, we chose PostgreSQL from the object-relational category, which does not natively support bitemporal tables. Finally, from the NoSQL category, we selected two different tools. These were XTDB, a document-based tool with native bitemporal support, and Neo4j, a graph-based tool without native bitemporal support. These tools and their core properties are shown in table 3.1.

3.3 Measurements & Comparisons

To aid us in answering our research questions, we identify what factors to evaluate in each implementation. Many factors come from previous work in the area, seen in section 2.9. We can divide these into *qualitative* and *quantitative* factors.

3.3.1 Qualitative

In short, the qualitative factors we investigate will mainly involve the functionality and features of the previously discussed tools. When comparing two or several tools, these factors will be put against one another.

Bitemporal Support

Perhaps the most critical factor in this thesis is how and which bitemporal features have been implemented. For this factor, we will primarily use the mentioned features in section 2.5 and, if appropriate, those mentioned in section 2.3.5. Among these, we will prioritize temporal primary keys, temporal foreign keys, temporal operations such as joins, and the types of queries available.

Configuration

Configuration involves managing system options. In our case, we narrow down the options to those that may affect performance, such as indexes and algorithms. If a tool offers many options, it increases the chances of being able to set up a system that works efficiently for the user's scenario. This may also be related to how well integration works with existing systems. Configuration is highly related to scalability and availability as these are affected by the available options and configurations.

Scalability

Scalability is the ability to add resources to a system or server to increase its performance. This property is usually split into horizontal scaling, and vertical scaling [79]. Vertical scaling is usually the most straightforward way and involves adding resources directly to the server. This could be done by switching to a more powerful CPU or adding more such as increasing the amount of RAM or disk space.

Horizontal scaling, on the other hand, also known as sharding, involves adding more servers who work in tandem with each other. The data is split across the servers in order to reduce workload. The second option is more expensive but gives additional benefits such as increased concurrency, better data distribution, and better backup options. As Parkster is still growing, so will their data, meaning that the ability to scale horizontally with a bitemporal solution would be desirable.

Availability

As mentioned in section 2.8, the A in "CAP" stands for availability and concerns the server's ability to respond to requests even during heavy load. Again, this does not guarantee that the response contains the most recent write. In our evaluation, we will consider if any response is returned during ingestion while also commenting on any eventual difference in response time compared to when not ingesting data.

Ease of use

This factor considers the many steps included when attempting to use a tool. These could be complications when setting up the tool, the query languages' complexity, and other quality-of-life features such as the user interface or documentation.

3.3.2 Quantitative

Considering the requirements of Parkster and their "Parking map", the following quantitative factors were considered relevant:

Ingestion

The speed at which we ingest data has an impact on choosing between our tools, as low speeds could increase the risk of having the database return data that is not up to date. However, ingestion, in our case, will solely be done through the Java application from Parkster, which is one of many ways to ingest data and not always the fastest method. Depending on the tool used, the dependent variable will be the number of rows or nodes ingested per second. This can be abstracted to the number of parkings ingested per second but can vary depending on the implementation and thus will be discussed later on.

Temporal Queries

While ingesting is done with insertion or updating queries, selection queries are done when one wishes to extract data from the database. In our case, we are mostly interested in queries where we set conditions on either valid time, transaction time, or both.

To begin with, we will once again consider the TPC-BiH paper described in section 2.9. Before presenting the queries, the paper discusses the importance of the tables used and their underlying data. For example, their first table of CUSTOMERS has a stable amount of rows but many UPDATE operations that would generate a large history. Their second table of ORDERS has a smaller history for each row, however has a significant number of rows due to a large number of insertions. In our case, we have different tables and implementations depending on the tool used. Still, the characteristics will be the same in that we will have many Parkings inserted with few updates, and fewer ParkingZones will have more updates when compared to Parkings. This paper will adopt the major bitemperal query categories listed in section 2.9:

- Point to point: query two different points in time, one for each axis. For example, query parkings at valid time 2022-12-31 and transaction time 2022-12-30.

- Slicing: keep one axis to a single point, query full range of the other.
- Pure-key audit: retrieve history for a single or small set of tuples across any single axis, or both. For example, track the evolution of some parkingZones and see how their attributes have changed over time.
- Bitemporal queries: These variants span both time dimensions and vary the usage of each.

The amount of parking data gathered at Parkster has increased each year. This means that the density of rows, in Parkster's case, may differ greatly across different periods. Queries regarding transaction time are a bit trickier when spacing out the dates since transaction time uses the current time. In our case, it means that the period depends on the ingestion time for each tool. Here, we employ four points that are equidistant to each other in regard to ingestion time. These are the timestamps at which 0%, 33%, 66%, and 100% of the data has been ingested.

	TT (% of ingestion)	VT (Year)	VT (Month)	# of Queries
Point-to-Point	0, 33, 66, 100	Year 1, Year 2	02, 05, 08, 11	32
Vertical Slice	Full range	Year 1, Year 2	02, 05, 08, 11	8
Horizontal Slice	0, 33, 66, 100	Full range	Full range	4
Pure-key, full	Full range	Full range	Full range	4
Pure-key	0, 33, 66, 100	Year 1, Year 2	02, 05, 08, 11	32
Bitemporal	33% increments	Year 1, Year 2	3-month incr.	21
Total				**101**

Table 3.2: Table of combinations of query types. First query, P2P, has VT = Year1-February and TT = 0%

Our chosen data set will be two full years of parking data. The chosen points, shown as dates, is listed in table 3.2. For example, with our Point-to-Point queries, we combine each possible value once, giving us 32 different kinds of queries. The first query would for example be setting valid time to February 2022 and transaction time to 0%. The next queries would iterate the valid time months May, August and November before switching to the next year using the same four months. When this has been done, we iterate transaction time by setting it to the point in time where 33% of the data had been ingested, and the valid time combinations are repeated for this transaction time point.

Full range means that we query across the whole axis rather than supply a given time instant. The pure-key queries are identical to the point-to-point queries, except that we include four chosen parkingIDs. Only in the bitemporal queries are ranges used, which are done in 33% increments for transaction time (such as 66% to 100%) and 3-month increments (such as May until August) for valid time.

3.4 Retroactivity

Parkster has provided an application that loads events related to parkings and parking zones. Ingestion times would be much faster if the data was parsed to, for example, a .csv and later

loaded. However, the end goal is to have this application forward future events so that the database may hold current data as well. Thus, ingestion is formed around Parkster's application where we, for each DBMS, must set up the code that is to be run for each received event.

Lastly, as Parkster uses event-driven architecture, there is a high probability that the application provides events in the wrong chronological order. For example, an event for "parking stopped" may come before the respective "parking started" for a specific parking ID. Another example is when a parking zone, initialized "2019-01-01", might have its name changed to "Lund University" on a certain date, say "2020-02-01". However, the next day we receive an event of another name change that actually took place "2020-01-01" meaning that the valid-time period for "Lund University" was not "2019-01-01" until "2020-02-01", but rather "2019-01-01" until "2020-01-01". This dilemma is demonstrated in figure 3.2 where changes required are shown in bold.

parkingZoneID	name	vt_start	vt_end
123	Lund High School	2019-01-01	2020-01-31
123	Lund University	2020-02-01	9999-12-31

parkingZoneID	name	vt_start	vt_end
123	Lund High School	2019-01-01	**2019-12-31**
123	**Lund College**	**2020-01-01**	**2020-01-31**
123	Lund University	2020-02-01	9999-12-31

Figure 3.2: Updates required when events arrive in wrong order

This, however, is a separate problem from solely modeling bitemporality. The solution to this would be to write ingestion code that detects where an insertion should happen on our valid-time axis and manage the intervals so that no overlaps occur unless the DBMS supports this natively. Any solution implemented for any tool will also be presented in this paper if possible. However, it will not be included in benchmarks as we primarily track the data of when parkings start and end.

Chapter 4

Execution & Results

In this chapter, we execute the setup, modeling, implementations, ingestion, and benchmarks for each selected tool as described in chapter 3. A table of our setup in terms of hardware and software is listed in section 4.1

4.1 MariaDB

4.1.1 Setup

MariaDB, unfortunately, does not offer an installation package for Mac OS. However, using Homebrew we can simply run "brew install mariadb". Starting and stopping the service is also done through brew without much trouble. Importing data can be done in many ways, however, we will as mentioned earlier use Parkster's application which will require us to use MariaDB's JDBC driver. The driver is loaded into our Java application as a Maven dependency, as with the remaining three tools.

Hardware	Software
Computer: Macbook Pro (2016)	macOS Monterey 12.1
Processor: 2,6 GHz Quad Core Intel i7	MariaDB 10.8-3 Homebrew
RAM: 16 GB 2133 Mhz LPDDR3	PostgreSQL 14.5-1
Storage: 500GB SSD	XTDB 1.20.0
	Neo4j 4.4.5

Table 4.1: Hardware and software versions of setup

4.1.2 Modeling & Implementation

As discussed in section 2.4, we have an array of possible ways to model bitemporal tables in relational schemes, however, they are mostly similar in that we need only track two time axes, most easily done by introducing four time columns to each table. This results in the general diagram shown in figure 4.1.

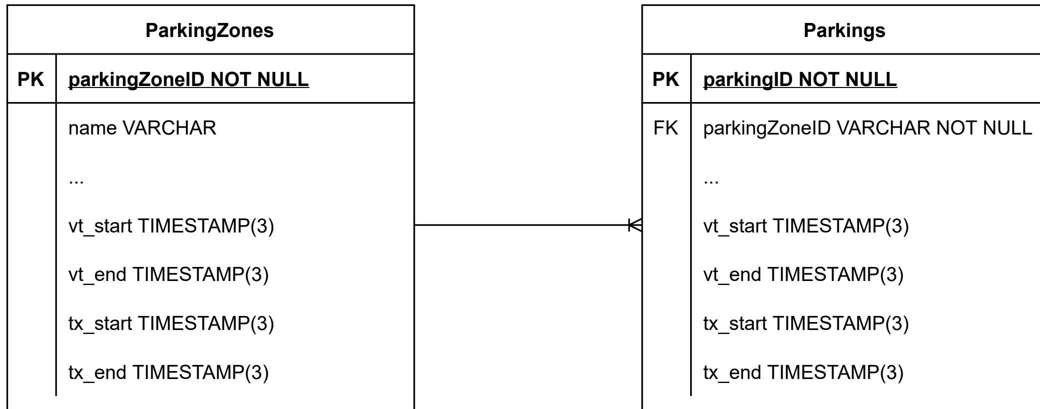


Figure 4.1: Modeling bitemporal tables in relational DBMSs

The implementation of bitemporal tables in MariaDB is very straightforward. The user creates a table in familiar SQL by giving the table and column names, along with their respective data types. Valid-time columns are applied with the PERIOD data type, as proposed by the SQL:2011 standard in section 2.5. The user supplies the name for the period, as well as the name for the period's start and end limits. The same applies when applying the period for introducing transaction time columns, however, the user must also add "WITH SYSTEM VERSIONING".

```
CREATE TABLE Parkings (
    parkingID           VARCHAR(64),
    parkingZoneID       VARCHAR(20),
    ...
    vt_start           TIMESTAMP(3),
    vt_end             TIMESTAMP(3),
    tx_start TIMESTAMP(6) AS ROW START INVISIBLE,
    tx_end TIMESTAMP(6) AS ROW END INVISIBLE,
    PERIOD FOR valid_time(vt_start, vt_end),
    PERIOD FOR system_time(tx_start, tx_end),
    UNIQUE (parkingZoneID, valid_time WITHOUT OVERLAPS)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 WITH SYSTEM VERSIONING;
```

Listing 4.1: SQL for creating a bitemporal table in MariaDB

System versioned tables that track transaction times in MariaDB have indexes created automatically. Since we will query across valid time as well, we create indexes on vt_start and vt_end. By default, B-tree indexes are created.

```
CREATE INDEX vt_index ON parkings (vt_start, vt_end)
```

Listing 4.2: SQL for creating indexes on valid time in MariaDB

Given the tables above, we recognize that *tuple* timestamping is applied. This means that for a change for any ID, a new row is appended with the updated information and new valid-time period. If the change were to be a correction, we instead close the end date of the row's transaction time, which signifies a "logical deletion" and add the new correct row of data.

Since "infinity" is not a supported data type, and the use of "PERIOD" sets a "NOT NULL" restriction on vt_start and vt_end, we may model an open-ended interval by setting the max value for either valid-time or transaction-time (or, for simplicity, 2038-01-01).

4.1.3 Ingestion

We primarily work with the loading of parking data received by Parkster's application. When we receive a "ParkingStarted" event, we are given the time at which a parking started along with its parkingZoneID. The starting time is set as "vt_start" and "vt_end" is set to open-ended, which in this case is represented by setting its value to "2038-01-01".

```
INSERT INTO Parkings (parkingID, parkingZoneID, vt_start, vt_end)
VALUES ('parkingid1', 'zone123', ("2022-01-01"), ("2038-01-01"));
```

Listing 4.3: SQL for inserting a "ParkingStarted" event

Unfortunately, the insert failed as MariaDB does not currently support using the "UNIQUE WITHOUT OVERLAPS" constraint together with Java and JDBC, which is what is used to ingest our data. However, since we believe this will be supported in the future, we will continue our discussions without changing the model. Testing can be performed by users manually through the terminal.

```
UPDATE Parkings
SET parkingZoneID = 'newZone'
WHERE parkingID = 'parkingid1'
AND vt_start = ('2022-01-01');
```

Listing 4.4: SQL for updating a parking

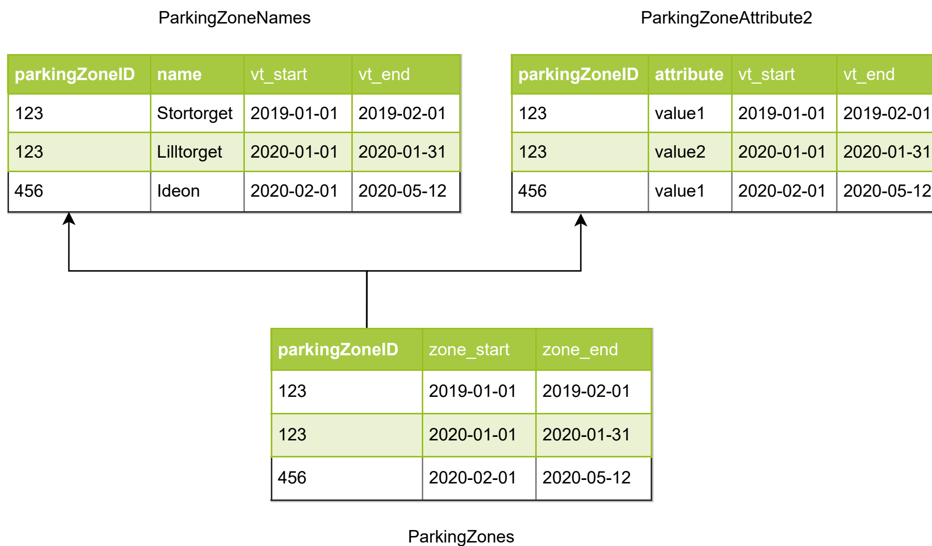
When a parking has ended, we logically delete the existing row and truncate it with a new row that contains the start and end time of the parking. Because MariaDB natively supports bitemporal tables, it is enough for us to run an UPDATE query. The parkingID we are working with will have a new row inserted with its parking start and end time filled, whereas the old row that only contained a start value is logically deleted. If MariaDB did not support bitemporal tables, these steps would need to be manually executed.

This procedure of using UPDATE cannot be used if the table contains multiple validity periods for a row. For example, in our case with the Parkings table, any parkingID will only have a single valid time period. After that, a parking will never be "re-activated", although it may have its timestamps corrected if we later realize that they were erroneous. ParkingZones may have multiple periods of validity when toggled active or inactive. In these cases, it is necessary to avoid the UPDATE query and instead manually perform logical deletions and insertions.

4.1.4 Retroactivity

As mentioned earlier, the order of events from Parkster may come in the wrong chronological order. This complicates our use case since no tool supports retroactive changes that are also carried forward to other rows of data. For example, in the ParkingZones table shown in figure 3.2, the name change of "Lund College" must also be applied to rows that share the same time period. This is problematic not only for data redundancy but troublesome when updating the time periods correctly.

With relational DBMSs, we can solve this by extracting *each attribute* into its own table, as shown in figure 4.2. Here, timestamps in the ParkingZones table now represent the period at which a zone is active.



ParkingZoneNames

parkingZoneID	name	vt_start	vt_end
123	Stortorget	2019-01-01	2019-02-01
123	Lilltorget	2020-01-01	2020-01-31
456	Ideon	2020-02-01	2020-05-12

ParkingZoneAttribute2

parkingZoneID	attribute	vt_start	vt_end
123	value1	2019-01-01	2019-02-01
123	value2	2020-01-01	2020-01-31
456	value1	2020-02-01	2020-05-12

parkingZoneID	zone_start	zone_end
123	2019-01-01	2019-02-01
123	2020-01-01	2020-01-31
456	2020-02-01	2020-05-12

ParkingZones

Figure 4.2: Event-order problem and using existing timestamps

When we want to update a ParkingZone's attribute, we must first find its position in the timeline compared to other values. For example, if the table is empty, we normally insert our value and set the end date of this value to "2038-01-01". However, if records exist, we want to know if we should insert before, after, or in between two records. The time periods of existing records are used with our insertions in order to prevent overlaps or gaps in our timeline. The insertion procedure is demonstrated in figure 4.3

4.2 PostgreSQL

4.2.1 Setup

PostgreSQL offers a native installation package for OS X. There is no native support for bitemporal data, however, there is an extension available with bitemporal helper functions [115]. These are cloned and loaded into our database as stored functions. As bitemporal foreign keys have not been fully implemented (they are included in the repo but not included in the loading script), we choose to leave these out for now.

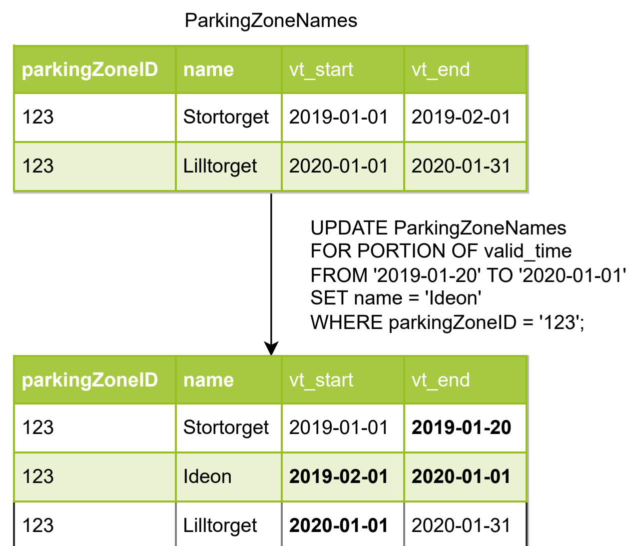


Figure 4.3: Example on how to handle event-order problem

4.2.2 Modeling & Implementation

PostgreSQL is an object-relational DBMS, which essentially means that the modeling of bitemporal tables will be the same as with MariaDB. What differs slightly is the queries that will be sent into the database since we will be using stored imported functions. Additionally, some functionality raised in section 2.10 will differ, which will be discussed in more detail later.

One of these is "table inheritance", which could technically be applied in our case. For example, we could let a parent table be called "ParkingZoneAttribute" and create a child table for each attribute that would then inherit the fields in the parent table. However, we may skip this in our case as the only thing really inherited are the bitemporal timestamps. Furthermore, since we are using stored functions to implement bitemporality, it might be best not to use inheritance. Listing 4.5 shows code for creating a bitemporal table in PostgreSQL, if done "manually" without the external functions.

```
CREATE TABLE parkings(
parking_key serial NOT NULL, -- key used by system
parking_id integer NOT NULL, -- actual, business key
parking_zone_id integer NOT NULL, -- would-be foreign key
field1 text NOT NULL,
effective_range TSRANGE,
asserted_range TSRANGE,
row_created_at timestamp with time zone NOT NULL
DEFAULT now(),
CONSTRAINT parking_id_asserted_effective_excl
EXCLUDE USING gist
(parking_id WITH =, asserted_range WITH &&, effective_range
WITH &&));
```

Listing 4.5: Implementing our model

The most important functions included are bitemporal inserts, updates, deletions, and so-called inactivations where the valid end time is set or closed for a specific row. Listing 4.6 shows example code of how we create bitemporal tables and perform insertions. Updates and deletions are done in a similar manner.

```
SELECT bitemporal_internal.ll_create_bitemporal_table(
        'public', -- schema
        'parkings', -- table name
        'parking_id integer, parking_zone_id integer, field1 text',
        'parking_id'
);

SELECT bitemporal_internal.ll_bitemporal_insert(
        'parkings', -- table name
        $$parking_id, parking_zone_id, field1$$, -- list of columns
        $$1682, 9, value1$$, -- list of values
        temporal_relationships.timeperiod(('2020-01-01'),
        'infinity'), -- valid time
        temporal_relationships.timeperiod(now(),
        'infinity') -- transaction time
);
```

Listing 4.6: Using the externally loaded bitemporal functions

4.2.3 Ingestion

We use the given functions as demonstrated in listing 4.6 but also set conditions depending on if the ID already exists. When it comes to parkings, it is safe for us to immediately attempt an insert with ParkingStarted and ParkingEnded events because of the valid time constraints set during table creation. If the constraints trigger, we can simply run the corresponding UPDATE: set valid time start if we received a ParkingStarted event, otherwise set valid time end date.

4.2.4 Retroactivity

The same procedure we applied with MariaDB can be used here. Upon attempted insertions to parkingZoneAttributes, we first compare the valid time of our row compared to other rows of the same ID. Then, we proceed to insert using either UPDATE if rows already exist, otherwise a normal insert. The external bitemporal update function works similarly to MariaDB's "UPDATE FOR PORTION OF" where we can provide valid start and end timestamps to the function. The function then uses the timestamps to "connect" adjacent rows on the valid timeline when inserting between two rows. This saves us the manual queries of "detaching" rows from neighbors, demonstrated in figure 3.2.

4.3 XTDB

4.3.1 Setup

XTDB can be run through a pre-built .jar, used with Java through their given library, or as a standalone in-memory instance with Clojure. For experimentation and modeling, we used their in-memory instance of XTDB. Clojure is required, which we installed through Leiningen [63] and its simple script. When finished, a new Clojure project is easily created with the "lein" command, as seen in listing 4.7. Then, XTDB needs to be added as a project dependency.

```
In a terminal:
lein new app xtdb-test

In project root folder, project.clj, add to :dependencies array:
[com.xtdb/xtdb-core "1.21.0"]

In core.clj, we use:
(ns xtdb-test.core
  (:require [clojure.string :as str])
  (:require [clojure.java.io :as io])
  (:require [xtdb.api :as xt])
  (:gen-class))
```

Listing 4.7: Running an in-memory XTDB node with lein

4.3.2 Modeling & Implementation

XTDB is a NoSQL, document-store DBMS. Since bitemporality is supported natively, we know that the transaction time is automatically logged, and timestamps for valid time are supplied whenever we insert a document. From a higher level, we may assume that the model used is similar to those mentioned in [10] where each document is timestamped on both axes. Each document, or entity, is free from any schematic constraints, which shifts the responsibility to the user to keep track of attributes when inserting data. As far as we know, the recommended constraint discussed in section 2.4 has *not* been applied. In other words, the user is responsible for enforcing these constraints should one wish to use nested document structures.

Since bitemporal support is native, we may instead discuss the structure of our documents. In our case, we need to distinguish between parking zones and parkings, as well query between them separately when needed. For starters, we let each Parking and ParkingZone be its own document. Since each parking document will have the ParkingZone it belongs to as an attribute, it can be queried easily and allows us to retrieve all parkings that belong to a certain ParkingZone, as seen in figure 4.4. Insertions are done with "put", as shown in listing 4.8.

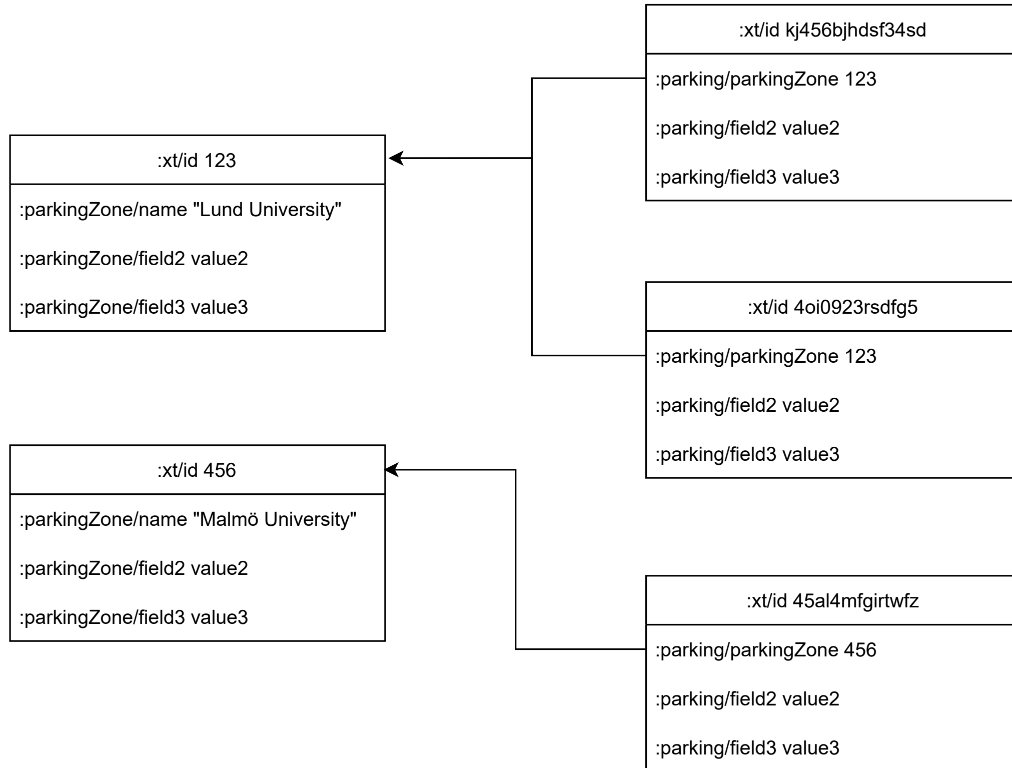


Figure 4.4: "Schema" for XTDB. Note that bitemporal timestamps are hidden

```
(xt/submit-tx
 node
 [[::xt/put
   {:xt/id 123
    :parkingZone/name "Lund University"
    :parkingZone/field2 value2
    :parkingZone/field3 value3
    }
    #inst "2000-01-01"
]])

(xt/submit-tx
 node
 [[::xt/put
   {:xt/id kj456bjhdsf34sd
    :parking/parkingZone 123
    :parking/field2 value2
    :parking/field3 value3
    }
    #inst "2000-01-01"
]])
```

Listing 4.8: Insertions of parking and parkingzone

4.3.3 Ingestion

As with the previous tools, we ingest our data through the XTDB Java API contining transaction functions such as put, delete and match. First, we create an empty map and add our fields. Then, we create the document and assign its ID, as shown in listing 4.9. The document is then submitted as a put transaction with our given valid time start. If no time is given, the current time will be used.

```
Map<String, Object> documentData = new HashMap<String, Object>();
XtdbDocument xtdbDoc;

documentData.put("parkingZone/name", "Lund University");
documentData.put("parkingZone/field2", value2);
documentData.put("parkingZone/field3", value3);
xtdbDoc = XtdbDocument.create('123', documentData);

node.submitTx(buildTx(tx -> {
    tx.matchNotExists(id); //stop transaction if doc already exists
    tx.put(xtdbDoc, new Date(vt_start));
}));
```

Listing 4.9: Using the XTDB Java API

4.3.4 Retroactivity

The way XTDB treats documents on the timeline greatly benefits our use case since data may arrive unordered. When a ParkingEnded event follows up a ParkingStarted event, the start and end time of the parking is aligned as expected. Whenever a ParkingEnded is instead received first, we can immediately submit a delete event planned for the given valid end time. At this stage, any query for said document ID will not give us any results. However, the transaction logs do contain the history of this event. Whenever a ParkingStarted event is finally received, its starting time is connected to the earlier delete event, giving us the proper valid timeline for this document ID. Now, queries for this document within its valid time period will return a result. In other words, we can put a delete/end timestamp in advance for a given document ID.

However, the same cannot be said when updating the attributes of other documents, such as a ParkingZone. In other solutions, we extracted the attributes into their own tables. With XTDB, we are only handling documents. A similar solution to Neo4j can be applied where we model a graph database by letting documents represent both nodes and relationships. Timestamping these middle-hand documents, which are now relation edges, allows us to insert past events so long as we correctly change overlapping documents' timestamps.

4.4 Neo4j

4.4.1 Setup

Neo4j offers an installation package for OS X. The desktop app lets the user create, delete and clone databases. Settings and logs are also accessible here instead of finding them in the application folder. The included "Neo4j Browser" allows for visualization of the graph, also possible by accessing the server locally through localhost.

4.4.2 Modeling & Implementation

As discussed in section 2.4.2, we can model bitemporal properties in graphs by letting each entity have its own set of state nodes and timestamp the edges. While schemas are not available per se, we can work with attaching labels to nodes in order to categorize them. A node may be labeled multiple times. Implementation is done by using the labels "Parking" and "ParkingZone" and their state labels, "ParkingState" and "ParkingZoneState", as demonstrated in figure 4.5.

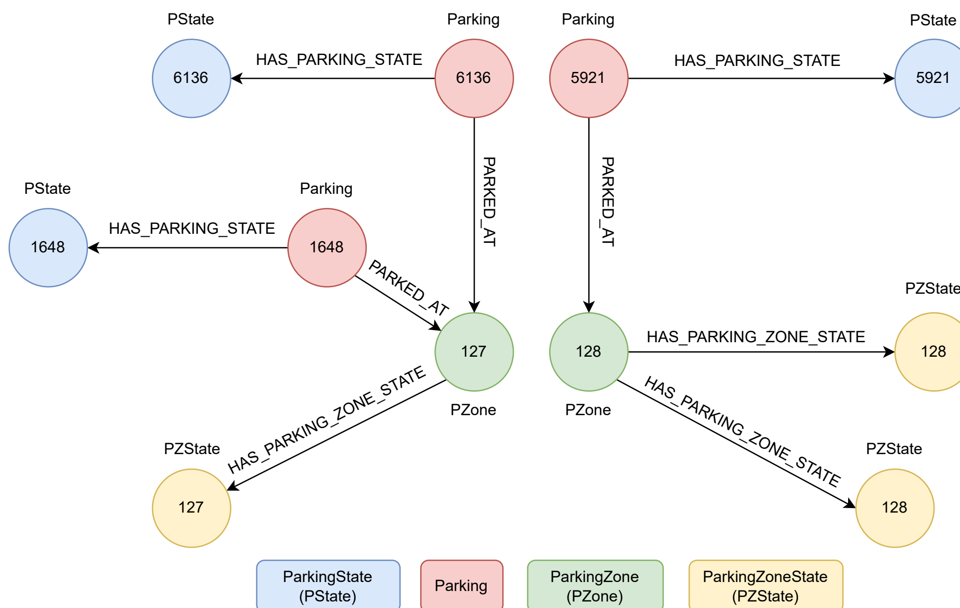


Figure 4.5: Bitemporal modeling for graph databases with relationships

4.4.3 Ingestion

As done earlier, we use the available JDBC driver for ingesting our data. Nodes are created using the CREATE query, however, MERGE can also be used. Merge searches the database for a node with the given properties (exact match) and creates the node if not found. In our case, we use this for both Parkings and ParkingZones. This is because they need to be created whenever a ParkingStarted, ParkingEnded, or ParkingZoneCreated event is received. A

MERGE can also be used for relationships. ON CREATE allows us to set a property on the node or relationship if it was created with a MERGE. ON MATCH allows us to set some values if the MERGE found a node with the given properties. Ingestion is demonstrated in listing 4.10.

```
WITH datetime.transaction('Z') AS timestamp
MERGE (p:Parking{parkingId: '5921'})
MERGE (pz:ParkingZone{parkingZoneId: '127'})
MERGE (p)-[rp:PARKED_AT]->(pz)
ON CREATE SET rp.txStart = timestamp

MERGE (ps:ParkingState{parkingId: '5921', field1: 'value1'})
WITH p, ps, timestamp
MERGE (p)-[r:HAS_PARKING_STATE]->(ps)
ON CREATE SET r.txStart = timestamp, r.vtStart = date("2020-01-01")
ON MATCH SET r.txEnd = timestamp

WITH p, ps, timestamp
MATCH (p)-[r:HAS_PARKING_STATE]->(ps)
WHERE r.txEnd = timestamp
CREATE (p)-[r2:HAS_PARKING_STATE
    {
    txStart: timestamp, vtStart: date("2020-01-01"), vtEnd: r.vtEnd
    }]->(ps);
```

Listing 4.10: Cypher queries to ingest data

The second and third block of code executes differently depending on if the ParkingState already exists. In our case, we assume only two events: ParkingStarted and ParkingEnded. We also assume that either may only arrive once for each parkingID. These can come unordered. In either case, we need to MERGE a ParkingState with our parking data and then MERGE the relationship between the parking and this state.

If the merge on the relationship to the ParkingState does not match, then we proceed as planned and timestamp our relationship with the current time as txStart and set our ParkingStarted time as vtStart. However, if the state already existed, meaning that a state with the same properties already existed, we know that our corresponding ParkingEnded event arrived first. This will give us a match on the ParkingState that holds the relationship, which will be timestamped with vtEnd. Then, we logically delete the past state by setting a transaction end time, txEnd, on the old relationship. Queries wanting the currently valid data should therefore search for relationships where txEnd does not exist.

The third block will only fully execute if r.txEnd = timestamp, or in other words, if we logically deleted the previous state. We finalize the query by creating a new relationship that contains both the end time of the previous relation along with the start time in our current ParkingStarted event. This is a costly part of the query since we run a MATCH, which is required in order to create relationships.

When it comes to indexing, we again wish to create indexes on the attributes that will be used in our queries. For any given Parking, we need only query for ParkingStates and check that the time period we are seeking falls within the time period of the valid time of the ParkingState. Because we store this information on the relation edges, the indexes are created on these. Cypher for creating B-tree indexes is demonstrated in listing 4.11.

```
CREATE INDEX parking_times_index
FOR ()-[r:HAS_PARKING_STATE]->()
ON (r.vtStart, r.vtEnd, r.txStart, r.txEnd);

CREATE INDEX parked_at_index
FOR ()-[r:PARKED_AT]->()
ON (r.txStart);

CREATE LOOKUP INDEX parking_label_lookup_index
FOR (n:Parking)
ON EACH labels(n)
ON (r.vtStart, r.vtEnd, r.txStart, r.txEnd);
```

Listing 4.11: Cypher for creating indexes on valid time in Neo4j

4.4.4 Retroactivity

The model we have implemented can be extended to support data arriving in the wrong order. This can be done in a similar way to our solution with MariaDB where we look up the existing values for a certain ID. In this case, we iterate the states for an ID and adjust the vtStart and vtEnd for the states depending on when in the timeline our current value is to be inserted.

4.5 Qualitative

We have had the opportunity to get an overview of the qualitative factors during the modeling and implementation of the tools. Most are rather objective as we try our best to list all of the reasons to why bitemporal support in one tool is greater than the other, or why scalability and availability are better. Ease of use, however, is mostly subjective and depends heavily on the user's technical background. Table 4.2 shows our findings in qualitative factors.

	Temporal Support	**Configuration**	**Scalability**	**Availability**	**UX**
MariaDB	Great	Great	Great	OK	Great
PostgreSQL	Temporal type only	Indexes	Vertical	OK	Great
XTDB	Good	Great	Vertical	OK	Good
Neo4j	Temporal type only	RAM, threads	Great	OK	Great

Table 4.2: Results of qualitative comparison between selected tools

4.5.1 Bitemporal Support

As per section 3.3, we will mostly consider the temporal support in terms of keys, operations, and the type of queries available. The results of our findings are shown in table 4.3. Again, temporal joins and coalesced model are not part of the SQL:2011 standard.

Temporal support may come natively, as in MariaDB and XTDB, or through extensions, as with PostgreSQL and Neo4j. The models may be implemented without extensions by timestamping with the supported "DateTime" types, as we did with Neo4j.

	MariaDB	PostgreSQL	XTDB	Neo4j
Transaction Time	Yes	Extension	Yes	Extension
Valid Time	Yes	Extension	Yes	Extension
Period	Yes	Yes	No	No
Period Predicates	No	Extension	No	No
Temporal PK	Yes	Extension	No	Extension
Temporal FK	No	No	No	No
Temporal Joins	Yes	Yes	Upcoming	Extension
Coalesced	No	No	Yes?	No

Table 4.3: Temporal properties of our chosen four DBMSs

MariaDB and PostgreSQL, our relational and object-relational tools, have a dedicated "Period" datatype, as specified in the SQL:2011 standard. The standard also includes predicates on periods such as "overlaps", "contains" or "precedes". These are supported in the PostgreSQL bitemporal extension, but not for our other options.

MariaDB is the only DBMS that natively supports temporal primary keys. XTDB will do so in their next major release, whereas PostgreSQL and Neo4j support it through extensions. None of the options currently support temporal foreign keys. However, XTDB will include this and temporal joins in the near future.

Temporal joins function as regular joins, but with the addition of some predicate on either time axis (or both). As long as the DBMS supports a date or period type, temporal joins are possible. An exception to this is XTDB which will include temporal joins later on.

Lastly, we have the property of coalesced DBMSs. We are unsure, but we suspect XTDB is coalesced. For example, say that we insert a document at the time "2022-01-02". If we insert the same document, but with the valid time of the day before, XTDB should only change the valid time start to "2022-01-01". The insertion will be saved in the transaction log and will be queryable on the transaction time axis. However, the valid time axis for the same document should be seen from "2022-01-01" onwards rather than seeing two (identical) documents at both dates. None of the remaining options are natively coalesced.

4.5.2 Configuration

The configuration factor, in our case, has to do with the number of options available when tuning or otherwise affecting the experience and performance on a technical level.

MariaDB uses InnoDB by default but offers over ten different engines in the categories of scaling, partitioning, archiving, search optimization and more [68]. B-Tree, R-Tree, Hash and full-text indexes are supported [72], depending on the engine used.

PostgreSQL offers only one type of storage as of now. A second one, zheap, is under development [28], althought it has not been updated in over four years. A more recent fork of zheap has been merged into PostgreSQL 14.1 [23]. The support for indexes is the most extensive one out of all of our tools: B-tree, Hash, GiST, SP-GiST, GIN and BRIN. Additionally, the GiST-index can implement other indexes, as discussed in section 2.6.3.

XTDB offers an impressive list of options for all of the three modular stores: transaction (7), document (10), and index (4) [120]. Some options may act as several stores. Therefore, there is a high chance that the user's existing system may integrate well with the transition

to XTDB. Each tool may then be further tuned to preference; for example, RocksDB directly accepts extra options passed into it.

Neo4j offers some configuration when it comes to tuning performance. It is possible to manage the number of threads in the thread pool, as well as manage the RAM that Neo4j uses by changing the heap size, page cache, and the transaction memory that holds uncommitted data. This was required in our case since Neo4j started to run out of memory during the ingestion of larger sets of data. The enterprise edition comes with additional options, such as being able to warm up the page cache faster [81]. B-tree, full-text, text indexes, and token lookups are supported, although B-tree will be replaced by range and point indexes in the future [82].

4.5.3 Scalability

When determining scalability, we consider the tools' possibilities to scale vertically (upgrade server) and horizontally (add servers). The actual performance of scaling must be measured for each scaling option and is unfortunately out of the scope of this thesis.

Vertical Scaling

In most cases, vertical scaling is limited by the underlying engine or storage solution used by the DBMS. In some cases, the server itself cannot scale further before the limitations of the DBMS, such as being unable to add more memory physically.

To start with, MariaDB uses InnoDB with an upper limit of 1017 columns per table, a maximum of 512 GB log file size, and a maximum of 256 TB tablespace size. Another option that circumvents these limitations is ColumnStore, based on InfiniDB, which handles petabytes of data and linear scalability [69].

PostgreSQL has an upper limit of at most 1600 columns per table. However, this number is reduced when columns are of types that use more space. There is no limit on the database size, implying that there is none for the tablespace size.

XTDB has no recommended or default configuration, and the documentation does not mention any limitations on XTDB itself. Any eventual limitations are linked to the configuration modules and the combination chosen by the user.

Neo4j has limitations on the number of nodes and relationships, 34 billion each, whereas properties are limited to 68 billion. The enterprise edition of Neo4j does not have these limitations [83].

Horizontal Scaling

MariaDB supports multiple options for partitioning data across multiple servers. These are Galera (Linux only), Spider, and ColumnStore. Neo4j also offers sharding, however, this is restricted to their enterprise edition.

PostgreSQL offers efficient replication of the main database for read purposes, however, does not support sharding as of now. The same applies to XTDB.

4.5.4 Availability

We tested availability by running ingestion on additional data and then executing the queries shown in section 4.6. All of the tools responded to the queries without fail. Most queries also had the same latency as when the queries were executed without parallel ingestion.

4.5.5 Ease of Use

MariaDB was easy to install and run. We set up tables and perform queries through the terminal with SQL. By default, no data visualizer is available, although third-party tools may fulfill this functionality. Documentation is excellent, and issues are tracked openly.

PostgreSQL is similar since the installation is easy and SQL is mostly used. PostgreSQL itself is also run through the terminal; however many use the tool pgAdmin to get an overview of schemas, tables, and functions while also being able to run queries.

Neo4j is installed through their given package, which also comes with an extensive user interface where settings and databases can be managed. An included tool, Neo4j Browser, connects to the DBMS and visualizes queries by building up the returned nodes and their relationships, as shown in figure 2.8.

Installation of XTDB is a bit less straightforward since we first needed Clojure, which we retrieved by installing Leiningen. Then, a Clojure project was created in which we added XTDB as a dependency. Although the documentation is rich with examples, it could take users some time to get used to Datalog and Clojure unless they have previous experience. Transaction functions, in particular, only support Clojure. This is most likely the largest hurdle, as these are required when the user wants to perform more complicated transactions. An extension to browse the database rather than using the terminal exists [109].

4.6 Quantitative

Our results are shown in the graphs below where each data point is the average of 10 iterations for each query. The raw data is unfortunately too large to fit in the appendix section, however may be sent out to any reader interested.

4.6.1 Ingestion

As described in earlier sections, ingestion is done through Parkster's Java application and the related API/JDBC for each tool. Due to time constraints, we could not have the same amount of ingestions for each tool, as some would take more than a day to complete. The results are shown in table 4.4.

	Ingestions	Average no. of Parkings/second
MariaDB	5	967
PostgreSQL	3	197
XTDB (RocksDB)	8	660
Neo4j	3	430

Table 4.4: Ingestion rate of our four chosen tools

4.6.2 Point-to-Point

Our first set of results is demonstrated in figure 4.6. These queries search for when the transaction time is set at the initialization of the database, meaning that no rows are returned. Valid time is set to cycle through different points throughout the two years with 3-month intervals. Overall, we see that PostgreSQL performs the best, Neo4j the next best, and MariaDB the worst from the 4th valid time point onwards.

The following graphs show queries where we shift the transaction time to 33%, 66%, and 100% of the ingestion time. These points are different for each tool as they vary in ingestion time. The first, second, and third graphs show similar results. However, the last graph shows that Neo4j performs worse when transaction time is set to the timestamp of the last insertion.

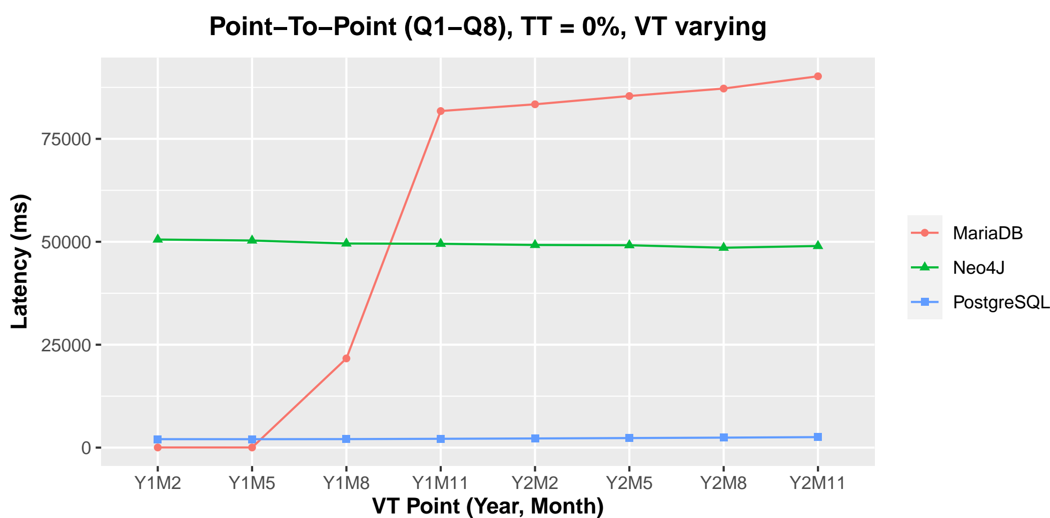


Figure 4.6: Average latency of P2P queries where transaction time is set to time of database initialization

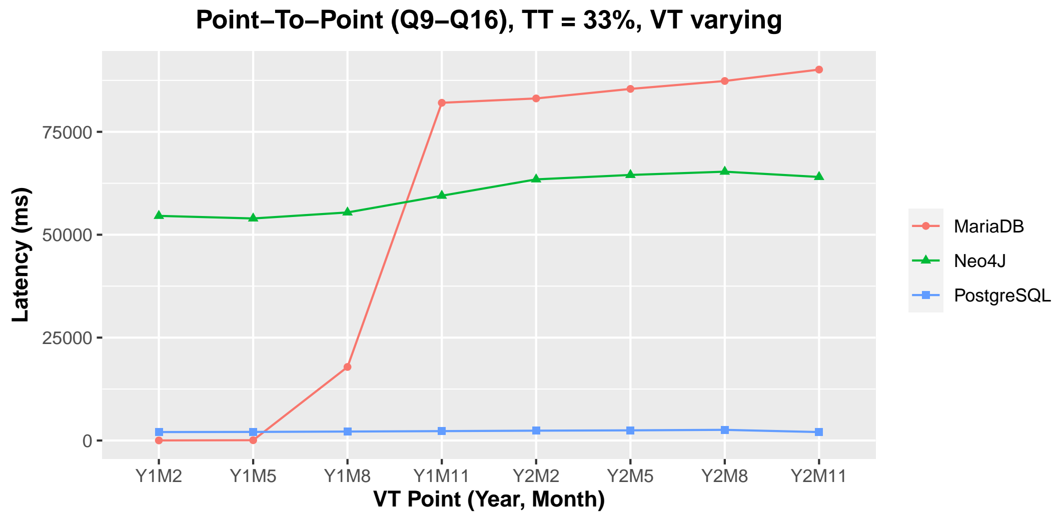


Figure 4.7: Average latency of P2P queries where transaction time is set to 33% of ingestion time

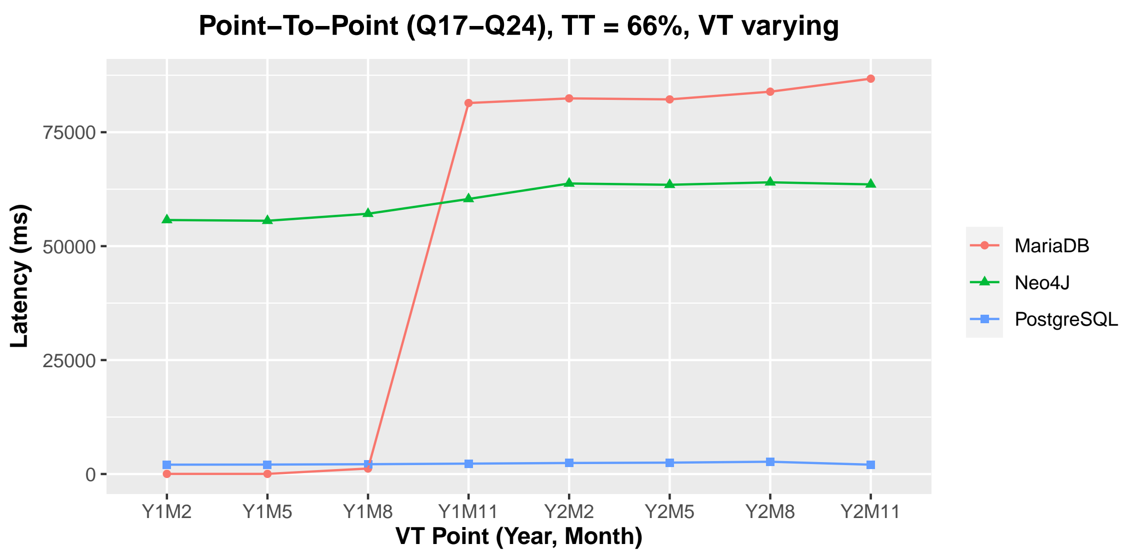


Figure 4.8: Average latency of P2P queries where transaction time is set to 66% of ingestion time

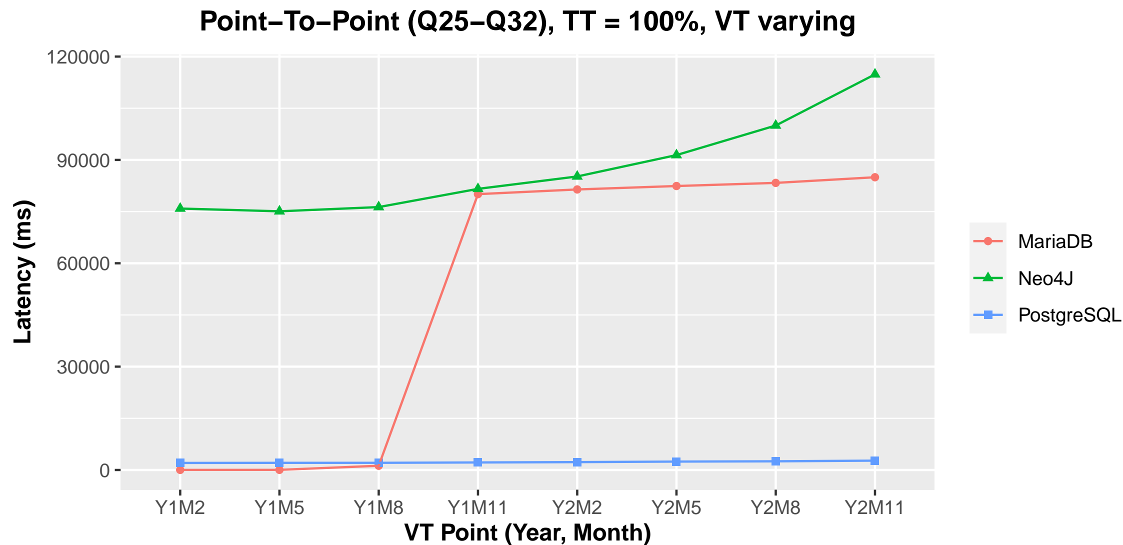


Figure 4.9: Average latency of P2P queries where transaction time is set to end of ingestion

4.6.3 Slicing

Horizontal slicing searches the full range of the transaction time axis, meaning that for any ID, we also fetch its historic rows that have been logically deleted. Valid time points are the same as in the previous graphs. The results are shown in figure 4.10, where we see similar results in earlier graphs: PostgreSQL performs the best overall, Neo4j the second best, and MariaDB the worst from the fourth point onwards. Compared to the P2P queries, we see a considerable increase in latency the further we go in the valid time axis. MariaDB shows a discrepancy from the trend where latency decreases on the fifth and sixth points.

Vertical slicing queries the transaction time at different points for each tool, and the full range of the valid time axis. The results are shown in figure 4.11 and show different results; PostgreSQL still performs the best, MariaDB the next best, and Neo4j the worst. Neo4j also shows a sharp increase in latency as the transaction time moves to the latest point.

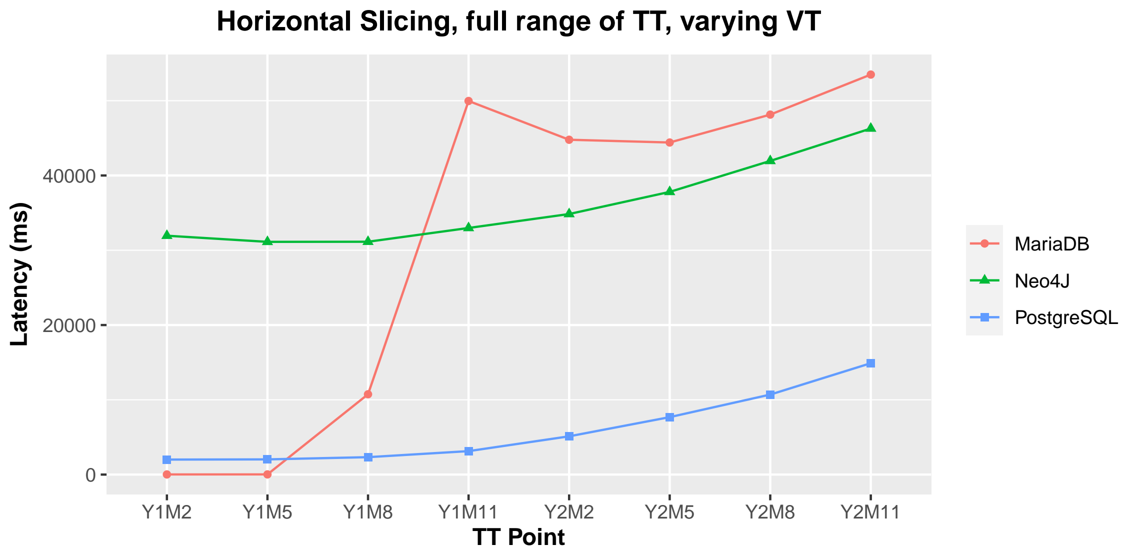


Figure 4.10: Average latency of horizontal slice queries

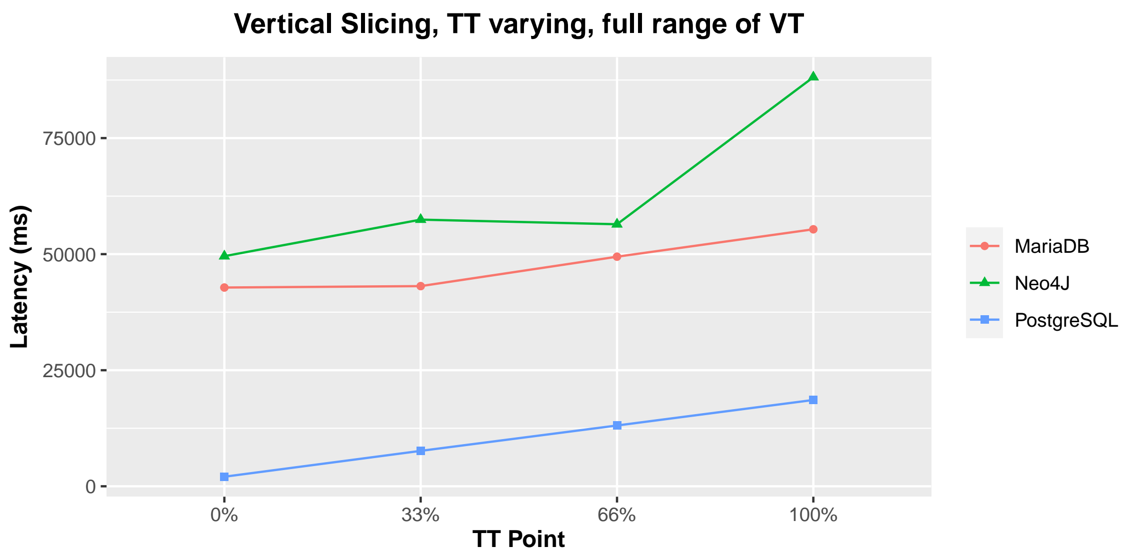


Figure 4.11: Average latency of vertical slice queries

4.6.4 Pure-key audit

These tests have been executed, however will not be listed. This will be discussed in section 5.2.

4.6.5 Bitemporal Queries

Our bitemporal queries search a range on both time axes, where the ranges are built from our previously used points. The results show that PostgreSQL once again performs the best.

Neo4j shows a similar trend for all graphs where the latency decreases as the valid time increases, although we note that the latency overall is greater in the first graph compared to the other two. MariaDB shows no clear trend across the graphs.

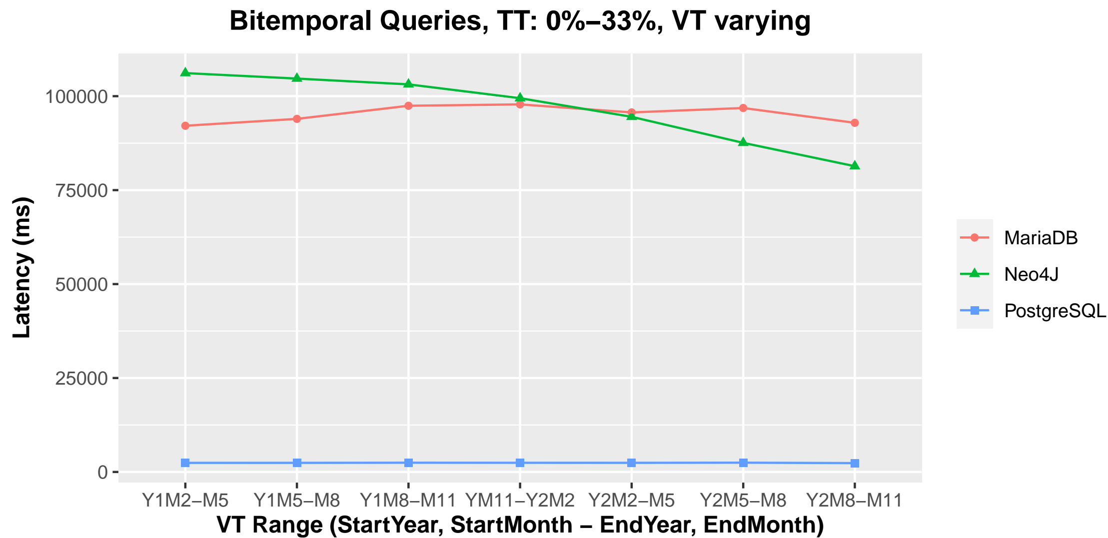


Figure 4.12: Average latency of P2P queries where transaction time is set to 0-33% of ingestion time

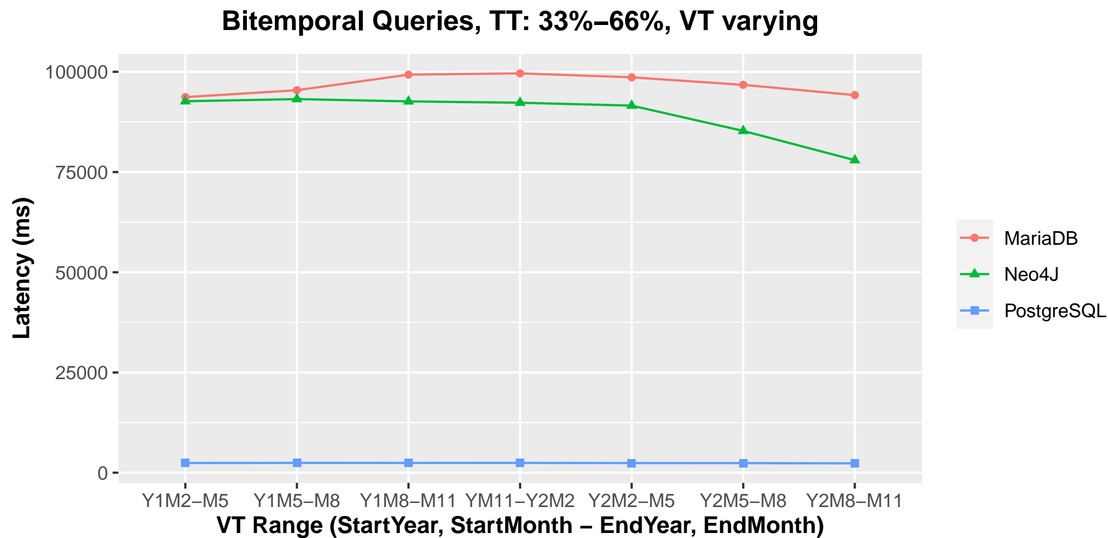


Figure 4.13: Average latency of P2P queries where transaction time is set to 33-66% of ingestion time

4.6.6 Summary

This section summarizes the practical work of this thesis. These have been divided into database tools, qualitative evaluation, and quantitative benchmarks. The first category sum-

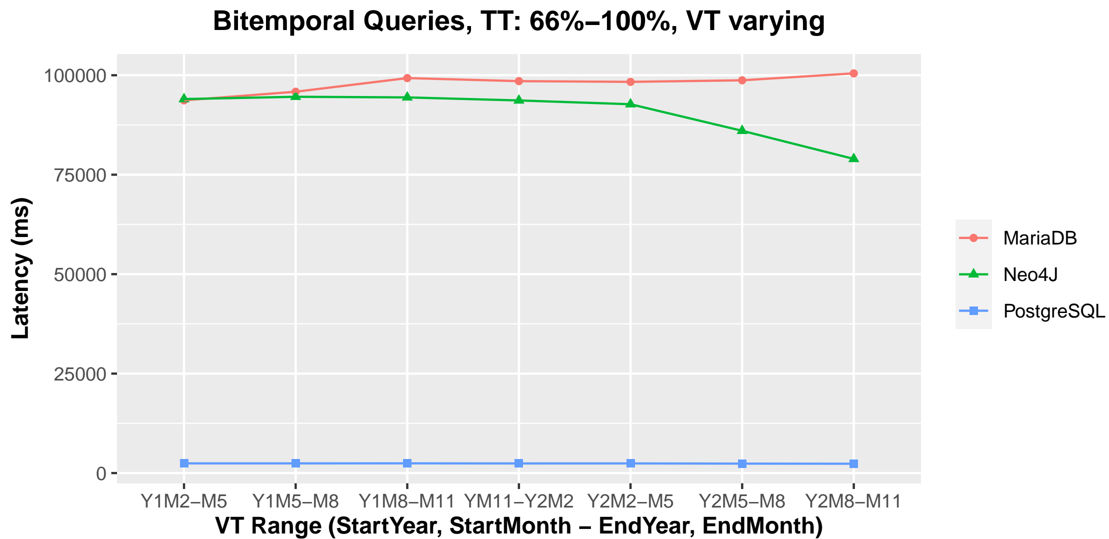


Figure 4.14: Average latency of P2P queries where transaction time is set to 66-100% of ingestion time

marizes the setup and implementation when ingesting our parking data into each tool. The second and third category summarizes our evaluation and benchmarks, respectively.

Database tools

This category consisted of the setup, modeling, and implementation for each tool. Using the proposed implementations in section 2.4 was relatively straightforward, with Neo4j being the only tool that needed manual code and modeling in order to achieve bitemporal properties. All tools supported JDBC, which made ingestion an easy feat. As explained earlier, retroactivity must be implemented manually for each tool. These general results are seen in table 4.5.

	MariaDB	**PostgreSQL**	**XTDB**	**Neo4j**
Installation	Homebrew	Installer	Leiningen	Installer
Model	Tuple, period	Tuple, period	Document, period	Graph, period
Implementation	Native	Extension	Native	Manual
Driver	JDBC	JDBC	JDBC	JDBC
Ingestion	Great	Great	Great	Great
Retroactivity	Manual	Manual	Manual	Manual

Table 4.5: Results of the different steps in implementing bitemporal properties and ingestion

Qualitative Evaluation

An overview of the qualitative results can be found in table 4.2 and table 4.3. In short, MariaDB had the best temporal support and also offered a significant amount of configuration options. Some of these included the ability to scale both horizontally and vertically. The

other tools had varying results, lacking either temporal support or configuration. The results on availability did not fully represent real-life use as we did not test the tools with heavy load.

Quantitative Benchmarks

The quantitative results differed considerably and yielded interesting patterns. MariaDB saw a repeating pattern for all Point-to-point and horizontal slicing queries: the first two or three queries had near instant query times, whereas the remaining queries showed a steep increase with latency that were considerably higher than those of Neo4j. Neo4j demonstrated nearly the same pattern for all queries, with latency increasing slightly as the valid time points increased. The exception to this and MariaDB's pattern occurred in vertical slicing, where Neo4j had the highest latency, and in the bitemporal queries, where Neo4j had decreasing latency as valid time increased. MariaDB had nearly the same latency for all queries, as opposed to low latency for the first few.

As evident in the graphs, PostgreSQL surpassed both Neo4j and MariaDB in query execution times. This was true for almost all queries, with the exception being the slicing queries where latency increased considerably as valid and transaction time increased. Overall, MariaDB was the slowest, Neo4j in the middle, and PostgreSQL the fastest. Why PostgreSQL demonstrated such patterns will be discussed in chapter 5.

Chapter 5

Discussion

In this chapter, we discuss our experience using the tools and our findings from chapter 4. For each category, we reason about the results in order to answer our research questions. Explanations of results will be given so long as sufficient evidence exists. If none can be found, relevant theories are suggested. Suggestions on how to improve the results for future reference are also given whenever possible.

5.1 Qualitative Results

While much of **RQ 1** has been investigated in chapter 2, our qualitative results contribute to this discussion by considering our selected tools and implementations in more detail. **RQ 2** can also be mentioned, although most of it stems from the bitemporal support discussion.

5.1.1 Bitemporal Support

Our evaluation showed that temporal support varied across the tools, even when native support for bitemporality was present. We mostly used the main temporal features introduced in the SQL:2011 standard as a basis during our evaluation. Our findings show that none of our chosen tools fully support the standard. XTDB may do so in the short future, whereas PostgreSQL and Neo4j currently need extensions or manual implementation IBM DB2 claims to support temporal foreign keys. However it did not seem to work as of late 2019 [53]. Additionally, we listed that most of the tools supported temporal joins. However, this is only true from a technical standpoint since it can be done whenever a temporal data type is supported. Native support in terms of adapted performance, which speeds up joins and queries, is not supported.

When applying this discussion to Parkster's parking map, we remind ourselves that the requirements involve bitemporal support in the sense that both valid time and transaction time are tracked. It should also be possible to extract the parking data, across a point or

period in time, and relate it to what parking zone it belongs to in order to display the correct information on the parking map. As of now, MariaDB supports this natively, PostgreSQL and Neo4j support it through extensions, and XTDB does not due to the lack of ranged query support.

5.1.2 Configuration

All tools in this thesis offer useful configuration options, and many happen to be unique among the tools. MariaDB and XTDB offer many engines that fit different needs, XTDB more so. PostgreSQL offers many indexes with some, such as GiST, being unique and not available in other tools. Neo4j offers some useful configurations, although some, perhaps minor ones, are locked behind their commercial Enterprise Edition.

5.1.3 Scalability

Our findings of scalability, which were mostly based on configuration options, showed that options for scaling varied. The choice of scaling method heavily depends on the scenario. Parkster has an increasing rate of data, which will most likely persist for a long time due to continued expansion. In this and similar cases, we recommend MariaDB or Neo4j, which support both bitemporality and horizontal scaling.

5.1.4 Availability

All tools passed our simple test of running queries while also ingesting data. Additionally, querying times seemed relatively unaffected. However, ingestion was performed locally rather than ingesting from other sources in parallel. Different results would likely be produced if concurrent queries and ingestion were executed from multiple sources. In order to get a more accurate result of availability, a heavier load would need to be introduced.

5.1.5 Ease of use

Installation was mostly easy for our chosen tools, where three out of four offered installation packages for (at least) OS X. It may take some time to fully use XTDB's potential due to transaction functions having to be written in Clojure. This was the case for us, although time constraints prevented us from experimenting with these and implementing retroactivity for our use case.

5.1.6 Takeaways

The temporal models from section 2.3.5 contribute to answering RQ 1 as most of the models and their respective implementations represent the state-of-the-art, despite the fact that many of these originated decades ago.

In our qualitative evaluation, we realized that none of the chosen tools fully supports the SQL:2011 standard. However, they have proven to be sufficient for the requirements of parking map. This contributes to answering both RQ 1 & 2. The results demonstrate the

differences in functionality when certain models are used over others. We note, however, that this comparison covers only our chosen tools in detail, whereas other tools have been given an overview in section 2.10 of their properties.

Additionally, the distinction between the temporal models and database models must be mentioned in the context of RQ 2. Although many temporal models exist as per section 2.3.5, most have not been implemented. Instead, the temporal "model" demonstrated in the SQL:2011 standard has been brought forward. In order to fully answer RQ 2, we would have to manually implement some of the remaining temporal models onto a variety of database models before evaluating the qualitative differences. The same would apply to the quantitative part of RQ 2, with the addition that the models be implemented in DBMSs as well.

The results of the remaining qualitative factors contribute partially to RQ 2. We realize that configuration options and scalability are highly prioritized factors for parking map, which must be considered alongside bitemporal support.

Finally, we may also discuss RQ 3. After having implemented bitemporality on our selected tools, we realize that the remaining models brought up in section 2.3.5 may be implemented as well, although their effectiveness may vary depending on the use case. The ability to define custom data types helps in this regard, something which is possible with PostgreSQL. We note that the ability to implement a temporal model onto a database model is heavily influenced by the tool in question. For example, most temporal models were based on the relational database and its set theory that relied on constraints to ensure data integrity. These are not supported in a bitemporal manner for most tools today.

5.2 Quantitative Results

The quantitative results partially answer RQ 2 since we compare the technical performance of the implementations of different models on bitemporal DBMS. These results generally contribute to answering RQ 3 by discussing the reasons behind the differences in benchmarks. Additional benefits and drawbacks for each model were discussed in our qualitative results section.

5.2.1 Ingestion

Our results showed that the rate of ingestion varied substantially. However, the data is more meant to show some sample numbers for our use case. The first reason behind the variance is not having the amount of trials for each tool be equal, which was due to time constraints. The second reason is that of our scenario; modeling and implementations. As seen in section 4.1.2, the models and implementations used differed. For example, our bitemporal implementation in Neo4j used multiple node labels. This means that for each parking, we need to consider matching or creating a Parking, a ParkingState and a relationship between these. Thus, an insertion of a single parking is initially represented by the creation of three entities. On the other hand, an insertion of a parking in MariaDB or PostgreSQL is done by appending a single row and logically deleting the now truncated row. In other words, the average rate of ingestion shows the performance given our implementation of how Parkings are to be represented in each DBMS.

The last factor revolves around the technical details and tuning, some of which relies on indexes. The authors of the Postgres bitemporal extension has "more efficient indexing" on their roadmap which hints at better performance in the future. This is in contrast to the remaining DBMSs, even those outside of this thesis, where B-tree indexes have long been the main option. Additionally, performance could be explained by detailing the inner workings of each DBMS and how they store their tables on a physical level. However, we have not been able to provide a proper assessment of ingestion rate and discussion around this next layer of performance factors due to time constraints.

5.2.2 Query Results

Point-to-point

In general, our benchmark showed that MariaDB performed the worst, Neo4j slightly better, and PostgreSQL the best. However, the graphs show a pattern that invokes some suspicion. Apparent from the first graphs, we see how MariaDB has near instant query time for the first two or three queries in the Point-to-point category. In this case, we believe the reason is related to the previously discussed indexes and query planning. We can use MariaDB's EXPLAIN function to investigate our suspicions, seen in listing 5.1. With valid time set to Y1M2 and Y1M5, the EXPLAIN function shows that the table will be accessed by the "range" type, which uses a key over one or several ranges [70]. In our case, this is the valid time period. The same applies to the next point, Y1M8, although there is a sharp increase in the number of rows here that explains heightened latency. The remaining points show that MariaDB uses the type "all", a full table scan.

As mentioned in section 2.7, the optimizer may skip using index access if the cost exceeds a full table scan. These last queries show a further steady increase in latency, all of which are accessed through full table scans. We believe this increase is due to the increasing amount of data or rows for the second year of this data set. The same pattern of instant access, which then transitions to long, full table scans, applies to all of MariaDB's Point-to-point graphs.

```
-------------+----------+-------+---------------+------------+
 select_type | table    | type  | possible_keys | key
-------------+----------+-------+---------------+------------+
 SIMPLE      | Parkings | range | vt_indexes    | vt_indexes
-------------+----------+-------+---------------+------------+

---------+------+------------------------------------+
 key_len  | ref  | Extra
---------+------+------------------------------------+
 6        | NULL | Using index condition; Using where
---------+------+------------------------------------+
```

Listing 5.1: MariaDB explaining the table access strategy for our first query

PostgreSQL shows an apparent and unfortunate trend in almost all of our graphs, with query times ranging between 2-3 seconds. It is evident that the results of the queries have been cached and are being returned, especially since the first run (which has been discarded) returns query times more on par with the other DBMSs. The EXPLAIN function shows us

that a full table scan is used for all queries. Unlike with MariaDB, this caching feature cannot be turned off; instead, a restart of the server is required [92]. All of our queries were executed directly on each server through the command-line. In an attempt to produce better results for PostgreSQL, we tried their included benchmark tool, "pgbench". Unfortunately, we saw the same pattern where consecutive runs after the initial few runs yielded query times of around 2-3 seconds.

The results for Neo4j show slightly different patterns for the point-to-point queries. This is interesting since the first graph shows nearly equal query times regardless of the given valid time points, hinting that nodes are fetched through a Parking label scan as well. This is confirmed when using Neo4j's "ANALYZE" function, which shows us that "NodeByLabelScan" is used to find labels before these are filtered on our query parameters, i.e. valid or transaction time. However, the second and third graph shows a considerable increase in latency overall. Moreover, the latency varies for all points, increasing slightly from the middle point onwards. ANALAYZE on these graphs returns the same query planning. Lastly, the final graph shows a sharper increase in latency across the board to such an extent that it surpasses MariaDB. We are not entirely sure how to explain this increase, but we suspect that the number of edges to potentially check and filter has impacted the query times significantly. Monitoring the resource usage of the computer does not show anything surprising with high I/O activity and rather low CPU usage.

Horizontal Slicing

The slicing queries show us interesting data, not only because the patterns differ but also because PostgreSQL does not exclusively show a flat line. As a small reminder, horizontal slicing queries the full range of the transaction time axis, meaning that historical rows for any ID will be included. Neo4j displays an arc similar to PostgreSQL, which applies to their query planning as well; for both types of slicing queries, full table scans are used.

MariaDB shows a similar pattern to previous graphs where the table access is first done by index but later switches to a full table scan. This is confirmed in EXPLAIN. Additionally, the latency lowered from Y1M11 to Y2M2, where it then gradually increased. This drop in latency is not explained through table access, as MariaDB still uses full table scans for these points. The amount of data stored in these later points is higher than those of earlier dates, which contradicts our thoughts regarding the general latency increase with the last queries.

PostgreSQL sees an increasing trend in latency, one of which we cannot explain since it is still performing full table scans. One of our theories is that the amount of data to be cached has grown to such an extent that instant access is no longer possible.

Vertical Slicing

With vertical slicing, we vary the transaction time points as a percentage of the total ingestion time for each DBMS. Here, we see in a rare case that Neo4j performs the worst for all points, although we cannot explain this result. Neo4j, PostgreSQL, and MariaDB again use full table scans, with PostgreSQL showing some difficulty fully utilizing its cache.

Pure-key audit

This category of queries searches for a specific entity through its ID, or a few entities, through different points in time. Because we applied indexes on all DBMSs, query times were almost instantaneous. It would not contribute much to showcase query times in the millisecond range; we have therefore opted not to include these benchmarks.

Bitemporal

For last, we discuss our bitemporal queries. Here, we provide period ranges on both time axes. Unfortunately, PostgreSQL again uses its cache to quickly return the query results, which is in contrast to the slicing queries, where it showed some increase in latency. MariaDB shows almost the same pattern, with the last graph increasing latency slightly towards the last points. Again, full table scans are used. Neo4j has an almost inverse pattern where the latency decreases as the valid time points increase. This also applies when the transaction time increases, which means that the pattern persists even though the amount of data to scan increases. As with all other graphs, Neo4j uses full table scans.

5.2.3 Takeaways

Regarding RQ 2 and parking map, it becomes clear that in quantitative terms, the DBMS that returns the lowest latency for our given queries is the most desirable. As the graphs have shown us, we believe this to be influenced by the table access method. However, this is related to the amount of data that the query is seeking. Unfortunately, the results did not satisfy Parkster and their requirements. Therefore, we conclude that Parkster will need to investigate other options, such as implementing different temporal models, or experimenting with a tool and tuning it to shorten query times.

When discussing the *models* used in terms of performance, the discussion becomes more difficult. What affects performance when using different database models? An initial thought we had was that the number of rows or entities used to store bitemporal data affected performance. For example, we mentioned earlier that Neo4j would require the creation of a Parking, a ParkingState, and the relationship between these that contains the temporal data. Furthermore, using ParkingZones would require an additional edge. This results in many nodes that would need to be checked and filtered out during querying. However, the same applies to, say, MariaDB. Multiple rows will be inserted for changes in states for any specific ID, corresponding to additional nodes in Neo4j. Furthermore, many of the graphs showed that Neo4j performed better. Here, the difference in performance could *still* be related to the model used when considering, for example, the search algorithm used in the graph model compared to the relational model. The same logic can be applied to XTDB, which searches "flat" documents without schemas or edges between them, although we could not perform these benchmarks.

Finally, although not applicable in our benchmarks, the model used could affect performance when wanting to query for related data. This would be done through temporal joins. For example, say we hypothesize that the query times mostly rely on the fetching and filtering of data according to our temporal predicates. Then, we can also theorize that the performance of temporal joins will rely on the querying algorithm for a specific database model

implemented in a specific DBMS. This is a takeaway that answers RQ 3. This discussion borders on a more technical discussion where further details and work which is, unfortunately, is out of the scope for this thesis.

5.3 Validity Threats

In short, threats to validity are the factors that impact the credibility of our results. These can be divided into threats to internal validity and external validity. The former considers the extent to which the independent variables have been affected by confounding factors, whereas the latter considers the extent to which we can generalize our findings to industrial practice [119].

5.3.1 Internal Validity

The first and most apparent evidence of internal validity threats was demonstrated with our PostgreSQL results. The presence of a query cache makes us consider configuration, a threat that may have been eliminated had we executed the queries in a different order.

The second related factor is the choice and use of indexes in the DBMSs. Our initial thought was to assign identical indexes on the same attributes, that is, B-tree indexes on both time axes. However, GiST was used in PostgreSQL. In the end, as discussed in section 5.2, the indexes were mostly not used due to our queries covering too large parts of the data set.

The last and perhaps most important factor to consider is that our independent variables are not properly "independent". An example scenario of proper methodology would be to test our chosen tools with a simple data set and then perform benchmarks on an equal ground. Modeling and implementing bitemporality means that we introduce additional variables, reducing the clarity around our experiment setup.

5.3.2 External Validity

Although the experiment in terms of variables could be improved, it could prove to be beneficial since it demonstrates a real-life application of bitemporal modeling. However, parts of our evaluation do not meet industry standards. This is apparent in our evaluation and testing of availability in our tools, where a real-life scenario would imply higher concurrent read and write requests.

Our query benchmarks are based on an excellent suite [55]. However, only a small portion of these queries has been selected. Additionally, they may suffer from the same drawback of not demonstrating industry requirements. This is also true when it comes to the hardware used during benchmarks.

5.4 Reflection

Our first thought on the thesis as a whole has to do with its scope, where we believe that our goals may have been a liability for the six-month planned time frame. Our work cul-

minated in the evaluation of qualitative and quantitative results. The qualitative evaluation was based on the comparison between different tools and the theory in chapter 2, both of which required considerable research. The quantitative part of the thesis was based on the modeling, implementation, and benchmarking of our four selected tools. While the modeling and implementation were straightforward, especially since two of our tools supported bitemporality natively, the benchmarks were considerably more difficult. The differing tools having differing data models was the first reason. The second is the abundance of configuration options among the DBMSs and the fact that indexes have not been properly utilized, as noted in the bitemporal benchmark conducted in 2014 [55].

The lack of time has affected the level of content in both of our evaluations. In the theoretical section, details are missing regarding the mathematical foundations of the temporal models. Perhaps more importantly, rather than attempting to implement multiple temporal models, we chose the simplest form of tracking both time axes as periods. In the qualitative section, testing on temporal joins and discussions on the types of queries available have been left out. In the quantitative section, our benchmark of the cached results for PostgreSQL may have been avoided if the query execution order differed. Additionally, thorough testing on DBMSs' availability has not been conducted.

5.5 Future Work

A handful of interesting suggestions for future work in this area exist beyond those already discussed. The first one would be to add an object-oriented database type to benchmarks and evaluations, as well as the remaining NoSQL types. Gemstone/S is one of the few commercially available object-oriented databases mentioned in section 2.3.2, which was originally planned to be included in this thesis.

Decision time is another form of time, purposefully left out of this thesis due to time constraints. It refers to the time at which some decision has been made, such as the time a promotion was decided. This is distinct from valid time, which in this case would reflect the promotion date [47]. Its use has been investigated in a blog post [97] along with both valid time and transaction time, effectively implementing a tri-temporal database.

A paper released in 2016 concluded that attribute timestamping with one-level nesting would be ideal for modeling bitemporality since it was both faster and used less disk space compared to tuple timestamping [6]. Although this form of timestamping was raised in this thesis, we have not demonstrated it in our implementations.

As previously mentioned, XTDB's Core2 will fully implement the SQL:2011 standard. It would be interesting to see how it fares against our tools in qualitative and quantitative evaluations.

Lastly, "big data" is defined as huge data stores. Although our data set is far from that, it would be interesting to see if those tools would help minimize query times in our implementations.

Chapter 6
Conclusion

The goal of this thesis is to investigate the state-of-the-art of bitemporal modeling, reflected in **RQ 1**, as well as the pros and cons of its implementations, reflected in **RQ 2** and **RQ 3**.

RQ 1 was answered by researching literature on bitemporal modeling and investigating its applications in tools available today. We conclude that while the SQL:2011 standard contributed to the status of temporal databases, the commercial implementations are not on par with the state-of-the-art bitemporal modeling. This thesis used the most basic features by considering valid time and transaction time as either points or periods, just as in the SQL:2011 standard. Most, if not all, of the models presented in section 2.3.5 may be implemented in a DBMS that supports custom types. In our case, this would be PostgreSQL and Neo4j.

RQ 2 has been more complicated to answer, mostly due to time constraints. In our discussion, we could draw some conclusions on the functional differences when using different *database models* and their respective tools. Additionally, our quantitative results showed some differences between the DBMSs. In the context of Parkster and their use case, we concluded that some tools were sufficient enough qualitatively but not quantitatively. We noted that our benchmarks and evaluations were done using a single temporal model. In order to fully answer this question, further implementations and evaluations using the remaining models are required. In Parkster's case, we recommend that they investigate other tools that may provide better performance than those demonstrated in our benchmarks.

At last, we may reflect on *RQ 3* regarding the takeaways of having implemented bitemporality in different ways. The first and most natural is that the temporal models, database models, and tools presented will fit differently depending on the use case. This is true even within database categories, as different tools offer different tuning solutions, which in turn fit the implementation of temporal models differently. We believe the ability to define custom data types to be beneficial to this purpose, possible with PostgreSQL. Unfortunately, it might not be possible to implement all of the temporal models in a bitemporal manner. One reason is that most temporal models are based on the relational model, implying data integrity through constraints. Most tools today do not offer bitemporal constraints.

In conclusion, it can be difficult to draw solid conclusions about our results. Our qualitative evaluation has not covered the majority of available tools, and some factors, such as temporal joins, have not been investigated thoroughly. Some uncertainty applies to our benchmarks as well, including cached results and the possibility that our code implementations have not been the most optimal. Furthermore, earlier benchmarks presented in section 2.9 cannot be immediately comparable to ours due to differing setups, and differing implementations. In some cases, the DeWitt-clause prevents the publication of benchmarks for some DBMSs. This is unfortunately true for [55], the paper on which our queries mainly were based.

Despite these shortcomings, we believe the thesis to have fulfilled its goal in terms of scientific contribution. We introduce the temporal research, the state-of-the-art, and some of the implementations currently applied in DBMSs. We also showcase and apply some of the temporal models to our case of parkings, culminating in the presentation of qualitative and quantitative results. These serve as a small example of temporal data and its use cases in the hopes that interest in bitemporal support will grow.

References

[1] Athiq Ahamed. Benchmarking Top NoSQL Databases. `http://dx.doi.org/10.13140/RG.2.1.2276.6969`, 2016.

[2] Mohammed Al-Kateb, Alain Crolotte, Ahmad Ghazal, and Linda Rose. Adding a Temporal Dimension to the TPC-H Benchmark. In *Selected Topics in Performance Evaluation and Benchmarking*, pages 51–59, Berlin, Heidelberg, 2013. Springer.

[3] Hibatullah Alzahrani. Evolution of Object-Oriented Database Systems. *Global Journal of Computer Science and Technology*, 2016.

[4] K. Anusha, Nichenametla Rajesh, M. Kavitha, and N. Ravinder. Comparative Study of MongoDB vs Cassandra in big data analytics. In *2021 5th International Conference on Computing Methodologies and Communication (ICCMC)*, pages 1831–1835, 2021.

[5] Gad Ariav. A Temporally Oriented Data Model. *ACM Trans. Database Syst.*, 11(4):499–527, dec 1986.

[6] Canan Atay. An attribute or tuple timestamping in bitemporal relational databases. *TURKISH JOURNAL OF ELECTRICAL ENGINEERING & COMPUTER SCIENCES*, 24:4305–4321, 01 2016.

[7] Sikha Bagui. Achievements and Weaknesses of Object-Oriented Databases. *Journal of Object Technology*, 2:29–41, 07 2003.

[8] R. Bayer and E. McCreight. Organization and Maintenance of Large Ordered Indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '70, page 107–141, New York, NY, USA, 1970. Association for Computing Machinery.

[9] J S Ben-Zvi. The Time Relational Model. 1982.

[10] Dmitrij Biriukov. *Implementation aspects of bitemporal databases*. PhD thesis, Vilniaus Universitetas, 2018.

[11] Michael H Böhlen, Anton Dignös, Johann Gamper, and Christian S Jensen. Database technology for processing temporal data. In *25th International Symposium on Temporal Representation and Reasoning (TIME 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[12] Eric Brewer. Towards robust distributed systems. page 7, 01 2000.

[13] Eric Brewer. CAP Twelve years later: How the "Rules" have Changed. *Computer*, 45:23–29, 2012.

[14] F.P. Brooks. *The Analytic Design of Automatic Data Processing Systems*. Harvard University, 1956.

[15] Carlo Strozzi. NoSQL: a non-SQL RDBMS. `http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/NoSQL/Home%20Page`, 2010. [Accessed: 2022-08-01].

[16] R. G. G. Cattell. The Object Database Standard: ODMG-93 (Release 1.1). 1994.

[17] Donald Chamberlin, Morton Astrahan, Mike Blasgen, Jim Gray, W. III, Bruce Lindsay, Raymond Lorie, James Mehl, Thomas Price, Gianfranco Putzolu, Patricia Selinger, Mario Schkolnick, Donald Slutz, Irving Traiger, Bradford Wade, and Robert Yost. A History and Evaluation of System R. *Commun. ACM*, 24:632–646, 1981.

[18] A Chaudhri and Peter Osmon. A Comparative Evaluation of the Major Commercial Object and Object-Relational DBMSs: GemStone, O2, Objectivity/DB, ObjectStore, VERSANT ODBMS, Illustra, Odapter and UniSQL. Technical report, Citeseer, 1996.

[19] Peter Pin-Shan Chen. The Entity-Relationship Model—toward a Unified View of Data. *ACM Trans. Database Syst.*, 1(1):9–36, 1976.

[20] James Clifford and Albert Croker. The historical relational data model (HRDM) and algebra based on lifespans. In *1987 IEEE Third International Conference on Data Engineering*, pages 528–537, 1987.

[21] E. Codd. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM*, 13:377–387, 01 1970.

[22] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[23] Cybertec. Cybertec-PostgreSQL - zheap. `https://github.com/cybertec-postgresql/postgres/tree/zheap_undo_record_set_pg_14.1`. [Accessed: 2022-08-01].

[24] Hugh Darwen and C. J. Date. The Third Manifesto. *SIGMOD Rec.*, 24(1):39–49, mar 1995.

[25] Joshua Duhl and Craig Damon. A performance comparison of object and relational databases using the sun benchmark. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, OOPSLA '88, page 153–163, New York, NY, USA, 1988. Association for Computing Machinery.

[26] Margaret H. Dunham, Ramez A. Elmasri, Mario A. Nascimento, and Marion G. Sobol. Benchmarking Temporal Databases - A Research Agenda. 1995.

[27] Andrew Eisenberg and Jim Melton. SQL: 1999, Formerly Known as SQL3. *SIGMOD Rec.*, 28(1):131–138, mar 1999.

[28] EnterpriseDB. EnterpriseDB - Zheap. `https://github.com/EnterpriseDB/zheap`. [Accessed: 2022-08-01].

[29] Mohammed Eshtay, Azzam Sleit, and Monther Aldwairi. Implementing bi-temporal properties into various NoSQL database categories. *International Journal of Computing*, 18(1):45–52, 2019.

[30] Anon et al, Dina Bitton, Mark Brown, Rick Catell, Stefano Ceri, Tim Chou, Dave DeWitt, Dieter Gawlick, Hector Garcia-Molina, Bob Good, Jim Gray, Pete Homan, Bob Jolls, Tony Lukes, Ed Lazowska, John Nauman, Mike Pong, Alfred Spector, Kent Trieber, Harald Sammer, Omri Serlin, Mike Stonebraker, Andreas Reuter, and Peter Weinberger. A Measure of Transaction Processing Power. *Datamation*, 31(7):112–118, apr 1985.

[31] etcd. Data model. `https://etcd.io/docs/v3.5/learning/data_model/`. [Accessed: 2022-08-01].

[32] Johann-Christoph Freytag. *Query Processing and Optimization in Object Relational Databases*, pages 2293–2297. Springer US, Boston, MA, 2009.

[33] Shashi K. Gadia. A Homogeneous Relational Model and Query Languages for Temporal Databases. *ACM Trans. Database Syst.*, 13(4):418–448, 1988.

[34] Dengfeng Gao. *Temporal Joins*, pages 2982–2987. Springer US, Boston, MA, 2009.

[35] Seth Gilbert and Nancy Lynch. Perspectives on the CAP Theorem. *Computer*, 45(2):30–36, 2012.

[36] Jim Gray. *The Transaction Concept: Virtues and Limitations*, page 140–150. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[37] Theo Haerder and Andreas Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Comput. Surv.*, 15(4):287–317, dec 1983.

[38] Thomas Haigh. The Chromium-Plated Tabulator: Institutionalizing an Electronic Revolution, 1954-1958. *IEEE Ann. Hist. Comput.*, 23(4):75–104, 2001.

[39] Thomas Haigh. How Data Got its Base: Information Storage Software in the 1950s and 1960s. *IEEE Annals of the History of Computing*, 31(4):6–25, 2009.

[40] Thomas Haigh. How Charles Bachman invented the DBMS, a foundation of our digital world. *Communications of the ACM*, 59:25–30, 06 2016.

[41] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized Search Trees for Database Systems. In *VLDB*, 1995.

[42] Martin Höst, Björn Regnell, and Per Runeson. *Att genomföra examensarbete*. Studentlitteratur AB, 2006.

[43] IBM. Data access through index scansl. `https://www.ibm.com/docs/en/db2/11.5?topic=methods-data-access-through-index-scans`. [Accessed: 2022-08-01].

[44] IBM. What is Data Modeling? `https://www.ibm.com/cloud/learn/data-modeling`. [Accessed: 2022-08-01].

[45] Christian Jensen, J. Clifford, Shashi Gadia, Fabio Grandi, P. Kalua, N. Kline, Angelo Montanari, S. Nair, E. Peressi, E. Robertson, John Roddick, Nandlal Sarda, M.R. Scalas, A. Segev, Richard Snodgrass, Abdullah Tansel, Paolo Tiberio, A. Tuzhilin, and G. Wuu. The TSQL benchmark. 1993.

[46] Christian Jensen, J. Clifford, Shashi Gadia, Fabio Grandi, P.P. Kalua, N. Kline, Nikos Lorentzos, Y. Mitsopoulos, Angelo Montanari, S.S. Nair, E. Peressi, E.L. Robertson, John Roddick, Nandlal Sarda, M.R. Scalas, A. Segev, Richard Snodgrass, Abdullah Tansel, and G.T.J. Wuu. *A consensus test suite of temporal database queries*. 01 1993.

[47] Christian Jensen, James Clifford, Shashi Gadia, Arie Segev, and Richard Snodgrass. A Glossary of Temporal Database Concept. *SIGMOD Record*, 21:35–43, 01 1992.

[48] Christian S. Jensen. Introduction to Temporal Database Research, 2001.

[49] Christian S. Jensen, Richard T. Snodgrass, and Michael D. Soo. *The TSQL2 Data Model*, pages 157–240. Springer US, Boston, MA, 1995.

[50] C.S. Jensen, L. Mark, and N. Roussopoulos. Incremental implementation model for relational databases with transaction time. *IEEE Transactions on Knowledge and Data Engineering*, 3(4):461–473, 1991.

[51] Susan Jones, Peter Mason, and Ronald Stamper. LEGOL 2.0: A relational specification language for complex rules. *Information Systems*, 4(4):293–305, 1979.

[52] Flanders Judith. *A Place for Everything: The Curious History of Alphabetical Order*. BASIC BOOKS, New York, 1st edition, 2020.

[53] Jungwirth, Paul A. Survey of SQL:2011 Temporal Features. `https://illuminatedcomputing.com/posts/2019/08/sql2011-survey/`. [Accessed: 2022-08-01].

[54] Patrick P Kalua and Edward L Robertson. *Benchmark Queries for Temporal Databases*. PhD thesis, 1993.

[55] Martin Kaufmann, Peter M Fischer, Norman May, and Donald Kossmann. Benchmarking Bitemporal Database Systems: Ready for the Future or Stuck in the Past? 2014.

[56] Martin Kaufmann, Peter M. Fischer, Norman May, Andreas Tonder, and Donald Kossmann. TPC-BiH: A Benchmark for Bi-Temporal Databases. In *In TPCTC*, 2013.

[57] M.A. Ketabchi, S. Mathur, T. Risch, and J. Chen. Comparative analysis of RDBMS and OODBMS: a case study. In *Digest of Papers Compcon Spring '90. Thirty-Fifth IEEE Computer Society International Conference on Intellectual Leverage*, pages 528–537, 1990.

[58] Samiya Khan, Xiufeng Liu, Syed Arshad Ali, and Mansaf Alam. Storage solutions for big data systems: A qualitative study and comparison. *arXiv preprint arXiv:1904.11498*, 2019.

[59] Keith Allen Kimball. *The DATA system*. PhD thesis, 1978.

[60] John Klein, Ian Gorton, Neil Ernst, Patrick Donohoe, Kim Pham, and Chrisjan Matser. Performance Evaluation of NoSQL Databases: A Case Study. In *Proceedings of the 1st Workshop on Performance Analysis of Big Data Systems*, PABS '15, page 5–10, New York, NY, USA, 2015. Association for Computing Machinery.

[61] Krishna Kulkarni and Jan-Eike Michels. Temporal features in SQL:2011. *ACM SIGMOD Record*, 41:34–43, 10 2012.

[62] Lazarevic, Lju. Nodes 2019: How to keep track of change - versioning approaches in Neo4J. `https://neo4j.com/online-summit/session/change-tracking-versioning-approaches-neo4j/`. [Accessed: 2022-08-01].

[63] Leiningen. Leiningen Homepage. `https://leiningen.org/`. [Accessed: 2022-08-01].

[64] Scott T. Leutenegger and Daniel Dias. A modeling study of the tpc-c benchmark. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, page 22–31, New York, NY, USA, 1993. Association for Computing Machinery.

[65] C. Levine. Why TPC-A and TPC-B are obsolete. In *Digest of Papers. Compcon Spring*, pages 215–221, 1993.

[66] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't Settle for Eventual Consistency: Stronger Properties for Low-Latency Geo-Replicated Storage. volume 12, page 30–45, New York, NY, USA, mar 2014. Association for Computing Machinery.

[67] João Ricardo Lourenço, Bruno Cabral, Paulo Carreiro, Marco Vieira, and Jorge Bernardino. Choosing the right NoSQL database for the job: a quality attribute evaluation. *Journal of Big Data*, 2(1):1–26, 2015.

[68] MariaDB. Choosing the right storage engine. `https://mariadb.com/kb/en/choosing-the-right-storage-engine`. [Accessed: 2022-08-01].

[69] MariaDB. Columnstore. https://mariadb.com/kb/en/mariadb-columnstore/. [Accessed: 2022-08-01].

[70] MariaDB. Explain. https://mariadb.com/kb/en/explain/. [Accessed: 2022-08-01].

[71] MariaDB. Query Cache. https://mariadb.com/kb/en/query-cache/. [Accessed: 2022-08-01].

[72] MariaDB. Storage Engine Index Types. https://mariadb.com/kb/en/storage-engine-index-types/. [Accessed: 2022-08-01].

[73] MarkLogic. A deep dive into bitemporal. https://www.marklogic.com/blog/bitemporal/. [Accessed: 2022-08-01].

[74] L. Edwin McKenzie. An Algebraic Language for Query and Update of Temporal Databases. 1988.

[75] Jim Melton. Advanced SQL:1999: Understanding Object-Relational and Other Advanced Features. 2002.

[76] Microsoft. Non-relational data and NoSQL - Azure Architecture Center. https://docs.microsoft.com/en-us/azure/architecture/data-guide/big-data/non-relational-data. [Accessed: 2022-08-01].

[77] Gianina Mihai (Rizescu). Comparison between Relational and NoSQL Databases. *Annals of Dunarea de Jos University of Galati Fascicle I Economics and Applied Informatics*, XXVI:38–42, 12 2020.

[78] MongoDB. Databases and Collections. https://www.mongodb.com/docs/manual/core/databases-and-collections/. [Accessed: 2022-08-01].

[79] MongoDB. Horizontal vs. Vertical Scaling Comparison Guide. https://www.mongodb.com/basics/horizontal-vs-vertical-scaling. [Accessed: 2022-08-01].

[80] Shamkant B. Navathe and Rafi Ahmed. TSQL: A Language Interface for History Databases. In *Temporal Aspects in Information Systems, Proceedings of the IFIP TC 8/WG 8.1 Working Conference on Temporal Aspects in Information Systems, Sophia-Antipolis, France, 13-15 May, 1987*, pages 109–122. North-Holland / Elsevier, 1987.

[81] Neo4J. Disks, Ram and other tips. https://neo4j.com/docs/operations-manual/current/performance/disks-ram-and-other-tips/. [Accessed: 2022-08-01].

[82] Neo4J. Index Configuration. https://neo4j.com/docs/operations-manual/current/performance/index-configuration/. [Accessed: 2022-08-01].

[83] Neo4J. Introduction. https://neo4j.com/docs/operations-manual/current/introduction/. [Accessed: 2022-08-01].

[84] Lara Nichols and Laurian M. Chirica. *A Comparison of Object-Relational and Relational Databases*. PhD thesis, California Polytechnic State University, 2007.

[85] Moira C. Norrie. *A collection model for data management in object-oriented systems*. PhD thesis, University of Glasgow, 1992.

[86] Ohlsson, Andreas and Persson, Mikael. A Comparison in Performance Between a Selection of Databases, 2019. Student Paper.

[87] Oracle. What is a Database? `https://www.oracle.com/database/what-is-database/`. [Accessed: 2022-08-01].

[88] Oracle. Defying Conventional Wisdom. `https://www.oracle.com/us/corporate/profit/p27anniv-timeline-151918.pdf`, May 2007. [Accessed: 2022-08-01].

[89] OrientDB. Time Series. `https://orientdb.org/docs/2.2.x/Time-series-use-case.html`. [Accessed: 2022-08-01].

[90] Zachary Parker, Scott Poe, and Susan V Vrbsky. Comparing nosql mongodb to an sql db. In *Proceedings of the 51st ACM Southeast Conference*, pages 1–6, 2013.

[91] Tariq Rao, Ehsan Haq, and Dost KHAN. Performance based Comparison between RDBMS and OODBMS. *International Journal of Computer Applications*, 180:42–46, 02 2018.

[92] Rouhaud, Julien. Re: Clear cache in postgresql. `https://www.postgresql.org/message-id/20210616063919.hrysiixqgr2wcyrx%40nol`. [Accessed: 2022-08-01].

[93] Lawrence Rowe. History of the Ingres Corporation. *Annals of the History of Computing, IEEE*, 34:58–70, 10 2012.

[94] W. B. Rubenstein, M. S. Kubicar, and R. G. G. Cattell. Benchmarking Simple Database Operations. SIGMOD '87, page 387–394, New York, NY, USA, 1987. Association for Computing Machinery.

[95] Rubik Sadeghi. *A Database Query Language for Operations on Historical Data*. PhD thesis, 1987. AAID-80693.

[96] Kosovare Sahatqija, Jaumin Ajdari, Xhemal Zenuni, Bujar Raufi, and Florije Ismaili. Comparison between relational and nosql databases. In *2018 41st international convention on information and communication technology, electronics and microelectronics (MIPRO)*, pages 0216–0221. IEEE, 2018.

[97] Salvisberg, Philipp. Multi-temporal features in Oracle 12C. `https://www.salvis.com/blog/2014/01/04/multi-temporal-database-features-in-oracle-12c/`. [Accessed: 2022-08-01].

[98] N. L. Sarda. Algebra and Query Language for a Historical Data Model. *Comput. J.*, 33(1):11–18, feb 1990.

[99] Vipin Saxena and Ajay Pratap. Article: Performance Comparison between Relational and Object-Oriented Databases. *International Journal of Computer Applications*, 71(22):6–9, June 2013.

[100] Schaefer, Lauren. NoSQL vs SQL databases. `https://www.mongodb.com/nosql-explained/nosql-vs-sql`. [Accessed: 2022-08-01].

[101] Nadia Ben Seghier and Okba Kazar. Performance Benchmarking and Comparison of NoSQL Databases: Redis vs MongoDB vs Cassandra Using YCSB Tool. In *2021 International Conference on Recent Advances in Mathematics and Informatics (ICRAMI)*, pages 1–6, 2021.

[102] S. K. Singh. *Database Systems: Concepts, Design & Applications*. Prentice Hall Press, USA, 1st edition, 2009.

[103] Karen Smith and Stan Zdonik. Intermedia: A Case Study of the Differences Between Relational and Object-Oriented Database Systems. volume 22, pages 452–465, 12 1987.

[104] Richard T. Snodgrass. *TSQL2*, pages 3192–3197. Springer US, Boston, MA, 2009.

[105] Andreas Steiner. A generalisation approach to temporal data models and their implementations. 1998.

[106] Hallam Stevens. Hans Peter Luhn and the birth of the hashing algorithm. *IEEE Spectrum*, 55(2):44–49, 2018.

[107] Michael Stonebraker and Jason Hong. Saying Good-Bye to DBMSs, Designing Effective Interfaces. *Commun. ACM*, 52(9):12–13, sep 2009.

[108] Michael Stonebraker and Lawrence A. Rowe. The POSTGRES papers. Technical Report UCB/ERL M86/85, EECS Department, University of California, Berkeley, Jun 1986.

[109] tatut. xtdb-inspector. `https://github.com/tatut/xtdb-inspector`. [Accessed: 2022-08-01].

[110] Peter Macleod Thompson and Anton W. Gelijn. *A Temporal Data Model Based on Accounting Principles*. PhD thesis, CAN, 1991. AAINN71151.

[111] TigerGraph. Graph gurus episode 10 - analyzing temporal data with a graph database. `https://info.tigergraph.com/graph-gurus-10`, 2019. [Accessed: 2022-08-01].

[112] TPC. Origins of the TPC and the first 10 years. `https://www.tpc.org/information/about/history5.asp`. [Accessed: 2022-08-01].

[113] TPC. TPC Benchmarks Overview. `https://www.tpc.org/information/benchmarks5.asp`. [Accessed: 2022-08-01].

[114] Bogdan George Tudorica and Cristian Bucur. A comparison between several NoSQL databases with comments and notes. In *2011 RoEduNet International Conference 10th Edition: Networking in Education and Research*, pages 1–5, 2011.

[115] Various. GitHub Repo - Bitemporal tables in postgres. `https://github.com/scalegenius/pg_bitemporal`. [Accessed: 2022-08-01].

[116] Paul Werstein. A performance benchmark for spatiotemporal databases. In *In: Proc. of the 10th Annual Colloquium of the Spatial Information Research Centre*, pages 365–373, 1998.

[117] Gio Wiederhold, James F. Fries, and Stephen Weyl. Structured Organization of Clinical Data Bases. In *Proceedings of the May 19-22, 1975, National Computer Conference and Exposition*, AFIPS '75, page 479–485, New York, NY, USA, 1975. Association for Computing Machinery.

[118] P. F. Windley. Trees, Forests and Rearranging. *The Computer Journal*, 3(2):84–88, 01 1960.

[119] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, and Björn Regnell. *Experimentation in Software Engineering.* Springer, 2012.

[120] XTDB. Configuring. `https://docs.xtdb.com/administration/configuring/`. [Accessed: 2022-08-01].

[121] XTDB. What is XTDB? `https://docs.xtdb.com/concepts/what-is-xtdb/`. [Accessed: 2022-08-01].

[122] Amal W. Yassien and Amr F. Desouky. RDBMS, NoSQL, Hadoop: A Performance-Based Empirical Analysis. In *Proceedings of the 2nd Africa and Middle East Conference on Software Engineering*, AMECSE '16, page 52–59, New York, NY, USA, 2016. Association for Computing Machinery.

Appendices

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2022-11-18

EXAMENSARBETE Bitemporal modeling: implementations, performance, and use cases
STUDENT Anthony Bui
HANDLEDARE Sergio Rico (LTH), Adam Nilsson (Parkster AB)
EXAMINATOR Roger Henriksson (LTH)

Res genom tiden med bitemporala databaser

POPULÄRVETENSKAPLIG SAMMANFATTNING **Anthony Bui**

Tidsaspekten av data har länge varit ett kraftfullt verktyg till beslutstagande. Oftast handlar det om vad vi visste, exempelvis en anställds lön, men på senare tid har man också lagt fokus på *när* man visste något. Vi har undersökt de motsvarande tidsaxlarna giltighet- och transaktionstid i dagens databaser.

I vanliga fall lagras information i elektroniska databaser utan någon tidsaspekt. En butik kan exempelvis uppdatera sina priser veckovis, och när en ny vecka är kommen skrivs dessa över med de nya. Historiska värden som vi eventuellt önskar av jämförelse ändamål hade krävt att vi explicit sparar undan tidsperioden för varje vara och dess pris - i detta fall veckovisa perioder. Ordnar vi detta kan vi "resa genom tiden" sett till prisnivåerna. Låt säga att de digitala prisskyltarna en måndagsmorgon fallerar och uppdaterar sina värden sent, vilket leder till att kunder fått varor till fel pris. Hur får vi reda på hur länge en skylt visade fel pris?

Hanterandet av tid i databaser har varit en forskningsfråga parallell med utvecklingen av digitala databaser redan under 1950-talet. Förslagen har varit många, men de flesta har introducerat tidsperioder till giltighetstiden, tiden då något var sant, samt transaktionstiden, tiden då vi registrerat fakta. Att hålla reda på båda samtidigt ger oss en så kallad bitemporal databas.

Frågan i exemplet med butiken hade man förvisso kunnat svara på genom att jämföra kundernas kvitton, dock utan exakta tidpunkter. I andra scenarion, så som inom aktiemarknaden, är det viktigt att veta priset vid ett visst klockslag,

parkingZoneID	name	vt_start	vt_end
123	Lund High School	2019-01-01	2019-12-31
123	Lund College	2020-01-01	2020-01-31
123	Lund College	2020-02-01	2038-01-01

↓

parkingZoneID	name
123	{<[2019-01-01, 2019-12-31], Lund High School> <[2020-01-01, 2038-01-01], Lund College>}

men också *när* vi fick reda på priserna för att kunna veta vilken data vi hade till hands när vi tog beslut. En bitemporal databas har med andra ord stor nytta i många användningsområden.

Vi undersöker tidsmodeller som föreslagits i forskningen samt hur de kan appliceras i dagens databaser. Vi implementerade också en enkel bitemporal modell till några av de mest populära databaserna och utförde prestandatester. Vår slutsats är att dagens databaser inte når upp till modellernas standard, trots SQL:2011 standarden som agerat som milstolpe. Prestandatesterna nådde inte upp till våra förväntningar, dock beror det nog på de tekniska begränsningarna. Avslutningsvis tror vi att arbetet bidrar till forskningen genom att ge en aktuell bild av tidsmodelleringen och dess appliceringar, samt genom praktiska implementeringar i fyra databasmodeller.