DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2018

Comparing the performance of relational and document databases for hierarchical geospatial data

ANDRÉ JOSEFSSON

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Comparing the performance of relational and document databases for hierarchical geospatial data

ANDRÉ JOSEFSSON

Master in Computer Science
Date: July 3, 2018
Supervisor: Mika Cohen
Examiner: Mads Dam
Swedish title: En prestandajämförelse av relationella och dokumentorienterade databaser vid lagring av hierarkisk geospatial data
School of Electrical Engineering and Computer Science

Abstract

The aim of this degree project is to investigate alternatives to the relational database paradigm when storing hierarchical geospatial data. The document paradigm is found suitable and is therefore further examined. A benchmark suite is developed in order to test the relative performance of the paradigms for the relevant type of data. MongoDB and Microsoft SQL Server are chosen to represent the two paradigms in the benchmark. The results indicate that the document paradigm has potential when working with hierarchical structures. When adding geospatial elements to the data, the results are inconclusive.

Sammanfattning

Det här examensarbetet ämnar undersöka alternativ till den relationella databasparadigmen för lagring av hierarkisk geospatial data. Dokumentparadigmen identiferas som särskilt lämplig och undersöks därför vidare. En benchmark-svit utvecklas för att undersöka de två paradigmens relativa prestanda vid lagring av den undersökta typen av data. MongoDB och Microsoft SQL Server väljs som representanter för de två paradigmen i benchmark-sviten. Resultaten indikerar att dokumentparadigmen har god potential för hierarkisk data. Inga tydliga slutsatser kan dock dras gällande den geospatiala aspekten.

Contents

Chapter 1

Introduction

This chapter begins by introducing relational databases, their limits and a problem related to those limits. NoSQL databases are then presented as a potential solution in need of further investigation. Lastly, the problem is formally defined and a study is proposed.

1.1 Problem Background

Relational databases are based on the relational model proposed by E.F Codd in 1970 [5]. This kind of database emphasizes consistency and fault tolerance, as can be seen in the ACID[1] properties central to the relational paradigm. From their early introduction on the market, relational database management systems (RDBMS) dominated the database world for a long time. Only during the past decade has there been any significant interest in alternative approaches.

The desire for new types of databases mainly stems from the trade-offs incurred by adhering to the relational model and ACID properties. These trade-offs include difficulties in parallelizing write-operations and scaling the database over several machines. The adherence to the relational model also makes for rigid data relationships, as all data must fit into the database's schema definition.

With the rise of Web 2.0 there has been an explosion in the amount of collected data, often referred to as "Big Data". Being huge, sparse, poorly structured and often in need of heavy analyses in order to be useful, this data has been a driving factor in the development of alternatives to the relational paradigm [23]. Geospatial information is one type of data that is often present in big data. In the year 2000, 80% of all corporate data

[1]**A**tomicity, **C**onsistency, **I**solation, **D**urability. Further explained in the theoretical chapter.

had spatial elements [15], a figure which is likely even higher today considering the many ways to retrieve a user's location. The geospatial parts can also add additional challenges to a database, such as unusually large records and deep hierarchies.

The term "NoSQL" was coined in 1998, but did not see wide usage until its popularization during a database meetup in 2009 [34][1]. The term is generally seen as a shortening of "Not only SQL", referring to a database with at least some characteristic not present in a typical relational database.

NoSQL databases have risen in popularity during the past decade as an attempt to make up for the shortcomings of the relational paradigm, especially when working with big data. One of the main design ideas in popular NoSQL database management systems (DBMS) is that by loosening the ACID requirements, the database can be improved in other aspects. Another key observation is that the optimal choice of data model depends on the structure of the data being stored and its access patterns. Altogether these observations have given rise to several paradigms of NoSQL, each optimizing for different use cases.

Triona is a Swedish IT company specializing in logistics and infrastructure related solutions. As such, there is often an element of geospatial data in their systems. One of their relational databases has been identified as exerting several of the characteristics problematic for RDBMSs. The data contains elements of geospatial and sparse data, with hierarchical data structures and inheritances. This data might therefore benefit from residing in another type of database.

This study attempts to investigate whether a viable alternative to relational databases exists for the dataset described from Triona. As such, the interest lies in finding a NoSQL paradigm suited for hierarchical data with geospatial elements. Focus is put on the relative performance of the NoSQL paradigm and relational databases. Most information on the area today comes in the form of blog posts and first-party publications, making it difficult to draw reliable conclusions solely from reading. This study therefore attempts to compare the performances through a benchmarking approach.

1.2 Problem Definition

The aim of the study is to investigate the relative performances of the relational paradigm and the NoSQL paradigm best suited to hierarchical geospatial data. Performance is measured in terms of throughput and latency. The study begins with a survey of existing NoSQL paradigms, evaluating their respective potential for hierarchical geospatial data. As further explained in the next chapter, the document paradigm is deemed most suitable and is therefore chosen.

Next, the relational and the document paradigm each get one DBMS chosen as their representative. The study attempts to choose these DBMSs to be as representative of the paradigms' potential as possible. A survey of DBMSs in the paradigms is used as the basis for the decision. To measure the performance, a benchmark suite is created. The real-world database from Triona is used as the dataset for that suite. The two representatives each get populated with appropriate models of the dataset. The suite is finally run against the two DBMSs to obtain results.

The research question of the study therefore becomes:

> *"How do relational and document databases compare in terms of throughput and latency when storing hierarchically structured geospatial data on a single node?"*.

Due to time constraints, the study is limited to comparing single-node executions. The study is also limited to choosing one NoSQL-paradigm for investigation beyond the initial survey. It is further limited to choosing one DBMS for each paradigm to act as representative in the performance tests. The study is also limited to a single modelling technique for each of the two paradigms. Lastly, the study is limited to working with the dataset from Triona and modified versions of that set.

The study aims to contribute to the field of databases and geographic information. The study does not touch on any ethically problematic areas, and so does not further discuss that topic. The study also does not touch on any issue related to sustainability, other than the possible resource savings that can come from an improved database performance.

Chapter 2

NoSQL Paradigms

This chapter establishes which NoSQL paradigm is most suitable for working with hierarchical geospatial data. The chapter begins by explaining key concepts necessary in the context. It then presents the most prevalent NoSQL paradigms and picks one based on that material.

2.1 Terminology

Horizontal Scaling is the extent to which a system can gain performance by distributing its data and workload onto several, non-memory sharing machines [4].

The **CAP Theorem** states that when distributing a database over several nodes, it is impossible to achieve more than two of the three properties: **C**onsistency, **A**vailability, **P**artition tolerance. Consistency in this context means that the system behaves as if all operations were sequentially executed on a single node. Availability means that all requests to a healthy part of the system will eventually be served. Partition tolerance means that if some part of the system fails, the other parts continue to operate [14].

ACID is an abbreviation of **A**tomicity, **C**onsistency, **I**solation, **D**urability. These four properties together are intended to guarantee the validity of data after transactions, even under errors such as power failure. Atomicity means that either the entire transaction happens or none of its changes occur. Consistency in this context refers to transactions always leaving the database in a semantically valid state. Isolation says that all concurrent operations must behave as if they were executed sequentially. Finally, durability ensures that once a transaction is committed, it must occur.

BASE is an abbreviation of **B**asically **A**vailable, **S**oft state, **E**ventually consistent. The abbreviation is not as explanatory, but in essence describes an abandonment of ACID guarantees in favor of increased availability. A BASE system cannot guarantee consistency at a specific point in time, but will always eventually reach such a state if given enough time to stabilize [1].

2.2 Established NoSQL Paradigms

NoSQL is a wide term, essentially referring to any database that is not purely SQL. There are however a few paradigms which can be used to classify the common NoSQL DBMSs. Shared by all of the listed paradigms is a lowered emphasize on consistency in return for higher availability in terms of the CAP theorem. NoSQL databases also tend to follow BASE rather than ACID to further improve availability. The exact definition of each paradigm can vary slightly over the literature. The descriptions given here attempt to incorporate only the most agreed upon features of each one.

Key-Value databases consist of a long list of records (values) which can be accessed using the corresponding keys. The access uses a hash of the key and can therefore be performed as an amortized O(1) operation [27]. This type of database completely lacks a schema, making it highly flexible. It is also memory efficient since none-applicant values are omitted from a record altogether, instead of storing it as NULL. In contrast to most other paradigms, values are generally opaque to the system. This allows for simpler DBMS implementations but also makes the system primitive. With opaque values, the database cannot be indexed, and consequently cannot be efficiently queried based on its values. With these limitations and the lack of a schema, the task of creating structure and efficiency is mostly left to the user.

Document databases store data in the form of semi-structured documents, such as JSON or XML. Like key-value stores, these documents are free of schemas and can be accessed with a key called the unique document identifier. Unlike key-value stores, document databases have transparent values, allowing for indexing and content-based querying. Document databases are built around the idea of keeping related data in the same document when possible. This leads to fast reads, but may require some redundancy for certain data structures.

Graph databases store data in the form of graphs. The nodes in this graph are records containing attributes. A unique idea in this paradigm is that relationships (or edges) too can contain attributes of their own. In some graph databases, edges are even first-class citizens in the data model [21]. Depending on the implementation, graph databases may offer O(1) neighbour transitions without the need for indexes. Common applications for this type of database are social networks, recommendation services or other types of systems where neighbour transitions are frequent. Graph databases are among the least adopted paradigms treated in this thesis [22].

Wide-Column databases have several similarities to the relational paradigm. Both store their entries as rows which in turn contain values for predefined columns. Instead of relational tables, this paradigm uses the concept of column families to house its rows. One difference between the two is that column-families do not need to store NULL values for unpopulated attributes [27]. This is a subtle difference that has big implications when dealing with sparse data. Where a relational database is filled mostly with NULL values, a wide-column database can represent the same data being almost empty. Several popular wide-column databases are also built on top of distributed file systems, having horizontal scaling built into the architecture. In general, these databases are built with huge and distributed applications in mind.

Object databases store objects from some specific object-oriented programming language. This approach gives rise to several advantages and disadvantages. Advantages include consistent data representation throughout the application and removing the time used to convert database objects to application objects. A clear disadvantage is that the database gets tied to a specific programming language. Another disadvantage is that the database inherits any limitation present in the used object type [27]. The object paradigm is currently the least adopted of the ones mentioned [22].

2.3 Choice of Paradigm

As the study only intends to investigate one NoSQL paradigm, a decision has to be made regarding which one to pick. Performance is the core focus of the study and is therefore the main concern when selecting a paradigm. To maximize the likelihood of a selecting a well performing paradigm, the three main criteria for the candidates are:

- Having some feature that can improve on relational databases when modelling hierarchies.
- Having an overall design that allows for efficient handling of geospatial data.
- Having mechanisms to efficiently deal with sparse data.

Starting with geospatial indexing, key-value stores can more or less be removed from the discussion. Since this paradigm generally does not support indexing at all, its DBMSs are deemed unable to perform efficient geospatial queries. The paradigm is niched towards situations where the user has a good idea of which record he wants, which is not the case here. While the object paradigm does not inherently prohibit spatial operations, none of the established DBMSs of the paradigm natively support it. The remaining paradigms show no immediate obstacles for geospatial data.

The second criterion revolves around modelling hierarchies. Graph databases offer O(1) neighbour transitions. This allows them to easily traverse hierarchies, making them suitable. Document databases can nest entire hierarchies in the same document. Unless the hierarchies are very large this too should provide fast access and easy modelling. Wide-Column databases however do not offer any feature that can improve significantly on the relational paradigm in this regard. Wide-Column databases are therefore removed from consideration.

The last criterion is to provide some mechanism to efficiently deal with sparse data. Both document and graph databases fulfill this criterion since they offer schema-free storage. With schema-free storage, only populated properties have to be defined for a record, shrinking the size of records with unpopulated properties.

With both graph and document databases fulfilling all criteria, the final decision is based on overall feasibility. Since document databases are more widely adopted, they are deemed more likely to succeed in practice, regardless of the two paradigms' equal theoretical potential. The document paradigm is therefore chosen. The higher level of adoption also makes the results interesting to a larger crowd. Graph databases will therefore not be further considered in this study, but are still of high interest for any future study.

Chapter 3

Theoretical Background

This chapter begins by introducing the relational and document paradigms. The chapter then presents the candidate representatives for the document paradigm. Finally, some theory behind data modelling and benchmarking is given.

3.1 Relational Databases

Relational databases are based on the relational model presented by E.F Codd in 1970 [5]. Relational databases structure their data according to a per-database defined schema. The schema defines a number of containers called tables. Each table is further defined to have a number of columns. A column defines the name and datatype of an attribute that every record in this table must contain. Records, also called rows, can now be inserted into the tables. Each row will consist of values corresponding to each column in its table. If a column is not applicable to a specific row, the value NULL can be specified.

Each table may define one or a set of columns to be the table's primary key. The primary key is used to uniquely identify each row in the table, and therefore must be unique for every record. A table can also define one or several foreign keys to create relationships between rows. A foreign key is a column or a set of columns used by one row to uniquely identify another row [9].

To manipulate the database, the relational paradigm makes use of the Structured Query Language (SQL). SQL defines the four core statements INSERT, UPDATE, SELECT & DELETE. These are used to add, modify, fetch and remove data from the database. To apply conditions to these statements, SQL defines several clauses. One of the most important ones is the WHERE clause, used to specify which rows a query should affect. Another important clause is JOIN. JOINs are used to combine two or more rows based on a relationship between their columns. The JOIN clause may be used on any set of columns, but is most commonly used in combination with foreign keys.

Viewed as an abstract model, Codd's relational model can perform all of its operations instantly. In order to fully use all of the features in the relational model, this is an essential property. Implementations, of course, cannot complete operations instantaneously. Instead, the relational paradigm uses ACID transactions to mimic this behaviour. While these transactions are not instantaneous, they are atomic, all-or-nothing operations that can be treated much like an instant operation in some regards. In order to provide these transaction guarantees in a multi-threaded environment, RDBMSs use locks. These locks can be placed on single rows or entire tables to prevent other threads from accessing data that is currently being modified. If two threads are interested in the same record, these locks can reduce performance.

Database normalization is the practice of organizing data into specific patterns called normal forms. Codd introduced several different normal forms as part of his relational model. While differing slightly, they all essentially consist of constraints that decrease redundancy and protect against certain data inconsistencies [9]. Whether normalization is technically part of the relational paradigm is debatable. It is however a widespread practice and links closely to the ACID properties fundamental to all relational databases.

Relational databases prioritize consistency over availability in terms of the CAP theorem. This makes them naturally apt at avoiding data inconsistencies and race conditions. The strictly defined query and modelling languages are designed to allow for efficient software implementations. Lastly, having been the goto paradigm in the industry for several decades, RDBMSs have seen a much larger development effort than any other paradigm. This means that they are generally highly optimized.

The strict design choices of relational databases have downsides as well though. The ACID requirements limit RDBMSs ability to deal with high loads in several ways. Since all operations must appear to be performed in isolation, high parallelization can be difficult to achieve. Requirements on consistency and durability also limit horizontal scaling, requiring a single, increasingly powerful computer as the load increases. Working with sparse data, RDBMSs need to store a high number of NULL values as every row must have a value for every column.

Another type of data that does not map perfectly with the relational paradigm is inheritance hierarchies. In this type of data, there is a finite and clearly defined set of subtypes of an entity. The root in the hierarchy, which may or may not be a concrete entity subtype in itself, contains those properties applicable to all subtypes. The children of the root, which also may or may not be concrete entity subtypes, continue adding properties applicable to its children. The leaves of the hierarchy must be concrete entity subtypes as no other entities inherit from these. Fowler [12] describes three distinct methods for modelling inheritance hierarchies in relational databases:

- **Single Table Inheritance** - Use a single table to represent a merge of all subtypes. This approach minimizes the number of tables and centralizes data. A downside is a large number of NULL values being stored wherever columns do not apply to every row.
- **Class Table Inheritance** - Use one table for each entity subtype, whether it is a concrete or an abstract type. This approach means that tables will only contain columns applicable to all of its rows, eliminating the need to use NULL values for this purpose. This is the most normalized approach of the three. The downside is that a concrete entity may have its columns spread out over many levels in the hierarchy.
- **Concrete Table Inheritance** - Use one table for each concrete subtype only. This approach compromises between the other two. Tables will only contain columns applicable to their subtypes and all data specific to one entity type will be gathered in a single table. The downside to this approach is a high level of redundancy, as columns shared by several subtypes are defined in several tables.

3.1.1 Microsoft SQL Server

Microsoft SQL Server is an RDBMS created by Microsoft. This study uses it as the representative for the relation paradigm, due to its leading performance and geospatial support. The choice will be further explained in the next chapter. This RDBMS closely follows the relational paradigm and provides support for SQL querying. Indexes can be created on common data types such as numbers, dates and text. Geospatial indexing is also supported [26]. The geospatial operations available are the basic within, intersects & near as well as contains, overlaps, touches and equality[25]. Like most other DBMSs, it uses a B-Tree index structure [26].

3.2 Document Databases

This section describes document databases in more detail, highlighting positive and negative sides of the paradigm. Towards the end, a short description is given of each of the most established DBMSs in the paradigm. MongoDB is given more space, as it is later chosen to represent the paradigm.

Document databases store data in the form of documents. A document is a semi-structured record of a format such as JSON or XML. Each document consists of a set of fields that contain either values or nested documents. Several documents can be grouped together into containers called *collections* [16]. These collections differ a lot from relational tables. For example, collections put no requirements on which fields or datatypes each document should contain. This means that every document in a database could have a unique set of fields. The only exception to the rule is that every document must have a unique identifier.

The properties mentioned so far are similar to key-value stores. The big difference is that the content of a document is transparent to the document database, as opposed to the values in a key-value store. This allows document databases to provide common features absent in key-value stores, such as indexing and a query language. To be able to efficiently use these features, while keeping the database schema-less, some level of structure among documents is needed in the database though. As opposed to the relational paradigm, the document paradigm mostly leaves this task to the user, for better and for worse.

The query capabilities can vary slightly between different document database implementations. However, a big part of the paradigm is to not rely on a JOIN clause for combining data. Instead, the query language is built around the practice of storing all related data in the same document when possible, avoiding normalization. Not relying on normalization also opens up for a higher level of data duplication wherever it can speed up the database. These ideas tie in closely with the paradigm's prioritization of availability over consistency. Positive effects of this strategy are an increased speed in reading data and a simplified data reading process. Storing related data together can also give a more intuitive match between the data model and the entities being modelled, eliminating the need for entity-relational mappings. Downsides include more complicated and potentially slower updates when data is duplicated. With the decreased focus on consistency, it is also possible to reach states where duplicated data is inconsistent across the database. Duplication will also increase the size of the database. However, with the size of modern memories compared to the speed of modern processors, the document paradigm still recognize this to be a valuable trade-off. The trade-off is mostly made viable because of the relaxed consistency model that comes from the paradigm's BASE-philosophy.

To model data in more sophisticated ways than mentioned so far, the document paradigm offers two concepts; document embedding and document referencing. Starting with document embedding, this is a technique made possible from the use of a semi-structured record format. Document embedding is the practice of nesting documents inside each other. The parent document stores a nested child document as if it was a piece of data like any other. This nesting can be performed recursively, although creating top-level documents that are too large would be inefficient in practice. The nested sub-document is stored and accessed together with its parent, eliminating the need to use a JOIN clause when fetching them together. The main application for document embedding is to store related data in the same record, while still maintaining some structure to it. Advantages to embedding include fast and simple retrieval, since the data is already grouped and organized. A major disadvantage is that not all relationships can be modelled in this way. For *1:1* relationships, embedding is a straightforward process. For *1:N* relationships, lists can be used to embed multiple documents. However, when faced with *N:1* or *N:M* relationships, document embedding encounters issues. If embedders and embeddees are distinct subsets, these relationship cardinalities can still be modelled through embedding, by duplicating data. In other cases however, these relationships cannot be modelled through embedding at all.

When document embedding is impossible or otherwise undesirable, document referencing can be used instead. In this approach, documents store a copy of another document's unique identifier. The reference is stored like any other value, meaning it can be placed in lists, indexed and queried on. While similar to relational foreign keys in some ways, document references are generally seen as a looser connection. Also, because they are not backed up by a JOIN clause, these references have less expressive power when querying. Records can for example not be selected based on the content of their referenced documents. The advantage of document referencing is that it can model any relationship cardinality. It also helps to prevent documents from growing into unmanageable sizes. The disadvantages are mainly that references are slower and offer less power when querying than embedding does.

3.2.1 Available Document Databases

This section provides brief descriptions of the currently most popularly ranking document DBMSs according to *db-engines.com* [22]. The section has been filtered to only include DBMSs with native support for geospatial datatypes.

ArangoDB provides several data models, one being document storage. It has a free community edition and a proprietary enterprise edition with increased features and support. It supports geospatial within and near queries [3]. It is ranked 10 in terms of popularity among all document databases according to db-engines.com [22].

CosmosDB is a part of Microsoft's Azure cloud services, having a free trial period but is otherwise proprietary. It is a so-called multi-paradigm DBMS, supporting document, graph, key-value and wide-column models. It supports geospatial within, intersects and near queries [13]. Ranked 5 in terms of popularity among all document databases according to db-engines.com [22].

Couchbase provides both an open-source community edition and a proprietary enterprise edition with increased support and features. It only supports querying for locations within a bounding-box [7]. Ranked 3 in terms of popularity among all document databases according to db-engines.com [22].

RethinkDB has no proprietary version. It supports geospatial intersects and near queries [31]. Ranked 9 in terms of popularity among all document databases according to db-engines.com [22].

MongoDB provides a free community edition and a proprietary version in the form of server software or a cloud service. It supports geospatial within, intersects and near queries [17]. Ranked 1 in terms of popularity among all document databases according to db-engines.com [22].

3.2.2 MongoDB

Being a document database, MongoDB stores documents inside collections. Collections are user-defined containers used to group documents together. As explained earlier, collections put no requirements on the structure of its documents, meaning the database is schema-free [24]. Some homogeneity is however needed for efficiency, as you need properties to place your indexes on.

Indexes are created on a collection basis. By default, all collections are indexed on the "_id" field, MongoDB's unique document identifier. The user may create additional indexes on any field. The schema-free nature implies that all documents in a collection might not contain the field of a specific index. In this case, those documents are ignored by the index.

MongoDB provides two ways of querying data; operators and aggregates. Operators can perform simpler tasks such as filtering, searching indexes, ordering and performing arithmetic. Operators are passed to MongoDB as a JSON object and compiled into a single query. This is the primary query language of MongoDB and supports all of the common functions needed in the document paradigm. The other approach is the use of aggregates. Aggregates allow for more advanced tasks, but come at a higher execution time. Aggregates can for example be used to perform multi-stage queries following a pipeline-like workflow. Since aggregates are more of a MongoDB feature than a paradigm property, they will not be further explored in this study.

MongoDB uses a JSON-like document format called Binary JSON (BSON). The format was created by MongoDB to allow for more data types than JSON [19]. When querying, it can be treated as JSON in most cases. MongoDB supports both single field indexes and compound ones. Compound indexes are used when a collection is often queried with a combination of several fields. It also supports geospatial indexes. All indexes use a B-tree structure, similar to most other DBMSs [18].

3.3 Converting Schemas to Documents

The dataset from Triona that this study is based around, currently has a relational representation. For a comparison to be possible, a document representation of the data is needed as well. There are several papers written on the topic of converting relational schemas to document structures. Different approaches can be more or less optimized for ease of migration or performance of the resulting database. The approaches can also differ in how much resemblance they strive to achieve between the resulting document structure and the original relational schema. One approach is to look at the schema and access logs of the RDBMS in order to determine a suitable document representation. This way you can construct the document representation based on access patterns and optimize it according to those. An algorithm to perform such a conversion is described by Jia et al. [24]. The algorithm can informally be explained as follows:

1. Store a complete model of the relational schema.
2. Traverse the log of the relational database. Add "description tags" to the stored model when certain constellations and values are found in the log. The following description tags exist:
    - **Frequent JOIN** - If two tables are found to be JOINed more frequently than a predefined threshold, add this tag to their relationship.
    - **Big Size** - If the average size of the rows in a table is found to exceed a predefined threshold, this tag is added to that table.

- **Frequent Modify** - If the frequency of ALTER or DELETE operations on a table exceed a predefined threshold, add this tag to the table.
- **Frequent Insert** - If the frequency of INSERT operations on a table exceed a predefined threshold, add this tag to the table.

3. When the description tags have been added, another set of tags called "action tags" are generated. Action tags are what will be used to create the final document structure. The action tags are generated as follows:

   (a) For each table, if marked with any of the description tags *Big Size*, *Frequent Modify* or *Frequent Insert*, mark the table as **not eligible for embedding**. Otherwise, mark the table as **eligible for embedding**.

   (b) For each *1:1* relationship, if the child table is marked as eligible for embedding, mark relationship as **embed child**. If the child is not eligible for embedding, but the parent is, mark the relationship as **embed parent**. If neither are eligible for embedding, mark relationship as **reference**.

   (c) For each *1:N* relationship, if the child entity is eligible for embedding and the relationship has a *Frequent JOIN* tag, mark relationship as **embed child**. Otherwise mark relationship as **reference**.

   (d) For each *N:M* relationship, mark relationship as **reference**.

4. The result is a directed graph where tables are nodes and embeddings are directed edges, going from the table being embedded to the table in which to be embedded. If this graph is not a DAG, remove edges (turning embedding into referencing) until all cycles have been removed. This is necessary because a cycle of embedding cannot be implemented.

5. The document schema is now complete, consisting of the original tables combined using the embedding and reference tags generated by the algorithm.

The above algorithm covers most of the common strategies in document modelling. It does however miss out on two-way embedding [33]. If two documents contain information that is often retrieved together, but the documents are not suited for regular embedding, a two-way embedding approach can be used instead. In this approach, the two documents, A and B, will each contain an embedded mini-version of each other. The mini-version of A, embedded into B, might only contain the most important fields and vice versa. This way both documents can get the querying speed advantages of embedding, while none of them get the modelling constraints that come with it. The two-way embedding approach does however imply duplication of data. So while it results in faster reads, it also makes for slower and more complex writes, as several locations need to be updated when a single record changes. Ultimately it is a speed-simplicity trade-off whose viability has to be determined on an application specific level.

MongoDB provides an official handbook for migrating a relational database to MongoDB [20]. Most of the hints provided are in some way covered in the above paragraphs, but a few new ones are presented. The first one is that large hierarchical data should not rely solely on embedding, due to the large memory footprint when dealing with those documents. Another tip is to not combine frequently used data and rarely used data in the same document. This is because a document database will generally work with documents in their entirety, and so can be cumbered by constantly loading seldomly used properties in and out of memory. The last hint is that pieces of data which need to be atomically updated together, need to be stored in the same document. This is because MongoDB, like other document databases, do not support atomic multi-document updates.

3.4 Benchmarking

Benchmarks can be used to attain a quantitative measurement of a database's performance. However, creating a good benchmark is a difficult task. Huppler, from the TPC organization, points out three characteristics of a good benchmark [28]:

- **Relevance** - A benchmark can only cover a limited amount of cases. It is important to choose meaningful and representative operations. Drawing conclusions about something not tested in the benchmark will be difficult.

- **Repeatability** - To ensure that separate runs can be compared in a fair manner, the benchmark must give similar results for repeated runs. Random elements must be performed in such a way that they even out as much as possible.
- **Fairness** - The benchmark must not make design choices that favor specific contenders, unless that choice is a deliberate and explicit part of the benchmark.

TPC is a non-profit organization that designs benchmarks for RDBMSs. They are widely acknowledged as the industry standard. They provide a range of benchmarks designed to simulate different business cases. However, none of the benchmark suites include geospatial data [36].

There are a few other suites around that aim specifically at benchmarking the geospatial functionality of RDBMSs. The most established ones are Jackpine, HSR benchmark and FOSS4G benchmark. The suite from FOSS4G specializes in web map services, which is outside the scope of this study [11]. The remaining two are described in the following paragraphs.

The HSR benchmark was developed at University of Applied Sciences Rapperswil. It is an open-source benchmark covering the most basic geospatial operations. It consists of 4 different operations performed on a real-world dataset. The first operation is a non-spatial count, as querying for metadata is an important part in the workload of a geospatial database. The second query is a spatial intersects-query with a bounding box. The third is a spatial near-query asking for all points within a distance from a point. The last one is a spatial intersection between railroads and lakes in the dataset. The queries are run three times each with random parameters (bounding box, point for the radius etc). Only the third run is timed, so as to benchmark the system in a 'hot' state [35].

Jackpine is an open-source benchmark developed by researchers at the University of Toronto. Like HSR, it performs its queries on an included real-world dataset. The test suite consists of querying for each of the 9 extended topological relations defined by Egenhofer [10]. The benchmarker begins by running warm-up rounds for each query and then goes on to perform several timed queries. The result is calculated as the sum of each query types' average execution time [30].

All of the previously mentioned benchmarks are implemented with RDBMSs in mind only. When it comes to benchmarking NoSQL DBMSs there is only one established option available. Yahoo Cloud Service Benchmark (YCSB) is an extensible open-source benchmarking framework aimed at all major NoSQL databases. It does not include support for RDBMSs. The framework does not come with any data, but instead includes a data generator. Its suites are constructed to be long series of reads and writes at a proportion defined by the user. The framework focuses on measuring latency and throughput. To measure these it makes use of several threads. Latency is determined by putting a moderate load on the database and taking the average response time. Throughput is determined by increasing the load on the database until its serve rate no longer increases. The throughput is then said to be the number of requests served on average per second at this state. YCSB currently does not support benchmarking of geospatial data [6].

3.5 Related Studies

There have been many studies performed around comparing DBMSs. Most only focus on relational databases, making them less relevant to this study. When it comes to explicitly comparing relational databases to NoSQL databases the amount of credible scientific sources are considerably fewer. Blog posts and self-published discussions exist in large numbers, but are of little value to this report due to the difficulty of establishing their validity. This section aims to summarize the most relevant sources of credible and relevant studies in the area.

Zhang et al. [37] identify the growing need to store large unstructured data with geospatial elements mixed in. This need stems from the rise of location-based services and similar applications. They explain why relational databases, which have dominated the geospatial domain for long, are not ideal for this kind of data. Instead, they propose MongoDB as an alternative due to its fitting data model and horizontal scaling among other reasons. The paper goes on to claim that MongoDB is an efficient alternative to RDBMSs for this application, but lacks data to back up this claim.

Agarwal & Rajan [2] acknowledge that NoSQL databases give both advantages and disadvantages over relational ones, depending on what is being compared. In their paper, they investigate the advantages MongoDB could present when working with spatial and non-spatial data mixed together. They perform a series of experiments around querying for restaurants and related properties from a real-world dataset. In their experiment, MongoDB is compared to the RDBMS PostgreSQL. The conclusion is that MongoDB is faster than PostgreSQL by about one order of magnitude for all categories tested.

Parker et al. [29] observe that very few comparisons between NoSQL databases and relational ones exist. They expand the knowledge in the area by performing a series of experiments on MongoDB and Microsoft SQL Server. They define a schema consisting of three different tables with relationships to each other. The three tables are represented by one collection each in MongoDB, using only references to relate documents. The authors perform a series of tests looking at different variations of inserts, updates and reads. Most of the results speak in favor of MongoDB although SQL Server was faster in a few cases. Each test was run 100 times to obtain an average, but a few anomalies in the data exist.

Schmid et al. [32] look at NoSQL databases as an option for storing geospatial data. They claim that only the document and graph paradigms of NoSQL are seeing any noticeable use in combination with geospatial data at the moment. The paper goes on to compare the performances of MongoDB, Couchbase and PostgreSQL. The results show an order of magnitude faster execution for the two document databases when retrieving non-spatial data. For the geospatial data, only MongoDB and PostgreSQL are tested. The results indicate that PostgreSQL is faster for data in the megabyte ranges, while MongoDB is faster in the gigabyte ranges.

Chapter 4

Method

This chapter begins by forming a hypothesis for the outcome of the study, based on the theoretical background. The chapter then describes the choice of paradigm representatives, along with the dataset used in the study. Finally, the benchmark suite is presented.

4.1 Hypothesis

Recall the research question:

> *"How do relational and document databases compare in terms of throughput and latency when storing hierarchically structured geospatial data on a single node?".*

Given the theory presented in the previous chapter, this section derives some hypotheses about what throughput and latency to expect in the experiment. First we consider read-queries only, and then expand to include writes.

To begin with, it seems reasonable to expect that that the throughput-latency relationship will look approximately the same for both paradigms. Having a high throughput relative to the latency would indicate superior concurrency. While the document paradigm is designed to handle many simultaneous requests, that ability usually comes from horizontal scaling. As this study only considers a use case with a single machine, horizontal scaling will not come into play. Also, the limitations imposed by ACID should have equal impact on both metrics. Since latency and throughput are hypothesized to behave similarly, they will both be referred to as 'performance' for the remainder of this section.

Secondly, the document paradigm can be expected to handle hierarchies faster whenever document embedding can be used. This is because the relational paradigm has to JOIN data from several tables when querying for a full hierarchy, incurring some overhead. Document embedding avoids this runtime collation of entities by keeping all data in the same document. However, document embedding is not suitable for every situation, as explained in the previous chapter. When not suitable, the document paradigm has to resort to document referencing. Looking up a document reference requires an index search similar to that in a relational JOIN. On top of that, looking up document references require an additional query. This means a reference should generally be slower to work with than a foreign key. However, this can be counter-balanced by the fact that fewer references are needed in a document model, than foreign keys in a relational model. It is therefore reasonable to expect that the two paradigms will perform about equally well on hierarchies where some document referencing is needed.

Thirdly, both paradigms can be expected to be about equally fast for read-only queries on geometries. This is because none of the paradigms have any outstanding properties that should affect this scenario.

Fourthly, the document paradigm can be expected to be faster when handling the metadata of the geometries. This metadata is usually of a shallow hierarchical structure, with sparsely populated properties and a lack of uniformity across records. The performance in regards to hierarchies has already been discussed and should apply here as well. However, because the hierarchies here are usually shallow, the document paradigm's advantage should not be as impactful. Regarding the non-uniformity and sparsities, the relational paradigm will likely see reduced performance due to its rigid schema model. The main problem with the relational schema in this case is that entities either have to be separated into very many tables, or have large parts of its record consist of mostly NULL values. Both approaches have negative performance implications as one requires additional JOINs and the other increases the memory footprint (which may indirectly decrease the performance). The document paradigm on the other hand can store the data in a single document and omit unpopulated properties. This technique should avoid the issues present in the two relational approaches.

Thus far only read queries have been considered. When introducing write queries, the relational paradigm can be expected to see a greater reduction in performance than the document paradigm. This would mainly be caused by the locks used by the relational paradigm. While likely affecting all queries, these locks may be particularly impactful on geospatial data for two reasons. Firstly, geospatial queries generally involve checks against many records, meaning that even locks on single records may put queries on hold. Secondly, geometric features have bigger variations in size than most other datatypes, increasing the likelihood of having to reorganize the record layout in memory after an update. During such a reorganization, a relational database may have to lock an entire table, greatly impacting the performance.

Finally, one other factor that could impact the performance in all of the above cases is the benefit of a strict schema. Since relational databases define how each record should look, the database has a better knowledge of its records. However, while this allows for certain optimizations, those are hypothesized to be less impactful than the previously mentioned performance factors.

4.2 Choice of Representatives

In order to have something to benchmark, one DBMS per paradigm is chosen as a representative. The representatives are chosen carefully in order for the results to be as generalizable as possible for the paradigms. Unfortunately, there is no such thing as a perfect choice here, but due to time constraints, this approach is still deemed the most feasible.

The most important requirement for the representative is to support geospatial operations and indexing. A threshold for this is set at supporting at least two of the three most common spatial operations (within, intersects, near). This is to ensure that some geospatial queries can be used in the benchmark suite. The representatives should also be among the most performant DBMSs in their respective paradigms to give a fair assessment of the paradigms' capabilities.

Starting with the relational paradigm, there are many DBMSs on the market. Geospatial operations are widely supported in this paradigm, giving a large selection of candidates. It is however difficult to find useful performance metrics. Most comparisons focus on specific business scenarios that have little or no similarity to this study. According to the userbase of G2 Crowd, an established peer-to-peer business review platform, the three most performant RDBMSs in general are Microsoft SQL Server, Oracle Database and MySQL [8]. These are also the three highest ranked RDBMSs in terms of popularity at *db-engines.com* [22]. All three DBMSs have among the most developed sets of spatial operations, offering the usual within, intersects & near operations along with many more. As they all have very similar qualifications, the choice between them is arbitrary. This study chooses to use Microsoft SQL Server.

The document representative is more difficult to determine, as the paradigm is younger and more volatile. CosmosDB is however immediately excluded from consideration due to supporting multiple paradigms. While the DBMS can be run in a document-mode, it is difficult to determine what is going on beneath. All of the surveyed DBMSs support geospatial indexing, but differ on which geospatial operations they offer. Couchbase only supports a within-bounding-box operation and is therefore excluded as well. Of the remaining ones, RethinkDB and ArangoDB each support two, while MongoDB supports all three considered geospatial operations. Regarding performance, it is difficult to find reliable metrics. Not a single benchmark could be found including all three DBMSs at the same time. Most of the benchmarks around are performed by parties with business interests in the area, making the numbers unreliable. MongoDB is currently seeing roughly 100 times more usage than the other two candidates according to *db-engines.com* [22]. While a widespread adoption does not ensure a high performance, it makes it unlikely that MongoDB would be considerably worse than the other two. This, combined with having the best support for geospatial operations, leads the study to choose MongoDB as its document paradigm representative.

4.3 The Dataset

The dataset chosen for this study consists of tables from a forestry database. The data centers around forest remedies and has a hierarchical structure. The hierarchies are formed both from inheritance and parent-child data relationships. It is difficult to give a precise description of the structure without favoring a specific modelling strategy. The complete data model will therefore be shown in both a relational representation (fig. 4.1) and a document representation (fig. 4.2). These representations are the same ones used in the benchmark.

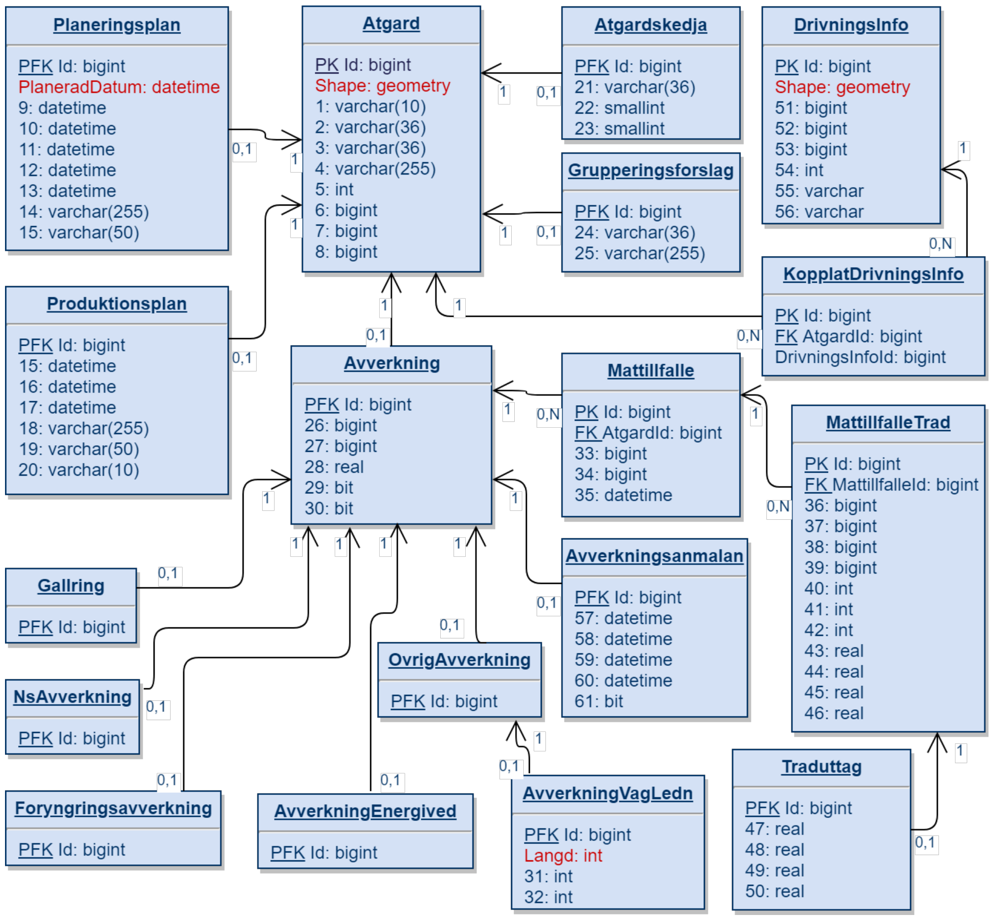
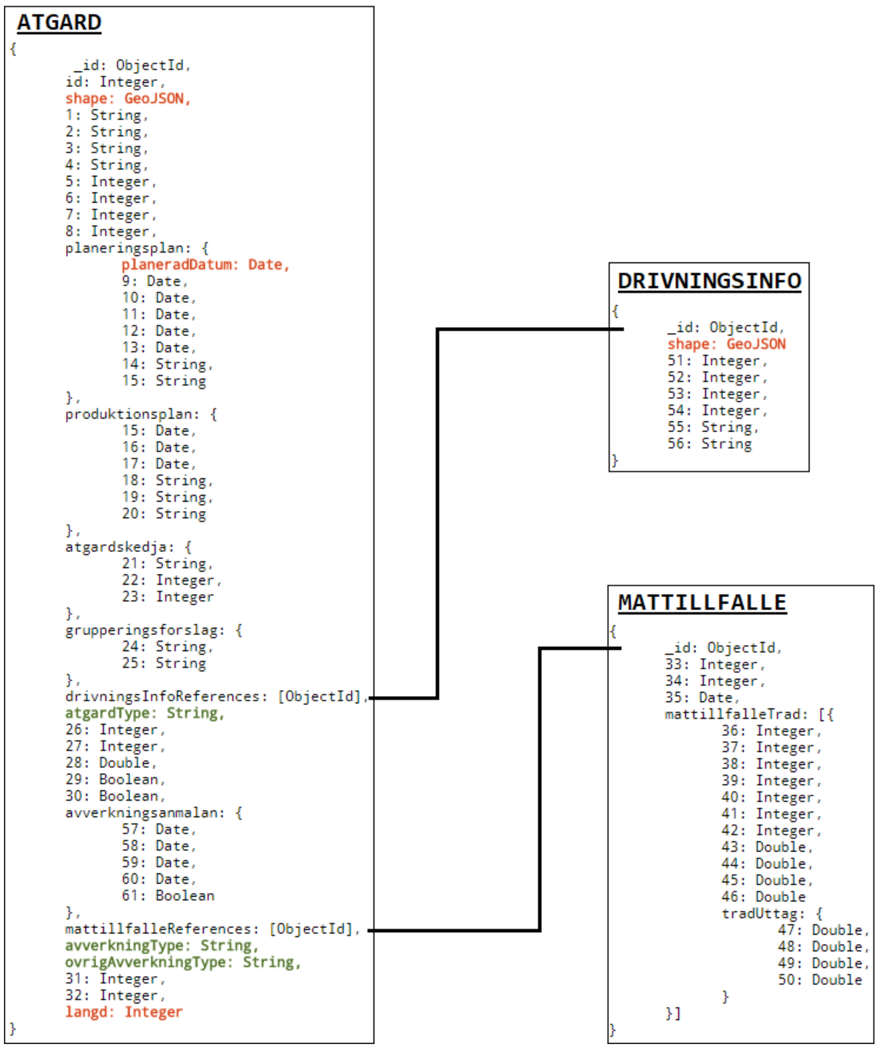


Figure 4.1: *The relational representation of the dataset. Columns that are not important for understanding the model and benchmarking process have been renamed to numbers. The columns central to the benchmark (apart from primary and foreign keys) have been highlighted in red.*

The modelled data is all rooted in being part of a remedy, called *Atgard* in the model. All remedies share some information such as a geometry, an id and a few more properties. There are several types of forest remedies available. The part selected for this study only includes one such remedy, logging or *Avverkning* in the schema. There are in turn several types of logging remedies, five of which are present in the selected part of the dataset (*Gallring*, *NsAvverkning*, *Foryngringsavverkning*, *AvverkningEnergived*, *OvrigAvverkning*). Finally one of the logging subtypes has yet another subtype, *AvverkningVagLedn*.



```
ATGARD
{
        _id: ObjectId,
        id: Integer,
        shape: GeoJSON,
        1: String,
        2: String,
        3: String,
        4: String,
        5: Integer,
        6: Integer,
        7: Integer,
        8: Integer,
        planeringsplan: {
                planeradDatum: Date,
                9: Date,
                10: Date,
                11: Date,
                12: Date,
                13: Date,
                14: String,
                15: String
        },
        produktionsplan: {
                15: Date,
                16: Date,
                17: Date,
                18: String,
                19: String,
                20: String
        },
        atgardskedja: {
                21: String,
                22: Integer,
                23: Integer
        },
        grupperingsforslag: {
                24: String,
                25: String
        },
        drivningsInfoReferences: [ObjectId],
        atgardType: String,
        26: Integer,
        27: Integer,
        28: Double,
        29: Boolean,
        30: Boolean,
        avverkningsanmalan: {
                57: Date,
                58: Date,
                59: Date,
                60: Date,
                61: Boolean
        },
        mattillfalleReferences: [ObjectId],
        avverkningType: String,
        ovrigAvverkningType: String,
        31: Integer,
        32: Integer,
        langd: Integer
}
```

```
DRIVNINGSINFO
{
        _id: ObjectId,
        shape: GeoJSON
        51: Integer,
        52: Integer,
        53: Integer,
        54: Integer,
        55: String,
        56: String
}
```

```
MATTILLFALLE
{
        _id: ObjectId,
        33: Integer,
        34: Integer,
        35: Date,
        mattillfalleTrad: [{
                36: Integer,
                37: Integer,
                38: Integer,
                39: Integer,
                40: Integer,
                41: Integer,
                42: Integer,
                43: Double,
                44: Double,
                45: Double,
                46: Double
                tradUttag: {
                        47: Double,
                        48: Double,
                        49: Double,
                        50: Double
                }
        }]
}
```

Figure 4.2: *The Document representation of the dataset. The same name simplification schema has been applied as in the relational model. Fields central to the benchmark have been highlighted in red. Fields specific to the document representation have been highlighted in green.*

Two distinct types of hierarchies can be identified in the dataset. The first one is the overall tree-like shape of the data. At the root is a remedy, which has several children, which in turn has children etc. The other kind of hierarchy is the remedy subtyping, resulting in an inheritance hierarchy. Both of these hierarchies can be modelled in several ways, described in the theoretical chapter. The preferred way to model them differs between the relational and document paradigms.

The model chosen for the relational database was constructed to normalize data as much as possible. Self-contained groups of data were broken out to form child tables, linked using the Atgard id as foreign key. This way the child tables only need to instantiate rows for an Atgard that defines data related to that child table. The child tables also prevent the Atgard table from growing into hundreds of columns, which would lead to a large number of NULL values.

The inheritance hierarchy in the dataset is structured in such a way that no useful abstract types can be created. The class table inheritance and concrete table inheritance approaches described in the theoretical chapter will therefore give the same result. This result is what was used in the relational representation. The other option described in the theoretical chapter is single table inheritance, which creates a single table including all columns from all subtypes. This would lead to a large number of NULL values for each Atgard row, since many columns are not applicable to most rows. Single table inheritance is therefore not used.

The model chosen for the document database is based on Jia's algorithm and MongoDB's migration guide, both introduced in the theoretical chapter. Instead of implementing Jia's algorithm in software however, it was used as a guideline when converting. The choice of Jia's algorithm was made somewhat arbitrarily and could have been replaced. It is however a straightforward algorithm and produces models that adhere to general guidelines for document databases.

Starting from the relational representation described earlier in this section, Jia's algorithm can be applied as follows. All tables are marked with FREQUENT JOIN, since the entire hierarchy will be retrieved together in some queries in the benchmark. Since modifications will not be the primary concern of the benchmark, none of the tables are marked with FREQUENT MODIFY or FREQUENT INSERT. As it is now, none of the tables are deemed to have a BIG SIZE.

Using the tags, we can begin by embedding all *1:1* relationships into

Atgard, except the ones that are part of the inheritance. *TradUttag* can also be embedded into *MattillfalleTrad* which can in turn be embedded into *Mattillfalle*. Mattillfalle has now become rather large and is therefore given the BIG SIZE tag. This tag indicates that Mattillfalle should be referenced instead of embedded to avoid creating documents that are too large. Since *KopplatDrivningsInfo* is only used to represent an *N:M* relationship it is omitted from the model. *DrivningsInfo* is instead referenced as prescribed by the algorithm. Finally, all fields from the inheritance hierarchy are embedded into the Atgard document. This can be performed without storing NULL values due to the schema-less nature of document databases. When embedded into Atgard the hierarchy can be further optimized by only using flat values to represent the subtypes. This results in the green fields in fig. 4.2. Note that of the fields in the representation, each document will only define those relevant to its data record. All in all the algorithm results in the three document collections Atgard, DrivningsInfo and Mattillfalle shown in fig. 4.2.

Last of all the databases were given indexes on all properties that could be beneficial in the benchmark. For the relational database, this meant all id columns and those marked with red in fig. 4.1. Clustering indexes were placed on the id columns. For the document database, indexes were placed on the _id fields and the id field in Atgard. They were also placed on all fields marked with red in fig. 4.2.

4.4 Benchmarking

The main goal of the study is to find out how the document and the relational paradigm compare in terms of latency and throughput when storing hierarchical geospatial data. With the data models completed and paradigm representatives chosen, only the performance measurement remains. One way to measure performance is through the use of a benchmark suite. However, as found in the theoretical chapter, there is no straightforward choice available for this particular study. While Jackpine and HSR are established benchmarks for geospatial data, they are constructed with only RDBMSs in mind. They rely on several geospatial operations not yet present in document DBMSs and only implement RDBMS interfaces. YCSB on the other hand, being the defacto standard for NoSQL benchmarking, provides no geospatial support. Its implementation also does not support RDBMSs as of today.

To solve the benchmarking problem, there are two possible approaches. Either extend one of the mentioned benchmarks to provide the correct support, or write one from scratch. For this study, it was deemed most feasible to implement one from scratch, as the modifications needed on the other benchmarks are substantial.

4.4.1 The Constructed Benchmark

To make sure the constructed benchmark yields reliable results, it has taken its main design ideas from Jackpine, HSR and YCSB. It is also constructed with Huppler's three characteristics of a good benchmark in mind.

The basic framework consists of a C# main thread that starts query tasks in asynchronous threads. The query tasks join a connection pool towards the desired database and proceed to start querying that database a set number of times. Each query is timed and in the end all result are reported to the main thread. The number of tasks can be varied in order to increase the load beyond what you could get with a single thread. One or two tasks can be used to measure the latency of the DBMS under a moderate load. The number of tasks can then be increased until the total throughput no longer rises, in order to find the maximum throughput. This architecture closely resembles that of YCSB.

Before starting the timer, each task will perform a number of warm-up queries. This ensures necessary parts of the database are loaded into memory and so gives a fair assessment. This idea is used in both YCSB and Jackpine. A set of queries were defined to test different properties of the paradigms. These queries were largely based on those present in HSR and Jackpine, with non-applicable ones excluded and a few new ones added. The added queries center around the hierarchical and sparse aspects of the data. The query set for the benchmark consists of:

- **Atgard by Id** - Retrieve an entire Atgard with all of its children. A random Atgard Id is used as key. This query tests how fast the DBMS can puzzle together all parts of the hierarchy.

- **Batch Mattillfalle by Atgard** - Retrieve all Mattillfalle linked to 100 random Atgard instances. A random range of 100 consecutive Atgard Ids is used as key. This query tests the performance of *1:N* relationships using foreign keys vs a mix of embeddings and references.
- **Count Atgard Complex** - Count all Atgard matching a constraint on the properties *Langd* and *PlaneradDatum*. Random values for the two properties are used as keys. This query tests the database's ability to filter documents on two sparsely populated properties at once. Note that Langd is situated 3 levels down in the relational inheritance hierarchy. Also note that only about 0.5% of all Atgard are of the subtype in which Langd is defined.
- **Count Within Atgard** - Count the number of DrivningsInfo geographically situated within the geometry of an Atgard. The Atgard is chosen randomly by Id. This query tests the performance of the geospatial within-geometry operation.
- **Count Within Box** - Count the number of Atgard geographically situated within a randomly generated bounding box. The box consists of the rectangle formed by two points randomly placed anywhere in Sweden. This query tests the performance for geospatial within with large bounding boxes.
- **Count Within Box and Langd** - Count the number of Atgard geographically situated within a randomly generated bounding box and matching a constraint on Langd. The box consists of the rectangle formed by two points randomly placed anywhere in Sweden. The value for Langd is chosen at random from the interval of values present in the database. This query tests the performance for large bounding boxes when also filtering the result on geospatial metadata.
- **Count Intersects** - Count the number of Atgard geographically intersecting a randomly generated box. The box consists of the rectangle formed by two points randomly placed anywhere in Sweden. This query tests the performance of the geospatial intersects operation for large geometries.
- **Count Near** - Count the number of Atgard geographically situated

within 10000 meters of a random point in Sweden. This query tests the performance of the geospatial near query.

To fulfill Huppler's criteria, the benchmark suite attempts to cover a wide selection of queries and run each many times to even out any unfairness introduced from randomized parameters. The suite also makes paradigm specific implementations of each query. This is to make sure the queries are implemented in an suitable manner for each paradigm.

Many benchmarks attempt to combine the results of several types of queries into a single scalar. This may be useful for very specific application areas where the proportions of operations are well known. This benchmark does however not attempt such a unified metric. Instead, each query type has its numbers presented individually. This approach, combined with a wide selection of queries, allow the results to be useful for many different application scenarios.

4.4.2 Running the Benchmark

The suite ran on three different database sizes, each differing in size by a factor five. The sizes used were 40k, 200k and 1000k Atgard records. The size variations were used to test performance scaling with database size. For size reference, the average record in the relational database was about 3kB large, indexes included.

Three different kinds of tests were performed. The first kind was latency. For these tests, a single reading task and no writing tasks were used. The task was executed for 20 minutes for each of the query types on each paradigm representative. This process was repeated for all three database sizes.

The second kind of test was throughput. For these tests, the number of reading threads was chosen to give as high throughput as possible. The number varied for each combination of parameters and was found through trial and error. The measured execution ran for 20 minutes per query type on each paradigm representative. The process was repeated for all three database sizes.

The third kind of test was read throughput during simultaneous writes. Since different applications can have varying ratios of read and write operations, several ratios were covered. The covered ratios of reads to writes were 0, 0.5, 1 and 2 respectively for each round. The number of total threads used in a run was determined the same way as in the previous test. The measured executions ran for 20 minutes per query type on each paradigm representative. Only the 1000k record database was used for these tests, as it would have taken too much time otherwise. The writing threads used a simple write operation designed to give changes on all properties used by queries in the benchmark. Only the reading threads were timed.

At the beginning of each measured execution, 100 warm-up queries were performed. This number was set arbitrarily, but experimenting indicated that a higher number gave no noticeable difference. Before starting the benchmarking process, the relational DBMS was given an opportunity to generate statistics for query optimization. MongoDB performs all such optimizations automatically and was therefore not subject to this.

All tests ran on a single machine. Below are the most relevant specifications for that machine.

- Processor: Intel(R) Core i7-4800MQ CPU @2.70GHz
- Secondary Memory: SAMSUNG SSD SM841 2.5″ 256GB
- RAM Size: 16 GB
- OS: 64-bit Windows 10

Chapter 5

Results

This chapter presents the results obtained from running the benchmark as described in the previous chapter. The results consist of 24 charts, gathered into three groups of eight. Each chart consists of two curves, showing the respective results of MongoDB and Microsoft SQL Server (MSSQLS).

The first group of charts consists of figs. 5.1-5.8. This group shows the average read latency for each of the eight queries in the benchmark. The latency has been measured for three different database sizes, labeled *S*, *M* and *L*. These labels correspond to having 40k, 200k and 1000k records respectively.

The second group of charts consists of figs. 5.9-5.16. This group shows the average read throughput for each of the eight queries in the benchmark. The throughput has been measured for the same database sizes as described in the previous paragraph.

The third group of charts consists of figs. 5.17-5.24. This group shows the average read throughput for each of the eight queries in the benchmark, while simultaneously experiencing writes. For these charts the database size is fixed at *L*. Instead, the varying parameter is the write-to-read ratio. The ratios used were 0, 0.5, 1 and 2 respectively. Write operations are not part of the throughput metric and only serve to stress the database.

The rest of this chapter consists of figures. Note that all charts use a logarithmic scale on the vertical axis. Also note that the vertical axis has been fitted to the data, meaning the axis does not start at zero. A discussion of the results is carried out in the next chapter.

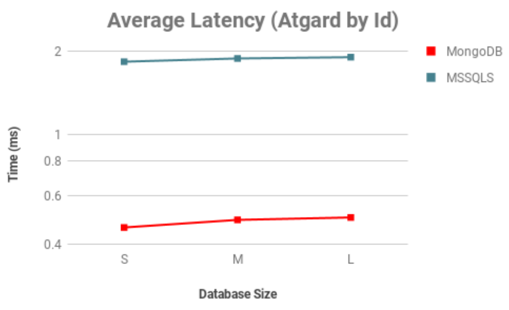


Figure 5.1: *Atgard by Id.*

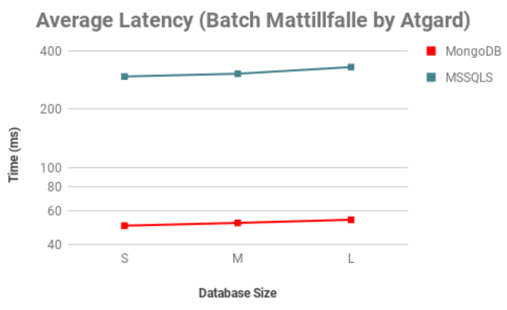


Figure 5.2: *Batch Mattillfalle by Atgard.*

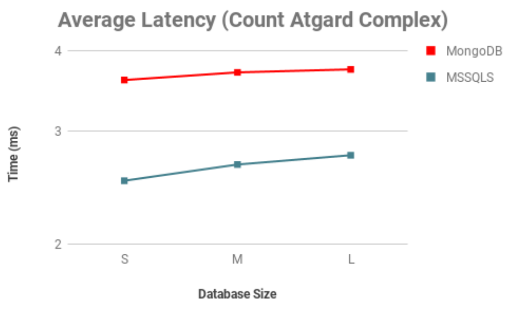


Figure 5.3: *Count Atgard Complex.*

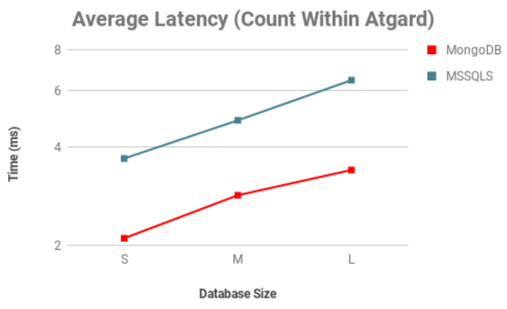


Figure 5.4: *Count Within Atgard.*

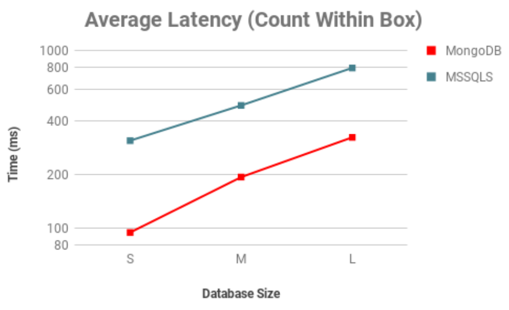


Figure 5.5: *Count Within Box.*

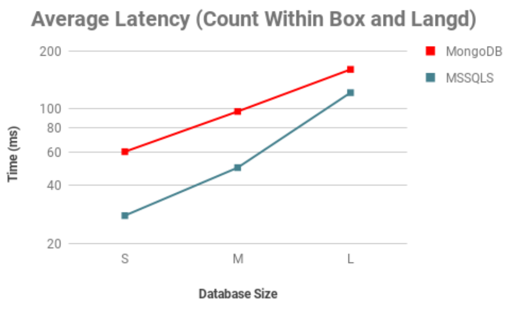


Figure 5.6: *Count Within Box and Langd.*

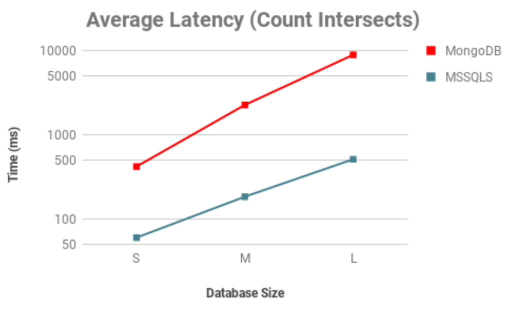


Figure 5.7: *Count Intersects.*

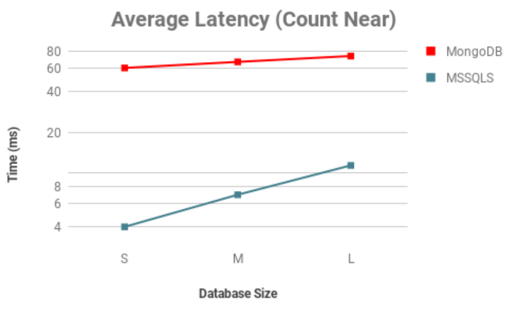


Figure 5.8: *Count Near.*

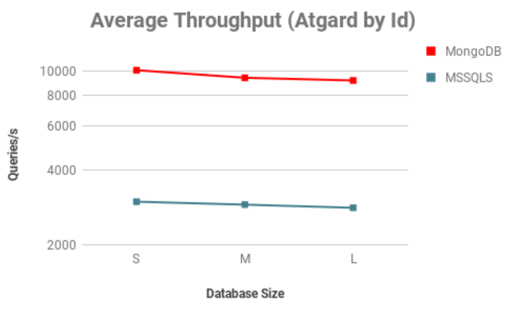


Figure 5.9: *Atgard by Id.*

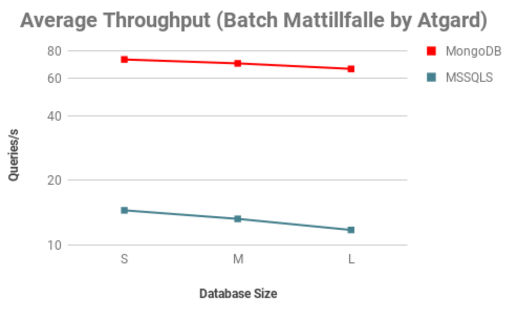


Figure 5.10: *Batch Mattillfalle by Atgard.*

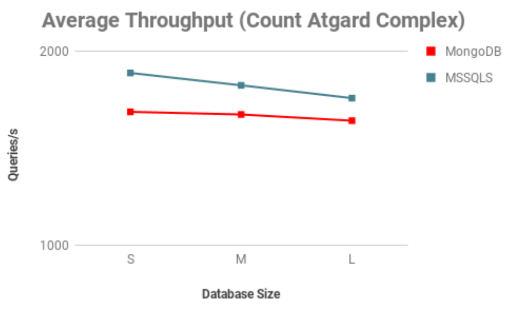


Figure 5.11: *Count Atgard Complex.*

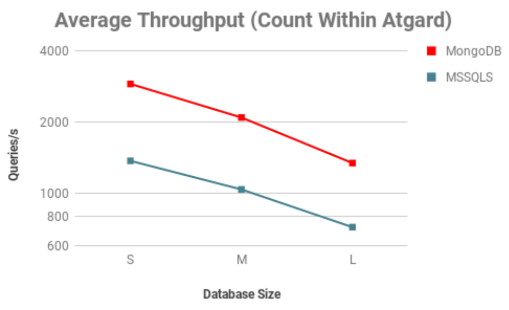


Figure 5.12: *Count Within Atgard.*

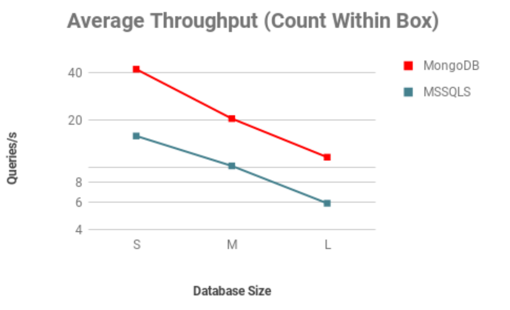


Figure 5.13: *Count Within Box.*

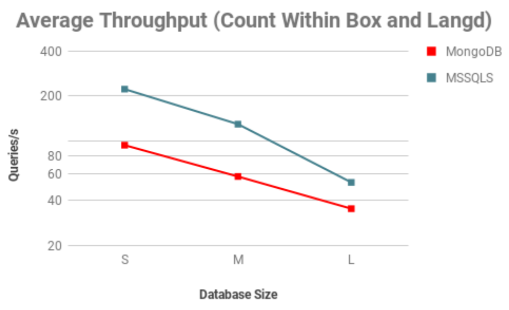


Figure 5.14: *Count Within Box and Langd.*

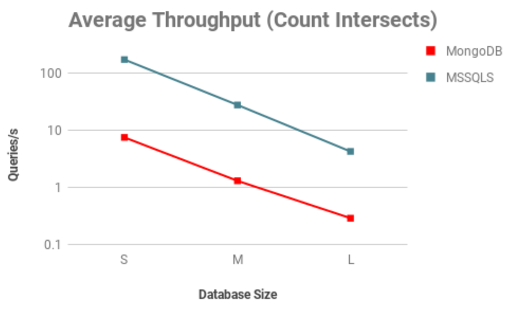


Figure 5.15: *Count Intersects.*

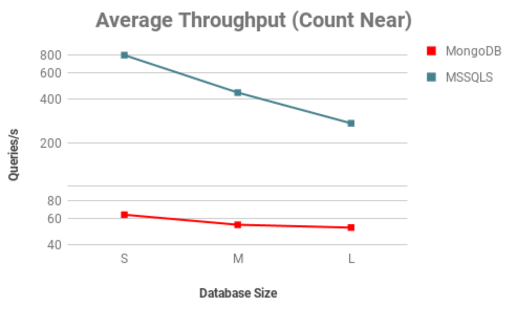


Figure 5.16: *Count Near.*

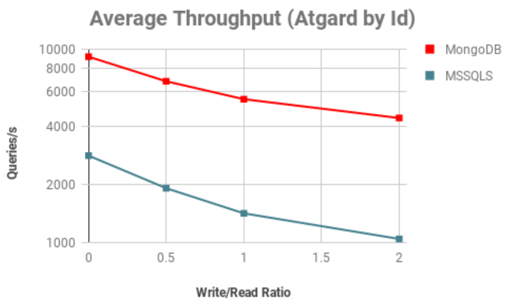


Figure 5.17: *Atgard by Id.*

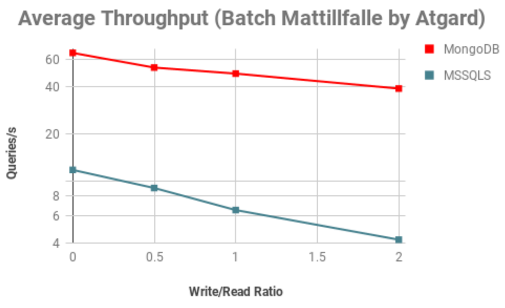


Figure 5.18: *Batch Mattillfalle by Atgard.*

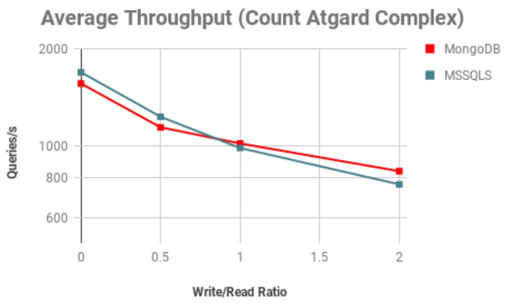


Figure 5.19: *Count Atgard Complex.*

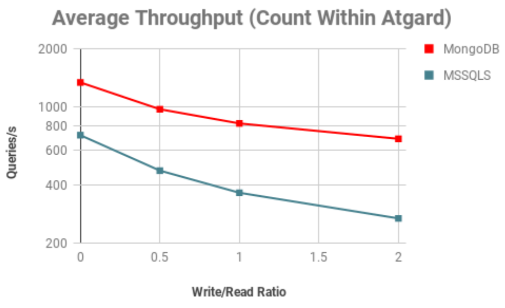


Figure 5.20: *Count Within Atgard.*

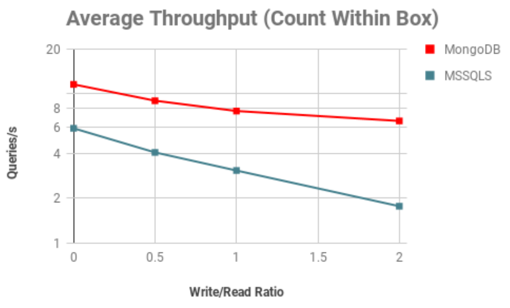


Figure 5.21: *Count Within Box.*

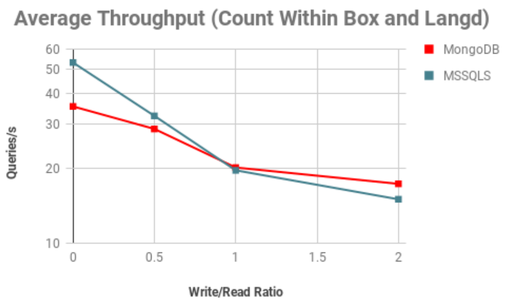


Figure 5.22: *Count Within Box and Langd.*

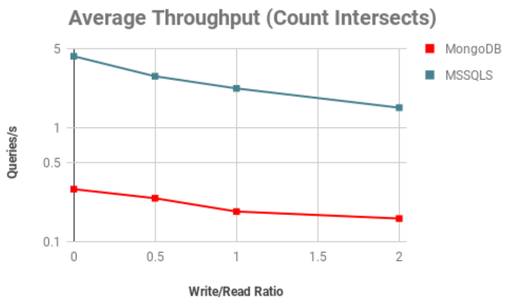


Figure 5.23: *Count Intersects.*

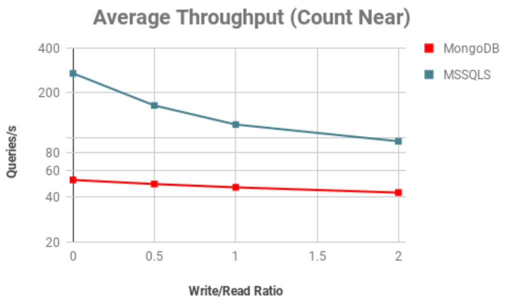


Figure 5.24: *Count Near.*

Chapter 6

Conclusion

This chapter discusses the obtained results, relating them to the theoretical chapter and the hypothesis. Towards the end, the research question is addressed and further studies are proposed.

6.1 Discussion

As hypothesized in the method chapter, throughput and latency behaved similarly throughout the benchmark. Comparing figs. 5.1-5.8 with 5.9-5.16, this observation is made evident. Given this result, the discussion can focus on throughput only (figs. 5.9-5.24) without loss of generality.

Looking at specific queries, it can be seen that the document paradigm achieves approximately three times higher throughput for *Atgard by Id* (fig. 5.9). This query consists of finding and collating all parts of an Atgard, spread over the hierarchy. The result falls in line with the second hypothesis, which predicted that the document paradigm would be faster for this type of query. As stated in the hypothesis, a reasonable explanation is that document embedding reduces the number of operations needed to collate an Atgard, giving the document paradigm an advantage. The results from this query are of particular importance, as hierarchy collation is a central part of the study.

Looking at *Batch Mattillfalle by Atgard* (fig. 5.10), the document paradigm achieves approximately four times higher throughput. This query consists of retrieving child entities in a *1:N* relationship. The result points towards the document paradigm being stronger here as well. It was hypothesized that these types of relationships, where embedding and referencing are mixed, would perform about equally well for both paradigms. This was however not the case here. A possible explanation is that the document version of the relationship only required one reference per entity, with the rest being modelled with embedding. The disadvantages from referencing might therefore have been outweighed by the advantages from embedding.

Count Atgard Complex (fig. 5.11) shows some interesting results. This query counts the number of Atgard having specific values for two sparsely populated properties far out in the hierarchy. This is again a situation where this study hypothesized that document embedding would outperform relational JOINs. However, the results from this query show a slight advantage in favor of the relational paradigm. A possible explanation for this unexpected result is that the document paradigm saw a disadvantage from its documents generally being much larger than individual table rows. To fully understand this speculation, consider that the query filters records on two properties. While these properties were both indexed, they did not reside in the same table/sub-document. The database therefore had to retrieve and correlate records based on the index hits. In the case of the document paradigm, this meant retrieving many large documents. The relational paradigm however, would only need to retrieve the relatively small rows needed to properly related the two index hits. The relational database model would therefore seem to have an important advantage that was not properly considered in the hypothesis.

Looking at the read-only spatial queries, there are some mixed results. The document paradigm performs better on two of the geometry-oriented queries (figs. 5.12 & 5.13). However, the relational paradigm performs better on the other two (figs. 5.15 & 5.16). This result falls more or less in line with the third hypothesis, which stated that none of the paradigms have any apparent reason to be faster at simple geometry reads. The fifth geospatial query (fig. 5.14) combines geometry and metadata querying. This type of query was hypothesized to favor the document paradigm. However, the query saw a higher throughput from the relational database. Due to the similarities with the query in fig. 5.11, the results can be explained similarly.

Finally, looking at the write influenced queries (figs. 5.17-5.24), a clear picture is painted. It was hypothesized that the document paradigm would suffer less throughput degradation during simultaneous write operations. This is precisely what the results seem to indicate. Figs. 5.18, 5.21 & 5.24 in particular show considerably stronger results for the document paradigm. To a lesser extent, this can also be seen in figures 5.19, 5.20 & 5.22. The hypothesis provided a reasonable explanation for this, revolving around the different consistency models employed by the paradigms. It stated that because the relational paradigm adheres to ACID, it cannot easily parallelize read-write mixes of operation to the same extent as a BASE-adhering paradigm. In the method chapter, it was also hypothesized that geospatial queries would be impacted more than other query types when adding writes to the relational paradigm. However, the results show no compelling evidence of that being the case.

6.2 Addressing the Research Question

The research question is once again recalled:

> *"How do relational and document databases compare in terms of throughput and latency when storing hierarchically structured geospatial data on a single node?"*.

Altogether, the document paradigm showed potential when querying hierarchically structured data alone. The paradigm saw superior performance for all such queries, except when querying on multiple, sparsely populated properties. With geospatial data added, no decisive conclusions could be drawn for read-only queries. However, when adding simultaneous write operations, the document paradigm saw less performance degradation than the relational paradigm did for all tested queries.

An important thing to remember in this study though, is that the DBMSs' performances are not only dependent on their underlying paradigm. While all DBMSs in the same paradigm have key design points in common, they also make DBMS specific design choices. As such, the results need to be interpreted as indications about the paradigms' relative performances, rather than conclusive evidence. With that in mind, the document paradigm has shown reasonable potential when working with hierarchical data, but is in need of further investigation when adding geospatial data to the mix.

6.3 Future Work

To further explore optimal databases for hierarchical geospatial data, a few areas of study are suggested. First of all, a study focused solely on geospatial data might lead to more decisive results than presented in this study. It could also be interesting to have a deeper look into how hierarchies should be modelled for optimal performance in a document database. A more extensive benchmark, perhaps including several DBMSs from each paradigm, could also be of interest. Finally, it could be fruitful to perform a similar study on the graph paradigm, as that paradigm also showed potential in the background chapter.

Bibliography

[1] V. Abramova and J. Bernardino. “NoSQL Databases: MongoDB vs Cassandra”. In: *ACM Conference C3S2E* (2013).

[2] S. Agarwal and Rajan K.S. “Analyzing the performance of NoSQL vs. SQL databases for Spatial and Aggregate queries”. In: *FOSS4G Conference Proceedings* 17 (2017).

[3] ArangoDB. *Geospatial Queries*. 2018. URL: `https://docs.arangodb.com/3.2/AQL/Tutorial/Geospatial.html` (visited on 03/01/2018).

[4] R. Cattel. “Scalable SQL and NoSQL Data Stores”. In: *ACM SIGMOD* 30, No 4 (2010).

[5] E. F. Codd. “A Relational Model of Data for Large Shared Data Banks”. In: *Communication of the ACM* 13 (1970).

[6] B.F. Cooper et al. “Benchmarking Cloud Serving Systems with YCSB”. In: *Yahoo! Research* (2010).

[7] Couchbase. *Querying Spatial Views*. 2018. URL: `https://developer.couchbase.com/documentation/server/4.0/views/sv-query-parameters.html` (visited on 03/01/2018).

[8] G2 Crowd. *Best Relational Databases Software*. 2018. URL: `https://www.g2crowd.com/categories/relational-databases?segment=all` (visited on 04/10/2018).

[9] C.J. Date. *Database Design and Relational Theory - Normal Forms and All That Jazz*. O’Reilly, 2012.

[10] M.J. Egenhofer and R. Franzosa. “Point-set topological spatial relations”. In: *Intl Journal of Geographic Information Systems* 5, No 2. (1991).

[11] *FOSS4G Benchmark*. 2013. URL: `https://wiki.osgeo.org/wiki/FOSS4G_Benchmark` (visited on 03/30/2018).

[12] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison Wesley, 2002.

[13] M. Gentz and A. Ramachandran. *Working with geospatial and GeoJSON location data in Azure Cosmos DB*. 2018. URL: `https://docs.microsoft.com/en-us/azure/cosmos-db/geospatial` (visited on 03/01/2018).

[14] S. Gilbert and N. Lynch. "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services". In: *ACM SIGACT News* 33 (2002).

[15] M. L. Gonzales. "Seeking spatial intelligence". In: *InformationWeek* 2, No 2. (2000).

[16] MongoDB Inc. *Documents*. 2018. URL: `https://docs.mongodb.com/manual/core/databases-and-collections/` (visited on 03/01/2018).

[17] MongoDB Inc. *Geospatial Queries*. 2018. URL: `https://docs.mongodb.com/manual/geospatial-queries/` (visited on 03/01/2018).

[18] MongoDB Inc. *Indexes*. 2018. URL: `https://docs.mongodb.com/manual/indexes/` (visited on 03/01/2018).

[19] MongoDB Inc. *JSON and BSON*. 2018. URL: `https://www.mongodb.com/json-and-bson` (visited on 03/01/2018).

[20] MongoDB Inc. *RDBMS to MongoDB Migration Guide - A MongoDB Whitepaper*. MongoDB Inc., 2017.

[21] Neo4j Inc. *What is a Graph Database?* 2018. URL: `https://neo4j.com/developer/graph-database/` (visited on 02/28/2018).

[22] solid IT gmbh. *DB-Engines Ranking*. 2018. URL: `https://db-engines.com/en/ranking` (visited on 03/01/2018).

[23] S. Jablonski and R. Hecht. "NoSQL Evaluation". In: *International Conference on Cloud and Service Computing* (2011).

[24] T. Jia et al. "Model Transformation and Data Migration from Relational Database to MongoDB". In: *IEEE International Congress on Big Data* (2016).

[25] Microsoft. *Geometry Methods Supported by Spatial Indexes*. 2018. URL: `https://docs.microsoft.com/en-us/previous-versions/sql/sql-server-2008-r2/bb895373(v=sql.105)` (visited on 04/10/2018).

[26] Microsoft. *Spatial Indexing Overview*. 2018. URL: `https://docs.microsoft.com/en-us/previous-versions/sql/sql-server-2008-r2/bb964712(v=sql.105)` (visited on 04/10/2018).

[27] A.S.P. Murugan. "A Study of NoSQL and NewSQL databases for data aggregation on Big Data". In: *DiVA* (2013).

[28] R. Nambiar and M. Poess. *Performance Evaluation and Benchmarking*. Springer, 2009.

[29] Z. Parker, S. Poe, and S.V. Vrbsky. "Comparing NoSQL MongoDB to an SQL DB". In: *ACMSE'13* (2013).

[30] S. Ray, B. Simion, and A.D. Brown. "Jackpine: A Benchmark to Evaluate Spatial Database Performance". In: *ICDE IEEE* (2011).

[31] RethinkDB. *Geospatial queries*. 2018. URL: `https://www.rethinkdb.com/docs/geo-support/ruby/` (visited on 03/01/2018).

[32] S. Schmid, E. Galicz, and W. Reinhardt. "Performance investigation of selected SQL and NoSQL databases". In: *AGILE 15 Lisbon* (2015).

[33] L. Stanescu, M. Brezovan, and D.D. Burdescu. "Automatic Mapping of MySQL Databases to NoSQL MongoDB". In: *Proceedings of the Federated Conference on Computer Science and Information Systems* 8 (2016).

[34] C. Strozzi. *NoSQL: A Relational Database Management System*. 2010. URL: `http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/nosql/Home%20Page` (visited on 02/27/2018).

[35] Hochschule für Technik Rapperswil. *HSR Texas Geo Database Benchmark*. 2014. URL: `https://giswiki.hsr.ch/HSR_Texas_Geo_Database_Benchmark` (visited on 03/30/2018).

[36] TPC. *Active TPC Benchmarks*. 2018. URL: `http://www.tpc.org/information/benchmarks.asp` (visited on 03/30/2018).

[37] X. Zhang, W. Song, and L. Liu. "An Implementation Approach to Store GIS Spatial Data on NoSQL Database". In: *22nd International Conference on Geoinformatics* (2014).