Master's thesis

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

Sondre Olav Dahl
Christoffer Synnestvedt

Evaluating a Flattened R-tree in MongoDB

Master's thesis in Computer Science
Supervisor: Svein Erik Bratsberg
June 2023

NTNU
Norwegian University of
Science and Technology

Sondre Olav Dahl
Christoffer Synnestvedt

Evaluating a Flattened R-tree in MongoDB

Master's thesis in Computer Science
Supervisor: Svein Erik Bratsberg
June 2023

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

NTNU
Norwegian University of
Science and Technology

Abstract

This thesis explores the development and evaluation of a flattened R-tree indexing structure for MongoDB. MongoDB, a leading NoSQL database, has been extensively used for big data applications, but its limitations in handling spatial data provoked the idea of implementing an R-tree index that could enhance performance in spatial querying.

The research began with a comprehensive literature review to gain a profound understanding of MongoDB's current indexing capabilities, R-tree structures, and the potential advantages of integrating the latter within the MongoDB environment. A previous attempt at implementing an R-tree alongside MongoDB was found and it served as the base for the implementation presented in this thesis. The implemented R-tree index's query operations were tested with four different data sets. Tests were primarily designed to measure and compare the performance of the newly implemented index against MongoDB's existing geospatial indexing structure.

Results show that the implemented R-tree does not outperform the existing indexing structure in MongoDB. However, the R-tree showed promise under certain query conditions, indicating that an R-tree index can perform better than the current 2dsphere index in MongoDB if implemented properly. The study emphasizes that the implementation of the R-tree is the most significant cause of the results and that with an improved implementation, the R-tree could be an enriching addition to MongoDB. While the implementation has several shortcomings, it improves MongoDB's handling of Cartesian coordinates. This could potentially improve big data processing and applications that demand high-performance spatial querying. The thesis concludes with propositions for further work aiming to improve the R-tree implementation.

Sammendrag

Denne oppgaven utforsker utviklingen og evalueringen av en flat R-tre-indeksstruktur for MongoDB. MongoDB, en ledende NoSQL-database, har blitt mye brukt for stordataapplikasjoner, men dens begrensninger når det gjelder håndtering av flerdimensjonal data ga ideen om å implementere en R-tre-indeks som kunne forbedre ytelsen ved flerdimensjonale spørringer.

Forskningen startet med en omfattende litteraturgjennomgang for å få en dyp forståelse av MongoDBs nåværende indekser, R-tre-strukturer, og de potensielle fordelene ved å integrere sistnevnte i MongoDB-miljøet. Et tidligere forsøk på å implementere et R-tre i MongoDB ble funnet og det ble brukt som et grunnlag for implementeringen presentert i denne oppgaven. Spørringene til den implementerte R-tre-indeksen ble testet med fire forskjellige datasett. Testene var primært utformet for å måle og sammenligne ytelsen til den nylig implementerte indeksen mot MongoDBs eksisterende indekseringsstruktur.

Resultatene viser at det implementerte R-treet ikke utkonkurrerer den eksisterende indekseringsstrukturen i MongoDB. Imidlertid viste R-treet lovende resultater under visse forhold, noe som indikerer at en R-tre-indeks kan prestere bedre enn den nåværende 2dsphere indeksen til MongoDB, hvis den blir implementert riktig. Studien understreker at implementasjonen av R-treet er den viktigste årsaken til resultatene, og at med en forbedret implementasjon kunne R-treet vært et gunstig tillegg til MongoDB. Selv om implementasjonen er mangelfull, så forbedrer den MongoDB sin håndtering av kartesiske koordinater Dette kan potensielt forbedre stordataprosessering og applikasjoner som har høye krav til prosesseringstid av romlige sprørringer. Oppgaven avslutter med forslag til videre arbeid som har som mål å forbedre R-tre-implementasjon.

Table of Contents

List of Figures

List of Tables

List of Listings

1 Introduction

The recent decades have seen a rise of massive spatial data sets from different sources [1]. With the rise of big spatial data, the need for spatial processing tools capable of handling massive data sets has become clear. Multiple surveys highlight the increase of research on big spatial data processing as traditional systems are not designed for handling big data [1]–[3]. There have been various research attempts aimed at improving the popular big data framework, Hadoop [4], to better handle spatial queries and analysis [5]–[7]. Other research attempts extend NoSQL databases such as MongoDB [8] and HBase [9] with spatial indexing structures to improve the processing of spatial data [10], [11]. Despite progress in using NoSQL databases for spatial data, the PostGIS [12] extension for PostgreSQL [13] still remains on top in usage according to a ranking by DBEngine [14].

PostGIS offers more spatial queries than available in the NoSQL databases and provides a multi-dimensional index for spatial data, the R-tree [15]. The R-tree was introduced by Guttman in 1984 [16]. It has received much attention in research and multiple variants have been proposed that improve the original version, such as the R+-tree [17] and the R*-tree [18]. A survey by Guo and Onstein on geospatial information processing in NoSQL shows that only Neo4j utilizes the R-tree index among the most popular NoSQL databases [19]. However, comparisons between PostGIS and MongoDB show that MongoDB can deliver better performance than PostGIS despite the lack of a multi-dimensional index [20]–[22]. MongoDB's spatial indexes transform multi-dimensional spatial data into one dimension and thus perform well on simple spatial data such as lines or points. On more advanced structures such as polygons, the calculations become more of a bottleneck as the complexity of the data increases. Another issue with spatial data in MongoDB is the lack of functionality when compared to the many queries available in PostGIS. Also, the support for Cartesian coordinates is lacking with even fewer functions and spatial data types available than for geodetic coordinates.

Despite the current challenges with spatial data in MongoDB, it has been shown capable of delivering better performance on larger data sets than the current state of the art [22]. The promising performance motivates further research in spatial processing improvements to the database system. In this thesis, we therefore aim to analyze the performance of the flattened R-tree implemented by Xiang et al. [10] in order to see how MongoDB can be improved with a multi-dimensional indexing structure. To see how the R-tree structure can extend the functionality of MongoDB, it will be tested with Cartesian data sets of both complex and simple geometries.

1.1 Purpose

Traditional relational databases are not designed for big data processing and thus NoSQL fits the task better. However, current NoSQL databases are not able to provide support for spatial data at the same level as their relational counterparts. The purpose of this thesis is thus to analyze how MongoDB, a NoSQL database, can be improved with a flattened R-tree implementation to provide better support for spatial data.

1.2 Research Goals

The following goals are obtained from the purpose of this thesis:

- Implement a flattened R-tree in MongoDB version 6
- Compare the performance of the index to MongoDB's current 2dsphere index
- Evaluate the results and the potential of an R-tree index in MongoDB

1.3 Structure of the Thesis

The rest of the thesis is divided into five sections. Section 2 covers the theoretical background required for the implementation. Section 3 gives an overview of the implementation and the development process. Section 4 presents the results achieved with the Flattened R-tree and MongoDB's 2dsphere index. This section also discusses the possible reasons for the processing times reached. Section 5 gives an overview of the main findings of the research, limitations of the study, implications, and final remarks. Lastly, Section 6 suggests further work that can be done to improve the implementation.

2 Background

This section will serve as an introduction to the theoretical aspects related to multi-dimensional index structures, NoSQL databases, and Geographical Information Systems. It will also cover a selection of related works.

2.1 Multi-Dimensional Index Structures

In order to understand what a multi-dimensional index structure is, one must first understand what an index is. An index in the view of a database system is a supportive data structure providing faster look-ups with given search conditions [23]. When data is written to a database, it will store the data in blocks on the computer's disk. In order to avoid looking through the records linearly to find a database record, which is a slow process, an index is necessary, although it adds overhead when writing data as an index record must be written along with the object. The B-tree is the most known index for databases and it has received much attention in research papers. It is a one-dimensional index where the leaf nodes of the tree contain pointers to objects on the disk. As the B-tree is one-dimensional, it cannot properly index on multiple dimensions unless the data is converted to one dimension for indexing. Employing hash functions on multi-dimensional data to retain a single hash value for indexing serves as an example of this. However, there are separate index structures designed for retaining the dimensions of the data as the next sections will show.

2.1.1 R-tree

The R-tree was introduced in 1984 by Guttman [16]. It is a height-balanced tree and each leaf node contains index records with pointers to data objects just like the B-tree. The non-leaf nodes of the tree contain entries that point to a child. Each non-leaf node entry is of the form *(I, child-node)* whereas each child node contains entries of the form *(I, tuple-identifier)*. A *tuple-identifier* is an identifier for a matching tuple in the database. The I represents the n-dimensional minimum bounding rectangle (MBR) of the entry, which is of the form:

$$I = I_0, I_1, ..., I_{n-1}$$

Each I represents a close-bounded interval *[a, b]* telling the extent of the object along axis i. If either a, b, or both equal infinity, this means that the object stretches out indefinitely in the given direction. An entry in a leaf node holds the MBR for the object it refers to, and an entry in a non-leaf node contains the MBR enclosing all MBRs of its children.

An R-tree, where M is the maximum number of entries that fit in one node and $m \leq \frac{M}{2}$ specifies the minimum number of entries in a node, must satisfy the properties formalized by Guttman [16]:

1. Every leaf node contains between m and M index records unless it is the root.
2. For each index record *(I, tuple-identifier)* in a leaf node, I is the smallest rectangle that spatially contains the n-dimensional data object represented by the indicated tuple.
3. Every non-leaf node has between m and M children unless it is the root.
4. For each entry *(I, child-pointer)* in a non-leaf node, I is the smallest rectangle that spatially contains the rectangles in the child node.
5. The root node has at least two children unless it is a leaf.
6. All leaves appear on the same level.

An important point to be noted from the properties above is the maintenance of the 4th property when a node-split occurs. Upon inserting new records, a node-split will happen at some point, which will incur the creation of two new MBRs in non-leaf nodes. The R-tree performs splits differently than the B-tree as it considers different criteria. The main goal of the split is to minimize the probability that both of the new nodes will be invoked by the same query. Three different algorithms for splitting are proposed in [16] with differing run times. The fastest one is the linear split which selects two objects as initial seeds for the new nodes and then assigns the remaining objects to the nodes based on how much the nodes need to expand their MBR to include the new object. The smallest possible increase in size is desired when assigning the objects. The exponential split is the slowest one as it uses brute force to find the best possible grouping of the objects with regard to the minimization of the enlargement of the MBR. The suggested algorithm to use is the quadratic split which also selects two nodes as the seed, but differs in how it selects the two seed nodes. The seeds are selected by checking which two objects create the most dead space in their MBR if put together where dead space is the space in an MBR not occupied by the objects in the MBR. After the seeds have been selected, the next object to be inserted in a node is the one maximizing the difference in dead space if assigned to both nodes given that the object is assigned to the node requiring the least MBR enlargement. The quadratic split was suggested as it still achieves reasonable performance while providing a better split than the linear split.

The deletion algorithm is straightforward, but should a node contain less than m entries, where m is the minimum number of entries in a node, the node is removed and all of its entries are reinserted into the tree. In a B-tree, the node would be merged with a sibling due to the nature

of one-dimensional data, but the same property does not hold for multi-dimensional data and thus the reinsertion strategy is applied to distribute the remaining entries. After removing the node, the update is propagated up in the tree towards the root with some MBRs potentially being resized as some entries have been removed.

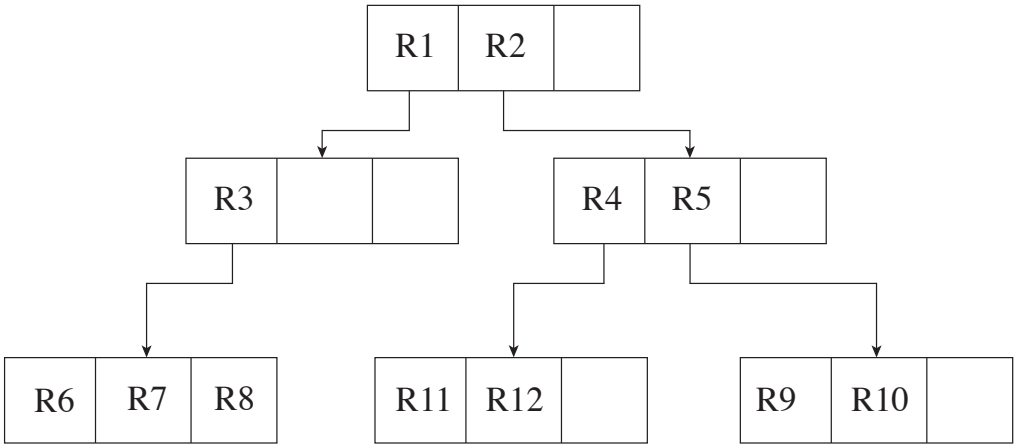
Figure 1 shows an example of a constructed R-tree. In Figure 1b multiple objects are shown along with their constructed MBR. To simplify the example, there are no overlapping MBRs, but note that it is possible for MBRs to overlap each other. In such a case where a user queries for overlap and the MBRs overlap, but not the objects themselves, they will still be returned as candidates to the user who must then verify if the objects truly overlap. The reason for this false alarm can also be seen in the figure since the MBR is not a perfect representation of the polygons and thus create some dead space around the polygons.

Numerous versions of the R-tree have appeared over the years with improvements to the original. We briefly present the R+ and the R* tree here, but refer the interested reader to Manolopoulos et al. [24] to learn more about other variants. The proposal of R+-trees [17] was motivated by the overlap problems in the original R-tree. A point location query in the original version could visit several paths from the root to leaves as the MBRs could be overlapping and the likeliness of multiple paths being visited increases with the amount of overlap found. The R+-tree does not allow MBRs at the same level to overlap and uses clipping to prevent overlap. Clipping introduces redundancy to the tree because the same MBR might be stored under two paths. Both insertion and deletion are affected by this change since multiple copies may be inserted and all of these copies must be deleted when delete is called on the object.

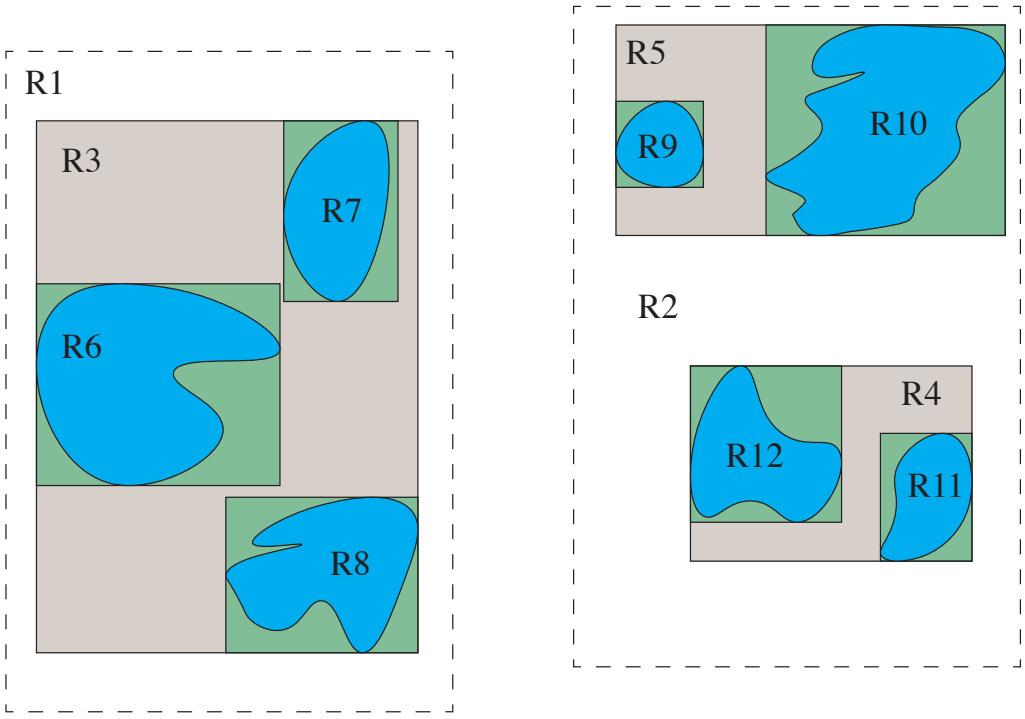
Another variant considered more performant is R*-trees [18]. The R*-tree utilizes an engineering approach to combine the following criteria in the best possible way [24]:

- Minimize the area covered by each MBR
- Minimize overlap between MBRs
- Minimize MBR margins
- Maximize storage utilization

The criteria are conflicting as reducing the area and overlap reduce storage utilization, but through extensive testing Beckmann et al. [18] found the optimal combination. When inserting a new entry, a branch must be selected. The R*-tree algorithm for selecting this branch is ChooseSubtree and it uses different criteria for selecting non-leaf nodes. The minimum MBR area enlargement is considered when selecting non-leaf nodes whereas the minimize overlapping criterion is examined for leaf-nodes as the extensive testing showed a better performance



(a) An R-tree with $M = 3$.



(b) Polygons with their corresponding MBRs.

Figure 1: A 2-dimensional R-tree along with its objects and constructed MBRs.

with these settings. The B+- and B*-tree are both dynamic R-trees, but there are variants of the R-tree made specifically for static data. Manolopoulos et al. [24] goes into more detail on the static versions of the R-tree and the interested reader is recommended to take a look there.

2.1.2 KD-tree

The kd-tree was introduced by Bentley [25] in 1975 as a generic tree structure for multidimensional data. It is a multidimensional binary search tree with the same space and time complexities as a regular binary search tree. Each record is considered a point in a k-dimensional space and the tree is built by partitioning the space along the k different axes. Every level of the tree is associated with an axis. The discriminator can be selected by cycling through the axes. To build the kd-tree, each record is inserted into the tree. The first record will become the root and the rest of the records will be put in either the left or right subtree of the first record. By finding the dimension with the highest spread and finding the median of this dimension, a more balanced split can be produced than using a randomly selected record. An algorithm to build the tree using the median is briefly explained by Skrodzki [26] in his thesis. A detailed explanation of a 2-dimensional tree can be found in chapter 2.3 of [27].

Performing range search on a kd-tree is straightforward. Consider the root to contain the entire search area. The root partitions the search space into two partitions, its left and right subtree. A query rectangle intersecting with the partitioning line of the root may contain points at both sides of the space and thus both subtrees must be searched. If the query rectangle is wholly contained on either side, one branch may be disregarded and the search continues in the other branch. Nodes with a search space that intersect the query rectangle are checked to see if they intersect with the query and added if they do. The search process applied at the root level is applied for the candidate subtrees. If a subtree is completely contained in the query rectangle, the entire subtree is added to the result.

2.2 NoSQL

In recent years, the prices of storage have decreased drastically [28], leading to the emergence of NoSQL (non-SQL or not only SQL) databases. This term covers a wide variety of databases that stores data differently from the traditional relational database. Since storing data became cheaper, there was no longer a need to create complex data models in order to avoid duplication. Developers were also becoming the primary expense of software development, rather than storage, so creating a database that was optimized for developer productivity was favorable.

Another reason for the emergence of NoSQL databases is that the ”one size fits all”-thinking

concerning databases has been questioned by both science and web companies [29]. There are many application scenarios that cannot be addressed with a traditional database approach as the amount of data is increasing and the nature of it is unstructured. Hence, creating a relational database schema is nearly impossible. This is what the different NoSQL databases try to solve, allowing the storage of huge amounts of unstructured data, and giving the developers a lot of flexibility.

As mentioned, there are a wide variety of NoSQL databases and each of them has its own features, but at a high level, many of the databases have the features listed in Table 1.

Table 1: Features of NoSQL Databases [30].

Feature	**Feature Description**
Flexible schemas	Since NoSQL does not require the developer to store data in a defined schema, making changes is easily done. This feature supports the Agile Manifesto [31], as it allows the developers to rapidly adapt the database to changing requirements.
Horizontal scaling	Most NoSQL databases scale out horizontally, meaning that developers can spread the data to different servers when the amount is too large to be handled by one server. This is different from most relational databases that require scaling vertically, which implies moving the data to a bigger server. Hence, scaling vertically is more expensive than scaling horizontally.
Fast queries due to the data model	Data in NoSQL databases are typically stored with regard to optimizing queries. So data that is accessed together is usually stored together. Consequentially, queries do not have to perform join operations, leading to faster queries. This is different from relational databases that usually are normalized, so queries may require performing join operations on data from multiple tables.
Ease of use for developers	Some NoSQL databases maps their data structures to those of popular programming languages. This allows developers to store data in the same way as they use it in their code, which may lead to less code, more effective development, and fewer bugs.

The different NoSQL databases have different ways of storing data. The four major types of databases today are: document databases, key-value stores, wide column stores, and graph databases [32]. Each of these suits different applications, so the choice of database to use depends on what type of data the developer wants to store. A document database stores data in JSON, BSON, or XML documents. Each document is a self-contained unit of data that contains all the information about an object. These are flexible, meaning that fields can be added or removed without affecting other documents in the database. This allows the objects to be in a form close to the ones used in applications, leading to less translation when using the data in an

application compared to relational databases that often have to assemble and disassemble data when retrieving and storing it. Typical use cases for document databases are e-commerce, web applications, and content management. The second type, key-value stores, is the simplest type of NoSQL database. It is similar to a relational database table with only two columns. Data is stored in key-value pairs, where the key represents the attribute name. Use cases for this type of database are shopping carts, user settings, and user profiles. Further, the wide column stores store information in columns, enabling users to access only the specific columns they need. It allows for a flexible and scalable data model, meaning that each row in the database can have a different set of columns, and each column can contain multiple values. This type is commonly used in applications that require fast and scalable data storage and retrieval, such as real-time analytics and time series data. Finally, the graph databases store data in nodes, and the relationship between nodes are stored in elements called links or relationships. As opposed to relational databases which imply these relationships through data, a graph database stores these directly. Hence, the database is optimized for searching and capturing the connections between data elements. Some use cases for this type are fraud detection, social networks, and knowledge graphs.

2.2.1 MongoDB

MongoDB is a widely used, open-source NoSQL database. It is used by companies in a broad range of industries, and as of now, it is around 35,000 that have used MongoDB to build their application [33]. MongoDB stores data as BSON documents composed of field and value pairs. BSON is a binary representation of JSON, however, it contains more data types. A value can have any of the BSON data types [34], but it can also be other documents, arrays, and arrays of documents. Therefore, a lot of different types of data can be stored. These documents are stored together in collections, and a database contains one or more collections. Collections are analogous to tables in relational databases, however, they do not require that the documents within a collection have the same schema by default. This makes it possible to modify the structure of a single document easily, and it does not affect the rest of the documents in the collection. Also, documents map directly to code so it correspond to native data types in many programming languages, giving the opportunity to have the same objects in both the application code and in the database.

Querying data in MongoDB is done through the Query API which consists of CRUD operations and Aggregation pipelines. CRUD stands for Create, Read, Update, and Delete, and the operations are self-explanatory. Aggregation pipelines on the other hand consist of one or more stages that process documents. It works like a pipeline with different stages where each stage performs operations on the input document before sending it through the pipeline to the next

stage. Examples of operations that can be performed are filter documents, group documents and calculate values. Examples of the results from the aggregation pipeline are total, average, maximum, and minimum values. The Query API also supports queries such as document join, graph and geospatial queries, full-text search, on-demand materialized views, and time series analysis. To support efficient query operations, MongoDB uses indexes. The database offers a wide variety of indexes that can be used on any field or sub-field in a document, all of which use the B-tree data structure. There is one default index, the _id index that creates a unique index on the _id field during the creation of a collection. This prevents the creation of multiple documents with the same _id. The different indexes that MongoDB provides, support different types of data and queries. Following is a short description of each index [35].

- *Single Field Index*: Besides the default _id index, MongoDB allows users and applications to create an ascending/descending index on any single field in a document. Listing 1 shows the command for creating a single field index on the field *item*. The number says if the sort order is ascending (1) or descending (-1).

```
db.collection.createIndex({item: 1})
```

Listing 1: Command for creating an index.

- *Compound Index*: A compound index is a single index structure that holds a reference to two or more fields in the documents. The order of the fields in the compound index is of significance as MongoDB will first sort on the first field, then within the first, it will sort on the second. The sort order (i.e. ascending or descending) of the index key can determine whether the index can support certain sort operations. Consider the index depicted in Listing 2. With the sort order on these index keys it will be possible for a

```
{"item": 1, "amount": -1}
```

Listing 2: Compound index.

query to return results that are sorted by ascending *item*, and then descending *amount*, and vice versa. However, the index cannot sort on ascending *item* and ascending *amount*.

```
{"item": 1, "amount": 1, "order_number": 1}
```

Listing 3: Compound index with three fields.

- *Multikey Index*: A multikey index is automatically created if the field the index is created on contains an array of values. MongoDB creates separate index entries for every element in the array, allowing queries that match an element or elements in the arrays. The array can contain both scalar values (i.e. strings, numbers) and nested documents.

- *Geospatial Index*: MongoDB provides two indexes to support efficient queries on geospatial data: the 2d index which uses planar geometry, and the 2dsphere which uses spherical geometry. These will be covered more in-depth in Section 2.6.1.
- *Text Index*: Text indexes support text search queries on string content. This can either be a single string or an array of strings, and the index can be a single field or compound. In addition, if the index is compound, it is possible to specify weights for each field. This is used to denote the significance of one field compared to others. MongoDB then uses this weight when calculating a score for each document by multiplying the number of matches with the weight. The index is also case-insensitive.
- *Hashed Index*: This index is used to support hash-based sharding. It indexes the hash of the field's value. A hash index has a more random distribution of values compared to other indexes, but because of the hashing, the index can only support equality matches and not range-based queries. More on hash-based sharding later in this section.

At its core, MongoDB is a distributed database with high availability, horizontal scaling, and geographic distribution built in. For high availability, MongoDB uses a replica set which is the replication of a group of MongoDB servers that hold copies of the same data. Under high workloads, this makes it possible for the database to route requests to the replications, spreading the workload over multiple machines. Further, MongoDB utilizes sharding to achieve horizontal scaling, which is a method for distributing data across multiple servers. Such a deployment of MongoDB is called a sharded cluster. The sharded cluster consists of three components:

- Shard – contains a subset of the data. A shard must be deployed as a replica set to provide redundancy and high availability
- Config servers – stores metadata and configuration settings for the cluster
- Mongos – acts as a query router, providing an interface between client applications and the sharded cluster

A shard consists of multiple servers since it must be deployed as a replica set. Together, the shards in a cluster hold the entire data set for that cluster. Further, the config server contains metadata that is required for the cluster's operation and maintains a mapping between the shard key and the corresponding shards. The information about the mapping is cached in mongos where it is used to direct the queries from the client application to the appropriate shard. The interaction between the different components is described in Figure 2.

MongoDB shards the data at the collection level using shard keys to distribute the documents within a collection across shards. A shard key is either a single indexed field or multiple

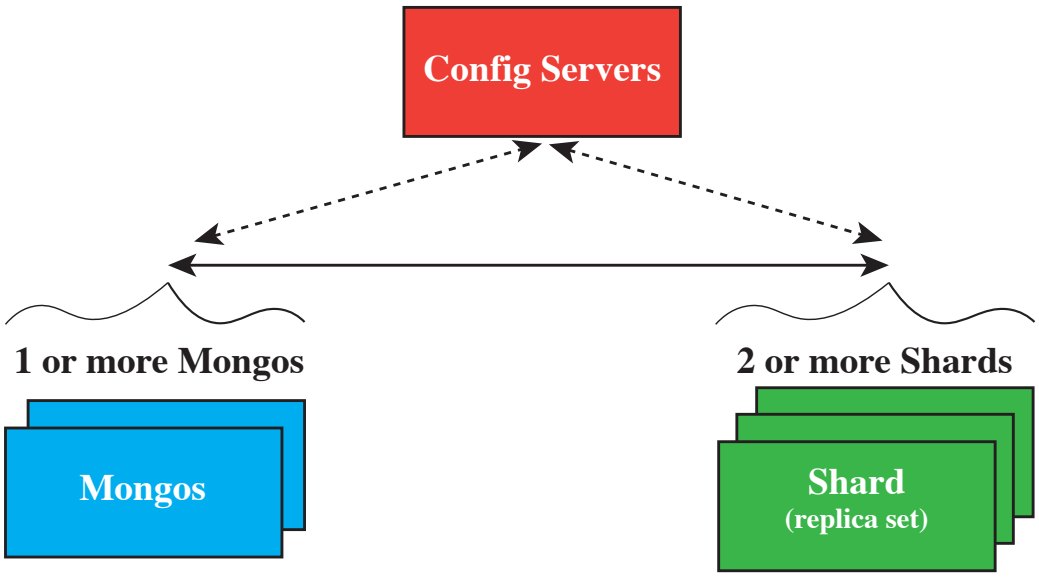


Figure 2: Shard components interaction.

fields covered by a compound index. MongoDB divides the values of the shard keys into non-overlapping ranges. Each range is associated with a chunk, a continuous range of shard key values, and MongoDB tries to distribute these evenly among the shards in the cluster. In addition, it is required that a sharded collection have an index that supports the shard key. If the collection that is to be sharded is empty, MongoDB creates the supporting index if the collection does not already have an index for the specified shard key.

Further, MongoDB supports two sharding strategies. The first is hashed sharding, which involves computing a hash for the shard key field's value. Then, each chunk is assigned a range of hashed shard key values. A visual representation can be seen in Figure 3. This strategy facilitates a more even distribution of the data. However, range-based queries are less likely to target a single shard since data that may have non-hashed shard key values close together most likely is spread over multiple shards.

The second strategy is ranged sharding. This involves dividing data into non-overlapping ranges based on the shard key values, as seen in Figure 4. This means that shard key values that are close are more likely to be in the same chunk, and range-based queries are more likely to target one shard. The efficiency of this sharding strategy depends on the shard key chosen. If the shard keys are poorly considered, it can lead to an uneven distribution, negating some of the benefits of sharding.

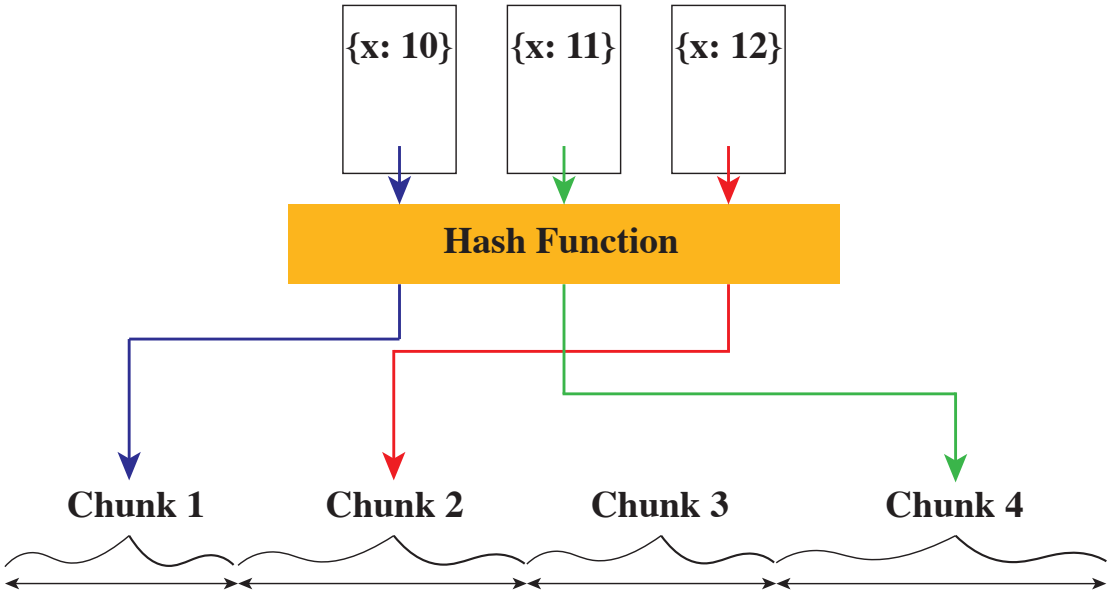


Figure 3: Hashed sharding.

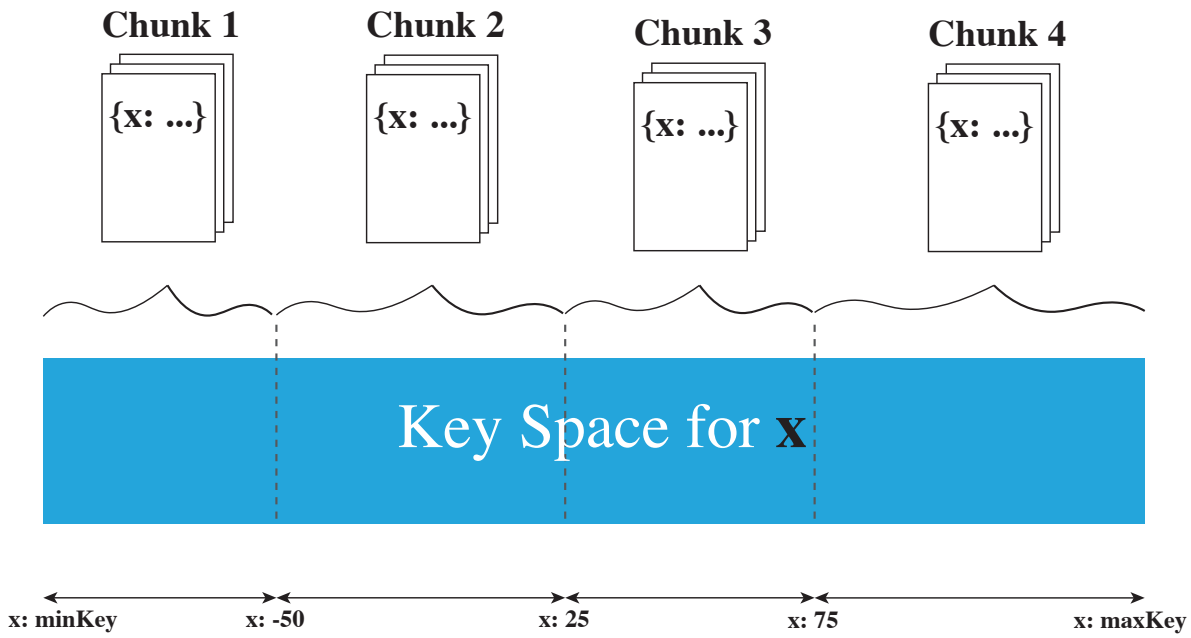


Figure 4: Ranged sharding.

2.2.2 Neo4j

Another NoSQL database that is worth mentioning is Neo4j. The content in this section is retrieved from Neo4j's documentation [36]. It is an open-source, highly scalable graph database. This means that it stores data as nodes, relationships, and properties instead of tables or doc-

uments. Since the relations are already represented in the graph, the database does not have to calculate them at runtime like other types of databases. This allows it to handle complex connections between data very fast.

Each node in the graph represents an entity in the domain, and a node can contain labels and properties. A label is a way of grouping nodes into sets where all nodes with a certain label belong in the same set. A label can for example be *Person*. This allows Neo4j to perform operations only on nodes with a certain label. Furthermore, a node can have zero to many labels, and these can be added and removed during runtime. Since labels can be modified during runtime, they can be used to indicate temporary states for nodes. Connections between nodes are described as relationships. These are unidirectional, so they tell how a source node is connected to the target node. To describe the relationship between the source node and the target node Neo4j uses types. An example of a type can be **:OWNS**. It is required that a relationship has one and only one type. To store data on nodes and relationships, Neo4j uses properties. These are key-value pairs that can hold different data types such as number, string and boolean, and homogeneous lists. Figure 5 shows an example of a simple graph, containing nodes, relationships, and properties.

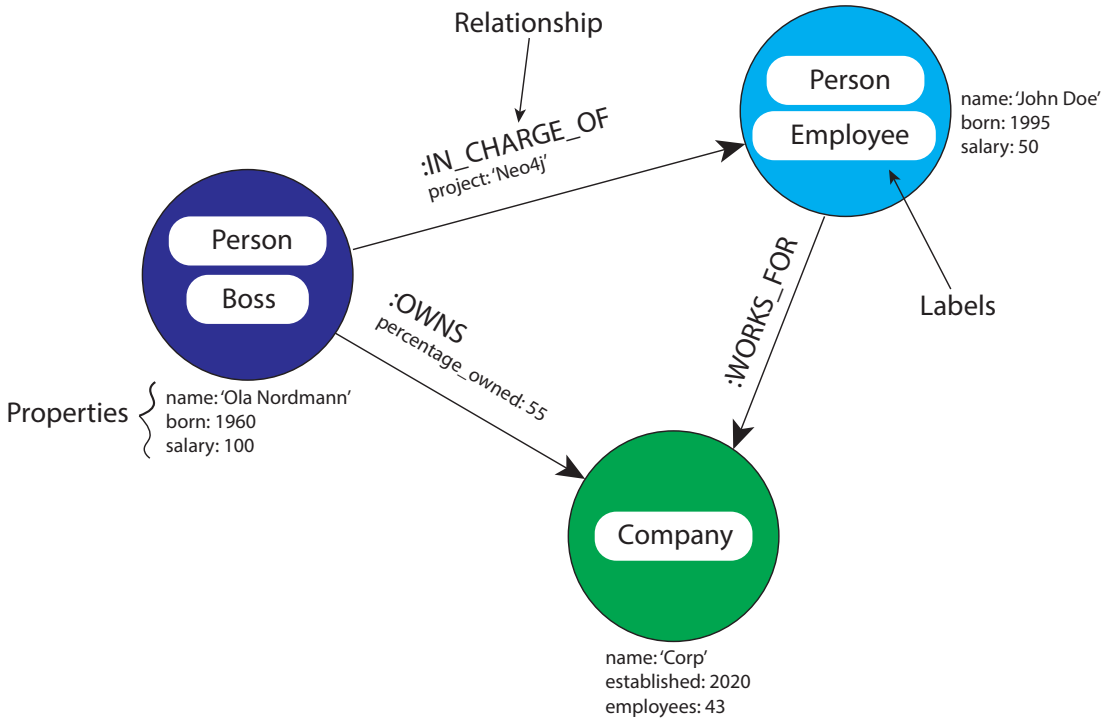


Figure 5: Nodes, relationships and properties in Neo4j.

Querying data is done by traversing the graph, following the relationships between the nodes.

The query language used to retrieve data is called Cypher. It is inspired by SQL in the sense that it lets the user focus on what data to retrieve from the graph and not how. The syntax was inspired by an ASCII-art type of syntax using rounded brackets for nodes and arrows for relationships as seen in Listing 4. This makes writing queries very visual. Cypher is also heavily based on patterns and uses this to match desired graph structures. Since pattern recognition is a fundamental way of how the brain works, the query language is said to be simple and logical for users to learn. A simple example of a pattern is: *a Person* LIVES_IN *a City*. This only consists of one relationship, but patterns can be composed of numerous relationships, expressing arbitrarily complex concepts.

```
(p:Person {name: 'John Doe'})-[rel:WORKS_FOR]->(c:Company {name:
↪  'Corp'})
```

Listing 4: Neo4j syntax. This shows the relation: John Doe works for Corp.

Neo4j is schema optional, meaning it does not require the user to define a schema for the database. A schema in Neo4j refers to indexes and constraints, meaning that it is not necessary to define these before data is stored in the database. However, in can be beneficial for performance and consistency in the data. The main reason for using indexes in a graph database is to find the starting point of graph traversal.

2.3 Relational Databases

The relational data model was introduced in 1970 by Codd [37]. It has proven to be an adaptable model capable of representing many different types of data and has managed to stay relevant for 50 years, an impressive feat in the field of computer science considering the rapid pace of development seen in the field. Four years after Codd's paper [37], the SQL (originally named SEQUEL) query language was released by IBM researchers Chamberlin and Boyce [38] and it is today used in all modern relational database management systems (RDMBS). We assume readers are familiar with both SQL and the relational data model and thus only a brief introduction of a popular open-source relational database, PostgreSQL, will be presented.

2.3.1 PostgreSQL

PostgreSQL is an open-source object-relational database management system. The POSTGRES project began in 1986 at Berkeley and is the ancestor of Postgres95 which today is known as PostgreSQL. PostgreSQL has been actively developed over two decades and is now one of the most advanced open-source databases out there with documentation spanning over 2000 pages

[39]. Data is stored in heap files with a page size of 8 kB and indexed with the B-tree by default. There are other indexes available and a full list of them is available in the documentation [39]. The database has support for a rich set of data types and is also capable of storing JSON and XML. It offers partitioning of tables which can be combined with remote server functionality to create shards. Thus it can also be scaled horizontally, although it is not able to provide the same performance as systems specifically designed to be scaled horizontally, such as MongoDB.

2.4 Geographical Information Systems

Geographical information systems (GIS) are systems that store, retrieve, manipulate, analyze, and map geographical data. It can store a multitude of different types of data as long as they contain location data. GIS is able to compare and overlay all the different types of information on a single map, allowing for analysis of the relationships between information. This section will cover the more general GIS.

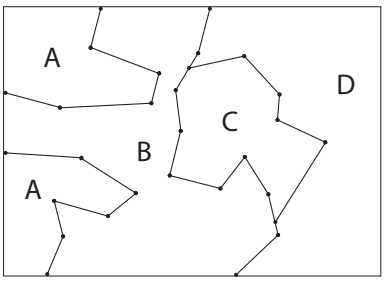
There are many ways to store geographic data in GIS, but the two most prominent ones are to store geographic information as raster or vector representation [40]. Raster data are often images where each pixel represents a cell in the grid. Every cell has an implied location based on its relationship to a single known location on the image. Vector data stores the geographic features using points that are connected by lines. Hence, an area is stored as a string of vectors where each vector starts where the last one ends. A visual representation of the two forms can be seen in Figure 6. The figures contain the same area, but as seen in Figure 6a, it looks like the B area is split in two. However, as seen in Figure 6b the B area is a single area. A finer raster grid with more cells would be able to depict the vector figure more accurately, but that requires more storage. So storing data in the vector format is more efficient than raster, as it is only necessary to store points and lines as opposed to raster where usually all the cells need to be stored. GIS can typically handle both vector and raster data, including switching between the formats. One can for example give a GIS a raster image and it can identify point features like buildings and line features such as streets, and save the information in vector format for further analysis.

Geographic data elements are those elements that one would recognize in the real world or on a map, and although GIS can utilize both raster and vector information, raster data are not truly geographic data. There are three elements of spatial objects that GIS uses to represent any geographic entity or the attributes of a geographic entity. These are:

- Points – The location of a point is represented by x, y coordinates, and the points are considered to have zero dimensions. Points are used for representing many different objects, such as buildings, crime locations, wells, trees, and so on. The scale and the

A	A	B	B	D	D	D
A	A	A	C	C	D	D
B	B	B	C	C	C	D
A	A	B	B	B	D	D
A	B	B	B	B	D	D

(a) Raster.



(b) Vector.

Figure 6: Representation of geographic information.

resolution play an important role in how an object is to be represented.

- Lines – Lines are represented by points that are connected by an arc, and they have one dimension. They can be used to represent roads, and the thickness of the line can represent the type of road.
- Polygons – Areas in GIS are represented by polygons of any shape. They consist of a closed set of lines that encloses the area. Polygons can be stacked on top of each other to form a three-dimensional representation. This can be useful for showing the elevation of a hill or a mountain.

An important aspect of GIS is that it retains the topological relationships between the geographic data elements. Topology is the study of the properties of a geometric object that remains unchanged by deformations such as bending, stretching, or squeezing but not breaking [41]. In GIS, this retention of information is used to represent that lines have direction and a starting and end point. The topology includes information as to which areas/polygons are on the left and which are on the right side of the line. Wieczorek and Delmerico [42] state that a system that does not retain topology is not truly a GIS, but is a collection of various lines and points that are unrelated.

Another important aspect of GIS is georeferencing, also called geocoding. Georeferencing is a type of coordinate transformation that binds raster or vector data to a Spatial Reference System (SRS). SRS, also called a Coordinate Reference System (CRS) defines how geometry is referenced to locations on the Earth's surface. There are three types of SRS [43]:

- A geodetic SRS – this uses longitude and latitude which map directly to the surface of the earth.
- A projected SRS – this flattens the spherical surface of the Earth onto a plane using a mathematical projection transformation. The result of this transformation is a Cartesian

coordinate system where location coordinates are assigned in a way that allows direct measurements of quantities such as distance, area, and angle.

- A local SRS – this is a Cartesian coordinate system which is not referenced to the Earth's surface.

GIS provides the ability to overlay multiple layers of information on top of each other and access these layers simultaneously. This can be used to combine different data about an area and analyze the relations between the data. An example of this is combining a map of the police districts in a city and a map of the locations of crimes and then GIS can be used to count the number of crimes within a given district. GIS also has spatial buffers that are used to specify an area around a geographic object, for example, a radius of 500 meters from a given point. This combined with other layers of information can be useful for decision making. A simple example of this is figuring out how many people live within a given distance from a potential new location for a hospital. All these features make GIS a powerful and important tool when analyzing spatial data.

2.5 GeoJSON

A way of storing geographical data is with GeoJSON. GeoJSON is a proposed standard (RFC7946) that describes a format specifically designed for encoding different geographic data structures using JSON [44]. The format is stricter than JSON and requires that objects contain certain fields and values. It requires that a GeoJSON object contain a field called *type*. This field must contain one of the geometry object types listed in Table 2, Feature or FeatureList. A Feature is a spatially bound entity and a FeatureList is a list of Features. Furthermore, a Feature contains a Geometry object, that represents points, curves, and surfaces in a coordinate space. A Geometry object is also a GeoJSON object, meaning that it must contain a field named *type*. However, this field can only take the value of one of the seven geometry object types in Table 2. In addition, a Geometry object of any other type than GeometryCollection contains a field named *coordinates*. The value of this field is an array and it is the type of the Geometry object that determines the structure of this array.

Furthermore, the GeoJSON standard specifies position as the fundamental geometry construct. A position is an array of two or more elements, where the first is the longitude and the second is the latitude. An optional third element, the altitude or elevation, can be added, but going beyond three elements is not recommended by the standard because the semantics of extra elements are unspecified and ambiguous.

Table 2: Geometry object types and description [44].

GeoJSON object type	**Description**
Point	The *coordinates* member is a single position
LineString	The *coordinates* member is an array of two or more positions
Polygon	The *coordinates* member must be an array of linear ring coordinate arrays
MultiPoint	The *coordinates* member is an array of positions
MultiLineString	The *coordinates* member is an array of LineString coordinates arrays
MultiPolygon	The *coordinate* member is an array of Polygon coordinate arrays
GeometryCollection	A Geometry object that can be a heterogeneous composition of smaller Geometry objects

2.6 Geospatial Objects in NoSQL

In recent years, NoSQL databases have started implementing support for geospatial objects, picking up the competition with relational databases. This section will cover how MongoDB and Neo4j handle geospatial objects.

2.6.1 MongoDB

Geospatial data in MongoDB is stored as GeoJSON objects (see Table 2 for details on the different objects) or as legacy coordinate pairs. GeoJSON objects are used for calculating geometries over an Earth-like sphere and are stored as an embedded document with a field for *type* and a field for *coordinates*. The *type* field specifies the GeoJSON object type and the *coordinates* field stores the coordinates as an array. On the other hand, legacy coordinate pairs are used to calculate distances on a Euclidean plane, and the data is stored in either an array or an embedded document. The format the data is stored in also decides what type of index will be used for queries. MongoDB provides two different geospatial indexes, 2d and 2dsphere. 2d indexes support queries that calculate geometries on a two-dimensional plane. The index is intended for data stored in legacy coordinate pairs used in MongoDB 2.2 and earlier. The 2dsphere supports data stored as both GeoJSON objects and legacy coordinate pairs. For legacy coordinate pairs, the index converts the data into GeoJSON points. The index supports all queries that calculate geometries on an earth-like sphere. MongoDB has a limited number of geospatial operations,

listed in Table 3, and not all operations are available on both indexes. The 2dsphere only supports spherical queries, while the 2d index supports flat queries and some spherical queries. As with the other indexes in MongoDB, both the 2d and the 2dsphere index use a B-tree structure, known for handling large amounts of data and fast inserts and deletions.

In the 2d index it is the geohashed value of the coordinates that are used for indexing. A geohash is a binary representation of the coordinates. It is calculated by recursively dividing a two-dimensional map into quadrants and assigning each quadrant a two-bit value. The value represents the quadrant and all the points within that quadrant. This process is continued to add precision, so each quadrant is divided into sub-quadrants that are represented by the concatenated bit value of the sub-quadrant and all parent quadrants.

The 2dsphere index utilizes a form of geohashing, based on the S2 library developed by Google [45]. The library works with spherical projections, mapping points on the Earth's surface to a perfect mathematical sphere. It then decomposes this sphere into a hierarchy of cells, where the cells are quadrilaterals bounded by four geodesics. The top level of the hierarchy contains 6 cells, and each of the cells can be divided into four sub-cells, which again can be divided into four. The levels range from 0 to 30, where level 30 covers about 0.74 cm^2 [46]. Each cell has a unique cell id, represented by a 64-bit integer. These cell ids are used when MongoDB indexes different GeoJSON objects. For example, for indexing a Polygon, MongoDB calculates a set of cells that cover the polygon and add the cell ids to the index.

2.6.2 Neo4j

Another NoSQL database that has support for geospatial data is Neo4j. It has a library called Neo4j Spatial that has utilities that enable spatial operations on data. It is currently in version 0.28.1. The key features of the library include [48]:

- Utilities for importing from ESRI Shapefile as well as Open Street Map files.
- Support for all common geometry types: Point, LineString, Polygon, etc.
- An R-tree index for fast searches on geometries.
- Support for topology operations during search (contains, within, intersects, covers, disjoint, etc.).
- The possibility to enable spatial operations on any graph of data, regardless of the way the spatial data is stored, as long as an adapter is provided to map the graph to the geometries.
- Ability to split a single layer or dataset into multiple sub-layers or views with pre-configured filters.

Table 3: MongoDB geospatial operation [47].

Operation	Spherical/Flat Query
$near (GeoJSON centroid point in this line and the following line, 2dsphere index)	Spherical
$near (legacy coordinates, 2d index)	Flat
$nearSpehere (GeoJSON point, 2dsphere index)	Spherical
$nearSphere (legacy coordinates, 2d index)	Spherical
$geoWithin: {$geometry: ...}	Spherical
$geoWithin: {$box: ...}	Flat
$geoWithin: {$polygon: ...}	Flat
$geoWithin: {$center: ...}	Flat
$geoWithin: {$centerSphere: ...}	Spherical
$geoIntersects: {$geometry: ...}	Spherical
$geoNear aggregation stage (2dsphere index)	Spherical
$geoNear aggregation stage (2d index)	Flat

Neo4j's primary type that defines a collection of geometries is called Layer. These are predefined in the library, so depending on the data that is being stored, one can choose the Layer that suits the data the best. For example, if there are only point data that is being stored, one can use the SimplePointLayer which only allows storing Points in the database, and it has methods specifically for this type of data, Layers can also be editable, which means that it is possible to modify the data in the Layer. It is also the Layer that contains the R-tree index.

Neo4j Spatial contains the Java Topology Suite, a library that provides an object model for planar geometry and a set of fundamental geometric functions. This enables the usage of all the capabilities of Java Topology Suite to operate on Geometry (Point, LineString, Polygon, etc.) instances obtained from the database. This allows for a wide variety of operations on the returned data from a query. Neo4j Spatial has implemented a larger amount of spatial queries than MongoDB. The queries that are implemented are: Contain, Cover, Covered By, Cross, Disjoint, Intersect, Intersect Window, Overlap, Touch, Within, and Within Distance.

2.7 Geospatial Objects in PostgreSQL

The extension PostGIS allows working with spatial data in PostgreSQL and it is a mature extension for working with GIS objects in the database. According to DBEngine, it is the most popular database for use as a spatial database [14]. The PostGIS extension uses an R-tree index implemented over a Generalized Search Tree (GiST) [15], [49]. Multiple geometries are available such as Point, LineString, Polygon, MultiPolygon, Triangles, and more. Three-dimensional data is supported and a measured value (e.g. time) as a fourth dimension is also accepted. Well-Known Text (WKT) [50] and Well-Known Binary (WKB) is used to represent geometries [15]. PostGIS offers both a Cartesian coordinate system and geodetic coordinates following the WGS 84 spatial reference system. The spatial reference system of a collection of geometries must be defined when creating the collection. Many functions for processing spatial data are offered and a complete list of functions can be found in the documentation [15].

2.8 Related Work

MongoDB has received attention from multiple papers in the GIS domain in recent years [19]. It has been compared to one of the most advanced spatial databases available, PostGIS, on performance in spatial processing [20]–[22]. Researchers have implemented multidimensional indexing structures in MongoDB to use the database for spatial data management [10], [51].

2.8.1 Performance Comparisons

Bartoszewski et al. [22] compared the performance of MongoDB versus PostGIS on different spatial queries. The test data was generated with a random sampling of points in a square containing the territory of Alaska. Their tests show MongoDB performs better than PostGIS on simple geometries and larger datasets for most queries. PostGIS performs better in the case of more complex shapes on nearest neighbor queries with a large area to check [22]. Agarwal and Rajan [21] tested point-within-polygon queries and found MongoDB to be on average ten times faster than PostGIS, with the performance difference increasing with the size of the data increasing. Both papers highlight the difference in functions available as PostGIS offers way more advanced spatial analysis with over 1000 spatial functions [21], [22]. Schmid et al. [20] ran geo-within tests simulating multiple concurrent user requests with the same results as the papers above. MongoDBs performance does not degrade as the size of the data grows, but performs slower on more complex geometry types. PostGIS performs better on small data sets where the geometries are more complex, but does not scale well [20]. MongoDB may be more performant with sharded collections as it is designed for scaling well horizontally, but none of

the above-mentioned papers ran tests on sharded collections.

2.8.2 Multidimensional Indexes in MongoDB

Li et al. [51] proposes an R-tree solution in MongoDB to store the position of patients providing hospitals with health data. NoSQL was proposed as a solution to handle the large amount of live stream data from medical devices. They used the R-tree as a global spatial index with local nodes containing hashed coordinate values and patient data. The results show their spatial index outperforms MySQL, but inserts slower than MongoDB without their spatial index by a small margin due to the overhead of building their global tree [51]. Xiang et al. [10] implements a flattened R-tree in MongoDB. The R-tree is written as an independent module and connected with MongoDB through an I/O layer using native MongoDB commands to build the R-tree as a collection. The 2dsphere index outperforms the solution in terms of speed for most cases, but performs worse on complex polygons of varying sizes. The R-tree implementation performs competitively in processing time for calculations, but suffers from I/O overhead of needing more I/O calls to perform each operation as seen in their results [10].

3 Design and Implementation

This section describes the development and implementation of the R-tree module for MongoDB version 6.3.0. The implementation uses the work of Xiang et al. [10] as a base for the code. The decision to reuse the R-tree implementation was made to save time given the short time frame of the project and it allowed for the development to be more focused on the I/O commands necessary for a working implementation in MongoDB. The main changes implemented are modifications to how the R-tree module communicates with MongoDB's source code as there have been changes to how internal command execution works from MongoDB version 3.2.0 to version 6.3.0. Minor alterations to the R-tree module were necessary to use a newer version of the GEOS library, but these changes do not affect the implementation. The code project consists of three parts: MongoDB source code, the R-tree module, and the I/O implementation.

3.1 MongoDB Rational

The reasons for implementing the R-tree in MongoDB are numerous. As mentioned in Section 2, the database is a widely used, open-source, NoSQL database. Since it is open source, it is easy to clone the database project from Github and modify it, and because it is widely used, one could assume there are many online resources that could aid in the implementation. The implementation described in this section is based on Xiang et al. [10] and the implementation they describe in their paper. This is also a reason why the choice fell on implementing the R-Tree in MongoDB. Due to the scope and the time limit of this project, it made more sense to build on previous work than implement the R-tree from scratch. However, their implementation of the R-tree was done in MongoDB version 3.2, which is three major releases behind the current version of the database. Therefore, it is reasonable to assume that the code has seen major changes and that parts of the implementation done by Xiang et al. [10] would no longer work in the current version of MongoDB. Lastly, MongoDB does have indexes for handling spatial data as mentioned in Section 2. However, these indexes contain a limited number of spatial operations. By implementing an R-tree, one could facilitate more operations and potentially more efficient querying.

3.2 Planning

There were several important challenges to consider when making the plan for the project. Firstly, C++ was an unfamiliar language at the time starting and thus the initial part of the development phase had to be dedicated to learning more C++. This introduced an initial delay which was important when deciding on the scope of the development project. Secondly, the

source code for the R-tree module had only been briefly scanned through and was of unknown quality and size. Finally, the inner workings of MongoDB were unfamiliar and required time to understand. These factors contributed to the decision to restrict the functionality to bring into the upgraded version. To be able to analyze the performance of the module, there must be at least two functions implemented; it must be possible to insert data and a query function must exist to test how well the index performs on the data. Preferably multiple query functions to have a stronger foundation for comparison against MongoDB and see how the results of the upgraded module compare against the results found by Xiang et al. [10]. Following the reasoning presented, it was decided to implement the insert operation along with three query functions: `$geoIntersects`, `$geoWithin`, `$near`. It was assumed the underlying R-tree module would work out of the box provided the correct data. The assumption includes the R-tree query algorithms, meaning no extra time was allocated to handle problems with the queries in the underlying module if they were to be discovered.

3.3 Development Methodology

The goal of the development was to upgrade a previously made R-tree implementation from MongoDB version 3.2.0 to version 6.3.0. C++ was selected to develop the implementation as MongoDB and the R-tree module are written in C++. Python scripts cleaned and formatted the data for the testing phase. GitHub was selected as the version control system for the project to make collaboration simple. The repository of the project was initialized as a copy of the MongoDB version 6.3.0 branch in the MongoDB repository. The R-tree module created by Xiang et al. [10] was then integrated into the repository. In order to upgrade the R-tree module within a short time frame, it was necessary to use an efficient method to port the code over to the newer version of MongoDB. An incremental strategy was applied, the commits in the GitHub repository of the previous R-tree module were added one by one. For each commit the compiler errors were fixed and MongoDB functions no longer available were removed. The goal of the first part of the development was to be able to compile the code into a working build without focusing on R-tree functionality. Skipping tests of individual components in the R-tree module saved time on porting the code and allowed for a test-ready build in a short amount of time. When the project could be built successfully with the source code from the R-tree module imported, the next part of replacing I/O operations began.

A significant challenge during the development of the I/O operations was dependencies between the operations. Therefore it was important to discover the relationships between the operations to allow for development to be executed efficiently. The R-tree implementation is structured around two meta collections which will be described in the sections below. The higher-level operations such as inserting data and running e.g. `$geoIntersects` require updating or query-

ing metadata. The metadata collections, however, are isolated from regular collections and therefore have no dependencies on other data. Since metadata operations are the base of other operations, they must be implemented first to handle the first dependency relationship. The next dependency occurs between the insert operation and the query operations. The query operations rely on the R-tree structure to fetch data, but the R-tree cannot be built without the insert operation in place. From this follows a natural order of development; the metadata operations are implemented first, then the insert operation, and finally the queries are implemented. Figure 7 shows how the implementation order is related to the commands in the R-tree module.

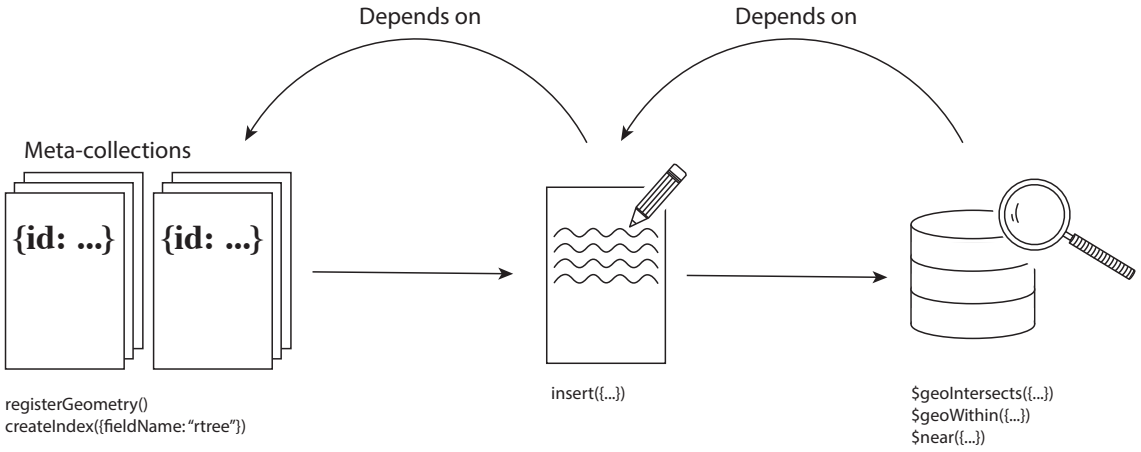


Figure 7: Implementation order.

3.4 The Flattened R-tree

The flattened R-tree is an implementation of the structure described in Section 2.1.1. To index a spatial collection (SC) in MongoDB, the R-tree structure has been flattened into a single collection, the R-tree collection (RC). There are two metadata collections, Spatial Metadata Collection (SMC) and R-tree Metadata Collection (RMC), to keep track of which index belongs to which spatial collection and maintain pointers to the root nodes of each R-tree.

3.4.1 The R-tree Collection

To flatten a tree structure into a collection, there are some important points to keep in mind. It must be possible to know which level each node resides in for parsing the R-tree. Also, storing each MBR in a separate document requires more I/O operations and will cause the solution to be slow. Therefore it is desirable to store child nodes as sub-documents within a node to fetch a node and its children with one read. As the number of children is preset to determine the fan

factor of the tree, the list of child nodes in a node can be pre-allocated to save more time. Since the list initially will be filled with empty nodes, a field to keep track of how many children contain data allows for simpler iteration. In addition, the sub-document of a child node contains a boolean to indicate whether or not it contains data allowing for quick filtering. Figure 8 shows how the level is kept as an integer to maintain the tree structure where level = 0 is the leaf node level. Each entry in the branches list is a sub-document as described above:

```
{ hasData: bool, mbr: MBR, childID: OID }
```

where childID points to an object in the SC if it is a leaf node, another R-tree node if not. The RC does not need to know its own root, as the RMC will keep track of the root node.

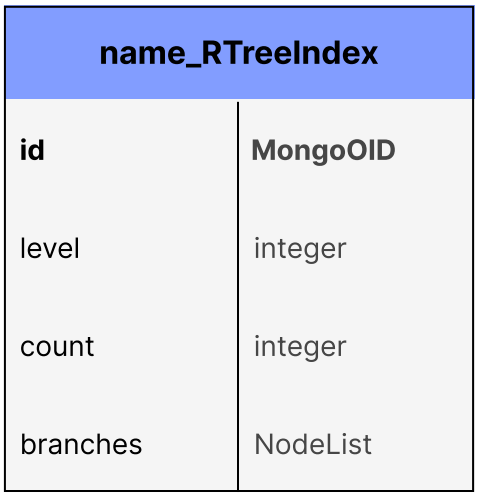


Figure 8: A document in the R-tree collection.

3.4.2 The Metadata Collections

The SMC keeps track of the connections between an R-tree index and the corresponding indexed collection. As seen in Figure 9, information about the contents of the spatial collection is stored in addition to a pointer to the R-tree metadata. For each spatial collection (SC), there will be one corresponding SMC document and RMC document. The fields *gtype*, *srid*, *crs_type*, and *tolerance* are intended for application usage and are not used to keep track of the R-tree index. The first two string fields identify the spatial collection and which field to index in the spatial collection. A datanamespace is on the format *dbname.collection_name* following the MongoDB format. The index type is set to 1 if the spatial collection is indexed by an R-tree, 0 otherwise. Each document in the SMC has a foreign key, *index_info* referencing an RMC document, which is only set after the R-tree index has been initialized. The *index_root* in the RMC document points to the root node in the R-tree collection and is updated whenever the root changes due

to splits. The *max_node* field refers to the *M* described in Section 2.1.1. *max_node* can also be considered as the branching factor of the three as it decides the length of the *branches* in the R-tree collection documents. *max_leaf* allows the user to increase the number of child nodes possible at the leaf level of the tree. Both values are set when creating the index.

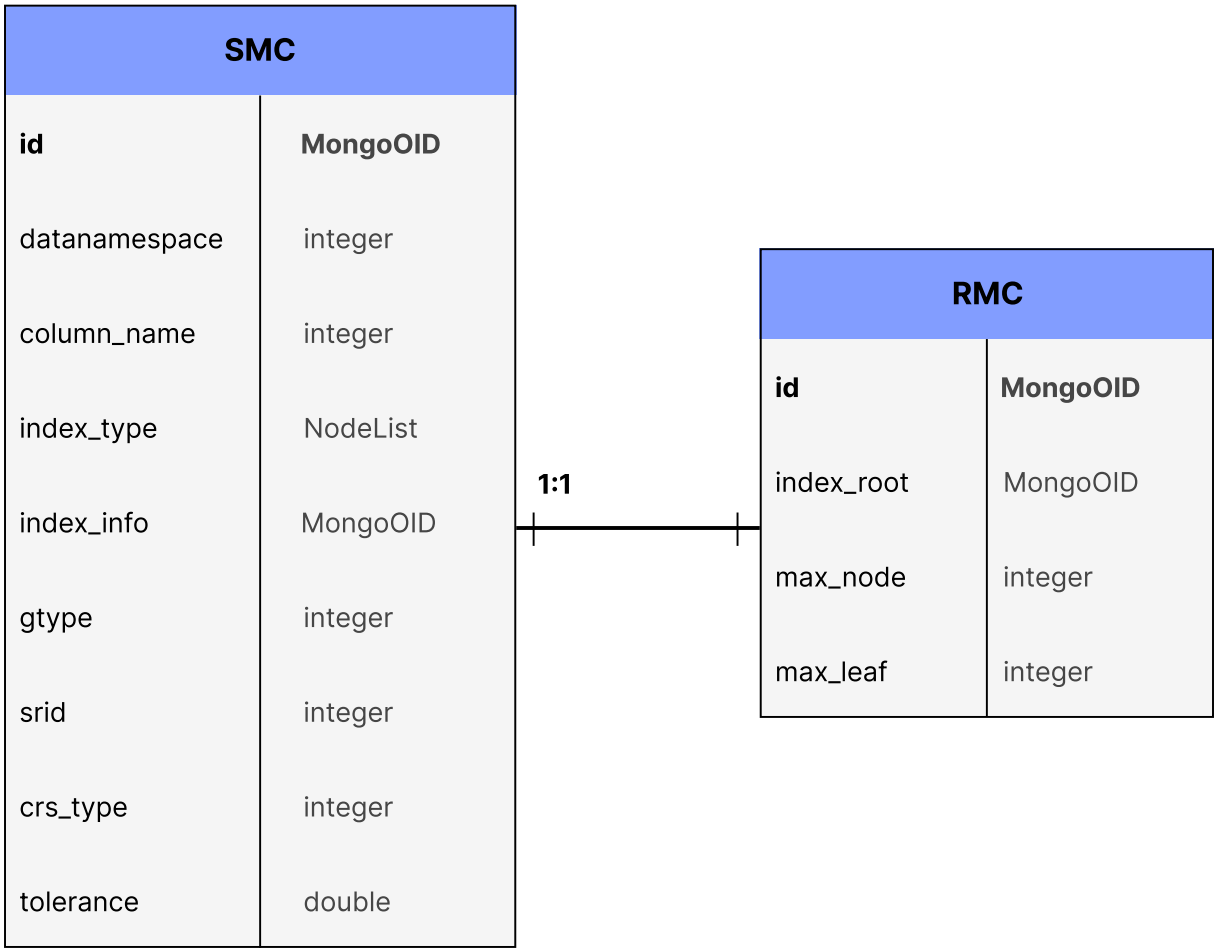


Figure 9: Document structure in the metadata collections.

3.5 System Architecture

A sharded cluster in MongoDB depends on three types of servers: a config server, a database server, and a routing server. The config and database servers are mongod processes whereas the routing server is a mongos process. Figure 10 shows how the processes are connected with the shell being the interface for user interaction. The user writes commands through the mongosh process which are received at the routing server. The routing server is responsible for running the commands on the correct shards and keeping the config database updated. To keep the architecture as simple as possible, only one shard and one config server is set up in the cluster for this project. In a real production setup there will be multiple shards and config servers. Additionally, there may be multiple routing servers to safeguard against hardware and software

faults. More details on how to deploy a sharded cluster for production environments can be found on MongoDB's guide for deploying shared clusters [52].

To communicate with both the config server and the database server, the R-tree module lies within the routing server. The R-tree module consists of the R-tree algorithms and an LRU cache. Additionally, there is an I/O interface between MongoDB and the module to separate the write operations from the R-tree algorithm. The R-tree utilize the I/O interface when parsing the tree by fetching one node at a time. The most recently used nodes will be stored in the cache for faster performance. Since the R-tree algorithms are separated from the DBMS they operate on, the module could in theory be ported to other systems as Xiang et al. argues in their paper [10]. However, much of the code relies on the object structure provided by MongoDB documents, meaning the code would require a significant refactoring to function in another system.

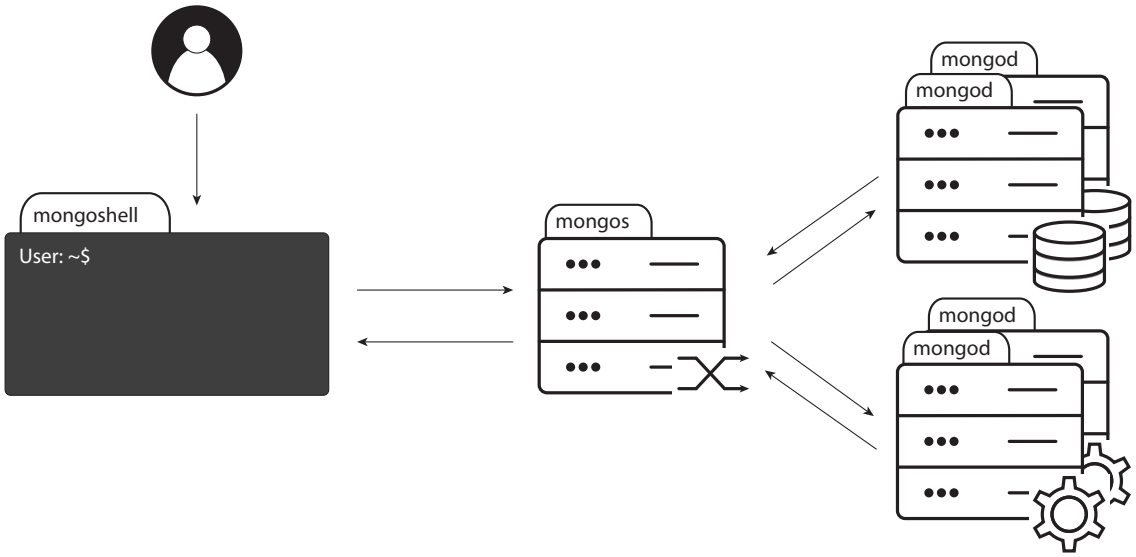


Figure 10: A sharded cluster with shell interaction.

3.6 Commands in MongoDB

The routing server, a *mongos* process, is responsible for routing the commands from the shell to be run on the different shard servers. Thus, the process must parse the command sent over the network to an internal command object and invoke the command object. To implement custom commands, it is necessary to understand parts of MongoDB's internal command process, and the parts relevant to the project will be covered here. A command is implemented as a C++ file and extend the base class Command. Each command created is added to a global command registry which can be queried. All commands in the R-tree module except one, `registerGeometry`, do not need to be registered as a separate command as they modify ex-

isting commands. To add the custom command `registerGeometry` to MongoDB and make it usable through the shell interface, there are two steps. First, the C++ file for the command must be created with functions informing MongoDB on how to run the command. Second, the command must be made available in the shell by adding it as a function on collections in the shell. By implementing the function in Listing 5, MongoDB will know how to run the command:

```
bool run(OperationContext *opCtx,
         const DatabaseName &dbName,
         const BSONObj &cmdObj,
         BSONObjBuilder &result
) {}
```

Listing 5: Run function in a MongoDB command.

Inside the run function, the R-tree module is used to register a spatial collection through R-tree function calls. The other commands which modify existing commands have their implementations inside of the run function in the appropriate command, e.g. R-tree inserts are handled in the MongoDB insert command. The key parameter to consider in the function is the *cmdObj* as it contains all parameters passed from the shell. By adding custom fields or values, specific branches of code can be targeted. This approach is used to trigger the R-tree module in modified MongoDB commands by looking for R-tree specific custom values and running the R-tree module only if the specific fields are present in the cmdObj. Note that the MongoDB command process presented here may differ greatly from how a MongoDB engineer would describe the process and does not serve as an accurate depiction of MongoDB command processing. It is presented in the simplified manner above to show how the R-tree module has been integrated into MongoDB from the perspective of the authors.

3.7 I/O Implementation

The implementation of I/O commands for the r-tree is supported by the native MongoDB commands `createIndex`, `insert`, `update`, and `find`. Using native commands allowed for ignoring the underlying data structure and focusing on the logic of the new commands. Table 4 lists the four available R-tree module operations.

Table 4: Available commands.

Command	**Mode**
registerGeometry	new
createIndex	extend
insert	override
find	extend

A full example of how the commands are executed can be found towards the end of this section.

Since there was no previous functionality for creating customized metadata for a collection, `registerGeometry` had to be made as a new command. A spatial collection must exist prior to running the command as it is run on the spatial collection. Running the command creates a document in the SMC registering the spatial collection but does not create an R-tree meta-document. After registering the geometry, it is possible to run `createIndex` to set up the R-tree index. The command call must specify if it should create an R-tree index and on which field to create the index. A check is performed upon creating the index and, if an R-tree index is not requested, the command executes as normal. As `createIndex` executes as normal without the R-tree specification, the method is denoted as extended and not overridden.

3.7.1 Create

After a spatial collection has been registered and the index has been created, the next step is to insert data. The operation has a high I/O overhead as multiple operations are required against both the metadata and the R-tree collection. A document insertion creates a new entry in the RC. Adding a new entry to the RC requires maintaining the R-tree structure and handling splits. If a split causes a new root to be made, the RMC must be updated to point at the new root. In addition to both inserts and updates to the RC, metadata is fetched at the beginning of each insert incurring read costs. With the amount of I/O operations required for each insert being high, a cache is necessary to reduce the overhead. To keep the scope of the project reasonable the LRU cache from the older R-tree module is employed. It stores recently used nodes in memory and improves performance by reducing the number of I/O operations needed.

3.7.2 Search

A common use case for the R-tree is range queries. The upgraded R-tree module provides both intersect and within searches. Searching a collection will first trigger a check to see if an R-tree exists for the collection queried. If the R-tree exists, the root will be fetched and the search begins. First, an MBR is constructed from the query object. Then the R-tree is traversed from the root, excluding branches that do not intersect with the query MBR. The R-tree traversal returns a set of candidates that need to be confirmed as matches. GEOS is used to refine the candidate set into true matches and the result is returned to the user. In modern databases, cursors are a common way to serve the user partial responses. Cursors are practical when the result set is large and it is time-consuming to provide the entire result in one batch. To provide cursor support for R-tree range queries, the tree is traversed with depth-first traversal. Each potential leaf-node candidate is verified as a match with GEOS and added to the return queue.

The queue is flushed to the user when a size threshold is reached and the traversal is paused until the user requests another result.

Xiang et al. [10] describe a novel algorithm for a cursor-based near query, which is also implemented in their R-tree module. The approach is to incrementally increase the distance of the search until it matches the max distance provided in the query, incrementally adding results as the distance increases. The upgraded module reuses the implementation but does not achieve correct results and is therefore not included in Section 4. The task of correcting the near query is added to the list of future improvements.

3.7.3 Update, Delete and Drop

Update, delete and drop operations were not implemented to reduce the scope of the project. The foundation for the operations exists in the R-tree module and the metadata operations for these operations are implemented. While the full commands have not been implemented, the approach to implement them will be described here for future development of the module. If an update to a spatial object in the SC changes the object's MBR, the corresponding index entry must be found and its index entry deleted. Then it must be reinserted into the tree with the updated MBR. Update is therefore even more I/O heavy than insert and requires the cache to increase performance. Deleting a spatial object requires removing the object from the spatial collection and maintaining the R-tree. A deletion may propagate changes all the way to the root as MBRs have to be adjusted along the way as described in [16]. To drop a spatial collection requires interaction with all the collections. The corresponding RC must be dropped, *index_type* and *index_info* must be reset to zero, and the RMC document for the collection must be deleted. The same operations are required if the user only wishes to drop the R-tree index but retain the spatial data.

3.7.4 Example

The following section provides an example of creating an R-tree index on a database called geodb and working with data in the database. All commands are run in the shell interface for MongoDB, mongosh. First, a spatial collection with the name parispoi is created with the two fields: *name* and *location*. The index will be created on *location*.

1. `db.createCollection("parispoi")`

   creates the spatial collection

2. `db.parispoi.registerGeometry({field: "location"})`

creates a document in the SMC with datanamespace as geodb.parispoi and columnName as location

3. `db.parispoi.createIndex({location: "rtree"})`

   creates a document in the RMC and updates the index type and info in the SMC document

After running steps 1.-3. the spatial collection will have an R-tree index on the field *location*. The next step is to insert data:

```
db.parispoi.insert(
  {name: "Louvre", location: {type "Point", coordinates: [35, 65]}}
)
db.parispoi.insert(
  {name: "Eiffel Tower", location: {type "Point", coordinates: [25, 25]}}
)
```

Listing 6: Inserting two points.

With data available, it is possible to check if any of the points are within the area [[20, 20], [20, 26], [26, 26], [26, 20], [20, 20]]. The query for `$geoWithin` is shown in Listing 7. Queries for intersects are the same, only the keyword *geoWithin* needs to be replaced. The query will return the second point inserted into the collection.

```
db.parispoi.find({"position": {"geoWithin": {"geometry": {"type":
↪  "Polygon", "coordinates": [[
                [20, 20],
                [20, 26],
                [26, 26],
                [26, 20],
                [20, 20]
            ]]
}}}})
```

Listing 7: geoWithin query.

3.8 Known Issues

A brief description of known system issues and bugs will be presented here. The most important issue is the `near` query not working as intended. Since the query executes, but does not return

correct results, the query is omitted from the test phase. It does not correctly calculate which objects to return and during testing it would return more results despite decreasing the search area. Since the query is important, it has been set to a high priority in further work. Another issue caused when a user tries to issue an R-tree query without an available R-tree index. This results in a segmentation fault, crashing the routing server. While a system crash is critical, the fault does not occur often as the user must forget to create the index before running the query or run the query on the wrong database. The error can be fixed by adding a check to see if an R-tree exists when the query is being processed. Since the error was discovered after the development had stopped, no fix was implemented. There is also a trivial bug when running the insert commando the first time after creating the R-tree index. A casting error occurs because of how the types for the fields in the meta-collections are handled. The current work-around is to attempt the insert a second time, which will work as the types are corrected during the first attempt to insert.

4 Results and Discussion

The goal of the implementation was to see how it fared up against MongoDB's current way of handling spatial data. Therefore, the testing of the R-tree's operations described in Section 3 are done against the equivalent operations of MongoDB. Data sets used for testing and the test setup will be covered in the following sub-sections.

4.1 Setup

All tests were run on a MacBook Air 2020 with an M1 CPU chip and 8GB memory. The operating system was MacOS Ventura 13.2.1. The project was built with Apple clang version 14.0.0 and python version 3.11. To only build the *mongos*, *mongod*, and *mongosh* binaries, the following command was run from the root directory of the project:

```
python3 buildscripts/scons.py install-devcore -j6
--disable-warnings-as-errors MONGO_VERSION=6.3.0
```

The flag `install-devcore` specifies which binaries to build. An instruction on how to build the project along with a description of different flags can be found in Appendix A. A mock of a sharded cluster was set up on the local network with different ports to simulate different IP addresses. However, only one shard was used, which means all data resided on the same shard. The setup was simple with one config database, one shard, and one routing server. MongoDB offers a tutorial for setting up a sharded cluster: Deploy a Sharded Cluster [52]. With the servers up, a connection was established between the routing server and the *mongosh*. *mongosh* was utilized for registering the spatial collections and creating an R-tree index for each of the spatial collections.

The MongoDB tools were used to test the build speed of the index. With each data set in a separate JSON file, the `mongoimport` command inserted the full files into collections. Each data set was inserted five times to achieve an average build time. Different branching factors were tested on the Hong Kong data set to find the optimal branching factor for build time. The CPU and memory usage was measured during a subset of the insertion tests with the `ps aum` command to see the load on the system. After the insertion tests the data was re-inserted with the most efficient branching factor for the query tests. MongoDB offers a method, `.explain()`, to measure query execution time, but the R-tree module cannot use it as it triggers a different command execution flow. A Python script per query was therefore used to test the time of fetching data. As a cursor object is returned upon initially calling the query, the script must iterate through the entire cursor with a loop to fetch all results. A single run of the test script runs the query five

times and returns the timing result of each query. The script for the `$geoIntersects` query can be found in Appendix B. The tests measure the round-trip time of a query, but in the next sections, it will be referred to as processing time because the network delay is considered to be equally low for both MongoDB and the R-tree module when both are run locally. The tables and figures in the next sections use the averages obtained from the iterations, see Appendix D for the results from all the iterations.

4.2 Data Sets

For testing, four data sets were used with various sizes and features. The data sets were chosen based on the data sets used for testing in Xiang et. al. [10]. For their testing, they used four data sets, where one contained Points, another LineStrings and the last two contained Polygons. The data sets used for testing the R-tree implementation in this thesis are of the same type, but the sets are different in size and cover different areas of the world. General information about the data sets can be found in Table 5. Each data set aims to test how the implementation handles the respective type of GeoJSON feature.

Table 5: Data sets.

Data set	GeoJSON Type	Num of features	Area covered
Paris	Point	96846	25.9km X 19.8km
New York	LineString	119005	18.6km X 21.6km
Barcelona	Polygon	66650	11.9km X 10.9km
Hong Kong	Polygon	21096	8.6km X 12.2km

The data sets were downloaded from overpass turbo, a software where the user can choose an area of the world map and query various features in that area [53]. The result of the query can be downloaded as GeoJSON files, which are JSON files with stricter rules on formatting. The coordinates retrieved from overpass turbo are on the WSG84 format, i.e. spherical data represented by latitude and longitude.

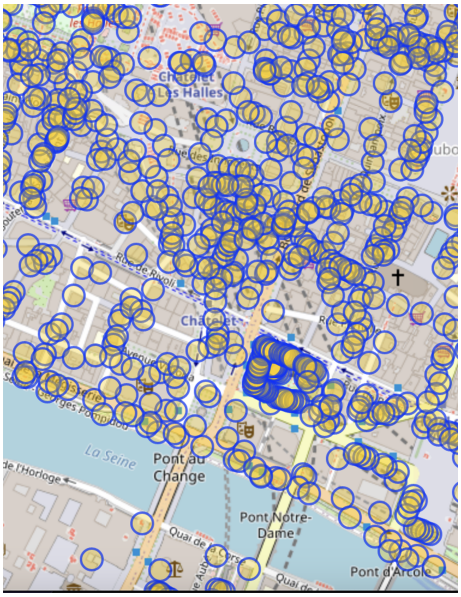
The Paris data set contains points of interest throughout the city center. A thumbnail of the data set can be seen in Figure 11a. Each point corresponds to a single entry GeoJSON Point, and it is represented by one coordinate on the format [latitude, longitude]. Next, the New York data set contains the roads in the city. Each line contains two or more coordinates, representing the starting point, end point, and possibly various points where the road changes direction slightly. Figure 11b depicts the thumbnail for the New York data set. Most of the roads are built up by several lines, meaning that they are represented by multiple points in the GeoJSON file. Further, the Barcelona data set is built up by polygons, represented as an array of lines, where the first and the last line meet and enclose the polygon. As seen in Figure 11c, the polygons are small,

in similar size, and they are clustered together. Lastly, the Hong Kong data set, similar to the Barcelona set, contains Polygons. However, the polygons in this data set are more scattered and they vary more in size. A thumbnail of the data set can be seen in Figure 11d.

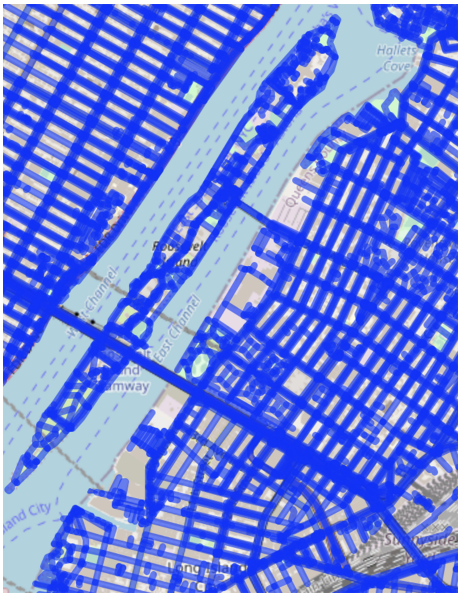
The data sets were processed with a python script that sanitized the set by iterating over all the features and removing any fields that did not contain information about the GeoJSON type or the coordinates. This was done so that there would not be any unnecessary data stored about the features that could affect the performance of the different operations in an inexplicable way. The script can be found in Appendix C.

Some problems arose when trying to insert the data sets with polygons into MongoDB's 2dsphere. This is due to restrictions that MongoDB enforces on polygons, namely that the outer ring of a polygon cannot self-intersect, the inner rings (holes) cannot intersect with the outer ring, and an inner ring cannot intersect with another inner ring. These are restrictions that the GeoJSON format does not contain, meaning that the R-tree implementation did not get these problems. To handle this, the script used data types and functions from the Shapely library for Python [54]. First, if the polygon contained any inner rings, the script transformed the coordinates into the datatype Polygon from Shapely. Then it ran a union function on the inner rings, creating a new Polygon or a MultiPolygon containing all the areas that the inner rings covered, merging any inner rings that intersected with each other. Then, it checked each of the new inner rings up against the outer ring's boundary to see if any of them intersected. If so, the inner ring would be subtracted from the outer ring resulting in a change in the outer ring's shape. This subtraction could lead to the outer ring being split into two outer rings. To deal with this the method for handling polygons returned a list of polygons that contained one if the subtraction did not lead to the outer ring being split, and more if it caused a split. Lastly, it checked whether or not the polygon was simple, meaning that the outer ring does not self-intersect. If it was not simple, the polygon was removed from the data set. When this was done, there were still some issues with a small number of polygons. The problem was that the insert in MongoDB failed due to self-intersection in the outer ring. The script had already checked for this and it was manually verified in the data set that this was actually not the case. But the coordinates were very close together, which may indicate that there is an inaccuracy in MongoDB's calculations that caused the failure. The polygons that caused these errors were removed from the data sets.

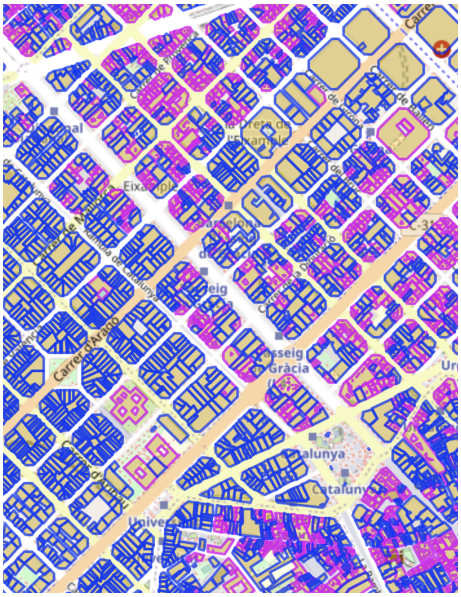
For the R-Tree, it was desirable to transform the coordinates from WSG84 format to a planar format. The script handles this as well, by using the transform function of the Python library pyproj [55]. To transform the coordinates, it was necessary to find the appropriate coordinate reference system to use. These were found using epsg.io, a website designed specifically for this purpose [56]. The mappings between the data sets and the coordinate reference systems can be seen in Table 6.



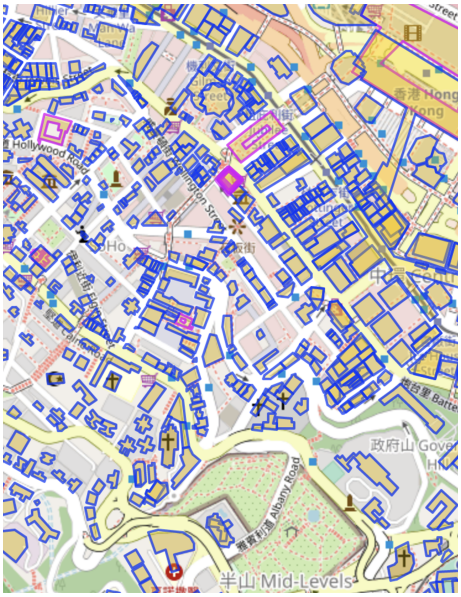
(a) Paris.



(b) New York.



(c) Barcelona.



(d) Hong Kong.

Figure 11: Data set thumbnails.

Table 6: Datasets with corresponding EPSG codes.

Data set	Grid system (EPSG)
Paris	27571
New York	2263
Barcelona	2062
Hong Kong	2326

The resulting data sets, now only containing fields with the GeoJSON type and coordinates, were considered ready for being inserted into the R-tree. Information about the size, middle coordinates, and the number of features per kilometer is listed in Table 7. Note that the coordinates in the New York data set are in feet, while the others are in meters. This is due to the coordinate referencing system used.

Table 7: Additional information about the data sets.

Data set	Size (megabyte)	Middle coordinates	Feature density
Paris	11.5	[601797.78, 1128578.58]	188.85 features/km^2
New York	52.5	[995114.33, 211032.65]	293.72 features/km^2
Barcelona	71,5	[1088352.63, 771624.53]	513.84 features/km^2
Hong Kong	21.5	[817296.83, 836973.61]	201.07 features/km^2

4.3 Insert Performance

The build speed of the R-tree was compared to the build speed of the 2dsphere index on the different types of geometries. The 2dsphere index is expected to be faster as it resides on the same server as the data and is a built-in index in MongoDB. The results in Table 8 show MongoDB's index outperforms the R-tree by orders of magnitude. The R-tree does not differ much in build speed on different types of geometries whereas MongoDB performs slower on more advanced geometries such as polygons. All data sets were fully inserted in less than five seconds on MongoDB while the R-tree version spent several minutes building for each collection. Since the R-tree does not differ much between the different geometries, it is plausible that most of the time spent is waiting for disk operations as each insert requires multiple I/O operations. The results from testing the branching factor, seen in Figure 12, support this theory as the build time decreases with a higher branching factor. A higher branching factor requires fewer I/O operations as there are fewer node splits caused by full nodes. Also, fewer documents are created as the number of internal nodes is reduced. Xiang et al. argued a branching factor of 96 was

the most performant due to the R-tree node document matching the block size of the database engine [10], but the results found here cannot be fully explained by block size. An R-tree with a branching factor of 32 will have a node document size of 3056 bytes which fits inside one block. It does not fully utilize the space on the block, but a branching factor of 64 produces a node document of 6064 bytes, bigger than the block size of MongoDB. Since a higher branching factor does not match a single block, but still builds faster, the block size is not likely to be an important factor. A node document's block size may have changed in the upgraded implementation as MongoDB may have changed how they store documents. Differences in the file system used for the database engine may also be related to the node document size and might be a part of the reason for the different node document sizes reported.

Table 8: Insert performance.

Data set	**R-tree**		**MongoDB**	
	Insert time (seconds)	**Docs/sec**	**Insert time (seconds)**	**Docs/sec**
Paris	2382.3	40.7	1.7	57800.4
New York	2823.7	42.2	2.7	43085.0
Barcelona	1474.9	45.2	3.6	18724.3
Hong Kong	451.1	46.8	1.2	17878.8

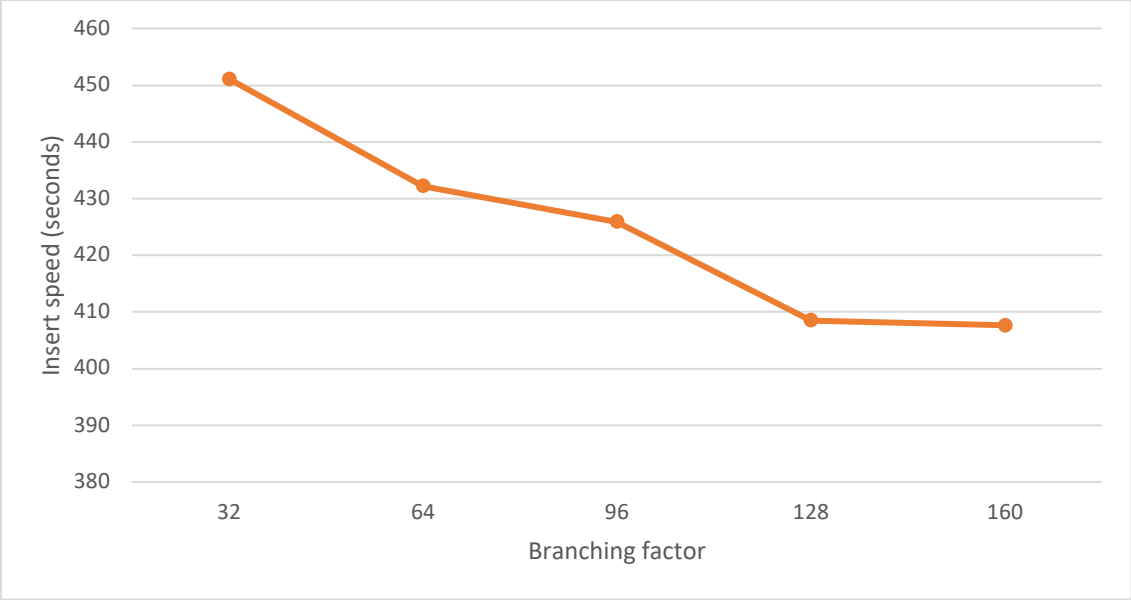


Figure 12: Hong Kong insert performance with different branching factors.

Table 9 displays key statistics for the R-tree constructed with different branching factors. The node entries display how many children a node has where each child counted contains data. All of the different branching factors achieve a high fill degree, but only a branching factor

of 64 achieves a satisfactory minimum count. The average storage utilization is acceptable, but node documents only containing two or three entries with data reduce performance. From the median value of the node count, it can be seen that more than half of the nodes achieve a minimum fill of at least 50%. Storage utilization is especially important for the MongoDB implementation as I/O operations are slow and in-memory processing is preferred. Currently, there is no parameter controlling the minimum fill degree for nodes, but such a variable could be introduced to improve the minimum node count and thus improve the storage utilization. However, a minimum fill degree has a significant effect on deletes as under full nodes require their entries to be re-inserted [16]. Therefore the minimum constraint cannot be too strict for applications where deletions are expected to occur frequently. An adjustable parameter would be preferable as it allows users to achieve better performance for their specific use case.

Table 9: Hong Kong key values with different branching factors.

Branching Factor	**Node entries**				**Number of Documents**	**Fill degree (%)**
	Avg	**Min**	**Max**	**Med**		
32	10	2	32	19	1072	64.55
64	41	13	64	38	538	62.64
96	61	5	96	58	346	64.35
128	82	3	128	74	259	64.09
160	100	2	158	93	212	62.74

The timing results were obtained with the `time` command, which also displays CPU usage. The CPU usage provides further insight into the performance of the indexes as the `mongoimport` command only utilizes 1 % of the CPU when inserting into the R-tree. Insertion with 2dsphere utilizes over 230 % of the CPU which means a parallel process is utilized to boost performance. CPU usage for the MongoDB processes, *mongod* and *mongos*, increase to between 20 %–30 % on building the R-tree index. The low CPU usage of `mongoimport` during R-tree building can be explained by the insert process in the R-tree module. Each document is processed one by one without optimizations for batch processing. The import process can therefore only send documents for processing and has to wait for each document to be processed by the R-tree module, making CPU utilization low. 2dsphere allows for the import command to do the batch insertion in parallel and is able to utilize multiple cores to work faster. This is an advantage of 2dsphere being a built-in index optimized for work with MongoDB tools and parallel processing.

Xiang et al. achieved an insertion speed of more than 400 documents per second for each of their data sets [10], but Table 8 shows the upgraded implementation is unable to achieve more

than 50 documents per second. A different approach may have been used to insert the data, such as looping over all the documents in a script and inserting them one by one, resulting in a different build time. This is more likely to affect the build time of 2dsphere than the R-tree as the R-tree does all the work sequentially and in a single process. The work will be moved from the import process to the routing server which then has to build the 2dsphere index, which is likely to result in a slower insert time as the documents are not batch processed. Problems with the LRU cache are seen as more likely because it has not been verified to work with the upgraded implementation. If the cache does not work as intended, the insert operation will be slower as R-tree nodes cannot be kept in memory, leading to a sharp increase in I/O operations. This is likely to cause a significant slowdown of the insert process, which matches the results obtained compared to Xiang et al. [10].

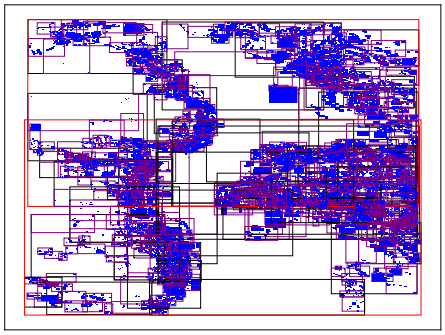
The MBRs constructed from the Hong Kong data set are shown in Figure 13a. The overlap is significant in the center area where there is a high density of buildings (marked with blue). Another issue is caused by the geography of Hong Kong. The selected map of Hong Kong is split in two by the ocean, which is observable as a large white segment in the middle of Figure 13a. However, the R-tree in the module is unable to recognize the large split and has thus produced many MBRs spanning across the water. Some queries will therefore have to consider candidate polygons which are across the ocean and therefore likely to be irrelevant to a user query from one side of the city. The R-tree implementation thus performs worse on segmented data sets such as Hong Kong where it generates unfavorable MBRs for queries. In Figure 13b, Figure 13c, and Figure 13d the overlap is caused by the high density of data. The densely colored areas indicate a high overlap between the lines of the MBRs which shows how the implementation introduces many overlapping MBRs when faced with high-density geographic data sets such as large cities. Overlapping can be reduced by swapping to an R*-tree [18].

4.4 Query Performance

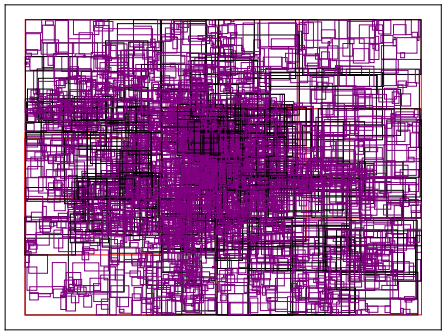
For query performance, the goal was to test the implemented queries presented in Section 3, against the equivalent query operations of MongoDB. Due to the way the R-tree is implemented, it is expected that it will be outperformed by the 2dsphere index. However, it is interesting to see how close the processing times are, as this will give an indication of whether or not an R-tree index is a valuable addition to MongoDB for handling geospatial data.

4.4.1 Intersect Query

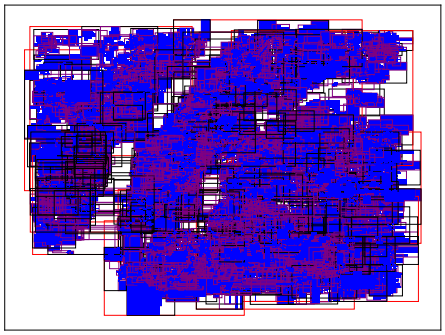
The first queries to be tested is the `$geoIntersect` query. These return the GeoJSON objects that intersect with a specified object. Meaning any object that is fully or partially contained



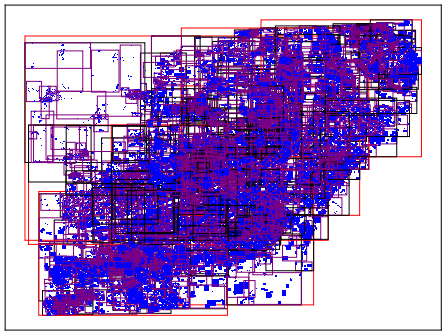
(a) Hong Kong



(b) Paris



(c) New York



(d) Barcelona

Figure 13: Plot of each collection's constructed MBRs.

within a given object. The object used for the `$geoIntersect` query can either be a Polygon or a MultiPolygon. The results from the test queries are presented in Figure 14, Figure 15 and Figure 16. The figures depict the results from the R-tree implementation and from MongoDB's 2dsphere index. The horizontal axis on the graphs represents the query window. A query window describes how much of the data set is being queried. The windows range from 0.5 % to 5.0 %, with intervals of 0.5 %. This is the same range and step size as Xiang et al. [10] used in their testing. However, the queries are not randomized but focused around a coordinate close to the center point of the data. A square was constructed around the point and gradually increased to fetch more documents. The vertical axis depicts the processing time in milliseconds. Each query with the given query window was performed 5 times and the average was used to create the graphs.

As seen in the results, the R-tree implementation follows a similar pattern in all of the data sets, except the Hong Kong set. It grows linearly when increasing the query window, drops at a certain point, and then keeps growing linearly. A possible reason is that the query MBR encloses most of the sub-trees and fewer Cartesian calculations are required. If the query MBR matches up with a full sub-tree, the entire sub-tree can be marked as a candidate and returned immediately, leading to fewer I/O operations. Also, if the query window does not include any overlapping MBRs, the query will be faster due to fewer path traversals and I/O operations.

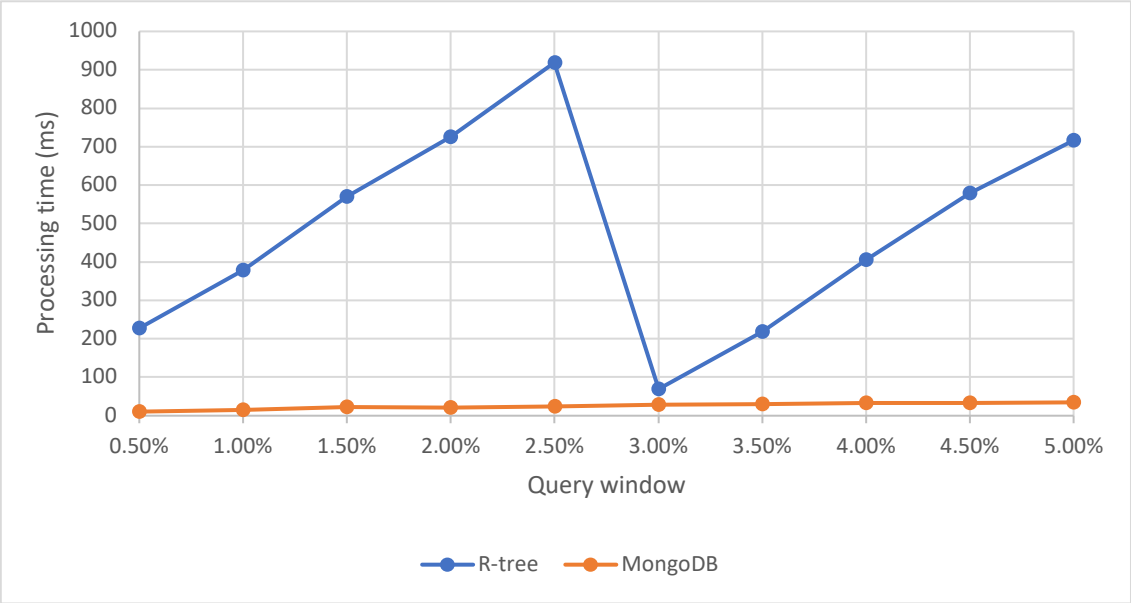


Figure 14: Intersect comparison – Point – Paris.

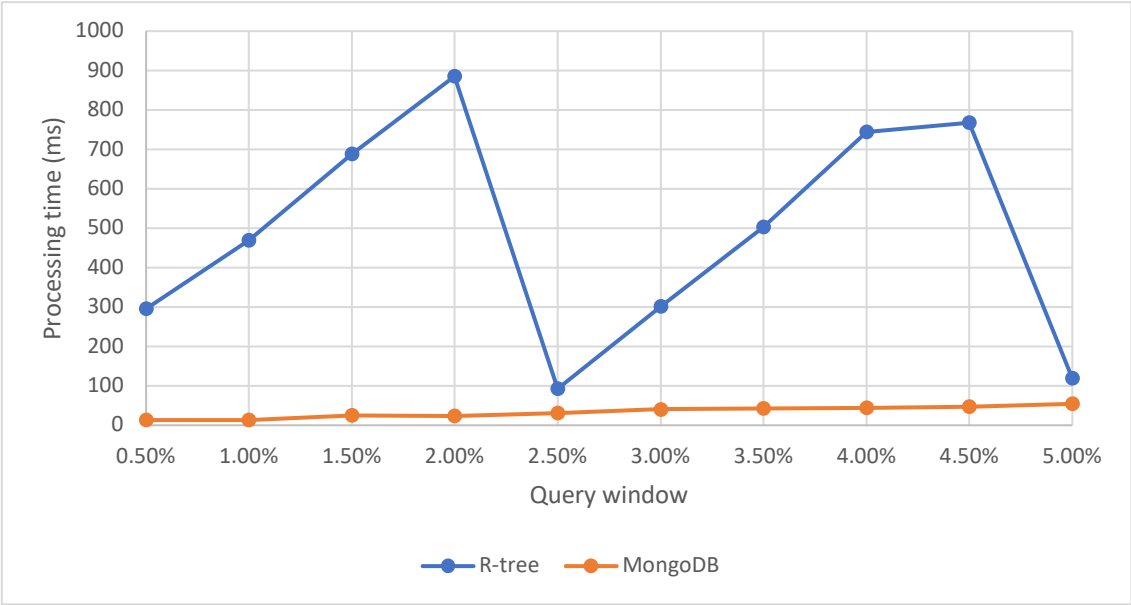
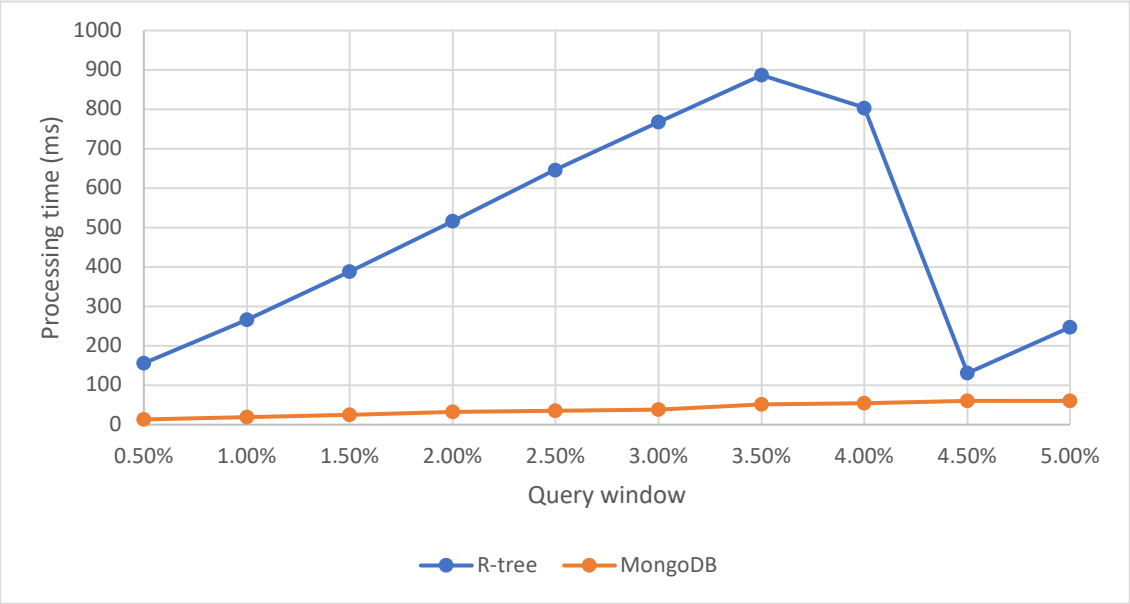


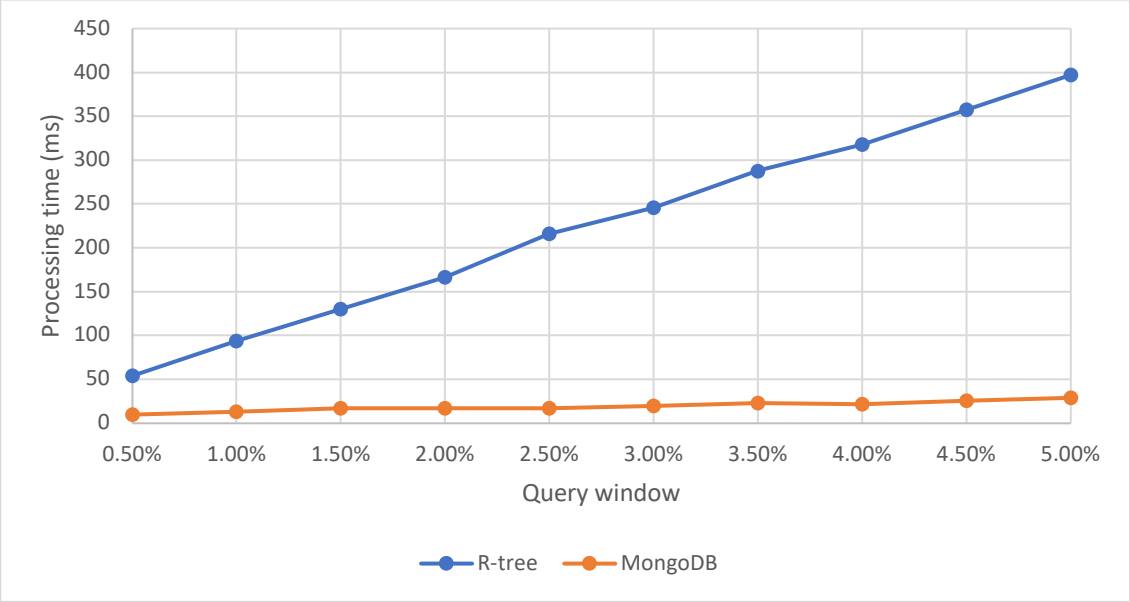
Figure 15: Intersect comparison – LineString – New York.

Furthermore, if the query window size matches well with the distribution of MBRs, the tree can more easily ignore the paths that do not intersect with the query window, leading to more efficient queries. For the Hong Kong data set, it can be seen that many MBRs stretch across the water where there are no buildings. Therefore it may require a larger query window to encapsulate sub-trees as the starting coordinate is not centered around the water area.

The R-tree is also significantly slower than the 2dsphere on all queries. Some of the processing delays can be attributed to I/O operations causing overhead. Furthermore, MongoDB stores the index on the same server as the data, while the R-tree implementation stores the index



(a) Barcelona.



(b) Hong Kong.

Figure 16: Intersect comparison.

in a collection on the config server. This means the mongos server first has to contact the config server and then the shard where the data is stored to retrieve the results from a query. Since the 2dsphere index resides on the same server as the data, it has an advantage over the R-tree module. The R-tree module is constructed to be portable to other database solutions (I/O independent), and therefore it does not have any optimizations with regard to storage or MongoDB-specific implementation details. The portability may lead to inefficient solutions for data retrieval as it must reconstruct a user query to query the R-tree index. All of these factors contribute to a longer processing time, but the R-tree is still able to produce results within a reasonable amount of time. When the query window is 3 % in the Paris intersect comparison in Figure 14, the processing time is only separated by 40ms. Similar results are found in Figure 16a and Figure 15 for specific query windows, meaning the R-tree can perform almost equally fast as the 2dsphere given the right conditions.

The drop in query time under given conditions may show that R-tree can scale well to larger data sets in the case where the R-tree produced has few overlapping MBRs. If the query time does not increase linearly with the size of the query window, it may perform well on larger queries whereas 2dsphere will have an increased processing time. A drawback of the unstable processing time is the lack of consistency. The processing time may be short, but Figure 16b shows it could also keep increasing linearly for a growing query window. In web applications where a user is awaiting a response, significant variation in the processing time may have a negative effect on the user experience as the app will have inconsistent response times.

The results presented differ from the ones presented by Xiang et al. [10] as they experienced a linearly increasing processing time on their queries. The speed is also significantly slower in the upgraded implementation, which may be caused by a faulty cache implementation as mentioned previously in Section 4.3. The I/O operations implemented in the previous version of the module may also use faster internal commands than the ones utilized in the upgraded version of this thesis, resulting in a smaller I/O overhead. The differences in test setup must also be considered as a different operating system has been used to run the tests for the upgraded version. Also, the test approach was different as Xiang et al. did not specify how they generated the queries [10]. The aforementioned factors may explain why the upgraded version produces slower results, but none of them explain the drop pattern seen in the processing time. Further testing is required to explain why some queries execute ten times faster while retrieving more data.

4.4.2 Within Query

The second query operation tested was the `$geoWithin` query. It was tested with the same query window range and the same step size as the `$geoIntersects` queries. This query returns the

GeoJSON objects that are fully within a specified object. The specified object is, as with the `$geoIntersects` queries, a Polygon or a MultiPolygon. Figure 17, Figure 18, and Figure 19 depict the average results of five iterations with each query window.

Figure 17: Within comparison – Point – Paris.

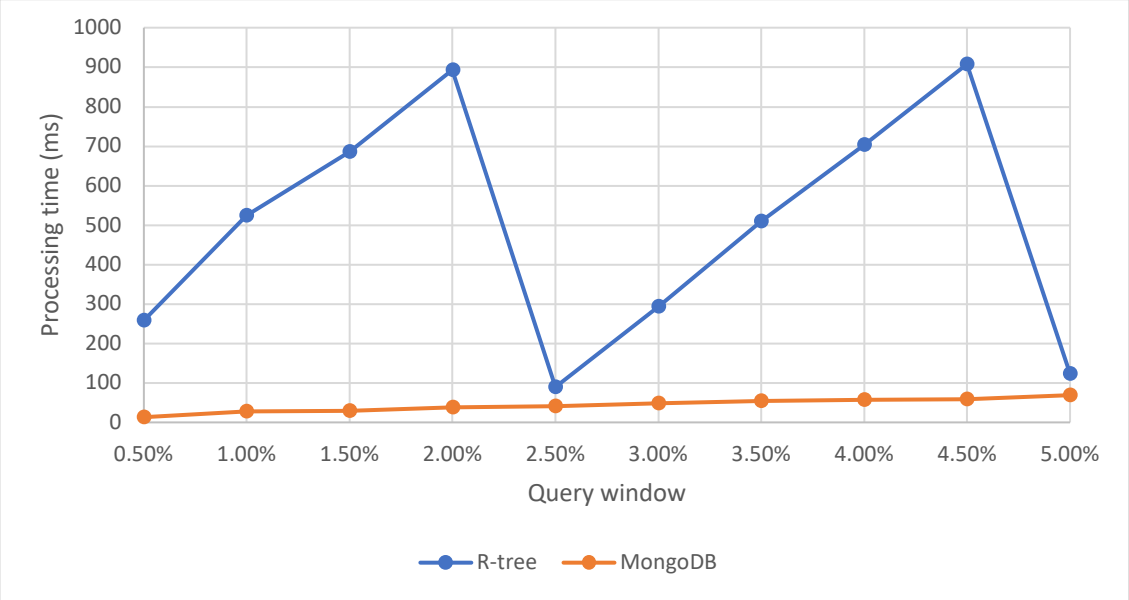
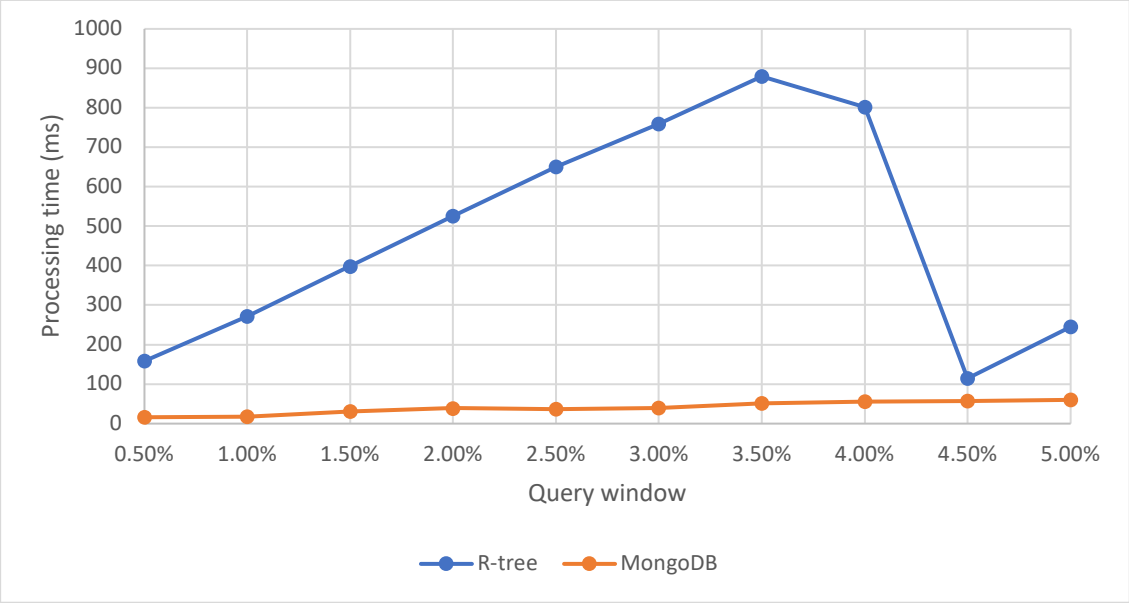
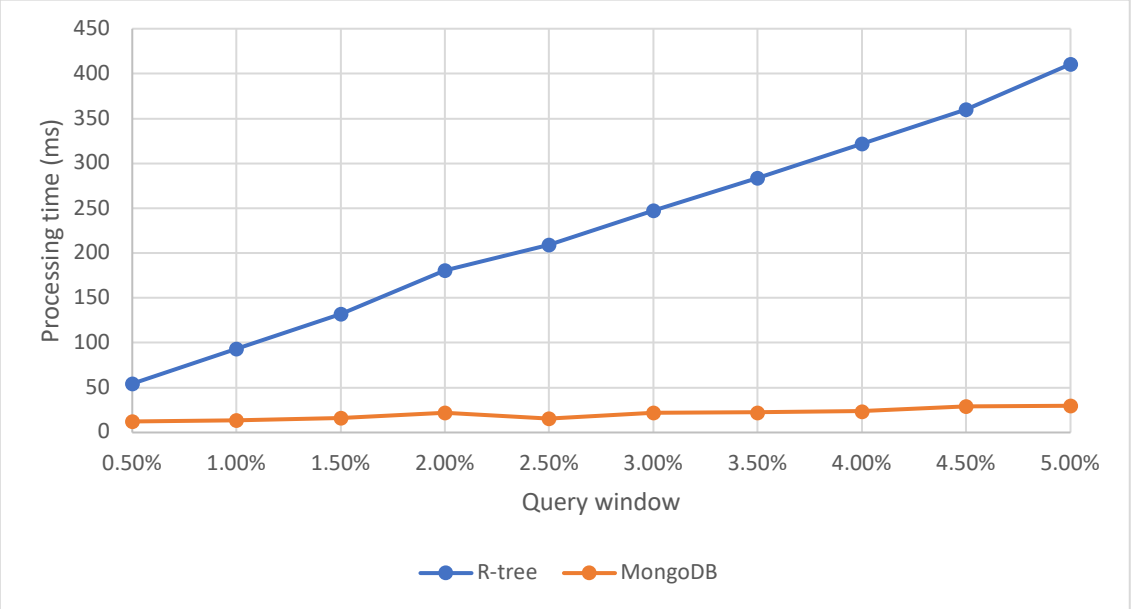


Figure 18: Within comparison – LineString – New York.

The results were expected to be similar to the intersect results as the queries are similar. All of the results from these queries follow a similar pattern as the ones for the `$geoIntersects` query. Since both of these queries retrieve GeoJSON objects that are within a given object, they have to do most of the same operations to retrieve the results. The `$geoWithin` is slightly slower than the `$geoIntersects` query. The reason is that within queries have to calculate



(a) Barcelona.



(b) Hong Kong.

Figure 19: Within comparison.

whether or not an object is fully within the specified object, while intersect queries do not have to do these calculations. Cartesian calculations are expensive and therefore a small increase in processing time is experienced. However, the difference between the processing times is small, which means that these calculations are not too expensive. Compared to the processing times of the `$geoWithin` query in MongoDB, the R-tree's processing times are slower. The reasons for this are the same as discussed with the `$geoIntersects` query.

4.4.3 Outliers

There are some outliers in the test results that are not visible in the graphs as they only affect the average result slightly. The R-tree occasionally produces results that are significantly faster than the average. As seen in Table 10, the first iteration with the R-tree is over 35 times faster than the rest of the iterations. The first iteration was performed right after the tests with the 3.5 % query window. It might be that the result from this query was cached in the LRU cache so the results were returned almost immediately with no additional processing required. However, the cache should then have been able to deliver the same processing time to the subsequent queries, but that was not the case. The built-in cache in the routing server is not able to drastically boost performance by a factor of more than 10. In fact, the impact from the built-in caching corresponds to a decrease in the processing time of around $50\,\text{ms} \pm 10\,\text{ms}$ for most queries as seen in Table 11. The only satisfactory explanation found is that the LRU cache sometimes works when certain conditions are met. If all nodes retrieved in the query reside in the cache it is possible to achieve times as low as 28.2 ms. More testing is required to discover the cause of this anomaly. Discovering why and how this happens is important as achieving this processing time consistently would make the performance of the R-tree index equal to that of the 2dsphere index on repeated queries. Of course, caching will not do all the work required as large queries will not fit in memory, but it is an important step towards better performance for the flattened R-tree in MongoDB.

Table 10: Barcelona within query with query window 4.0 %.

Iteration	R-tree	MongoDB
1	28.2 s	97.2 s
2	994.5 s	49.5 s
3	999.0 s	45.8 s
4	998.4 s	40.1 s
5	989.3 s	42.6 s

Table 11: Barcelona within query with query window 1.0 %.

Iteration	R-tree	MongoDB
1	314.5 s	31.5 s
2	261.9 s	11.4 s
3	260.3 s	11.5 s
4	259.6 s	16.6 s
5	257.6 s	14.9 s

4.5 I/O Speed

The Paris and New York data sets were also tested on a desktop machine running Ubuntu 22.04 OS with different hardware than the MacBook Air used for testing otherwise. The results can be seen in Table 12. The desktop machine is approximately twice as fast as the MacBook Air. The CPU and memory usage was found to be similar when testing, meaning the difference is not likely to be caused by CPU or memory differences. However, the desktop was equipped with a Samsung 980 Pro SSD which, according to Samsung, can deliver a sequential read speed of 7000 MB/s and a sequential write speed of 5100 MB/s [57]. The desktop computer is assumed to come close to those speeds as it is equipped with similar parts to those used by Samsung. To compare the read and write capabilities of the MacBook Air to the Samsung SSD, a benchmark test was executed. The MacBook Air delivered a read speed of 2976 MB/s and a write speed 2527 MB/s, which indicates the results are caused by a difference in I/O speed. The effect of write and read speed on secondary storage was expected to be significant since, as explained in Section 3, the bottleneck of the implementation is the insertion and retrieval of index nodes. Thus, the results are as expected with faster secondary storage.

Table 12: Desktop vs. MacBook Air insert speed.

Data set	MacBook Air		Desktop Ubuntu	
	Insert time (seconds)	Docs/sec	Insert time (seconds)	Docs/sec
Paris	2382.3	40.7	1100.0	85.6
New York	2823.7	42.2	1390.7	88.0

5 Conclusion

This thesis aimed to implement and evaluate a flattened R-tree index for geospatial data in MongoDB version 6.3.0, and compare it to MongoDB's 2dsphere index. The implementation presented is a modified version of Xiang et al.'s [10] flattened R-tree module for MongoDB version 3.2. The integration between the R-tree module and MongoDB required considerable adaptation due to significant upgrades in MongoDB since their work.

The performance of the implemented R-tree proved to be slower than the 2dsphere index for all query operations tested. The reason for this was found to most likely be the implementation of the R-tree rather than the R-tree structure itself. However, the R-tree showed promising results when the query window was of an optimal size. This was likely due to fewer I/O operations. Additionally, the build time of the R-tree index was way higher than that of the 2dsphere index, showing that there is still work to be done on improving the build speed before the implementation is usable.

In the broad field of NoSQL databases, this research aimed to test if an R-tree could be used as an indexing structure akin to SQL systems like PostgreSQL. Since R-trees are known to handle multidimensional data better than B-trees, an optimally implemented R-tree for MongoDB could improve the performance of geospatial queries. The current study, however, does not verify this presumption, emphasizing the need to improve the implementation strategy. The most significant contribution from the implementation is the possibility of inserting Cartesian coordinates as GeoJSON objects, which is currently not possible in MongoDB. This is an important step in providing better support for spatial data in MongoDB as it should be able to support multiple spatial reference systems for different use cases.

The study faced several constraints, including the complexity of MongoDB's system, the use of relatively small data sets, the limited number of query operations implemented, and the reliance on a single prior implementation. All these factors may have influenced the results achieved and are important considerations for further research.

To summarize, the findings in this thesis show that an R-tree index implemented as a module on the side of MongoDB does not match the performance of the 2dsphere. Yet, its potential was clear under certain query conditions which can indicate that with an optimized implementation, the R-tree could improve MongoDB's geospatial query performance. This study emphasizes not just the choice of index structure, but also its optimal implementation, thus paving the way for future research on this subject. This underlines the potential of the R-tree structure and the compelling need for a specialized implementation in MongoDB for harnessing this potential fully.

To see the source code for the implementation, see the GitHub link in Appendix E.

6 Further Work

This section will cover some proposals for further work that can be done to improve the R-tree implementation in MongoDB.

6.1 Near Query

Due to the limited scope of the project, the broken `$near` query was not fixed. However, finding the nearest neighbors of a geometry is an important spatial query, e.g. finding restaurants near you. The underlying operations have been implemented so a client can execute the `$near` query, but the results returned are not entirely correct and contain some invalid results. Thus, the future work will be to implement an efficient, functioning algorithm for returning the query results in batches such as the algorithm proposed by Hjaltason et al. [58]. Fixing the query has a high priority compared to other suggested future work as the implementation is not complete without it.

6.2 Optimize Implementation for MongoDB

As mentioned in Section 4, the current implementation leads to a lot of I/O overhead. The overhead is a significant factor in the slow processing times experienced compared to the 2dsphere. With more time and research, it could be possible to implement the R-tree index directly in MongoDB alongside the other indexes provided. This will eliminate the extra communication between the servers required in the implementation presented here. Since R-trees are designed for handling spatial data, it is reasonable to assume that a proper implementation would outperform the 2dsphere on more advanced spatial calculations, since the 2dsphere is a B-tree and has been shown to perform better on simpler geometries.

6.3 Caching

As mentioned in Section 4, the cache is inconsistent and does not function correctly. To provide stable performance on repeated queries and improve build time, the cache must be updated to work consistently with the newer version. Different caching techniques should be tested to find the best-performing cache as the LRU cache presented by Xiang et al. [10] may not be the most optimal solution. A different caching approach is presented by Jensen et al. where operations are buffered in main memory and flushed to the disk [59]. An adaption of their cache for the

flattened R-tree could be interesting to test as their benchmarks show the cache performs better than an LRU-cache.

6.4 Improving Build Speed

The current process of building the R-tree index is painfully slow, with over 40 minutes for the larger data sets used in the tests. Considering these data sets are on the smaller side, the build speed renders the R-tree unusable for applications with big data sets. Reducing I/O operations required to build the tree will improve the build time, but will not solve the problem of low CPU usage mentioned in Section 4. A parallelized approach is necessary to utilize more of the CPU and thus boost build speed. A parallel bulk-loading algorithm based on the Map-Reduce framework is presented by Achakeev et al. [60]. Luo et al. proposes another parallel approach utilizing the GPU to achieve fast parallelism [61]. Implementing such parallel bulk-loading algorithms could drastically improve the build speed of the R-tree and be utilized in combination with the `mongoimport` command.

Bibliography

[1] A. Eldawy and M. F. Mokbel, 'The era of big spatial data', in *2015 31st IEEE International Conference on Data Engineering Workshops*, IEEE, 2015, pp. 42–49.

[2] R. R. Vatsavai, A. Ganguly, V. Chandola, A. Stefanidis, S. Klasky and S. Shekhar, 'Spatiotemporal data mining in the era of big spatial data: Algorithms and applications', in *Proceedings of the 1st ACM SIGSPATIAL international workshop on analytics for big geospatial data*, 2012, pp. 1–10.

[3] R. S. A. Usmani, I. A. T. Hashem, T. R. Pillai, A. Saeed and A. M. Abdullahi, 'Geographic information system and big spatial data: A review and challenges', *International Journal of Enterprise Information Systems (IJEIS)*, vol. 16, no. 4, pp. 101–145, 2020.

[4] *Apache hadoop*. [Online]. Available: https://hadoop.apache.org/ (visited on 28th May 2023).

[5] A. Eldawy and M. F. Mokbel, 'Spatialhadoop: A mapreduce framework for spatial data', in *2015 IEEE 31st international conference on Data Engineering*, IEEE, 2015, pp. 1352–1363.

[6] R. T. Whitman, M. B. Park, S. M. Ambrose and E. G. Hoel, 'Spatial indexing and analytics on hadoop', in *Proceedings of the 22nd ACM SIGSPATIAL international conference on advances in geographic information systems*, 2014, pp. 73–82.

[7] L. Alarabi, M. F. Mokbel and M. Musleh, 'St-hadoop: A mapreduce framework for spatio-temporal data', *GeoInformatica*, vol. 22, pp. 785–813, 2018.

[8] *Mongodb*. [Online]. Available: https://www.mongodb.com/ (visited on 28th May 2023).

[9] *Apache hbase*. [Online]. Available: https://hbase.apache.org/ (visited on 28th May 2023).

[10] L. Xiang, J. Huang, X. Shao and D. Wang, 'A mongodb-based management of planar spatial data with a flattened r-tree', *ISPRS International Journal of Geo-Information*, vol. 5, no. 7, p. 119, 2016.

[11] S. Nishimura, S. Das, D. Agrawal and A. El Abbadi, 'Md-hbase: A scalable multi-dimensional data infrastructure for location aware services', in *2011 IEEE 12th International Conference on Mobile Data Management*, IEEE, vol. 1, 2011, pp. 7–16.

[12] *Postgis*. [Online]. Available: https://postgis.net/ (visited on 23rd May 2023).

[13] *PostgreSQL*. [Online]. Available: https://www.postgresql.org/ (visited on 1st Mar. 2023).

[14] solidIT consulting & software development gmbh, *DB-engines ranking*. [Online]. Available: https://db-engines.com/en/ranking_trend/spatial+dbms (visited on 1st Mar. 2023).

[15] T. P. D. Group. 'Postgis 3.3.3dev manual'. (2023), [Online]. Available: https://postgis.net/docs/using_postgis_dbmanagement.html (visited on 23rd May 2023).

[16] A. Guttman, 'R-trees: A dynamic index structure for spatial searching', in *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, 1984, pp. 47–57.

[17] T. Sellis, N. Roussopoulos and C. Faloutsos, 'The r+-tree: A dynamic index for multi-dimensional objects.', Tech. Rep., 1987.

[18] N. Beckmann, H.-P. Kriegel, R. Schneider and B. Seeger, 'The r*-tree: An efficient and robust access method for points and rectangles', in *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, 1990, pp. 322–331.

[19] D. Guo and E. Onstein, 'State-of-the-art geospatial information processing in nosql databases', *ISPRS International Journal of Geo-Information*, vol. 9, no. 5, p. 331, 2020.

[20] S. Schmid, E. Galicz and W. Reinhardt, 'Performance investigation of selected sql and nosql databases', in *Proceedings of the AGILE*, 2015, pp. 1–5.

[21] S. Agarwal and K. Rajan, 'Analyzing the performance of nosql vs. sql databases for spatial and aggregate queries', in *Free and Open Source Software for Geospatial (FOSS4G) Conference Proceedings*, vol. 17, 2017, p. 4.

[22] D. Bartoszewski, A. Piorkowski and M. Lupa, 'The comparison of processing efficiency of spatial data for postgis and mongodb databases', in *International Conference: Beyond Databases, Architectures and Structures*, Springer, 2019, pp. 291–302.

[23] R. Elmasri and S. B. Navathe, *Fundamentals of Database Systems, 7th edition*. Pearson, 2016, pp. 601, 622–630, 633–634.

[24] Y. Manolopoulos, A. N. Papadopoulos and Y. Theodoridis, *R-Trees: Theory and Applications: Theory and Applications*. Springer Science & Business Media, 2006.

[25] J. L. Bentley, 'Multidimensional binary search trees used for associative searching', *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[26] M. Skrodzki, *Neighborhood data structures, manifold properties, and processing of point set surfaces*. Freie Universitaet Berlin (Germany), 2019.

[27] F. P. Preparata and M. I. Shamos, *Computational geometry: an introduction*. Springer Science & Business Media, 2012.

[28] *What is nosql?* [Online]. Available: https://www.mongodb.com/nosql-explained (visited on 27th Feb. 2023).

[29] C. Strauch, U.-L. S. Sites and W. Kriha, 'Nosql databases', *Lecture Notes, Stuttgart Media University*, vol. 20, no. 24, p. 79, 2011.

[30] *Nosql vs. sql databases*. [Online]. Available: https://www.mongodb.com/nosql-explained/nosql-vs-sql (visited on 27th Feb. 2023).

[31] *Manifesto for agile software development*. [Online]. Available: https://agilemanifesto.org/ (visited on 27th Feb. 2023).

[32] *Understanding the different types of nosql databases*. [Online]. Available: https://www.mongodb.com/scale/types-of-nosql-databases (visited on 27th Feb. 2023).

[33] *Our customers*. [Online]. Available: https://www.mongodb.com/who-uses-mongodb (visited on 27th Feb. 2023).

[34] *Bson types*. [Online]. Available: https://www.mongodb.com/docs/manual/reference/bson-types/ (visited on 28th Feb. 2023).

[35] *Mongodb indexes*. [Online]. Available: https://www.mongodb.com/docs/manual/indexes/ (visited on 29th May 2023).

[36] *Neo4j documentation*. [Online]. Available: https://neo4j.com/docs/getting-started/ (visited on 11th May 2023).

[37] E. F. Codd, 'A relational model of data for large shared data banks', *Communications of the ACM*, vol. 13, no. 6, pp. 377–387, 1970.

[38] D. D. Chamberlin and R. F. Boyce, 'Sequel: A structured english query language', in *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, 1974, pp. 249–264.

[39] PostgreSQL Development Team, *PostgreSQL Documentation*. [Online]. Available: https://www.postgresql.org/docs/ (visited on 1st Mar. 2023).

[40] R. L. Church, 'Geographical information systems and location science', *Computers & Operations Research*, vol. 29, no. 6, pp. 541–562, 2002, Location Analysis, ISSN: 0305-0548. DOI: https://doi.org/10.1016/S0305-0548(99)00104-5. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0305054899001045.

[41] S. C. Carlson. 'Topology'. (2023), [Online]. Available: https://www.britannica.com/science/topology (visited on 6th Mar. 2023).

[42] W. F. Wieczorek and A. M. Delmerico, 'Geographic information systems', *WIREs Computational Statistics*, vol. 1, no. 2, pp. 167–186, 2009. DOI: https://doi.org/10.1002/wics.21. eprint: https://wires.onlinelibrary.wiley.com/doi/pdf/10.1002/wics.21. [Online]. Available: https://wires.onlinelibrary.wiley.com/doi/abs/10.1002/wics.21.

[43] *Spatial reference systems*. [Online]. Available: https://postgis.net/docs/using_postgis_dbmanagement.html#spatial_ref_sys (visited on 6th Mar. 2023).

[44] H. Butler, M. Daly, A. Doyle, S. Gillies, T. Schaub and S. Hagen, *The GeoJSON Format*, RFC 7946, Aug. 2016. DOI: 10.17487/RFC7946. [Online]. Available: https://www.rfc-editor.org/info/rfc7946.

[45] *S2 geometry*. [Online]. Available: https://s2geometry.io/ (visited on 18th May 2023).

[46] *S2 cell statistics*. [Online]. Available: https://s2geometry.io/resources/s2cell_statistics (visited on 19th May 2023).

[47] *Geospatial queries*. [Online]. Available: https://www.mongodb.com/docs/manual/geospatial-queries/ (visited on 20th Mar. 2023).

[48] *Neo4j spatial v0.24-neo4j-3.1.4*. [Online]. Available: https://neo4j-contrib.github.io/spatial/0.24-neo4j-3.1/index.html (visited on 24th Mar. 2023).

[49] J. M. Hellerstein, J. F. Naughton and A. Pfeffer, 'Generalized search trees for database systems', 1995.

[50] O. G. Consortium. 'Well-known text representation of coordinate reference systems'. (2019), [Online]. Available: https://www.opengis.net/doc/is/wkt-crs/2.0.6 (visited on 23rd May 2023).

[51] Y. Li, D. Kim and B.-S. Shin, 'Geohashed spatial index method for a location-aware wban data monitoring system based on nosql', *Journal of Information Processing Systems*, vol. 12, no. 2, pp. 263–274, 2016.

[52] *Deploy a sharded cluster*. [Online]. Available: https://www.mongodb.com/docs/manual/tutorial/deploy-shard-cluster/#deploy-a-sharded-cluster (visited on 14th May 2023).

[53] *Overpass turbo*. [Online]. Available: https://overpass-turbo.eu/ (visited on 12th May 2023).

[54] *The shapely user manual*. [Online]. Available: https://shapely.readthedocs.io/en/stable/manual.html (visited on 12th May 2023).

[55] *Pyproj documentation*. [Online]. Available: https://pyproj4.github.io/pyproj/stable/index.html# (visited on 12th May 2023).

[56] *Epsg.io*. [Online]. Available: https://epsg.io/ (visited on 13th May 2023).

[57] *Samsung 980 Pro Product Page*. [Online]. Available: https://semiconductor.samsung.com/consumer-storage/internal-ssd/980pro/ (visited on 31st May 2023).

[58] G. R. Hjaltason and H. Samet, 'Distance browsing in spatial databases', *ACM Transactions on Database Systems (TODS)*, vol. 24, no. 2, pp. 265–318, 1999.

[59] C. S. Jensen, S. Šaltenis and L. Biveinis, 'Main-memory operation buffering for efficient r-tree update', 2007.

[60] D. Achakeev, M. Seidemann, M. Schmidt and B. Seeger, 'Sort-based parallel loading of r-trees', in *Proceedings of the 1st ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data*, 2012, pp. 62–70.

[61] L. Luo, M. D. Wong and L. Leong, 'Parallel implementation of r-trees on the gpu', in *17th Asia and South Pacific Design Automation Conference*, IEEE, 2012, pp. 353–358.

Appendix

A Building MongoDB

Following is the building.md file from the MongoDB repository on GitHub. A slightly modified command had to be run instead of the ones listed under SCons, namely:

```
    $ python3 buildscripts/scons.py install-devcore -j6
     ↪  --disable-warnings-as-errors MONGO_VERSION=6.3.0
```

```
Building MongoDB
================

Please note that prebuilt binaries are available on
[mongodb.org](http://www.mongodb.org/downloads) and may be the easiest
way to get started, rather than building from source.

To build MongoDB, you will need:

* A modern C++ compiler capable of compiling C++20. One of the following
 ↪  is required:
    * GCC 11.3 or newer
    * Clang 12.0 (or Apple XCode 13.0 Clang) or newer
    * Visual Studio 2022 version 17.0 or newer (See Windows section below
     ↪  for details)
* On Linux and macOS, the libcurl library and header is required. MacOS
 ↪  includes libcurl.
    * Fedora/RHEL - `dnf install libcurl-devel`
    * Ubuntu/Debian - `libcurl-dev` is provided by three packages.
     ↪  Install one of them:
      * `libcurl4-openssl-dev`
      * `libcurl4-nss-dev`
      * `libcurl4-gnutls-dev`
    * On Ubuntu, the lzma library is required. Install `liblzma-dev`
    * On Amazon Linux, the xz-devel library is required. `yum install
     ↪  xz-devel`
* Python 3.7.x and Pip modules:
```

  * See the section "Python Prerequisites" below.
* About 13 GB of free disk space for the core binaries (`mongod`, `mongos`, and `mongo`) and about 600 GB for the install-all target.

MongoDB supports the following architectures: arm64, ppc64le, s390x, and x86-64. More detailed platform instructions can be found below.

MongoDB Tools

The MongoDB command line tools (`mongodump`, `mongorestore`, `mongoimport`, `mongoexport`, etc) have been rewritten in [Go](http://golang.org/) and are no longer included in this repository.

The source for the tools is now available at [mongodb/mongo-tools](https://github.com/mongodb/mongo-tools).

Python Prerequisites

In order to build MongoDB, Python 3.7+ is required, and several Python modules must be installed. Python 3 is included in macOS 10.15 and later. For earlier macOS versions, Python 3 can be installed using Homebrew or MacPorts or similar.

To install the required Python modules, run:

```
$ python3 -m pip install -r etc/pip/compile-requirements.txt
```

Installing the requirements inside a python3 based virtualenv dedicated to building MongoDB is recommended.

Note: In order to compile C-based Python modules, you'll also need the Python and OpenSSL C headers. Run:

```
* Fedora/RHEL - `dnf install python3-devel openssl-devel`
* Ubuntu (20.04 and newer)/Debian (Bullseye and newer) - `apt install
 ↪  python-dev-is-python3 libssl-dev`
* Ubuntu (18.04 and older)/Debian (Buster and older) - `apt install
 ↪  python3.7-dev libssl-dev`
```

SCons

```
If you only want to build the database server `mongod`:

    $ python3 buildscripts/scons.py install-mongod

***Note***: For C++ compilers that are newer than the supported
version, the compiler may issue new warnings that cause MongoDB to
fail to build since the build system treats compiler warnings as
errors. To ignore the warnings, pass the switch
`--disable-warnings-as-errors` to scons.

    $ python3 buildscripts/scons.py install-mongod
     ↪  --disable-warnings-as-errors

***Note***: On memory-constrained systems, you may run into an error such
 ↪  as `g++: fatal error: Killed signal terminated program cc1plus`. To
 ↪  use less memory during building, pass the parameter `-j1` to scons.
 ↪  This can be incremented to `-j2`, `-j3`, and higher as appropriate to
 ↪  find the fastest working option on your system.

    $ python3 buildscripts/scons.py install-mongod -j1

To install `mongod` directly to `/opt/mongo`

    $ python3 buildscripts/scons.py DESTDIR=/opt/mongo install-mongod

To create an installation tree of the servers in `/tmp/unpriv` that
can later be copied to `/usr/priv`
```

```
    $ python3 buildscripts/scons.py DESTDIR=/tmp/unpriv PREFIX=/usr/priv
     ↪  install-servers
```

If you want to build absolutely everything (`mongod`, `mongo`, unit tests, etc):

```
    $ python3 buildscripts/scons.py install-all-meta
```

SCons Targets

The following targets can be named on the scons command line to build and install a subset of components:

* `install-mongod`
* `install-mongos`
* `install-core` (includes *only* `mongod` and `mongos`)
* `install-servers` (includes all server components)
* `install-devcore` (includes `mongod`, `mongos`, and `jstestshell` (formerly `mongo` shell))
* `install-all` (includes a complete end-user distribution and tests)
* `install-all-meta` (absolutely everything that can be built and installed)

NOTE: The `install-core` and `install-servers` targets are *not* guaranteed to be identical. The `install-core` target will only ever include a minimal set of "core" server components, while `install-servers` is intended for a functional end-user installation. If you are testing, you should use the `install-core` or `install-devcore` targets instead.

Where to find Binaries

The build system will produce an installation tree into

`$DESTDIR/$PREFIX`. `DESTDIR` by default is `build/install` while `PREFIX` is by default empty. This means that with all of the listed targets all built binaries will be in `build/install/bin` by default.

Windows

Build requirements:
* Visual Studio 2022 version 17.0 or newer
* Python 3.7

Or download a prebuilt binary for Windows at www.mongodb.org.

Debian/Ubuntu

To install dependencies on Debian or Ubuntu systems:

```
    # apt-get install build-essential
```

OS X

Install Xcode 13.0 or newer.

FreeBSD

Install the following ports:

  * `devel/libexecinfo`
  * `lang/llvm70`
  * `lang/python`

Add `CC=clang12 CXX=clang++12` to the `scons` options, when building.

OpenBSD

Install the following ports:

  * `devel/libexecinfo`
  * `lang/gcc`
  * `lang/python`

B Intersect Query

A python script to run a $geoIntersects query.

```
from datetime import datetime, time
from pymongo.mongo_client import MongoClient
from pymongo.server_api import ServerApi

# Local connection
uri = "localhost:27019"

# Set the Stable API version when creating a new client
client = MongoClient(uri, server_api=ServerApi('1'))
db = client.geodb
collection = db.paris


def main():
    count = 0
    maxCount = 97846

    try:
        centerX, centerY = 600511, 1128382
        size = 1010
        lowX, highX = centerX - size, centerX + size
        lowY, highY = centerY - size, centerY + size
        for i in range(5):
            results = []

            startDate = datetime.now()
            cursor = collection.find({"position": {"geoIntersects":
             ↪  {"geometry": {"type": "Polygon", "coordinates": [[
                [
                    lowX,
                    lowY
                ],
                [
                    highX,
```

```
                    lowY
                ],
                [
                    highX,
                    highY
                ],
                [
                    lowX,
                    highY
                ],
                [
                    lowX,
                    lowY
                ]
            ]]}}}})
            for doc in cursor:
                results.append(doc)
            endDate = datetime.now()
            difference = endDate - startDate
            if count == 0:
                count = len(results)
            resultNumber = str(difference.microseconds/1000).replace(".",
             ↪  ",")
            print(resultNumber, end=" ")


    except Exception as e:
        print(e)
    print()
    print(count)
    print(count/maxCount)

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

C Format data script

Python script for sanitizing data sets so they could be used for the `mongoimport` method

```
import argparse
import json
import multiprocessing
from functools import partial

import geojson
from pyproj import CRS, Transformer
from shapely.geometry import MultiPolygon, Polygon, shape
from shapely.ops import orient, unary_union

EPSG = {
    "Paris": 27571,
    "NewYork": 2263,
    "Barcelona": 2062,
    "HongKong": 2326}


geojson_types = ["Point", "LineString", "Polygon"]
field_name = ["geometry", "position"]


def create_argument_parser():
    parser = argparse.ArgumentParser(
        prog='FormatData',
        description="Formats geodata to suit the mongodb import script",
    )
    parser.add_argument('filename', help="Path of the input file")
    parser.add_argument('-t', '--type', choices=geojson_types,
                        required=True, help="GeoJSON of the input data.
                        ↪ REQUIRED")
    parser.add_argument('-m', '--mongo', action='store_true',
                        help="Flag to set if the data is gonna be used by
                        ↪ original MongoDB, if so the geometry field is
                        ↪ replaced with 'location'")
```

```
    parser.add_argument('-e',
                        '--epsg', choices=['Paris', 'NewYork',
                        ↪  'Barcelona', 'HongKong'], required=False,
                        ↪  help="Specifies the EPSG to convert to")
    return parser.parse_args()


def format(filename, type, mongo, epsg):
    with open(filename, 'r') as openfile:
        input_file = json.load(openfile)

        pool = multiprocessing.Pool(6)

        modify_objects_partial = partial(
            modify_objects, type=type, mongo=mongo, epsg=epsg)

        new_objects = []
        if (type == geojson_types[2]):
            result = pool.map(
                modify_objects_partial, input_file["features"])
            for obj in result:
                if obj is not None:
                    new_objects.extend(obj)

        else:
            new_objects = pool.map(
                modify_objects_partial, input_file["features"])

        new_objects = [obj for obj in new_objects if obj is not None]

        pool.close()
        pool.join()

    with open('formatted_'+('mongo_' if mongo else
     ↪  'epsg'+str(EPSG[epsg])+'_')+filename, 'w') as outfile:
        json.dump(new_objects, outfile, indent=2)
```

```
def modify_objects(object, type, mongo, epsg):
    for field in object:
        if field == 'geometry' and object[field]["type"] == type:
            if type == geojson_types[0]:
                return handle_points(
                    field, object, type, mongo, epsg)
            elif type == geojson_types[1]:
                return handle_line_string(field, object, type, mongo,
                 ↪  epsg)
            elif type == geojson_types[2]:
                return handle_polygon(
                    field, object, type, mongo, epsg)
    return None


def geodetic_to_cartesian(lon, lat, epsg):
    lonlat = CRS.from_user_input(4326)
    geocent = CRS.from_user_input(EPSG[epsg])
    transformer = Transformer.from_crs(
        crs_from=lonlat, crs_to=geocent)
    x, y = transformer.transform(lat, lon)
    return [round(x, 2), round(y, 2)]


def handle_points(field, object, type, mongo, epsg):
    coordinates = object[field]["coordinates"]
    if mongo:
        new_coordinates = coordinates
    else:
        new_coordinates = geodetic_to_cartesian(
            coordinates[0], coordinates[1], epsg)
    return {"type": "Feature", field_name[0] if mongo else field_name[1]:
     ↪  {"type": type, "coordinates": new_coordinates}}


def handle_line_string(field, object, type, mongo, epsg):
    new_coordinates = []
    for coordinate in object[field]["coordinates"]:
```

```
        if mongo:
            new_coordinates.append(coordinate)
        else:
            cartesian_coordinate = geodetic_to_cartesian(
                coordinate[0], coordinate[1], epsg)
            new_coordinates.append(cartesian_coordinate)
    return {"type": "Feature", field_name[0] if mongo else field_name[1]:
     ↪  {"type": type, "coordinates": new_coordinates}}


# Only use the outer polygon as inner polygons may overlap and that
 ↪  causes errors in MongoDB
def handle_polygon(field, object, type, mongo, epsg):
    polygons = []
    sanitized_polygons = process_polygons(object[field]["coordinates"])
    for sanitized_polygon in sanitized_polygons:
        new_coordinates = []
        for coordinates in sanitized_polygon:
            coordinates_to_add = []
            for coordinate in coordinates:
                if mongo:
                    coordinates_to_add.append(coordinate)
                else:
                    cartesian_coordinate = geodetic_to_cartesian(
                        coordinate[0], coordinate[1], epsg)
                    coordinates_to_add.append(cartesian_coordinate)
            new_coordinates.append(coordinates_to_add)
        outer_ring, *inner_ring = new_coordinates
        polygon = Polygon(outer_ring, inner_ring)
        if not polygon.is_simple:
            print("Polygon is not simple")
            return None
        if (not polygon.is_valid):
            print("something went wrong with this polygon", polygon,
             ↪  "\n")
            return None

        polygon = geojson.Polygon(new_coordinates)
```

```
        new_polygon = {"type": "Feature",
                       field_name[0] if mongo else field_name[1]:
                        ↪  polygon}
        polygons.append(new_polygon)
    return polygons


def process_polygons(polygon):
    if len(polygon) == 1:
        return [polygon]

    outer_ring, *inner_ring = polygon

    # Make outer and inner ring polygons
    outer_polygon = Polygon(outer_ring)

    inner_polygons = [Polygon(x) for x in inner_ring]

    inner_polygons = handle_inner_polygons(inner_polygons)

    new_polygons = handle_outer_ring(
        outer_polygon, inner_polygons)

    return [format_polygons(x, [*y]) for x, *y in new_polygons]


def format_polygons(outer_polygon, inner_polygons):
    outer_polygon = [[x, y] for x, y in outer_polygon.boundary.coords]
    new_inner_polygons = []
    for poly in inner_polygons:
        new_inner_polygons.append(list([x, y]
                                       for x, y in poly.boundary.coords))

    return [outer_polygon, *new_inner_polygons]


def handle_outer_ring(outer_polygon, inner_polygons):
```

```
    outer_polygon = orient(outer_polygon, sign=1.0)
    inner_indexes_to_remove = []
    for i, poly in enumerate(inner_polygons):
        orient(poly, sign=-1.0)
        if not outer_polygon.contains_properly(poly):
            outer_polygon = outer_polygon.difference(poly)
            if isinstance(outer_polygon, Polygon):
                outer_polygon = Polygon(outer_polygon.exterior)
            inner_indexes_to_remove.append(i)

    new_inner_polygons = [poly for index, poly in enumerate(
        inner_polygons) if index not in inner_indexes_to_remove]

    result = []

    if isinstance(outer_polygon, MultiPolygon):

        result = result + \
            [handle_outer_ring_v1(outer, new_inner_polygons)
             for outer in outer_polygon.geoms if outer is not None]

    elif isinstance(outer_polygon, Polygon):
        result.append([outer_polygon, *new_inner_polygons])

    return result


def handle_outer_ring_v1(outer, inner):
    outer_polygon = orient(outer, sign=1.0)
    inner_indexes_to_remove = []
    for i, poly in enumerate(inner):
        orient(poly, sign=-1.0)
        if not outer_polygon.contains_properly(poly):
            outer_polygon = outer_polygon.difference(poly)
            inner_indexes_to_remove.append(i)

    new_inner_polygons = [poly for index, poly in enumerate(
        inner) if index not in inner_indexes_to_remove]
```

```
    if isinstance(outer_polygon, MultiPolygon):
        return None

    elif isinstance(outer_polygon, Polygon):
        return [outer_polygon, *new_inner_polygons]


def handle_inner_polygons(inner_polygons):
    combined_inner_polygons = unary_union(inner_polygons)

    if isinstance(combined_inner_polygons, MultiPolygon):
        inner_ring = list(combined_inner_polygons.geoms)
    elif isinstance(combined_inner_polygons, Polygon):
        inner_ring = [combined_inner_polygons]

    return inner_ring


if __name__ == "__main__":
    args = create_argument_parser()
    format(args.filename, args.type, args.mongo, args.epsg)
```

D Results

Iterations	Data sets (seconds)			
R-tree	**Paris**	**New York**	**Barcelona**	**Hong Kong**
1	2377.1	2822.5	1538.7	461.3
2	2460.7	2864.7	1450.0	450.3
3	2349.1	2838.6	1463.8	455.1
4	2424.3	2710.1	1444.6	453.7
5	2300.4	2882.6	1477.2	435.1
MongoDB	**Paris**	**New York**	**Barcelona**	**Hong Kong**
1	2.1	2.8	3.6	1.2
2	1.6	2.7	3.5	1.3
3	1.6	2.8	3.5	1.1
4	1.6	2.7	3.6	1.2
5	1.6	2.7	3.5	1.2

Table 13: Raw insert results

Table 14: GeoWithin query Paris (results in milliseconds).

Iterations	Query Window Size (%)									
R-tree	**0.5**	**1.0**	**1.5**	**2.0**	**2.5**	**3.0**	**3.5**	**4.0**	**4.5**	**5.0**
1	261.8	438.7	614.2	767.9	944.0	133.1	265.6	437.7	613.0	786.7
2	208.5	361.1	524.6	711.9	885.0	43.2	202.3	370.2	559.9	698.8
3	209.6	364.3	523.5	706.0	879.4	47.3	208.3	377.2	552.3	693.1
4	207.7	364.3	576.3	695.9	887.6	46.6	204.6	462.9	644.2	670.8
5	207.9	366.0	524.8	714.3	880.7	41.0	211.2	377.8	526.5	736.0
MongoDB	**0.5**	**1.0**	**1.5**	**2.0**	**2.5**	**3.0**	**3.5**	**4.0**	**4.5**	**5.0**
1	20.6	30.6	33.9	41.3	38.4	44.5	58.5	57.4	51.6	49.9
2	9.8	12.6	20.0	11.3	16.7	16.5	18.3	22.8	27.7	34.2
3	7.2	7.1	8.6	10.0	13.4	16.6	18.6	22.8	25.9	23.3
4	7.3	5.8	8.5	9.7	12.6	16.1	23.5	26.4	28.2	24.3
5	6.2	5.6	8.8	9.7	15.1	19.1	18.5	20.7	23.7	24.1

Table 15: GeoWithin query New York (results in milliseconds).

Iterations	**Query Window Size (%)**									
R-tree	**0.5**	**1.0**	**1.5**	**2.0**	**2.5**	**3.0**	**3.5**	**4.0**	**4.5**	**5.0**
1	310.7	572.1	737.6	940.8	140.5	331.1	558.8	759.4	966.7	206.7
2	239.0	509.6	675.5	900.9	89.1	282.5	503.5	690.4	897.4	110.6
3	236.5	510.5	670.7	872.0	77.9	273.5	497.3	685.4	901.0	115.6
4	273.8	516.8	668.7	870.9	81.1	310.7	503.1	695.1	870.0	106.5
5	235.5	510.7	677.8	883.7	63.4	270.9	489.0	692.5	904.8	75.1
MongoDB	**0.5**	**1.0**	**1.5**	**2.0**	**2.5**	**3.0**	**3.5**	**4.0**	**4.5**	**5.0**
1	27.6	52.8	41.9	67.7	63.2	83.4	78.7	95.9	85.4	105.6
2	16.0	24.7	25.3	37.9	41.8	48.1	55.6	52.7	59.4	59.2
3	8.8	22.9	26.5	35.0	39.3	35.8	43.4	44.6	49.1	58.8
4	8.7	21.1	29.1	25.8	30.9	33.6	42.7	47.5	50.7	53.9
5	8.4	17.9	23.1	23.8	29.9	38.8	53.4	44.8	48.7	66.8

Table 16: GeoWithin query Barcelona (results in milliseconds).

Iterations	**Query Window Size (%)**									
R-tree	**0.5**	**1.0**	**1.5**	**2.0**	**2.5**	**3.0**	**3.5**	**4.0**	**4.5**	**5.0**
1	198.7	314.5	439.1	571.6	747.7	800.2	908.2	28.2	158.1	293.9
2	147.3	261.9	392.5	527.1	621.1	753.5	882.4	994.5	112.1	234.7
3	146.6	260.3	386.0	512.4	625.5	750.6	870.9	999.0	94.3	238.9
4	145.9	259.6	387.2	503.3	632.4	748.2	872.6	998.4	108.0	223.9
5	155.5	257.6	387.3	511.2	628.4	742.8	863.6	989.3	100.5	234.0
MongoDB	**0.5**	**1.0**	**1.5**	**2.0**	**2.5**	**3.0**	**3.5**	**4.0**	**4.5**	**5.0**
1	31.7	31.5	52.8	78.7	49.7	57.8	67.0	97.2	96.9	84.8
2	13.0	11.4	24.8	40.0	35.8	40.8	54.6	49.5	51.1	65.0
3	11.8	11.5	30.6	28.8	33.6	31.7	53.7	45.8	45.8	51.9
4	10.5	16.6	24.2	24.2	30.9	33.4	37.7	40.1	46.5	47.7
5	10.0	14.9	20.5	20.9	29.0	33.7	39.0	42.6	42.4	49.3

Table 17: GeoWithin query Hong Kong (results in milliseconds).

Iterations	**Query Window Size (%)**									
R-tree	**0.5**	**1.0**	**1.5**	**2.0**	**2.5**	**3.0**	**3.5**	**4.0**	**4.5**	**5.0**
1	93.0	132.7	171.4	256.9	251.8	284.8	329.1	362.8	402.5	430.4
2	45.4	83.8	124.6	134.2	194.8	238.9	273.7	312.8	349.6	414.7
3	45.0	79.2	121.0	176.5	196.2	234.5	271.9	314.7	351.8	396.0
4	45.0	85.5	121.3	168.4	195.8	242.6	271.2	310.7	350.0	386.8
5	44.9	83.4	120.8	167.0	206.3	235.9	271.2	307.5	346.9	424.5
MongoDB	**0.5**	**1.0**	**1.5**	**2.0**	**2.5**	**3.0**	**3.5**	**4.0**	**4.5**	**5.0**
1	20.9	28.5	31.6	43.2	32.9	37.3	35.2	43.5	45.8	46.4
2	10.5	10.9	10.2	24.5	10.8	16.6	17.9	16.9	23.2	24.7
3	17.2	8.3	9.3	22.8	10.2	16.2	16.0	16.3	29.7	30.1
4	5.5	10.7	18.3	9.3	10.6	16.9	21.5	22.0	24.6	25.7
5	6.1	7.7	11.5	9.3	11.6	22.7	20.8	19.1	22.3	22.1

Table 18: GeoIntersect query Paris (results in milliseconds).

Iterations	**Query Window Size (%)**									
R-tree	**0.5**	**1.0**	**1.5**	**2.0**	**2.5**	**3.0**	**3.5**	**4.0**	**4.5**	**5.0**
1	270.9	439.7	570.0	753.2	943.1	109.7	265.6	437.7	613.0	786.7
2	217.8	360.3	531.3	708.0	954.0	72.1	202.3	370.2	559.9	698.8
3	216.4	366.2	633.1	744.4	885.1	78.3	208.3	377.2	552.3	693.1
4	217.1	366.8	576.9	716.1	890.3	45.7	204.6	462.9	644.2	670.8
5	217.6	359.4	537.9	702.4	919.7	37.8	211.2	377.8	526.5	736.0
MongoDB	**0.5**	**1.0**	**1.5**	**2.0**	**2.5**	**3.0**	**3.5**	**4.0**	**4.5**	**5.0**
1	17.0	25.8	64.1	40.4	43.9	56.1	48.3	57.8	48.4	62.2
2	11.6	18.4	13.3	16.2	16.1	23.5	26.7	29.4	28.3	29.5
3	10.0	15.5	11.3	14.2	18.3	22.3	24.8	28.1	27.4	28.5
4	8.1	6.8	12.0	14.9	18.9	19.4	27.3	26.4	30.4	24.3
5	4.6	5.9	10.8	15.2	18.2	20.4	20.8	20.6	26.0	23.6

Table 19: GeoIntersect query New York (results in milliseconds).

Iterations	**Query Window Size (%)**									
R-tree	**0.5**	**1.0**	**1.5**	**2.0**	**2.5**	**3.0**	**3.5**	**4.0**	**4.5**	**5.0**
1	339.3	512.7	769.3	930.6	146.3	354.7	568.6	797.4	999.0	191.1
2	286.0	462.7	671.6	882.0	82.8	305.9	494.0	788.9	900.1	150.4
3	282.4	463.1	670.2	867.5	73.6	280.6	475.3	703.3	908.9	376.8
4	282.5	452.6	663.0	872.5	77.9	279.3	489.2	716.5	897.0	137.3
5	289.1	453.0	663.6	869.2	88.3	286.2	486.2	711.2	130.9	143.8
MongoDB	**0.5**	**1.0**	**1.5**	**2.0**	**2.5**	**3.0**	**3.5**	**4.0**	**4.5**	**5.0**
1	28.2	29.7	42.2	44.0	49.8	59.6	70.8	60.2	67.4	91.4
2	14.1	15.0	19.3	18.1	32.7	47.6	46.3	50.5	49.5	53.1
3	9.6	8.5	19.0	22.0	29.7	36.0	34.7	39.4	43.0	45.0
4	8.1	8.1	27.2	17.7	22.5	31.4	30.3	37.3	40.0	40.3
5	8.7	8.4	21.5	18.2	21.7	30.3	30.9	33.2	36.9	44.4

Table 20: GeoIntersect query Barcelona (results in milliseconds).

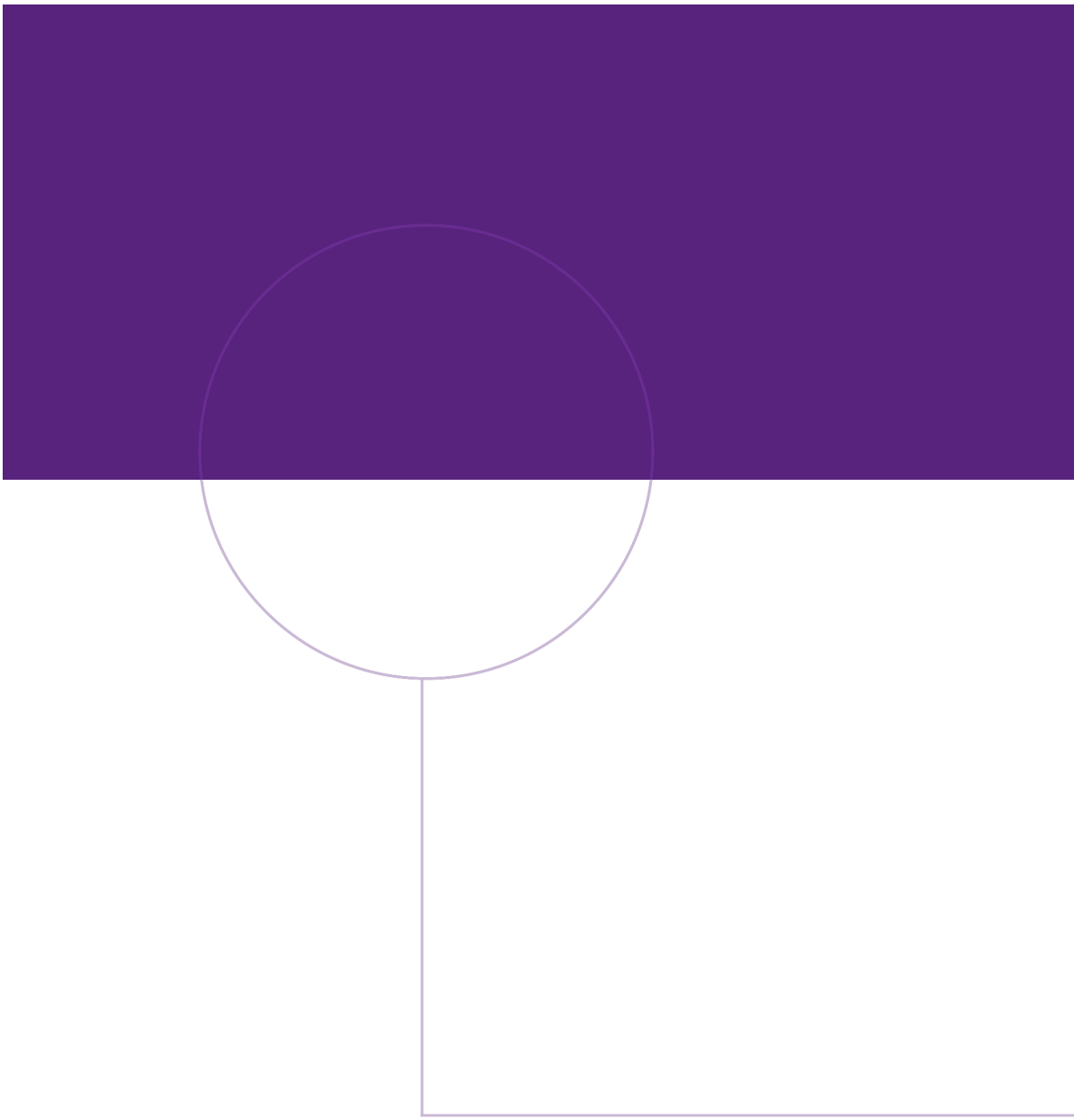
Iterations	**Query Window Size (%)**									
R-tree	**0.5**	**1.0**	**1.5**	**2.0**	**2.5**	**3.0**	**3.5**	**4.0**	**4.5**	**5.0**
1	195.6	308.4	434.3	560.1	714.9	808.9	922.6	72.3	177.2	308.8
2	144.7	254.9	377.4	516.2	623.4	755.5	868.9	986.3	119.8	235.6
3	143.3	261.5	379.8	493.9	633.1	761.1	882.7	980.2	116.2	228.2
4	144.2	254.7	376.2	507.9	633.9	750.8	864.6	998.0	121.6	228.1
5	152.5	254.5	374.8	503.7	626.1	759.2	895.4	980.8	121.7	233.1
MongoDB	**0.5**	**1.0**	**1.5**	**2.0**	**2.5**	**3.0**	**3.5**	**4.0**	**4.5**	**5.0**
1	24.6	37.0	42.8	53.9	52.3	53.2	89.3	99.5	111.9	98.9
2	19.4	24.5	23.6	35.0	28.5	45.7	52.5	48.9	51.2	60.8
3	9.0	11.0	26.4	25.7	34.4	34.9	42.6	46.3	46.0	45.2
4	8.4	14.5	19.2	25.0	31.9	32.9	36.0	44.6	46.4	48.2
5	8.5	12.5	16.8	23.3	31.7	27.3	39.2	38.3	48.4	50.7

Table 21: GeoIntersect query Hong Kong (results in milliseconds).

Iterations	**Query Window Size (%)**									
R-tree	**0.5**	**1.0**	**1.5**	**2.0**	**2.5**	**3.0**	**3.5**	**4.0**	**4.5**	**5.0**
1	93.8	134.1	170.2	202.3	275.9	287.7	342.6	346.2	401.7	437.2
2	44.4	85.3	120.5	159.3	200.9	234.8	276.2	305.2	349.6	385.9
3	43.9	84.4	120.6	157.8	208.9	232.5	274.4	318.6	256.9	395.8
4	44.0	82.8	118.5	157.3	191.0	239.6	271.5	309.7	331.9	381.9
5	44.0	83.2	120.7	156.5	201.6	234.5	273.6	310.0	346.4	385.0
MongoDB	**0.5**	**1.0**	**1.5**	**2.0**	**2.5**	**3.0**	**3.5**	**4.0**	**4.5**	**5.0**
1	17.1	28.6	33.9	37.7	33.8	37.4	39.2	37.6	39.0	47.7
2	10.7	14.0	18.9	11.8	12.6	11.7	15.9	15.2	20.0	23.4
3	11.2	7.9	10.1	11.1	11.4	15.5	16.7	17.2	25.3	26.9
4	4.5	7.3	10.1	11.6	11.6	16.4	23.0	20.5	21.9	23.7
5	4.4	7.9	10.8	12.0	13.9	17.5	18.3	17.7	21.5	23.2

E GitHub repository

For interested readers, the source code for this thesis can be found on https://github.com/CSynnestvedt/r-tree-mongodb



NTNU

Norwegian University of
Science and Technology