Design and Implementation of a NoSQL Data Access Interface

Nikolaos Koutroumanis

Supervisor: Assistant Prof. Christos Doulkeridis

Postgraduate Studies Programme “Digital Systems & Services”
Area of Study “Big Data and Analytics”
Department of Digital Systems
University of Piraeus

This dissertation is submitted for the degree of
Master of Science

April 2019

This thesis is dedicated to Nikolaos Bellias.

Acknowledgements

I would like to render my warmest thanks to my supervisor, Assistant Professor Christos Doulkeridis for his invaluable guidance, advice, encouragement and academic stimulus. It was a pleasure to be under his supervision for second time, since I kept learning fundamentals of conducting scientific research. His knowledge and experience has really inspired me and helped for completing this thesis.

I would like also to express my gratitude to Panagiotis Nikitopoulos for his indications on the code and the fruitful discussions we had during my thesis preparation. I greatly value the personal rapport that Panagiotis and I have forged. Special mention goes to Akrivi Vlachou and Orestis Telelis with whom we had illuminating discussions and excellent collaboration during my studies. Last but not least, my sincere thanks goes to the rest teaching stuff of our department for imparting the knowledge of data science and responding promptly to my questions.

Abstract

In recent years, the development of positioning technologies and the prevalence of GPS-equipped devices have generated vast amount of data with location and time information. Uber, one of the most known transportation network companies, records about 15 million trips every day over 600 cities worldwide. A Uber car drives a passenger from a departure to a destination location. During a trip, the embedded GPS of the car reports its geo-location in time, and the sequence of these locations forms a trajectory. Foursquare, a location technology company that provides an application for personalized recommendations of places on mobile devices, reports 8 million check-ins on a daily basis from its users. Many challenges arise with the rapid expansion of spatial and spatio-temporal data in volume and velocity, but the main target that remains is their efficient management and access.

The amount of such data exceeds the storage and processing capabilities of a single machine, thus distributed database systems are adopted. Namely, NoSQL databases provide a promising solution for handling massive data, offering high performance, availability and scalability. Many NoSQL databases do not support directly spatial or spatio-temporal indexing, but several studies propose techniques and methods for supporting this type of data. Targeting to provide a unified way to access big data stored in NoSQL databases, we present an API that makes the procedure of data accessing simple, hiding implementation details. The API can stitch together with analytics tasks, as it offers many primitives and operators for expressing data access operations. In addition to the offered functionality, the API is extended to spatial and spatio-temporal data access. Moreover, we conduct extensive experiments on MongoDB database with real and synthetic mobility dataset, so as to study its performance in terms of efficiency and scalability for spatial and spatio-temporal data.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

Nowadays, with the prevalence of GPS-enabled mobile devices, a huge amount of information tagged with geographic location is generated. Many are the challenges that data management systems face, since spatio-temporal data is increasing in an unprecedented rate in volume and velocity. The adoption of a NoSQL database system instead of a relational datababase allows the management of voluminous data, as they provide scalable distributed storage and querying. However, many of them do not support directly spatial or spatio-temporal indexing, but many studies propose techniques and methods for supporting this type of data.

Motivated by the lack of an unified language that could operate upon NoSQL storages, we present an API for accessing data on such storages. Every NoSQL storage has its own query language, putting a burden to developers who want to access the stored data. The API supports query primitives and operators for the formation of access operations. Since our primary interest is spatio-temporal data, we integrate to the API the functionality of accessing spatio-temporal data.

1.1 Goals of the thesis

This thesis aims at implementing a developer-friendly interface for accessing data stored in NoSQL stores. The interface was designed on the basis of providing a toolbox of functions for expressing data access operations in an abstract and unified way upon agnostic NoSQL systems. This facilitates to a great extent the procedure of data accessing, since the user focuses only to the task of forming operations through comprehensive query primitives and operators. The functionality of the interface is extended to spatial and spatio-temporal data, being thus capable of accessing mobility data.

The API is instantiated over MongoDB. We focus on the use-cases of spatio-temporal data accessing, and therefore we study thoroughly the performance of MongoDB in terms of

efficiency and scalability on such data. Our objective is to acquire a deeper understanding of performance issues for this database, as it can support geospatial data and index types for performing spatial queries. By taking into account that MongoDB does not support k-NN type queries, we adopt an approach based on grid partitioning for serving such queries via the implemented API.

1.2 Document structure

The remaining of this document is structured as follows:

- Chapter 2 covers the state-of-the-art in the field of NoSQL solutions for spatial and spatio-temporal data, providing some techniques and methods for managing such data. The review of the related work shows that some of the NoSQL databases provide spatial indexes, addressing the problem of managing spatial data at scale.
- Chapter 3 delineates the implementation of the NoSQL data access API which provides a unified and simple way for accessing data on NoSQL storages. A usage documentation of the API is listed, and use cases of accessing spatial and spatio-temporal data type are presented. API's architecture is also demonstrated, in order to provide a deep insight of its structure.
- Chapter 4 describes the query types of MongoDB that are used from the implemented API for performing queries. An approach is presented for performing k-NN queries, as they are not supported in MongoDB.
- Chapter 5 reports a detailed experimental evaluation on MongoDB, regarding spatial and spatio-temporal data. Three experimental studies are conducted over three mobility datasets, so as to acquire a deep understanding about the efficiency and scalability when utilizing the built-in spatial and compound indexes.
- Chapter 6 summarizes the conclusions drawn from the thesis and reports the possible future work.

Chapter 2

Review of NoSQL solutions for spatial and spatio-temporal data

In this section, we review scalable and distributed storage solutions for voluminous spatial and spatio-temporal data that need to be persistently stored.

The modern trend for scalable storage of massive data sets is by means of a NoSQL store [3, 6]. The exact choice depends on numerous parameters, including the type of data, the data access patterns, the purpose of data processing (read/write, read-only, etc.), as well as any special requirements with respect to the consistency, availability, and partition-tolerance (also known as CAP).

In the context of the thesis, spatial and spatio-temporal data is of our primary interest. We turn our attention towards support of such kind of data in NoSQL systems, by using built-in indexes and functionality. This involves the overview of existing NoSQL systems and their capabilities for managing this kind of data.

2.1 NoSQL stores

The categorization of NoSQL stores is based on the underlying data model that is supported. Essentially, a data model specifies how real-world entities and their relationships are represented and operated [15]. Thus, NoSQL stores are mainly classified into: key-value, wide-column, document, and graph stores.

Today, there exist literally dozens of NoSQL systems, each targeting a vertical dimension of the big data management landscape. We select a couple of representative NoSQL systems to delineate from each category, using as selection criterion its popularity and wide user base. Then, some of the selected NoSQL stores are presented in more detail. We focus on any

built-in support or functionality for handling spatial data, which is of major importance. Our selection is as follows:

- Document-stores: MongoDB (Section 2.1.1), Elasticsearch (Section 2.1.2)
- Wide-column: HBase (Section 2.1.3)

2.1.1 MongoDB

MongoDB[1] is one of the most known and widely-used NoSQL databases. Its data model is document-oriented, meaning that instead of storing records in tables like other databases, it stores documents in collections. Documents are binary JSON documents (BSON) composed of field-and-value pairs which are stored in collections.

Collections are containers for structurally (or conceptually) similar documents which are not forced to have the same fields. Collections are stored in databases.

Databases are namespaces for physical grouping of collections. In other words, databases hold collections of documents. Each database has its own set of files on the file system.

The lack of a pre-defined schema offered by documents confers some advantages. Schema-less models can handle the cases of storing data whose schema is changing frequently. Moreover, every document stores in fields all of the information related to it, supporting sub-documenting (a document can be nested in a document). This is beneficial for queries because no joins are required since every document includes all of its required data. For that reason, MongoDB does not support join operations. [1].

A document-based model leads to denormalized data sets, because related data are replicated across several documents. Consequently, this results to a hierarchical data structure. Suppose that we are modeling products for an e-commerce site. A normalized relational data model would require a query with joins in order to fetch all of the information about a product in contract to the document model in which we would have to read a single document that contains all of the product's information.

Many databases (especially the relational ones) are difficult to scale horizontally which is essential for handling vast databases because they are distributed across multiple machines. MongoDB was designed to be easy to scale by making the horizontal scaling manageable. This is achieved by supporting *sharding*. Similar to that feauture, MongoDB supports also data replication for automated failover via replica set.

A way to administer the MongoDB database is to use its command shell which is a tool based on the JavaScript language. The shell is similar to MySQL's shell, but the big

[1]https://docs.mongodb.com/manual/

difference is that SQL is not used. Another way is to use a driver which is offered in many languages such as C, C++, Java, Scala, Python and etc. A driver is the code that provides functionality to query, read, write and run commands on the database.

Like other databases, MongoDB provides the tools for atomic operations meaning that a single document can be processed. This is supported by the command *findAndModify* which allows to atomically update a document (or a sub-document if included) and return it in the same round-trip. Atomic update can not be interrupted by other operations. If a user tries to change a found document before we modify it, must wait until the atomic update finishes. A special feature of *findAndModify* command is that it returns the document after updating it. This is useful because if we fetch and then update a document, changes may be made to that document by another user between those two operations. In general terms, the atomic update capability is big. It enables to build job queues and state machines, being thus able to implement basic transactional semantics. This expands the range of applications we can build by using the MongoDB. Furthermore, the option of operating on multiple documents is available via the *updateMany* command. Although the operation as a whole is not atomic, the modification of each document that occur is atomic.

The concept of CRUD operations exist in MongoDB, offered by its query language.

- *Create* operation adds new documents to a collection of a database. If the collection does not exist, the collection is created with the first document insertion.
- *Read* operation retrieves documents from a collection. Query criteria can be specified for fetching the needed documents. Criteria is expressed by operators.
- *Update* operation modifies existing document/s in a collection. Criteria can be specified so as to identify the documents to update.
- *Delete* operation removes document/s from a collection. Criteria can be specified in order to identify the documents to delete.

MongoDB supports comparison, logical, element, evaluation, geospatial, array and bitwise operators for forming a query criteria.

- *Comparison* operators specify a range of values on a field
- *Logical* operators are used to define the logical relationship between operators
- *Element* operators specify criteria about the contained fields of documents
- *Evaluation* operators specify miscellaneous criteria on a field such as matching a given modulo operation on a field

- *Geospatial* operators specify a geospatial criteria on a field
- *Array* operators specify a criteria that is applied on the elements of an array field
- *Bitwise* operators specify a criteria that is applied on a binary field

2.1.2 Elasticsearch

Elasticsearch is a document-oriented database. It can be used in several ways such as search engine, analytics framework and Data store (mainly for log). It has a schema-less data store. The main data container is called index and is considered similar to the database in relational databases. In an index, the data is groupped in mappings similar to tables in relational databases. A mapping is composed of records stored as JSON object that contain fields.[13]

Elasticsearch supports sharding in order to manage the volumes of records Every record is stored only in one shard, so many operations that require loading of records and modifications are achieved without hitting all the shards.

The supported operations are divided into;

- Cluster\Index Operations - All write operations are locking and at first are applied to the master node and then to the secondary node. Read operations are broadcasted to all nodes
- Document Operations - All write operations are locking only for a single hit shard. Read operations are balanced among all the shard replicas.

Elasticsearch natively supports storing geolocation types; special types called Geo-Shapes, that allow localizing a document with geographical coordinates (longitude, latitude). These types are represented as GeoJSON objects.

The Geo-Shape data type facilitates the indexing of arbitrary geo shapes such as rectangles and polygons. This type is indexed by decomposing the shape into a triangular mesh and indexing each triangle as a 7 dimension point in a BKD tree. This has the advantage of providing perfect spatial resolution (down to 1e-7 decimal degree precision) because of using an encoded vector representation for the computation of spatial relations of the original shape instead of a raster-grid representation.[2]

[2]https://www.elastic.co/guide/

2.1.3 Apache Hbase

Apache Hbase[3] is an implementation of Google's Bigtable [4] for the Hadoop ecosystem. The data model is the same as Cassandra's (and Bigtable's) described above. One important distinction is that in Hbase there is no need for the valueless column design. The are no data types like String, int or long. Everything is stored as a byte array in a cell by using a serialization framework. So, the database gives out byte arrays which are implicitly converted to the data equivalent representation. At the conceptual level, Hbase table can be seen a set of rows, but in fact it is stored as per a column family. When a table is created, the name of the column family and the number of the contained columns must be already decided. A Column can be added at a column family at any point in time while storing the data [14].

In Hbase, large read and write operations are avoided since rows are divided into column families, enabling horizontal and vertical scaling of tables. Table is composed of the following components;

- Row
  - Column Family
    - Column
      - Cell

Row is a unique key for every record in a table. Internally, it is stored as a byte array no matter what data type we choose as row key.

Column Family is a group of columns. A table contains columns families which are stored separately on disk. This is advantageous because we can retriece columns faster.

Column is a set of data values

Cell is the smallest basic unit of storage in a column where the actual value is stored. Cells are accessed by using <*row, columnfamily:column, version*> tuple.

Hbase provides several advanced features such as:

- *Filters* allow the user to specify the subset of objects from the query result that will be returned to the client. The benefit is that they run on the server-side, thus reducing the amount of data that needs to be transferred. Examples include filtering rows based on key prefixes or some regular expression. Filters can also be combined in a *FilterList* (a set of filters), which is applicable on the result using either AND or OR semantics.

[3] https://hbase.apache.org/

- *Coprocessors* provide even more flexibility than filters in terms of querying. They are essentially user code that can be deployed in the Hbase cluster. Mainly, there are 2 types of coprocessors:
  - *Observers*, where for each base operation, i.e., *put*, the user can deploy custom code in form of hooks that can either pre-process the input parameters or post-process the results.
  - *Endpoints*, that act like functions and can be invoked through Remote Procedure Calls (RPC).

Hbase supports Range Scans and Prefix Range Scans, which is the same as a Range Scan and a PrefixFilter.

2.2 MongoDB data indexing

An index is a data structure which can speed up the query evaluation process, by enabling efficient search over stored data, avoiding the need to process every stored element. An index however, inflicts an additional cost for write operations and storage space, since it needs to update and store the corresponding index. Indexes are generally considered to be a crucial aspect of data management, and most database management systems support several index types.

MongoDB supports indexing on stored documents, by using a data structure called B-Tree [5]. It assigns a document id on every stored document, which is a unique identifier, that is indexed by default. This index, also called the *primary index*, accelerates the process of document search based on a provided document id. MognoDB also supports indexing of any other field of the stored documents. These indexes, also called *secondary indexes*, can be created and configured by the database administrator, in order to speed up query execution for queries with predicates on the indexed fields. A list of supported *secondary indexes* is provided in the following:

- *Single-field index* - In addition to the primary key which is indexed by default, a custom field of a document collection can be indexed with either an ascending or descending order. A single-field index is actually a B-Tree index, which supports efficient equality matches and range queries.
- *Compound-key index* - This index resembles to the single-field index with the difference that it indexes two or more fields of documents. At most, 32 fields can be indexed by using a compound index.

- *Text index* - This index is oriented on text search queries. It supports indexing of fields which contain unstructured text content.
- *Multikey index* - This index can be used over fields of array type, indexing either an array of scalar values (e.g. string, numbers) or an array of nested documents.
- *2dsphere index* - This is a spatial index that indexes geometries projected on an earth-like sphere. The indexed field should be a GeoJSON object or a legacy coordinate pair (for point geometries).
- *2d index* - This is a spatial index like the 2dsphere index, with the difference that it calculates geometries on a two-dimensional plane (in other words it supports calculation on a flat, euclidean plane). The indexed field should be a legacy coordinate pair.
- *geoHaystack index* - This is a special spatial index that aims to improve the efficiency of queries concerning a flat geometry over a small geographic area. It stores buckets of documents which are located on the same geographic area in order to improve the performance of spatial queries over that area.
- *Hashed index* - This index maintains entries with hashes of the values of the field that is indexed. It is utilized for sharding support by using hashed shard keys. In order to compute the hash of the value of the index field, a hashing function is used.

The supported MongoDB's indexes have properties, defining rules about the indexed data. Every index type can have one or more of the following properties:

- *Unique index* - This index type restricts the indexed values to be unique. By default, MongoDB creates a unique single-field index for storing the primary key of each document.
- *Partial index* - Only specific values of field/s are indexed, specified by a filter expression. It is reasonable that partial indexes usually have lower storage requirements, since they contain less values than expected.
- *Case insensitive index* - This index is used for string fields, performing comparisons without considering case sensitivity.
- *TTL index* - This index is a single-field index with the property of removing automatically the documents from a collection after a certain amount of time or at a specific clock time. If the document does not contain the indexed field, the document will not be removed.

- *Sparse index* - This index contains only entries for documents that contain a value for the indexed field, even if that value is null. If a document does not contain the indexed field, it will not be indexed. *Primary indexes* cannot be configured as sparse type.

In order to benefit by the natively supported indices provided by MongoDB, we need to know in advance the spatial and temporal fields from every data source. Provided with the spatial and temporal fields, we will create a spatial *2dsphere* index on longitude and latitude fields and a *single-field* index on date field for all of our document collections. These indices, are expected to increase the efficiency on queries with spatial and/or temporal predicates.

2.3 Underlying structure of spatial indexes of NoSQL stores

The majority of document-oriented NoSQL stores (e.g. MongoDB) provide many types of indexes for data as RDBMSs do. Spatial and spatio-temporal indexes are also provided, but their underlying structure is not based on the data structures that used in relational databases ([10], [7]). Wide-column stores do not provide directly spatial indexes at all, but many works that are related to the efficient managing of spatial and spatio-temporal data upon this kind of NoSQL database exist (Section 2.4).

The spatial index approaches used in RDBMSs have been proved to be efficient in databases that operate on a centralized machine, but R-Trees have severe difficulties on parallelizing. QuadTrees are more suited for clusters but are inefficient for non-point data since the indexing overhead is increased for geometries [11]. These difficulties point towards to the need of a robust distributed spatial and spatio-temporal index mechanism for NoSQL storages.

Thus, most of the existing spatial indexes on NoSQL stores use the *geohash* method for handling spatial and spatio-temporal data in a distributed way. *Geohash* was invented by Gustavo Niemeyer, specifying a point as an encoded string of bits, in which every bit indicates the divisions of the longitude and latitude ([-180, 180] x [-90, 90]) rectangle. The division starts from splitting the rectangle into two squares ([-180, 0] x [-90, 90]) and ([0, 180] x [-90, 90]). Points belonging to the left of the vertical division begin with 0 and the one in the right with 1. Then the next split that occurs is horizontal. The points below the horizontal split receive 0 and the ones above 1. The splitting continues until achieving the desired resolution [8]. By storing points as encoded strings (geohashes), the stores are capable of distributing them accross a cluster, enabling thus horizontal scalability for spatial data.

2.4 Techniques for managing mobility data on NoSQL stores

Several studies exist for spatial and spatio-temporal index techniques upon NoSQL databases. Many of them focus on column-oriented NoSQL databases ([12], [8], [16], [2]) as they do not provide directly a spatial index mechanism because of their underlying data structure. There are also several studies for spatial and spatio-temporal data management on document-oriented NoSQL stores. Below, we delineate those that use MongoDB, as this is the primary database we focus in this thesis.

SIFT [11] is a distributed spatial index whose structure is based on a tree combining features from both R-Trees [10] and QuadTrees [7]. Specifically, spatial objects are represented as MBRs (like in R-Tree) in a structure that resembles to QuadTrees' structure. Every node of the tree is given a bounding box being a logical storage for spatial objects. Parent nodes enclose the bounding box of their children, so the root node is the bounding box of the entire space. SIFT is unbalanced and can remain unbalanced without performance detioration. It does not impose a bound on the number of objects stored per node. It only imposes a bound for the available memory; this means that a node can contain more spatial objects than another node. Owing to the fact that the tree remains unbalanced, it is not necessary of object movement when nodes are created or destroyed. Thus, high ingestion rate can be reached. Also, the performance of queries are efficient because of skew mitigation.

The skewing problem is known for impairing the load balancing of the data in a cluster. In this work, it is addressed by adding dimension on the spatial data derived from their characteristics. This happens because the follwing assumption in the work is made; if it is impossible to obtain uniform partitioning of spatial data in its native dimensional space N, then is it possible to be obtained in a higher dimensional space. In other words, by adding a couple of dimensions to spatial data results in a higher probability of achieving uniform distribution.

MongoDB uses the Google's S2 Geometry Library for spatial indexing. This is an open-source library that utilizes the Hilbert space-filling curve for performing geometric operations on a sphere. SIFT reuses the S2 library by utilizing a related space-filling curve. It provides the following three library modules; *KeyGenerator*, *QueryPlanner* and *ShardManager*. Each of the modules have a unique role for the spatial index mechanism provision. The MongoDb codebase was modified for SIFT implementation.

The *KeyGenerator* module is used for the determination of the tree node(s) in which the geometry would be stored, deleted or modified. It generates the (one-dimensional) key(s) that represents specific tree node(s). Given these tree node(s), *ShardManager* finds the owning partition(s) of node(s), and forwards the data to the corresponding partition(s). Then, the server (partition) is responsible for the data processing on a B-Tree index.

In case of data retrieval, the *ShardManager* module determines the partitions that should be contacted. Specifically, the contained geometry in a given query is examined as if it is to be stored, thus determining its supposed location. Then, the SIFT querying algorithm (*QueryPlanner* module) is used to find the potential tree node(s) that need to be queried. The query is forwarded to the owning partition(s) of the candidate node(s), and the nodes are searched.

ST-Hash [9] extends the Geohash space-partitioning index (Section 2.3) by adding the temporal dimension in it. The 3D data structure is converted into a sequence of characters (1D String) which are stored in MongoDB. It supports both box and cylinder spatio-temporal queries since the index is a spatio-temporal space consisting of subspaces defined by a unique ST-Hash String.

Only one collection is used to store the trajectory data. Each trajectory point is stored as a JSON format document in the collection, containing fields with values that represent different kind of information. The fields encompass data that are correlated with the point's location stigma (longitude, latitude), temporal stigma (Date) and other features of the stigma such as its ID which correspond to a specific moving object (Taxi). Also, a field called ST-Hash is used for indexing purposes, storing both the spatial and the temporal part of the point (as its name signifies), in a hashed string. A B-Tree index is constructed on this field in order to accelerate spatio-temporal queries.

The ST-Hash field consists a string composed of the point's stigma year, a dashed character (-) and a sequence of five characters. The sequence of the five characters is an encoded representation of the rest of the temporal part (month, day, hour, minute, seconds) and the location (longitude-latitude) of the point. An instance of such a string is "2015-Re+BP". The encoding procedure includes the conversion of the values of longitude, latitude and date (without the year) to a long binary code. Then, the binary code is separated and every smaller part is converted to a character using the Base64 binary-to-text encoding schema, forming ultimately a string.

The hashed strings, implicitly represents a cube in the three-dimensional space, where each dimension corresponds to the longitude axis, latitude axis and time axis. The whole spatio-temporal cube is formed by smaller cubes where each one is labeled by a an ST-Hash String. When a spatio-temporal query is performed, the given coordinates and time are defined by upper $P_1(x_1, y_1, t_1)$ and lower $P_2(x_2, y_2, t_2)$ bounding points. These points are encoded, producing two hashed strings. Then, the B-Tree index is used in order to match some of the hashed strings contained in the range of the two hashed strings. In other words, the spatio-temporal query is depicted by a cube where the intersection with the ST-Hash labeled cubes is returned as result. Indexing the encoded strings that represent the spatio-

temporal part of a point, reduces to a large extend the number of points that have to be accessed during search operations.

Chapter 3

NoSQL data access API implementation

3.1 API Description

The implementation of the API was based on the concept of creating a toolbox of data query primitives for NoSQL database systems. The main objective of this idea was to facilitate the process of accessing the data stored in a NoSQL database for the developers who need to serve their purposes by performing query operations. Actually, this is accomplished by providing a user-friendly data access interface for expressing and performing queries in a *simple* and *unified* manner. *Simple* due to hiding as many implementation details as possible (such as establishing a database connection to a cluster of computing machines) and thus focusing only to the process of querying the data. *Unified* because a set of well-defined query primitives (functions) are provided, through which common data access tasks can be expressed abstractly (as a sequence of query primitives) and performed on different NoSQL stores. Also, a set of operators are provided for defining the functional behaviour of some primitives.

In a few words, the API groups most of the common (widely-used) query primitives and offers them in a comprehensive manner for performing query operations on NoSQL databases. This relieves the developers from using directly a native NoSQL database API (such as the MongoDB Java Driver for the MongoDB database), who would be loaded with the task of "translating" the desired query operation to the corresponding commands of the native API. Essentially, this is done by the API. Its functional part exploits the native libraries of the supported NoSQL databases. All of the supported query primitives and operators are implemented upon the functions of the native libraries that reflect their functionality. The API maps the supported query primitives and operators with its functional part which is hidden from users since the goal is to provide abstraction.

The supported query primitives cover most of the ordinary querying operations like fetching a field that fulfills a specific condition or fetching a sorted field in ascending order. The primitives were designed to be flexible in terms of expressing simple and complex query operations as well. This is possible because the primitives can be used in combination with each other and in combination with the provided operators which customize their behaviour. A simple operation e.g. grouping the values of a field to find the distinct ones, can be imprinted by an individual (distinct) primitive, whereas a complex operation can be expressed as a primitive combined with operators affecting over one dimension of data. In addition to the common operators such as the boolean and the comparison ones, the API supports also operators oriented to spatial (two-dimensional) and spatio-temporal (three-dimensional) data. This type of operators named Geographical Operators (or Geo-Operators in short) enables access to mobility data.

Listing 3.1: Find the max value of a field and count the number of records of a MongoDB collection by using the implemented API

```
1  import gr.unipi.api.nosqldb.NoSqlDbOperators;
2  import gr.unipi.api.nosqldb.NoSqlDbSystem;
3  import java.util.Optional;
4
5  public static void main(String args[]){
6      NoSqlDbSystem noSqlDbSystem = NoSqlDbSystem.MongoDB().host("83.212.102.163")
7      .database("test").username("user").password("pass").port(28017).build();
8
9      NoSqlDbOperators noSqlDbOp = noSqlDbSystem.operateOn("geoPoints");
10
11     Optional<Double> max = noSqlDbOp.max("aField");
12
13     int count = noSqlDbOp.count();
14
15     noSqlDbSystem.closeConnection();
16 }
```

Listing 3.2: Find the max value of a field and count the number of records of a MongoDB collection by using its native API (Java Driver)

```
import com.mongodb.MongoClient;
import com.mongodb.MongoClientOptions;
import com.mongodb.MongoCredential;
import com.mongodb.ServerAddress;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoCursor;
import org.bson.Document;
import org.bson.conversions.Bson;

import java.util.ArrayList;
import java.util.List;

public static void main(String args[]){
    MongoCredential credential = MongoCredential.createCredential("user", "test",
    "pass".toCharArray());

    MongoClientOptions options = MongoClientOptions.builder().build();

    MongoClient mongoClient = new MongoClient(new ServerAddress("83.212.102.163",
    28017), credential, options);

    MongoCollection mongoCollection = mongoClient.getDatabase("test")
    .getCollection("geoPoints");

    List<Bson> b1 = new ArrayList<>();
    b1.add(Document.parse("{ $group: { _id:null, max_val: { $max:\"$aField\"} } }"));

    MongoCursor<Document> cursor1 = (MongoCursor<Document>) mongoCollection
    .aggregate(b1).iterator();

    int max =  cursor1.next().getInteger("max_val");

    cursor1.close();

    List<Bson> b2 = new ArrayList<>();
    b2.add(Document.parse("{ $count: \"totalRecords\" }"));

    MongoCursor<Document> cursor2 = (MongoCursor<Document>) mongoCollection
    .aggregate(b1).iterator();

    int count =  cursor2.next().getInteger("totalRecords");

    cursor2.close();

    mongoClient.close();
}
```

3.2 API Documentation

The API was developed in Java programming language, offering various query data operations stored in NoSQL database systems.

3.2.1 Setting up a connection to a NoSQL database

In order to access the data on a NoSQL store via the API, a connection to the database should be set up at first. This can be done easily by using one of the below code Listings (3.3 or 3.4) that indicate a connection to a specific NoSQL system (MongoDB database). The NoSQL system is defined directly after having called the *NoSqlDbSystem* object. Then, by calling its rest methods, the required database credentials are provided.

The difference between the two Listings is the integration of Apache Spark processing engine. By integrating a Spark session into the API, the user is able to export the results of the data operations to Dataset<Row> (a.k.a. Dataframe) object (see the query primitive toDataframe on Table 3.1). This enables the parallel processing of data on a distributed environment.

Line 9 of the Listing 3.3 and line 16 of the Listing 3.4 define the table (or collection for the Document-oriented NoSQL databases) which will be accessed for performing data querying operations. Also, the lines 13 and 20 of the two Listings respectively, close any open connection to the NoSQL system. If a user opens simultaneously connections on different NoSQL database systems for querying multiple data sources, then all of them can be closed at once by calling *NoSqlDbSystem.closeConnections()*.

The code Listings of subsections 3.2.2 and 3.2.3 are declared in the static method *doOperations* of the classes *DataOperations* or *DataOperationsWithSpark* for demonstrating examples of operations via the API. It is implied that they are called from one of the below two Listings (lines 11 and 18 correspondingly).

Listing 3.3: Set up a connection to a NoSQL database

```
import gr.unipi.api.nosqldb.*;

public class NoSqlDbQueryOperations {
    public static void main(String args[]) {

        NoSqlDbSystem noSqlDbSystem = NoSqlDbSystem.MongoDB().host("83.212.102.163")
        .database("test").username("user").password("pass").port(28017).build();

        NoSqlDbOperators noSqlDbOp = noSqlDbSystem.operateOn("collection");

        DataOperations.doOperations(noSqlDbOp) //doing data query operations

        noSqlDbSystem.closeConnection();
    }
}
```

Listing 3.4: Set up a connection to a NoSQL database by combining it with a Spark Session

```
import gr.unipi.api.nosqldb.*;
import org.apache.spark.sql.SparkSession;

public class NoSqlDbQueryOperations {
    public static void main(String args[]) {

        NoSqlDbSystem.initialize();

        SparkSession session = SparkSession.builder().master("local")
        .appName("MongoSparkConnectorIntro").getOrCreate();

        NoSqlDbSystem noSqlDbSystem = NoSqlDbSystem.MongoDB().host("83.212.102.163")
        .database("test").username("user").password("pass").port(28017)
        .sparkSession(session).build();

        NoSqlDbOperators noSqlDbOp = noSqlDbSystem.operateOn("collection");

        DataOperationsWithSpark.doOperations(noSqlDbOp) //doing data query operations

        noSqlDbSystem.closeConnection();
        session.close();
    }
}
```

3.2.2 Query primitives and operators

The API offers the query primitives listed on the Table 3.1, the Filter type operators listed on the Tables 3.2, 3.3, 3.4, the Aggregate operators listed on the Table 3.5 and the Sort operators listed on the Table 3.6.

The functionality of the query primitives is described as follows;

Table 3.1 Supported Query Primitives

Primitives	Arguments	Phase
filter	(FilterOperator fop, FilterOperator... fops)	Definition
groupBy	(String fieldName, AggregateOperator... aops)	Definition
distinct	(String fieldName)	Definition
max	(String fieldName)	Execution
min	(String fieldName)	Execution
sum	(String fieldName)	Execution
avg	(String fieldName)	Execution
sort	(SortOperator sop, SortOperator... sops)	Definition
limit	(int limit)	Definition
project	(String fieldName, String... fieldNames)	Definition
toDataframe	()	Execution

- *filter* - performs filter operation/s given some (at least one) FilterOperator object type arguments. More than one defined arguments entails that they are operands of an and ($\cap$) boolean operator.
- *groupBy* - performs a group operation on a field, the name of which is passed as the first argument. Specifically, it arranges the identical values of a specific field into groups. It can be optionally used in conjunction with aggregate operators (functions) which are passed as AggregateOperator object type arguments after the field name. If aggregate operators are not defined, the primitive acts as a distinct statement, finding the unique values of the field.
- *distinct* - performs a distinct operation on a field, finding the distinct (different) values. The name of the field is passed as an argument.
- *max* - finds the maximum value of a field, the name of which is passed as an argument.
- *min* - finds the minimum value of a field, the name of which is passed as an argument.
- *sum* - finds the sum value of a field, the name of which is passed as an argument.
- *avg* - finds the average value of a field, the name of which is passed as an argument.
- *sort* - performs sort operation/s given some (at least one) SortOperator object type arguments. If more than one SortOperator object arguments are declared, then they will be performed sequentially.
- *limit* - performs a limit operation concerning the records, retaining a specific number of them which is passed as an argument.

- *project* - performs a project operation concerning the fields, retaining those of which name are passed as argument
- *toDataframe* - fetches the results as a Dataset<Row> (a.k.a. Dataframe) object given that the user has created a spark session.

The primitives that correlate with the definition phase can be used in conjunction with each other as many times as needed (Listing 3.5). They are lazy in nature meaning that they define a sequence of operations without being performed in fact. These operations are executed when a primitive belonging to the execution phase is called, returning either a primitive data type (*int* for the count query primitive) or a reference type (*Optional<Double>* for the max, min, avg query primitives and *Dataset<Row>* for the toDataframe primitive) that can be exploited afterwards.

By using the filter primitive more than once entails that all of their arguments are operands of an and ($\cap$) boolean operator.

Listing 3.5: Definition and Execution Phase of Primitives

```
1  import gr.unipi.api.nosqldb.NoSqlDbOperators;
2  import org.apache.spark.sql.*;
3
4  public class DataOperationsWithSpark {
5      public static void doOperations(NoSqlDbOperators noSqlDbOp) {
6
7          Dataset<Row> dataset = noSqlDbOp.filter( ... ).filter( ... ) //definition phase
8          .groupBy( ... ).sort( ... ).project( ... ) //definition phase
9          .toDataframe(); //execution phase
10     }
11 }
```

Table 3.2 Supported Comparison Operators

Comparison Operators	Arguments
eq	(String fieldName, T_1 fieldValue)
gt	(String fieldName, T_2 fieldValue)
gte	(String fieldName, T_2 fieldValue)
lt	(String fieldName, T_2 fieldValue)
lte	(String fieldName, T_2 fieldValue)
ne	(String fieldName, T_1 fieldValue)

$T_1 \in [short, int, long, float, double, boolean, Date, String]$

$T_2 \in T_1$ - $[boolean, String]$

The Comparison operators can be used as arguments of the filter query primitive since they are a subtype of FilterOperator type. Their functionality is described as follows;

- *eq* - selects the records whose values of a specific field (its name is passed as the first argument) equals ($=$) to a given value (passed as the second argument).
- *gt* - selects the records whose values of a specific field (its name is passed as the first argument) is greater than ($>$) to a given value (passed as the second argument).
- *gte* - selects the records whose values of a specific field (its name is passed as the first argument) is greater than or equal ($\geq$) to a given value (passed as the second argument).
- *lt* - selects the records whose values of a specific field (its name is passed as the first argument) is less than ($<$) to a given value (passed as the second argument).
- *lte* - selects the records whose values of a specific field (its name is passed as the first argument) is less than or equal ($\leq$) to a given value (passed as the second argument).
- *ne* - selects the records whose values of a specific field (its name is passed as the first argument) is not equal ($\neq$) to a given value (passed as the second argument).

Listing 3.6: Example of using a comparison operator - Find all of the 5 Star hotels in Greece

```
1  import gr.unipi.api.nosqldb.NoSqlDbOperators;
2  import org.apache.spark.sql.*;
3  import static gr.unipi.api.filterOperator.FilterOperators.*;
4
5  public class DataOperationsWithSpark {
6      public static void doOperations(NoSqlDbOperators noSqlDbOp) {
7
8          Dataset<Row> dataset = noSqlDbOp.filter(eq("star", 5)).toDataframe();
9      }
10 }
```

Table 3.3 Supported Boolean Operators

Boolean Operators	Arguments
or	(FilterOperator fop1, FilterOperator fop2, FilterOperator... fops)
and	(FilterOperator fop1, FilterOperator fop2, FilterOperator... fops)

The Boolean operators can be used as arguments of the filter query primitive since they are a subtype of FilterOperator type. Their functionality is described as follows;

- *or* - performs the logical or (∪) operation by selecting the records that satisfy the expression of at least one FilterOperation object types that are passed as arguments (at least two are needed).

- *and* - performs the logical and (∩) operation by selecting the records that satisfy the expression of all FilterOperation object types that are passed as arguments (at least two are needed).

Listing 3.7: Example of using a boolean operator - Find all of the 5 Star hotels in Greece located in the city of Piraeus

```
1  import gr.unipi.api.nosqldb.NoSqlDbOperators;
2  import org.apache.spark.sql.*;
3  import static gr.unipi.api.filterOperator.FilterOperators.*;
4
5  public class DataOperationsWithSpark {
6      public static void doOperations(NoSqlDbOperators noSqlDbOp) {
7
8              Dataset<Row> dataset = noSqlDbOp
9              .filter(and(eq("star", 5), eq("city","Piraeus"))).toDataframe();
10     }
11 }
```

Table 3.4 Supported Geographical Operators

Geographical Operators	Arguments
inGeoPolygon	(String fieldName, Coordinates c1, Coordinates c2, Coordinates c3, Coordinates... cs)
inGeoBox	(String fieldName, Coordinates lowerBoundPoint, Coordinates upperBoundPoint)
inGeoCircleKm	(String fieldName, Coordinates point, double radius)
inGeoCircleMeters	(String fieldName, Coordinates point, double radius)
inGeoCircleMiles	(String fieldName, Coordinates point, double radius)
nearestNeighbors	(String fieldName, Coordinates point, int neighbors)

The Geographical operators can be used as arguments of the filter query primitive since they are a subtype of FilterOperator type. Their functionality is described as follows;

- *inGeoPolygon* - selects the records whose spatial extent that is represented by a specific field (its name is passed as the first argument), is entirely within a polygon. The polygon is defined by its corner points (its coordinates are passed as arguments - at least three are needed).

- *inGeoBox* - selects the records whose spatial extent that is represented by a specific field (its name is passed as the first argument), is entirely within a box. The box is defined by its lower and upper bounding points (the coordinates of which are passed as the second and third argument respectively).
- *inGeoCircleKm* - selects the records whose spatial extent that is represented by a specific field (its name is passed as the first argument), is entirely within a circle. The circle is defined by its center point (the coordinates of which are passed as the second argument) and its radius in the kilometer unit (passed as the third argument).
- *inGeoCircleMeters* - selects the records whose spatial extent that is represented by a specific field (its name is passed as the first argument), is entirely within a circle. The circle is defined by its center point (the coordinates of which are passed as the second argument) and its radius in the meter unit (passed as the third argument).
- *inGeoCircleMiles* - selects the records whose spatial extent that is represented by a specific field (its name is passed as the first argument), is entirely within a circle. The circle is defined by its center point (the coordinates of which are passed as the second argument) and its radius in the mile unit (passed as the third argument).
- *nearestNeighbors* - selects a specified number of records (passed as the third argument) whose spatial extent that is represented by a specific field (its name is passed as the first argument), is the nearest to a specific point (the coordinates of which are passed as the second argument).

Listing 3.8: Example of using a geographical operator - Find the 10 nearest hotels from a location

```
1  import gr.unipi.api.nosqldb.NoSqlDbOperators;
2  import gr.unipi.api.filterOperator.geographicalOperator.Coordinates;
3  import org.apache.spark.sql.*;
4  import static gr.unipi.api.filterOperator.FilterOperators.*;
5
6  public class DataOperationsWithSpark {
7      public static void doOperations(NoSqlDbOperators noSqlDbOp) {
8
9          Dataset<Row> dataset = noSqlDbOp.filter(nearestNeighbors("location",
10         Coordinates.newCoordinates(23.65, 37.94), 10)).toDataframe();
11     }
12 }
```

The Aggregate operators can be used as arguments of the groupBy query primitive, performing an operation for each resulting group. Their functionality is described as follows;

Table 3.5 Offered Aggregate Operators

Aggregate Operators	Arguments
max	(String fieldName)
min	(String fieldName)
avg	(String fieldName)
sum	(String fieldName)
count	()

- *max* - finds the maxiumum value of a field (whose name is passed as an argument) for each group.
- *min* - finds the minimum value of a field (whose name is passed as an argument) for each group.
- *avg* - calculates the average value of a field (whose name is passed as an argument) for each group.
- *sum* - calculates the sum value of a field (whose name is passed as an argument) for each group.
- *count* - calculates for each group the number of identical values that formed the group.

Listing 3.9: Example of using an aggregate operator - For every city in Greece find how many 5 Star hotels exist and an average price per day

```
1  import gr.unipi.api.nosqldb.NoSqlDbOperators;
2  import org.apache.spark.sql.*;
3  import static gr.unipi.api.aggregateOperator.AggregateOperators.*;
4  import static gr.unipi.api.filterOperator.FilterOperators.*;
5
6  public class DataOperationsWithSpark {
7      public static void doOperations(NoSqlDbOperators noSqlDbOp) {
8
9          Dataset<Row> dataset = noSqlDbOp.filter(eq("star", 5))
10         .groupBy("city", count(), avg("approximate_price_per_day")).toDataframe();
11     }
12 }
```

Table 3.6 Offered Sort Operators

Sort Operators	Arguments
asc	(String fieldName)
desc	(String fieldName)

The Sort operators can be used as arguments of the sort query primitive. Their functionality is described as follows;

- *asc* - sorts a given field (whose name is passed as an argument) in ascending order.
- *desc* - sorts a given field (whose name is passed as an argument) in descending order.

Listing 3.10: Example of using a sort operator - Find all of the 5 Star hotels in Greece located in the city of Piraeus and sort them in ascending order by their approximate price per day

```
import gr.unipi.api.nosqldb.NoSqlDbOperators;
import gr.unipi.api.sortOperator.SortOperators;
import org.apache.spark.sql.*;
import static gr.unipi.api.filterOperator.FilterOperators.*;

public class DataOperationsWithSpark {
    public static void doOperations(NoSqlDbOperators noSqlDbOp) {

        Dataset<Row> dataset = noSqlDbOp
        .filter(and(eq("star", 5), eq("city","Piraeus")))
        .sort(SortOperator.asc("approximate_price_per_day")).toDataframe();
    }
}
```

3.2.3 Use case of spatial and spatio-temporal query operations

The geo-operators of Table 3.4 give access to mobility data as they can be used for expressing either spatial or spatio-temporal query operations. The listings below concern the cases of spatial range (rectangle and circle) query operations and spatio-temporal range (box and cylinder) query operations.

Listing 3.11: Spatial rectangle query operation - Count the points that are in a specific spatial rectangle given the coordinates of its lower and upper bounds

```
import gr.unipi.api.nosqldb.NoSqlDbOperators;
import gr.unipi.api.filterOperator.geographicalOperator.Coordinates;
import static gr.unipi.api.filterOperator.FilterOperators.*;

public class DataOperations {
    public static void doOperations(NoSqlDbOperators noSqlDbOp) {

        int count = noSqlDbOp.filter(inGeoBox("location",
        Coordinates.newCoordinates(23.65, 37.94),
        Coordinates.newCoordinates(23.67, 37.96))).count();
    }
}
```

Listing 3.12: Spatial circle query operation - Count the points that are at most 300 meters far from a point given its coordinates

```
import gr.unipi.api.nosqldb.NoSqlDbOperators;
import gr.unipi.api.filterOperator.geographicalOperator.Coordinates;
import static gr.unipi.api.filterOperator.FilterOperators.*;

public class DataOperations {
    public static void doOperations(NoSqlDbOperators noSqlDbOp) {

        int count = noSqlDbOp.filter(inGeoCircleMeters("location",
        Coordinates.newCoordinates(23.65, 37.94), 300)).count();
    }
}
```

Listing 3.13: Spatiotemporal box query operation - Count the points that are in a specific spatial rectangle given a particular time period defined by lower and upper bounds

```
import gr.unipi.api.nosqldb.NoSqlDbOperators;
import gr.unipi.api.filterOperator.geographicalOperator.Coordinates;
import java.text.SimpleDateFormat;
import java.util.Date;
import static gr.unipi.api.filterOperator.FilterOperators.*;

public class DataOperations {
    public static void doOperations(NoSqlDbOperators noSqlDbOp) {

        SimpleDateFormat s = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss.SSS") ;
        Date d1 = s.parse("2017-12-01T00:00:00.000Z"); //lower time bound
        Date d2 = s.parse("2017-12-04T23:59:59.000Z"); //upper time bound

        int count = noSqlDbOp.filter(and(inGeoBox("location",
        Coordinates.newCoordinates(23.65, 37.94),
        Coordinates.newCoordinates(23.67, 37.96)), gte(d1), lte(d2))).count();
    }
}
```

Listing 3.14: Spatiotemporal cylinder query operation - Count the points that are in a specific spatial circle given a particular time period defined by lower and upper bounds

```
import gr.unipi.api.nosqldb.NoSqlDbOperators;
import gr.unipi.api.filterOperator.geographicalOperator.Coordinates;
import java.text.SimpleDateFormat;
import java.util.Date;
import static gr.unipi.api.filterOperator.FilterOperators.*;

public class DataOperations {
    public static void doOperations(NoSqlDbOperators noSqlDbOp) {

        SimpleDateFormat s = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss.SSS") ;
        Date d1 = s.parse("2017-12-01T00:00:00.000Z"); //lower time bound
        Date d2 = s.parse("2017-12-04T23:59:59.000Z"); //upper time bound

        int count = noSqlDbOp.filter(and(inGeoCircleMeters("location",
        Coordinates.newCoordinates(23.65, 37.94), 300), gte(d1), lte(d2))).count();
    }
}
```

3.3 API Architecture

3.3.1 Packages and classes

The API is composed of following standard classes and interfaces, arranged in packages.

- *gr.unipi.api*
  - *Operator* interface

    This interface contains an abstract method for every NoSQL system (e.g. MongoDB) that is supported from the API. Each of the defined methods return an object type that can be utilized from a specific NoSQL database, expressing an operator. The interface is extended by the FilterOperator interface and implemented by the AggregateOperator and the SortOperator abstract class.

- *gr.unipi.api.nosqldb*
  - *NoSqlDbSystem* class

    This class is implemented with the builder design pattern since it is used directly from the users of the API for defining the NoSQL database, its ip, its port, its credentials and the table name for performing query operations.
  - *NoSqlDbOperators* interface

This interface defines all of the offering query primitives of the API. It is implemented by **X**Operators classes where **X** is the name of a NoSQL database such as MongoDB. Essentially, these classes implement the query primitives for a specific NoSQL database system.

- *NoSqlDbConnector* interface

  This interface represents a NoSQL database connector (its requiring elements for establishing a connection), defining all of the methods that should be implemented by a connector of a specific NoSQL database system. It accepts as a generic parameter the object type that represents an established connection to a specific NoSQL database. The connector classes which implement the interface are named **X**Connector where **X** is the name of a NoSQL database.

- *NoSqlDbConnectionManager*

  This class represents the connection manager of a NoSQL database, defining abstract methods that should be implemented for a connection manager of a specific NoSQL database system. It accepts as a generic parameter the object type that represents the connection to a specific NoSQL database for storing it to a HashMap. There are some methods that are already implemented in this class (getConnection) since their functionality is common to all NoSQL database systems. The ConnectionManager classes which extend the abstract class are named **X**ConnectionManager where **X** is the name of a NoSQL database.

- *NoSqlDb* enum class

  This class defines for every supported NoSQL database system an enum type. Its abstract methods are implemented in the body of each enum type, mapping thus the offered abstraction level (NoSQLConnector and NoSQLDbOperators interface) with object types that are oriented to a specific NoSQL database system.

- *gr.unipi.api.filterOperator*

  - *FilterOperator* interface

    This interface defines the FilterOperator type for the Filter operators. It is implemented by the ComparisonOperator, GeographicalOperator and LogicalOperator abstract classes.

  - *FilterOperators* class

    This class contains static methods, offering to the API users all of the instantiable classes that are subtypes of FilterOperator type.

- *gr.unipi.api.filterOperator.comparisonOperator*
  - *ComparisonOperator* abstract class

    This class defines the ComparisonOperator type for the Comparison operators. It is extended by all classes of the same package. It contains two instance variables whose types are String and generic (the generic type is passed as a parameter), storing a field name and a value respectively. These variables are utilized by all children classes in order to apply a specific type of comparison operator.
  - *OperatorEqual* class

    This class represents the equal condition that is applied on a field, given a specific value. It accepts a generic parameter that defines the type of value. The class or its parent class (ConditionOperator) should override the methods that are declared in Operator Interface.
  - *OperatorGreaterThan* class

    This class represents the greater than condition that is applied on a field, given a specific value. It accepts a generic parameter that defines the type of value. The class or its parent class (ConditionOperator) should override the methods that are declared in Operator Interface.
  - *OperatorGreaterThanEqual* class

    This class represents the greater than or equal condition that is applied on a field, given a specific value. It accepts a generic parameter that defines the type of value. The class or its parent class (ConditionOperator) should override the methods that are declared in Operator Interface.
  - *OperatorLessThan* class

    This class represents the less than condition that is applied on a field, given a specific value. It accepts a generic parameter that defines the type of value. The class or its parent class (ConditionOperator) should override the methods that are declared in Operator Interface.
  - *OperatorLessThanEqual* class

    This class represents the less than or equal condition that is applied on a field, given a specific value. It accepts a generic parameter that defines the type of value. The class or its parent class (ConditionOperator) should override the methods that are declared in Operator Interface.
  - *OperatorNotEqual* class

This class represents not equal condition that is applied on a field, given a specific value. It accepts a generic parameter that defines the type of value. The class or its parent class (ConditionOperator) should override the methods that are declared in Operator Interface.

- *gr.unipi.api.filterOperator.geographicalOperator*
  - *Coordinates* class

    This class represents the coordinates of a point. It contains two instance variables whose types are double, storing the longitude and latitude value.
  - *GeographicalOperator* abstract class

    This class defines the GeographicalOperator type for the Geographical operators. It is extended by the abstract classes GeographicalOperatorBasedOnPoints and GeographicalOperatorBasedOnSinglePoint. It contains two instance variables whose types are String and array of coordinates, storing a field name and coordinates. These variables are utilized by children classes in order to apply a specific type of geographical operator.
  - *GeographicalOperatorBasedOnPoints* abstract class

    This class represents a geographical operator that is applied on a specific field based on multiple coordinates (up to one pair). It is extended by OperatorInGeographicalBox and OperatorInGeographicalPolygon classes.
  - *GeographicalOperatorBasedOnSinglePoint* abstract class

    This class represents a geographical operator that is applied on a specific field based on one coordinates pair. It is extended by OperatorInGeographicalCircle and OperatorNearestNeighbors classes.
  - *OperatorInGeographicalBox* class

    This class represents a box geographical operator that is applied a field, given its minimum and upper bound coordinates. The class or its parent class (GeographicalOperatorBasedOnPoints) should override the methods that are declared in Operator Interface.
  - *OperatorInGeographicalCircle* class This class represents a circle geographical operator that is applied a field, given the coordinates of the center of the circle and a radius (in meters) which is stored as a double instance variable. The class or its parent class (GeographicalOperatorBasedOnSinglePoint) should override the methods that are declared in Operator Interface.

- *OperatorInGeographicalPolygon* class

  This class represents a polygon geographical operator that is applied a field, given its coordinates (at least three are needed for polygon formation). The class or its parent class (GeographicalOperatorBasedOnPoints) should override the methods that are declared in Operator Interface.

- *OperatorNearestNeighbors* class

  This class represents a nearest neighbor geographical operator that is applied a field, given a coordinates pair and a number of neighbors (k) which is stored as an integer instance variable. Based on the coordinates, the operator finds the radius of the circle that contains for certain at least k neighbors. Radius determination is achieved by exploiting histrograms that store the number of points in each cell that have been resulted from equi-width space partitioning.

- *gr.unipi.api.filterOperator.logicalOperator*

  - *LogicalOperator* abstract class

    This class defines the LogicalOperator type for the Logical operators. It is extended by all classes of the same package. It contains an instance variable whose type is FilterOperator array, storing FilterOperator object types. This enables the children classes to apply a specific logical type on many filter operators.

  - *OperatorAnd* class

    This class represents the and operator ($\cap$) that is applied on two or more filter operators. The class or its parent class (LogicalOperator) should override the methods that are declared in Operator Interface.

  - *OperatorOr* class

    This class represents the or ($\cup$) operator that is applied on two or more filter operators. The class or its parent class (LogicalOperator) should override the methods that are declared in Operator Interface.

- *gr.unipi.api.aggregateOperator*

  - *AggregateOperator* abstract class

    This class defines the AggregateOperator type for the Aggregate operators. It is extended by all classes of the same package. It includes two instance variables whose types are String, storing a field name and an alias. All of the children classes apply a specific aggregate operator on the field name, projecting the result with the alias.

  - *AggregateOperators* class

    This class contains static methods, offering to the API users all of the instantiable classes that are subtypes of AggregateOperator type.

  - *OperatorAvg* class

    This class represents the average aggregator that is applied on a given field, projecting its result with an alias. The class or its parent class (AggregateOperator) should override the methods that are declared in Operator Interface.

  - *OperatorCount* class

    This class represents the count aggregator that is applied on the records, projecting its result with an alias. When the class is instantiated, the field name that is passed on its parent class is empty since it is not used. The class or its parent class (AggregateOperator) should override the methods that are declared in Operator Interface.

  - *OperatorMax* class

    This class represents the max aggregator that is applied on a given field, projecting its result with an alias. The class or its parent class (AggregateOperator) should override the methods that are declared in Operator Interface.

  - *OperatorMin* class

    This class represents the min aggregator that is applied on a given field, projecting its result with an alias. The class or its parent class (AggregateOperator) should override the methods that are declared in Operator Interface.

  - *OperatorSum* class

    This class represents the sum aggregator that is applied on a given field, projecting its result with an alias. The class or its parent class (AggregateOperator) should override the methods that are declared in Operator Interface.

- *gr.unipi.api.sortOperator*

  - *SortOperator* abstract class

    This class defines the SortOperator type for the Sort operators. It is extended by all classes of the same package. It contains two instance variables whose types are String and int, storing a field and and an order (1 for ascending, -1 for descending). This enables the children classes to apply a specific sorting on a field.

- *SortOperators* class

  This class contains static methods, offering to the API users all of the instantiable classes that are subtypes of SortOperator type.

- *OperatorAsc* class

  This class represents the ascending sorting of values that is applied on a field. The class or its parent class (SortOperator) should override the methods that are declared in Operator Interface.

- *OperatorDesc* class

  This class represents the descending sorting of values that is applied on a field. The class or its parent class (SortOperator) should override the methods that are declared in Operator Interface.

3.3.2 Connector and connection manager

A NoSqlDbConnectionManager type class is a singleton class named **X**ConnectionManager where **X** is a NoSql database system. This class stores for the **X** database all of the open connections that have not been closed in order to reuse them, avoiding thus the overhead of a new connection establishment. Open connections are stored in a HashMap where the key is a Connector type class and the value is an object that represents an open connection of **X** database system. When defining the parameters that are required for accessing a NoSQL database (see code lines 6-7 and 12-14 of Listings 3.3 and 3.4 respectively), a Connector type object is created on the background and checked if it is contained in the HashMap. If the connector is contained in the HashMap, then its value is used directly for accessing the database. Otherwise, the connector creates a connection and is subsequently stored with the established connection in the HashMap. A **X**ConnectionManager singleton class is used only from the **X**Operators class which is a type of NoSqlDbOperators object.

A Connector class type (**X**Connector) represents all of the elements needed for establishing a connection (ip, port, username, password and database name) on a specific database (**X**). The connection is established by the connector only if it is not contained in the HashMap that is managed from a ConnectionManager. A **X**Connector class is created by the method createNoSqlDbConnector of **X** enum type of the NoSqlDb enum class.

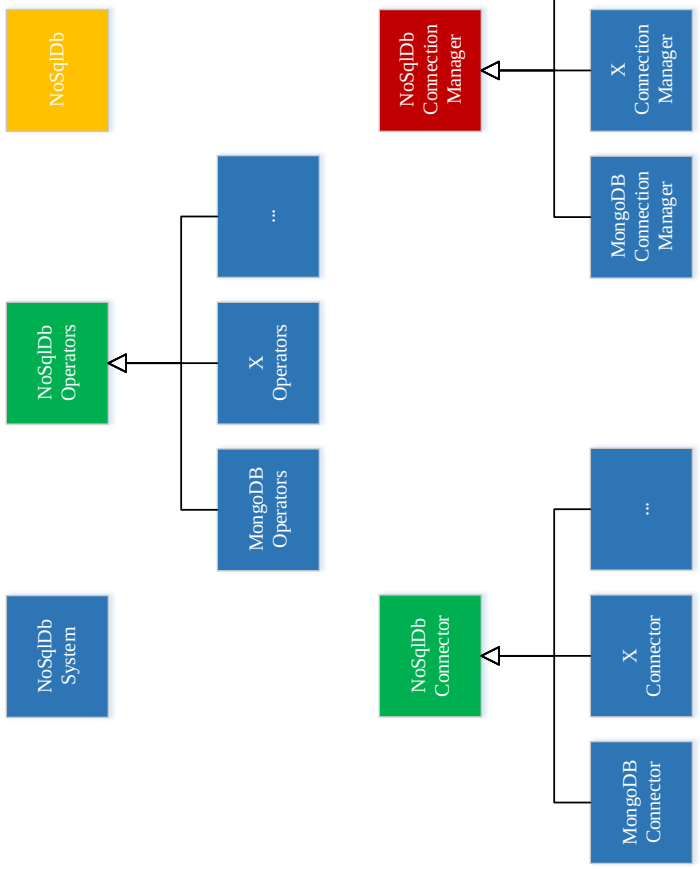

Fig. 3.1 Class diagram of gr.unipi.api.nosqldb package. Interfaces, classes, abstract classes and enum classes are illustrated with green, blue red and orange color respectively.

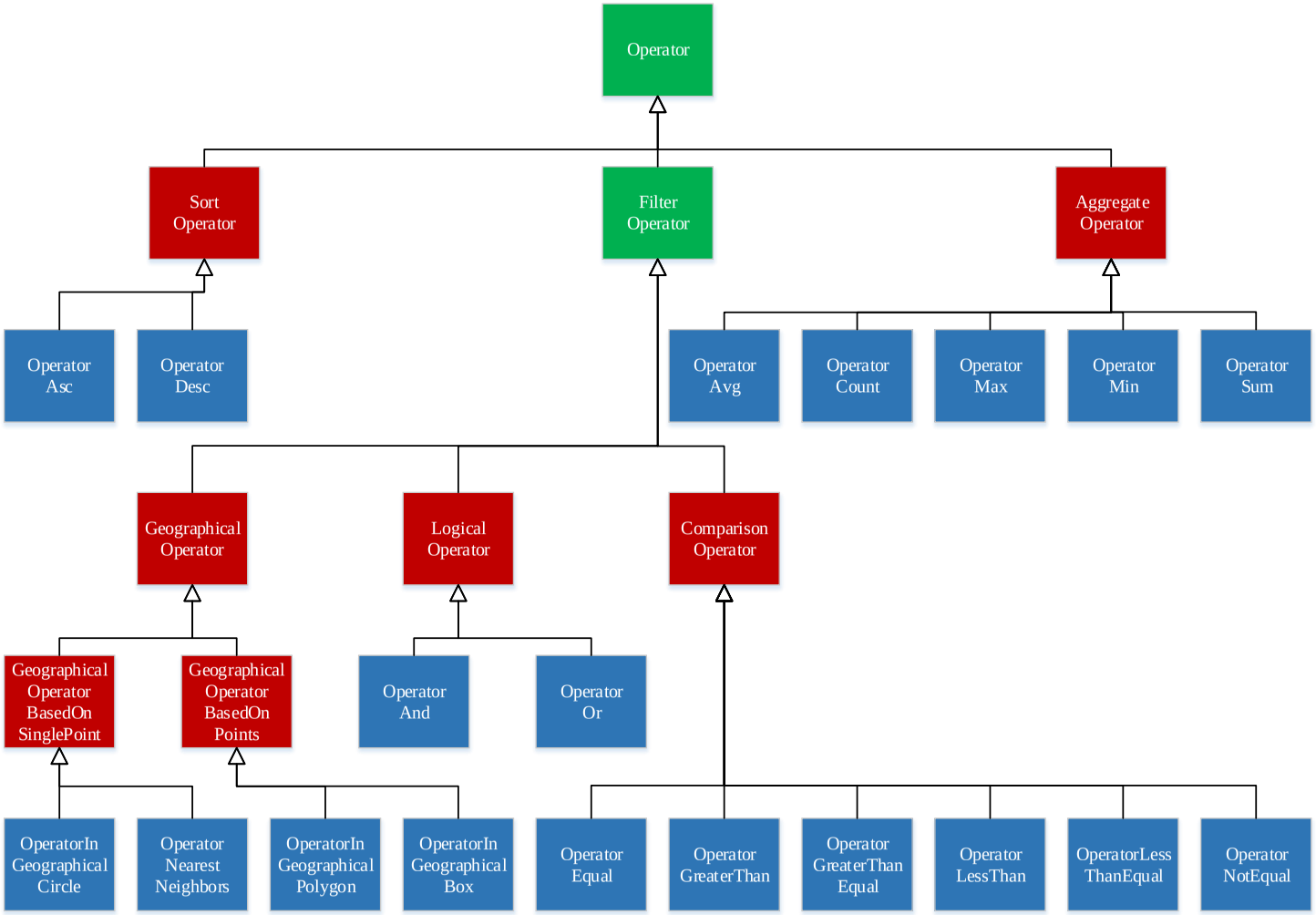

Fig. 3.2 Class diagram of API operators. Interfaces, classes and abstract classes are illustrated with green, blue and red color respectively.

3.4 Extending the API for supporting a NoSQL database system

To extend the API for supporting a NoSQL database system named **X**, at first, the following classes should be created in package gr.unipi.api.nosqldb;

- *XOperators*
- *XConnector*
- *XConnectionManager*

These classes should be arranged in the class hierarchy as shown in Figure 3.1, implementing their parents abstract methods. The generic type arguments of classes **X**Connector and **X**ConnectionManager must be an object type through which the database can be managed. This type is offered by the native client API of database.

The **X**Connector class should implement the methods createConnection, hashCode and equals which are used in the **X**ConnectionManager for creating and storing a connection to the HashMap. This class must contain only the elements as instance variables that are used for the connection establishment via the native API of the database. The functionality of equal method should be based on the defined instance variables.

The **X**ConnectionManager class must be a singleton class, implementing the closeConnection and closeConnections methods, used for closing open connections through the object whose type is defined by the generic argument.

XConnectionManager and **X**Connector classes are used in the **X**Operators class (XConnector type object is passed as an argument through constructor). The XOperators class implements all of the query primitive methods that are defined in the NoSqlDbOperators interface.

In order to get the expression of an operator in **Y** object type which we assume that it can be used by the NoSQL database for performing operations, a method get**Y** has to be declared in Operator interface and be implemented from all classes whose name start from the 'Operator' word (Fig. 3.2). The methods get**Y** are called from the object type parameters of the query primitives in XOperators class. Note that the primitives that belong to the definition phase (Table 3.1) should store the **Y** object types to a list. The query primitives belonging to the execution phase will use this list, the passed connector and the connection manager for performing operations on the database.

Furthermore, an enum type named **X** has to be declared in the NoSqlDb enum class and implement all of its abstract methods. This enum type should be also declared in a new static method that returns a Builder object type in NoSqlDbSystem class.

Chapter 4

Description of query types

In this section we describe the query operator types that are used for accessing the MongoDB through the NoSQL data access API, focusing on the spatial query types. We also report a technique for performing k-NN spatial queries upon MongoDB, considering that they are not supported.

4.1 Aggregation pipeline stages & API operators

Since the API provides the flexibility of executing a sequence of operations (query primitives), we use for the MongoDB database the provided aggregation pipeline framework. The framework is modeled on the concept of data processing pipelines, meaning that the documents enter a multi-stage pipeline that transforms them into aggregated results. By defining multiple primitives (Listing 3.5 in subsection 3.2.2), multiple stages are declared in the pipeline. The stages are executed in sequence, composing a series of operations that are performed on the documents.

All of the aggregate operators that are passed as arguments in the group by primitive (Table 3.1) and the max, min, sum avg and distinct primitives are declared in the *$group* stage. The group stage, groups documents by some specified expression and outputs to the next stage a document for each distinct grouping. The limit, sort and project primitives are declared in the *$limit*, *$sort* and *$project* stages. The *$limit* stage, limits the number of documents passed to the next stage in the pipeline. The *$sort* stage, sorts all input documents and returns them to the pipeline in sorted order. The *$project* stage, passes along the documents with the requested fields to the next stage in the pipeline. All of the filter operators that are passed as arguments in the filter primitive, are declared in the *$match* stage, except the nearestNeighbors geographical operator which is declared in the *$geoNear* stage. The *$match* stage, filters the documents to pass only the documents that match the specified

condition(s) to the next pipeline stage. The *$geoNear* stage, outputs documents in order of nearest to farthest from a specified point.

Pipeline stages have a limit of 100 megabytes of RAM. MongoDB produces an error if this limit is exceeded. To handle the limit for large datasets, we can use the allowDiskUse option which enables aggregation pipeline stages to write data to temporary files. Moreover, the *$geoNear* stage should be declared as the first stage of the pipeline. This means that the geographical nearestNeighbors operator should be passed as an argument only to the first-declared (filter) primitive in the NoSQL data access API, when operating on MongoDB.

4.2 Geospatial query operators & API geographical operators

All of the geographical operators of the API are declared in the *$match* stage of the aggregation pipeline framework, except the nearestNeighbors geographical operator which is declared in the *$geoNear* stage. The geographical box and polygon operators, use the *$geoWithin* with the *$geometry* query operator of MongoDB, selecting documents with geospatial data that exists entirely within a specified shape under the *$match* stage (Listing 4.1). The geographical circle operator uses the *$geoWithin* with the *$centerSphere* query operator of MongoDB, selecting documents within the bounds of the circle under the *$match* stage (Listing 4.2). The geographical nearestNeighbors operator is declared directly under the *$geoNear* stage (Listing 4.3). This stage outputs sorted documents by their distance from a specific point (nearest to farthest). A maximum distance can be set from the center point in order to limit the results to those documents that fall within the distance (circle query). Similarly, minimum distance can be set from the center point so as to limit the results to those documents that fall outside the distance. Maximum and minimum distances can be combined, forming thus an annulus query.

The *$geoWithin* query operator does not require a geospatial index for its performance. However, a geospatial index will improve query performance. Both of the offered spatial indexes (2d and 2dSphere) support *$geoWithin*. The *$geoNear* stage does require a geospatial index for its performance.

Listing 4.1: Format of match stage with $geoWithin and $geoWithin geospatial query operators

```
db.geoPoints.aggregate([
    {
        $match:{
            <location field>:{
                $geoWithin:{
                    $geometry:{
                        type:"Polygon",
                        coordinates:[ <coordinates> ]
                    }
                }
            }
        }
    }
])
```

Listing 4.2: Format of match stage with $geoWithin and $centerSphere geospatial query operators

```
db.geoPoints.aggregate([
    {
        $match:{
            <location field>:{
                $geoWithin:{
                    $centerSphere:[
                        [ <x>, <y> ], <radius>]
                }
            }
        }
    }
])
```

Listing 4.3: Format of geoNear stage

```
db.geoPoints.aggregate([
    {
        $geoNear:{
            near:{
                type:"Point",
                coordinates:[ <x>, <y> ]
            },
            distanceField:"dist.calculated",
            maxDistance: <radius in meters>,
            includeLocs:"dist.location",
            num:5,
            spherical:true
        }
    }
])
```

4.3 Serving k-NN over spatial circle queries

Considering that MongoDB does not support k-NN type queries, we manage to perform such queries over spatial circle queries.

Specifically, given a specified point and the number of neighboors (k), we perform a circle query with an approximated radius, so that it will return at least k documents. The circle query is performed by using the *$geoNear* stage as it outputs documents in order of nearest to farthest from a specified point. This facilitates the cases where the fetched documents are more than k, as the results are limited to the first k since they are sorted by distance.

The ideal condition of executing k-NN queries over circle queries would be to know the exact radius that would return precisely k results. By finding the radius for the query that will return at least k results, limits the circle query to be executed only once for serving a k-NN query. However, the trap of radius overestimation exists (finding a far larger radius than the actual). It should be mentioned that the occurrence of overestimation state depends on the distribution of the data. For instance, if data follows uniform distribution, overestimation of the radius will not be occurred.

The approach that is adopted for the determination of the radius of a circle query that returns at least k documents, is based on the grid partitioning method on the 2D space. This approach results to space cells whose number depends on the number of splits on the two dimensions. For each cell, the number of the contained points is calculated and then stored in an in-memory key-value data structure. The key and the value represent respectively the unique number of a cell and the number of points that encloses. This key-value data structure constitutes a histogram and is extracted to a serializable form on the magnetic disk. The procedure of histogram's creation presumes the scanning of the whole dataset and therefore is considered as a pre-processing step (Figure 4.1).

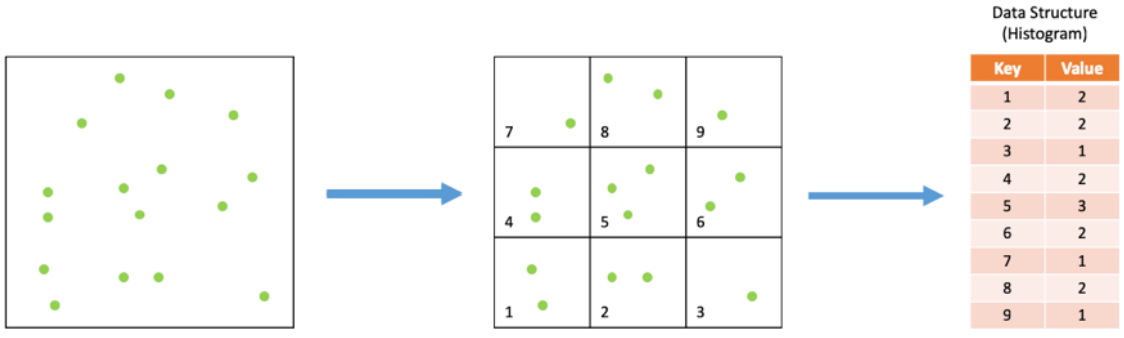

Fig. 4.1 The procedure of producing a histogram of a dataset containing points, by using the grid partitioning method.

Having the histogram been formed, it can be exploited directly for serving k-NN queries over circle queries. Assuming that our dataset contains coordinates of restaurants and we want to find the k nearest restaurants from a given point (red point in Figure 4.2), our algorithm firstly finds the point's cell id (which is 5). Then, by using the histogram, the number of contained restaurants in the cell is found. If more than k restaurants are enclosed (or equal), the distances from the point to the four corners of the cell are calculated (blue lines). From the found distances, the radius of the circle query that will be performed is set to the maximum one (bold blue line). By setting the radius to the maximum distance from the calculated distances, we make sure that the cell will be totally included in the (red) circle query (and a part outside of it). Thus, we definitely expect that the circle query will return at least k restaurants. If less than k restaurants are included in the cell, the number of the included restaurants of the adjacent cells are summed up and added to the restaurants of the cells that have already been examined (cell 5). By scanning the adjacent cells is like forming a larger cell (orange box in Figure 4.3) and determining the enclosed restaurants of it (by using the histogram). If the resulting number of restaurants are equal or more than k, then the distances from the given point to the four corners of the larger cell are calculated. From the found distances, the radius of the circle query that will be performed is set to the maximum one (bold blue line). By setting the radius to the maximum distance from the calculated distances, we make sure that the larger (orange) cell will be totally included in the (red) circle query (and a part outside of it). If the resulting number of restaurants is less than k, the same procedure is applied and a larger cell is examined every time. The procedure is repeated until the resulting number is equal or greater than k.

The implemented algorithm has two advantages in terms of performance:

- When a larger cell is formed, the number of the contained points of the cells that have been examined so far (not the adjacent cells), is already summed up and cached. Therefore, when adjacent cells are examined, there is no reason of scanning all of the cells that constitute the larger cell.

- The radius is calculated in the final stage of the algorithm's execution - when the final larger cell has been formed, including at least k points. The calculation of the radius has a constant performance since only the distances from the point to the four corners are computed.

The performance of our algorithm is affected if numerous cells are to be accessed in order to reach at least k neighbors. To some extent, this can be alleviated if the space partitioning result to less cells.

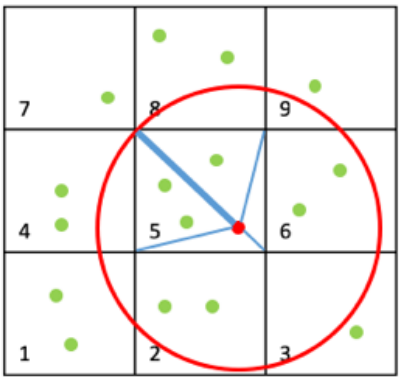

Fig. 4.2 The circle query that will be performed if the desired (k) neighbors is set equal or less than 3. The bold blue line is the radius of the (red) circle query which contains the cell 5.

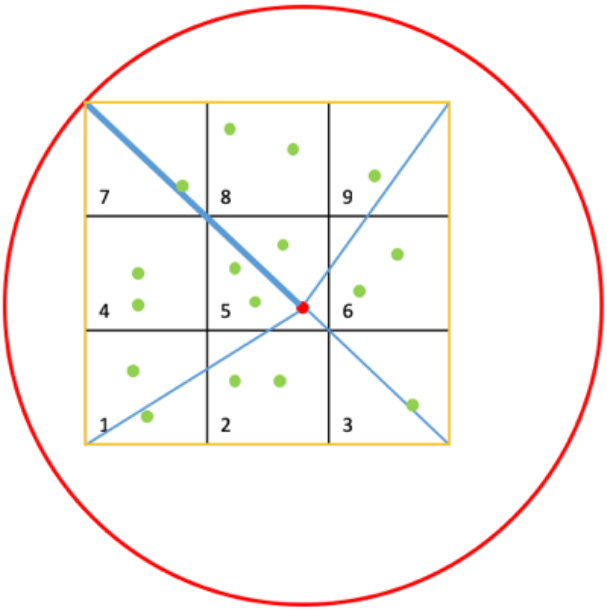

Fig. 4.3 The circle query will be performed if the desired (k) neighbors are set equal of greater than 4. The bold blue line is the radius of the (red) circle query which contains the cells 1-9. The orange box represents the larger cells which is examined in order to determine if it contains at least k points.

Chapter 5

Experimental evaluation

In this chapter, extensive experiments are conducted on mobility data stored in MongoDB database so as to study the performance of querying spatial and spatio-temporal data on NoSQL Document Stores.

The following three mobility datasets are used;

- Real mobility dataset (hereafter *RE*) - This dataset contains 35.5 million points of trajectories, formed by the GPS traces of transport vehicles in the region of Greece. The points are located on the road network (or near the road due to noisy location information provided by GPS measurement). The trajectories cover the time period 01/06/2017 - 30/06/2018. The mobility data is illustated in Figure 5.1

- Synthetic mobility dataset (hereafter *SYNTH1*) - This dataset was generated with 35.5 million spatio-temporal points in the region of Greece (specifically in the bounding rectangle illustrated in Figure 5.2), covering the same timespan as the real mobility dataset does. Both spatial and temporal generated information follows uniform distribution. As a result, the points may not exist only on the road network, but anywhere in the specified area.

- Synthetic mobility dataset ($\times 2$) (hereafter *SYNTH2*) - This dataset has the same features as the synthetic mobility dataset does have, being twice the size as it contains 71 (35.5×2) million spatio-temporal points.

These datasets were inserted in separate databases of a MongoDB single-node instance, installed on a computer equipped with 3.6GHZ Intel core i7-4790 processor, 16GB DDR3 1600MHz RAM, 1TB hard disk drive and Ubuntu 18.04.1 LTS operating system. The version of the used MongoDB database was 4.0.3.

Fig. 5.1 Data distribution of *RE* dataset.

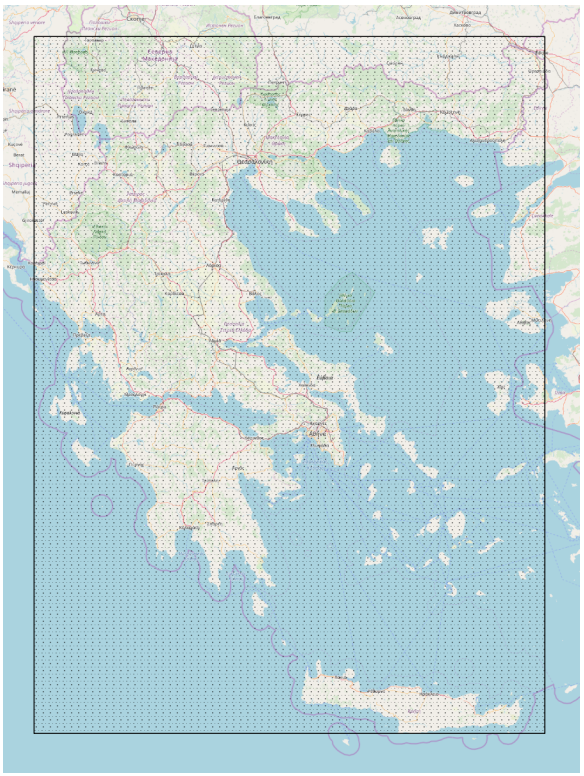

Fig. 5.2 Illustration of the bounding rectangle in which the synthetic data was generated. The coordinates (longitude, latitude) of the lower and upper bounds are (20.1500,34.9199) and (26.6041, 41.8269) respectively.

The three datasets exist in the form of CSV files and are parsed record by record in order to be inserted in the databases of MongoDB instance. The CSV files that compose each dataset contain the following columns;

1. Object (Vehicle) ID
2. Longitude
3. Latitude
4. Date

These columns are inserted as fields of every document in the databases. Object ID column is inserted as String format, longitude and latitude columns as GeoJSON object and date column as Date format. The values of the columns of the synthetic CSV datasets are uniformly distributed in their respective ranges (the Object ID values were generated as two underscore-separated random three-digit Integers).

We notice that the size of the datasets (Table 5.1) on the MongoDB is different than their corresponding size on the magnetic disk (CSV files) since the data is stored compressed (snappy block compression is used).

The spatial index that is used for the experiments purposes is the 2d sphere index that MongoDB offers, built on the GeoJSON field. This index supports queries on an earth-like sphere. It matches our case since the datasets we are using contain geographical coordinates. MongoDB also offers the 2d index, supporting queries on a two-dimensional plane.

All of the performed queries are executed in the context of experimental procedures in cold state, meaning that the system's cache has been previously cleaned. A query execution is equivalent to a count operation, calculating number of points that are contained in it. In some cases we execute the queries 3 times consecutively so as to specify the utility of cache.

Table 5.1 Mobility Datasets

Dataset	**# Points**	**Size on MongoDB**	**Size on Magnetic Disk**	**# CSV files of dataset**
RE	35.5 million	1.35 GB	1.93 GB	10,328
SYNTH1	35.5 million	1.97 GB	1.77 GB	10,328
SYNTH2	71 million	3.94 GB	3.54 GB	20,656

5.1 Experimental study of spatial filtering

In this section we investigate the performance of spatial range queries on MongoDB by exploiting the offered 2d sphere spatial index. We conduct three sets of experiments on the datasets *RE*, *SYNTH1* and *SYNTH2*. In the first set (subsection 5.1.1), we study the case of rectangular spatial queries with and without using a spatial index, in order to obtain a quantitative view of how much time can be saved by using the appropriate built-in indexes. In the second set (subsection 5.1.2), we study the minimum time needed to perform a spatial range query on the stored data. This experiment, facilitates our experimental study towards defining a minimum overhead needed to query the MongoDB instance, which might be affected by the network overhead, and other internal MongoDB operations. In the third set (subsection 5.1.3), we study the scalability of range spatial queries by experimenting with various range sizes.

Table 5.2 Spatial Indexes

Dataset	Spatial index construction time	Spatial index size
RE	2.15 min	417.21 MB
SYNTH1	2.68 min	498.64 MB
SYNTH2	5.4 min	994.64 MB

As shown in Table 5.2, the spatial index size of the *SYNTH2* dataset is approximately twice the size compared to the indexes of the other datasets as it contains double number of points, requiring more time for its construction. The indexes of other two datasets have a small size difference because the *RE* dataset is a little bit smaller than the *SYNTH1*. This has a small impact on their index construction time.

5.1.1 Performance of spatial filtering with and without index

In this set of experiments, we study for each dataset the effect of the 2dsphere index on the performance of a rectangular query (Tables 5.4 and 5.5). Simultaneously, we study the benefits gained from the usage of MongoDB internal cache by executing each query three times. The first query execution evaluates query results in cold state, meaning that the data reside in hard disk drives, and no data have been loaded in MongoDB caches. The other two executions, evaluate the query results in warm state, since a subset of the stored data is already loaded in MongoDB caches by previous query executions. We pick the rectangular queries shown in Table 5.3, which are executed on the respective datasets.

Table 5.3 Spatial rectangle queries Q_1^S, Q_2^S

	Q_1^S	Q_2^S
Lower spatial bound (lon, lat)	(23.6266, 37.9262)	(23.5500, 37.9262)
Upper spatial bound (lon, lat)	(23.6682, 37.9477)	(23.9500, 38.3000)
Area	8.71 km^2	1452.93 km^2

Fig. 5.3 Illustration of Q_1^S spatial range query in red color, covering the area of Piraeus.

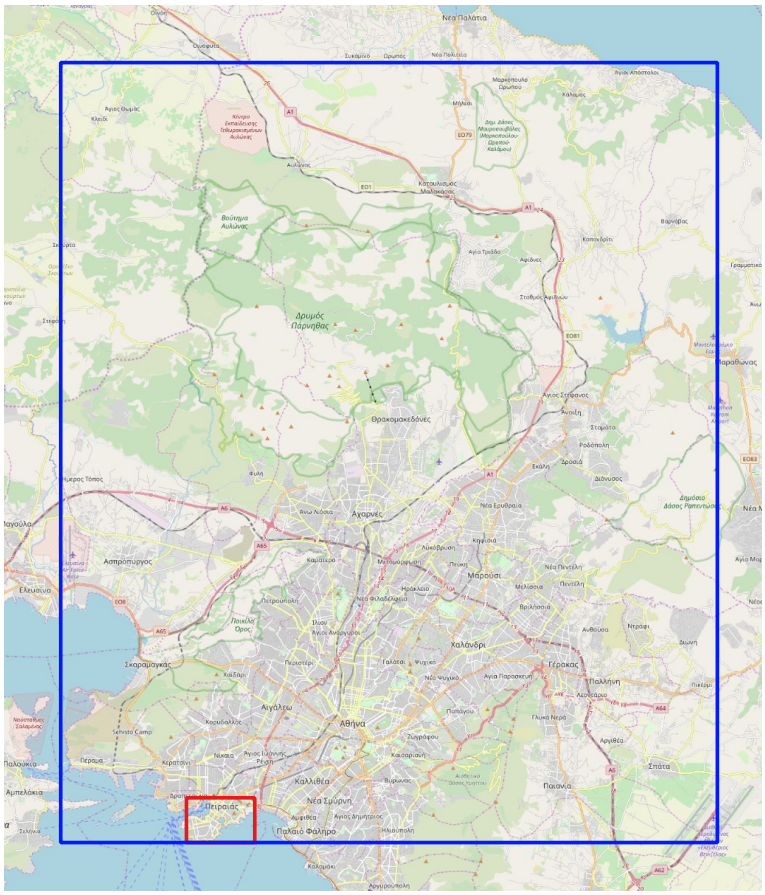

Fig. 5.4 Illustration of Q_2^S spatial range query in blue color, covering the area of Athens and Piraeus. The rectangle with the red color is the Q_1^S spatial query.

Table 5.4 Performance of spatial rectangle queries Q_1^S, Q_2^S with spatial (2d sphere) index usage

Run time execution	Q_1^S **- *RE* dataset**	Q_2^S **- *SYNTH1* dataset**	Q_2^S **- *SYNTH2* dataset**
1	65.90 sec	783.16 sec	1581.34 sec
2	1.29 sec	0.54 sec	1.96 sec
3	1.27 sec	0.53 sec	1.77 sec
# Examined index keys	508,444	167,627	334,720
# Counted documents	294,961	120,091	239,281

Table 5.5 Performance of spatial rectangle queries Q_1^S, Q_2^S without spatial index usage

Run time execution	Q_1^S **- *RE* dataset**	Q_2^S **- *SYNTH1* dataset**	Q_2^S **- *SYNTH2* dataset**
1	26.76 sec	34.50 sec	66.30 sec
2	18.53 sec	20.41 sec	45.39 sec
3	18.51 sec	20.40 sec	45.54 sec

Comparing the Tables 5.4 and 5.5, we notice that querying the MongoDB with a 2d sphere index in cold cache state, more time is required than without using index. Since the index of each dataset is relatively large, MongoDB needs additional time to load the index in main memory before being able to query it. This results to additional reads from disk, requiring thus more time for the execution of a query.

The queries are executed faster in warm state than in cold state in both cases. The difference between the warm and cold state is much more sensible when index is used. This happens because the index is loaded (maintained) in main memory in warm state, thus accelerating the execution of a query. Also, when using index, the execution time of a query in warm state depends on the number of documents (counted documents for our queries) that are retrieved. For example, Q_2^S was executed on *SYNTH1* dataset in less time than Q_1^S on *RE* dataset and Q_3^S on *SYNTH2* dataset, as fewer documents were counted.

Furthemore, despite that Q_1^S examines more documents on *RE* dataset than Q_2^S on the other datasets, it requires less time for its execution in cold state when index is used. Q_2^S covers 170.5 times more area than Q_1^S, requiring to access more blocks on disk because a larger part of the corresponding indexes should to be loaded in-memory for Q_2^S than Q_1^S.

5.1.2 Minumum overhead of spatial filtering

In this set of experiments, we measure the time needed to retrieve just a single document from each dataset that is stored in a collection of a distinct database, when using a spatial circular range as filtering criterion (Table 5.6).

Table 5.6 Spatial tiny circle queries Q_3^S, Q_4^S

	Q_3^S	Q_4^S
Center coordinates (lon, lat)	(25.751467, 35.023487)	(22.317873, 38.565775)
Radius	0.02 m	0.02 m
Area	0.0012 m^2	0.0012 m^2

Table 5.7 Performance of tiny circle queries Q_3^S, Q_4^S with spatial index usage

	Q_3^S **- *RE* dataset**	Q_4^S **- *SYNTH1* dataset**	Q_4^S **- *SYNTH2* dataset**
Execution time	289 msec	276 msec	322 msec
# Examined index keys	17	13	14
# Counted documents	1		

Table 5.7 reports for each dataset the minimum required time for the execution of a spatial range (circle) query in cold cache by using index. The minimum overhead is found by selecting the queries Q_3^S and Q_4^S which fetch (count in our case) only a single document from the respective datasets. Q_3^S requires 13ms more for its execution on *RE* than Q_4^S execution on *SYNTH1*, as 4 more index keys are examined. Additionally, the execution of Q_4^S on *SYNTH2* requires a few more milliseconds than its execution on *SYNTH1*, although the examined keys are about the same (the difference is 1). The extra overhead of *SYNTH2* results from its index size, as it contains double entries from *SYNTH1*.

5.1.3 Scalability of spatial filtering

In this set of experiments, we study the time needed for each dataset to evaluate range queries over increasing range sizes. We pick the circle queries shown in Table 5.8. For every subsequent size factor, the range size doubles.

Table 5.8 Spatial Circle Queries Q_5^S, Q_6^S with size factor

	Q_5^S		Q_6^S	
Size factor	**Radius**	**Area**	**Radius**	**Area**
f1	2.18 × 1 km	14.93 km^2	27.165 × 1 km	2318.29 km^2
f2	2.18 × 2 km	59.72 km^2	27.165 × 2 km	9273.19 km^2
f3	2.18 × 4 km	238.88 km^2	27.165 × 4 km	37092.77 km^2
f4	2.18 × 8 km	955.52 km^2	27.165 × 8 km	148371.08 km^2
Center coordinates (lon, lat)	(23.7613, 37.9864)			

Table 5.9 Performance on varying the size factors for circle queries Q_5^S, Q_6^S with spatial index usage

	Size factor	Q_5^S **- *RE* dataset**	Q_6^S **- *SYNTH1* dataset**	Q_6^S **- *SYNTH2* dataset**
Execution time	f1	75.80 sec	916.46 sec	1831.92 sec
	f2	213.12 sec	964.05 sec	1928.85 sec
	f3	264.45 sec	977.97 sec	2008.73 sec
	f4	282.22 sec	1026.89 sec	2263.99 sec
# Counted documents	f1	421,934	190,235	379,813
	f2	1,456,145	759,739	1,516,315
	f3	6,517,900	3,035,062	6,068,820
	f4	11,817,952	12,138,692	24,275,830

Table 5.9 shows how a spatial range (circle) query scales on each dataset in cold cache state by using index. By increasing the radius of the respective query that is performed on each dataset, we notice that more time is needed for its execution (Figure 5.5 and 5.7). This is reasonable because the covered area of the query gets larger and thus more documents are fetched (Figure 5.6 and 5.8). This results to access more disk blocks, loading a larger part of the index in the main memory as the queries are executed in cold cache.

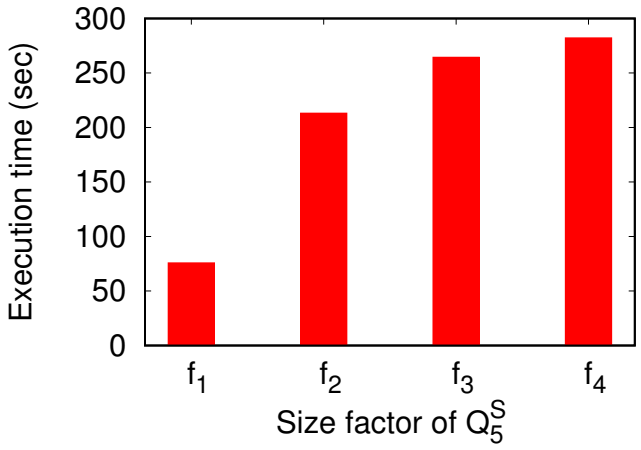

Fig. 5.5 Performance of the size factors of Q_5^S on *RE* dataset.

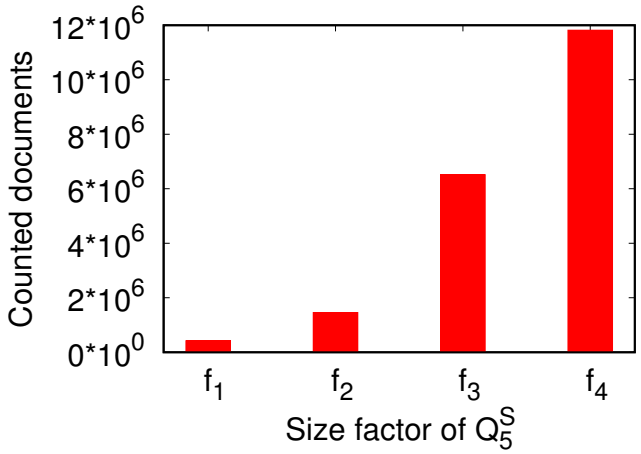

Fig. 5.6 Output size (# counted documents) of the size factors of Q_5^S on *RE* dataset.

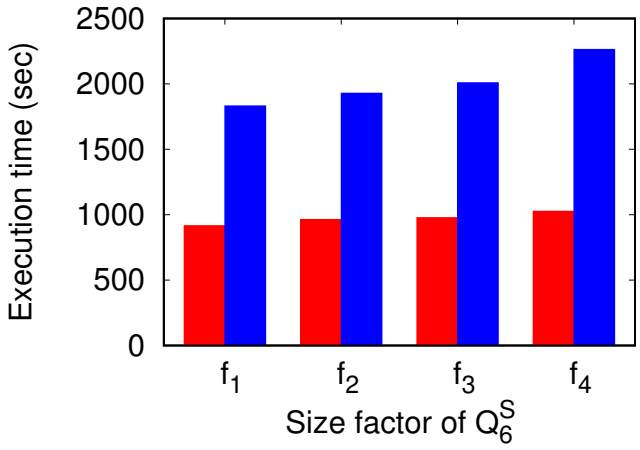

Fig. 5.7 Performance of the size factors of Q_6^S on *SYNTH1* (red color) and *SYNTH2* (blue color) datasets.

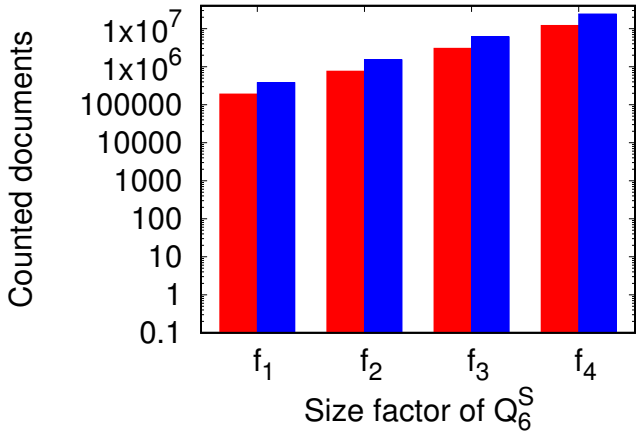

Fig. 5.8 Output size (# counted documents) of the size factors of Q_6^S on *SYNTH1* (red color) and *SYNTH2* (blue color) datasets. The scale of y axis is logarithmic.

5.2 Experimental study of spatio-temporal filtering

In this section we investigate the performance of spatio-temporal range queries on MongoDB by exploiting the offered compound index (spatial index combined with the time field). We conduct three sets of experiments on the datasets RE, SYNTH1 and SYNTH2. In the first set (subsection 5.2.1), we study the case of box spatio-temporal queries by using compound and

spatial index, in order to obtain a quantitative view of how much time can be saved by using the appropriate built-in indexes. In the second set (subsection 5.2.2), we study the minimum time needed to perform a spatio-temporal range query on the stored data. This experiment, facilitates our experimental study towards defining a minimum overhead needed to query the MongoDB instance, which might be affected by the network overhead, and other internal MongoDB operations. In the third set(subsection 5.2.3), we study the scalability of range spatio-temporal queries by experimenting with various range sizes.

Table 5.10 Compound Indexes

Dataset	**Compound index construction time**	**Compound index size**
RE	3.05 min	747.68 MB
SYNTH1	3.80 min	840.17 MB
SYNTH2	7.46 min	1677.66 MB

As shown in Table 5.10, the compound index size of the SYNTH2 dataset is approximately twice the size compared to the indexes of the other datasets as it contains double number of spatio-temporal points, requiring more time for its construction. The indexes of other two datasets have a small size difference because the RE dataset is a little bit smaller than the SYNTH1. This has a small impact on their index construction time.

5.2.1 Performance of spatio-temporal filtering with compound and spatial index

In this set of experiments, we study for each dataset the effect of the compound index on the performance of a box query (Tables 5.13 and 5.12). Simultaneously, we study the benefits gained from the usage of MongoDB internal cache by executing each query three times as we did with the rectangular queries in subsection 5.1.1. We pick the box queries shown in Table 5.11, which are executed on the respective datasets.

Table 5.11 Spatio-temporal box queries Q_1^{ST}, Q_2^{ST}

	$\boldsymbol{Q_1^{ST}}$	$\boldsymbol{Q_2^{ST}}$
Spatial Part	Same as Q_1^S	Same as Q_2^S
Lower time bound	2017-06-29 00:00:00	
Upper time bound	2017-12-31 23:59:59	

Table 5.12 Performance of spatio-temporal box queries Q_1^{ST}, Q_2^{ST} with spatial ((2d sphere) index usage

Run time execution	Q_1^{ST} **- *RE* dataset**	Q_2^{ST} **- *SYNTH1* dataset**	Q_2^{ST} **- *SYNTH2* dataset**
1	69.35 sec	785.97 sec	1585.26 sec
2	1.28 sec	0.55 sec	2.14 sec
3	1.27 sec	0.55 sec	1.96 sec

Table 5.13 Performance of spatio-temporal box queries Q_1^{ST}, Q_2^{ST} with compound index usage

Run time execution	Q_1^{ST} **- *RE* dataset**	Q_2^{ST} **- *SYNTH1* dataset**	Q_2^{ST} **- *SYNTH2* dataset**
1	31.68 sec	515.51 sec	1121.744 sec
2	0.79 sec	0.41 sec	0.83 sec
3	0.79 sec	0.39 sec	0.82 sec
# Examined index keys	456,047	167,644	334,737
# Counted documents	127,133	60,969	121,296

Comparing the Tables 5.13 and 5.12, we notice that spatio-temporal queries are executed faster on compound index than on spatial index, both in cold and warm state. By executing a spatio-temporal query with spatial index, the query is first filtered on its spatial part, and then on its temporal part. When using a compound index for such queries, spatial and temporal part is filtered simultaneously as its structure holds references to two fields (location and time field).

By using a compound index, the execution time of a query in warm state depends on the number of documents (counted documents for our queries) that are retrieved. For example, Q_2^{ST} was executed on *SYNTH1* dataset in less time than Q_1^{ST} on *RE* dataset and Q_3^{ST} on *SYNTH2* dataset, as fewer documents were counted.

The queries are executed faster in warm state than in cold state in both cases. The difference between the warm and cold state is much more sensible when index is used. This happens because the index is loaded (maintained) in main memory in warm state, thus accelerating the execution of a query. Also, when using index, the execution time of a query in warm state depends on the number of documents (counted documents for our queries) that

are retrieved. For example, Q_2^S was executed on *SYNTH1* dataset in less time than Q_1^S on *RE* dataset and Q_3^S on *SYNTH2* dataset, as fewer documents were counted.

Moreover, despite that Q_1^{ST} examines more documents on *RE* dataset than Q_2^{ST} on the other datasets, it requires less time for its execution in cold state when index is used. Q_2^{ST} covers 170.5 times more space than Q_1^{ST}, requiring to access more blocks on disk because a larger part of the corresponding compound indexes should to be loaded in-memory for Q_2^{ST} than Q_1^{ST}.

5.2.2 Minumum overhead of spatio-temporal filtering

Table 5.14 Spatial tiny cylinder queries Q_3^{ST}, Q_4^{ST}

	Q_3^{ST}	Q_4^{ST}
Spatial Part	Same as Q_3^S	Same as Q_4^S
Lower time bound	2018-02-22 08:55:52	2018-04-30 18:49:49
Upper time bound	2018-02-22 08:55:54	2018-04-30 18:49:51

Table 5.15 Performance of tiny cylinder queries Q_3^{ST}, Q_4^{ST} with compound index usage

	Q_3^{ST} **- *RE* dataset**	Q_4^{ST} **- *SYNTH1* dataset**	Q_4^{ST} **- *SYNTH2* dataset**
Execution time	292 msec	283 msec	317 msec
# Examined index keys	17	13	14
# Counted documents	1		

Table 5.15 reports for each dataset the minimum required time for the execution of a spatio-temporal range (cylinder) query in cold cache by using compound index. The minimum overhead is found by selecting the queries Q_3^{ST} and Q_4^{ST} which fetch (count in our case) only a single document from the respective datasets. Q_3^{ST} requires 9ms more for its execution on *RE* than Q_4^{ST} execution on *SYNTH1*, as 4 more index keys are examined. Additionally, the execution of Q_4^{ST} on *SYNTH2* requires a few more milliseconds than its execution on *SYNTH1*, although the examined keys are about the same (the difference is 1). The extra overhead of *SYNTH2* results from its index size, as it contains double entries from *SYNTH1*.

5.2.3 Scalability of spatio-temporal filtering

Table 5.16 Spatial cylinder queries Q_5^{ST}, Q_6^{ST} with size factor

			Q_5^{ST}	Q_6^{ST}
Size factor	**Lower time bound**	**Upper time bound**	**Spatial part**	**Spatial part**
f1	2017-06-29 00:00:00	2017-09-30 23:59:59	Same as f1 of Q_5^S	Same as f1 of Q_6^S
f2		2017-12-31 23:59:59	Same as f2 of Q_5^S	Same as f2 of Q_6^S
f3		2018-03-31 23:59:59	Same as f3 of Q_5^S	Same as f3 of Q_6^S
f4		2018-06-30 23:59:59	Same as f4 of Q_5^S	Same as f4 of Q_6^S

Table 5.17 Performance on varying the size factors for cylinder queries Q_5^{ST}, Q_6^{ST} with compound index usage

	Size factor	Q_5^{ST} **- *RE* dataset**	Q_6^{ST} **- *SYNTH1* dataset**	Q_6^{ST} **- *SYNTH2* dataset**
Execution time	f1	16.35 sec	458.02 sec	889.94 sec
	f2	102.91 sec	952.27 sec	1909.68 sec
	f3	208.37 sec	969.54 sec	1973.13 sec
	f4	278.63 sec	1009.24 sec	2263.58 sec
# Counted documents	f1	69,750	48,816	97,263
	f2	544,977	385,385	768,719
	f3	4,550,150	2,283,537	4,565,344
	f4	11,817,952	12,138,692	24,275,830

Table 5.17 shows how a spatio-temporal range (cylinder) query scales on each dataset in cold cache state by using index. By increasing the radius of the respective query that is performed on each dataset, we notice that more time is needed for its execution (Figure 5.9 and 5.11). This is reasonable because the covered space of the query gets larger and thus more documents are fetched (Figure 5.10 and 5.12). This results to access more disk blocks,

loading a larger part of the index in the main memory as the queries are executed in cold cache.

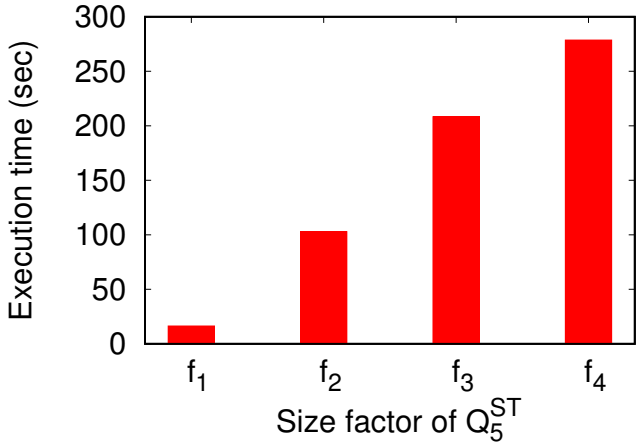

Fig. 5.9 Performance of the size factors of Q_5^{ST} on *RE* dataset.

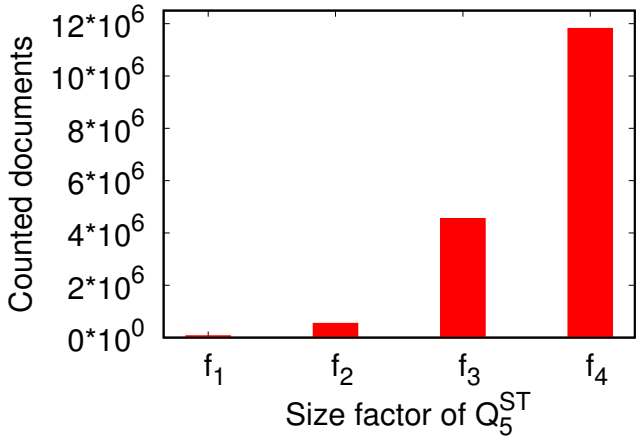

Fig. 5.10 Output size (# counted documents) of the size factors of Q_5^{ST} on *RE* dataset.

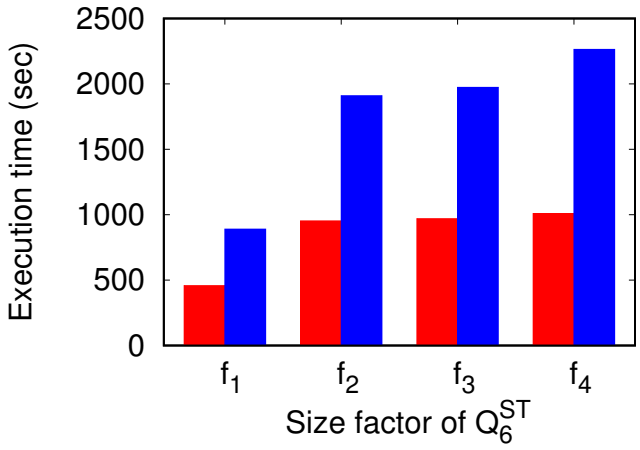

Fig. 5.11 Performance of the size factors of Q_6^{ST} on *SYNTH1* (red color) and *SYNTH2* (blue color) datasets.

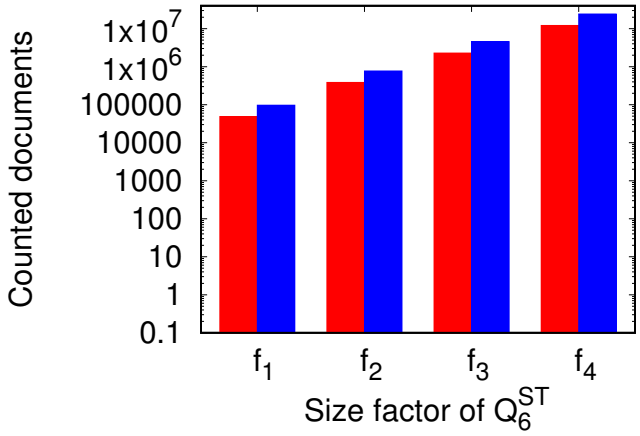

Fig. 5.12 Output size (# counted documents) of the size factors of Q_6^{ST} on *SYNTH1* (red color) and *SYNTH2* (blue color) datasets. The scale of y axis is logarithmic.

5.3 Experimental study of k-NN spatial queries

In this section we conduct experiments with k-NN spatial queries so as to study the effect of both k parameter and the number of buckets that compose the histogram that is used. The number of buckets are resulted from the number of splits that are defined for partitioning the 2d space in which the data exist. As described in section 4.3, the k-NN queries are executed

upon range queries by using histograms, in order to determine of the radius of the circle query that would fetch at least k documents. This is considered as an approximate way to find the radius, whereas the ideal case would be to know beforehand the radius of the circle query that would return exactly k documents. For this reason, we define the following two *metrics* (fractions) that represent the deviation of the ideal and the approximated (found) query.

- *Radius fraction* $\frac{r'-r}{r}$ (hereafter m_1) - Represents the percentage deviation of the found radius (r') and the actual radius (r) of the circle query for fetching exactly k documents.
- *Returned documents fraction* $\frac{n'-n}{n}$ (hereafter m_2) - Represents the percentage deviation of the fetched number of documents (n') and the actual number of documents (n - is equivalent to k) that should be fetched from the circle query.

The two metrics indicate the additional cost that is spent for a k-NN query, caused by the radius approximation of the circle query that is executed over it. In case we knew for a k-NN query the corresponding range query that would return exact k documents, no additional cost would occur.

For *RE*, *SYNTH1* and *SYNTH2* datasets we build two histograms by partitioning the space in which the data exist, composed of 200 x 200 cells and 300 x 300 cells; the 2d space that is taken into account for being partitioned for the three datasets is illustrated in Figure 5.2. We measure m_1 and m_2 for some k parameters (10, 50, 100, 300, 800) on each dataset, by executing 250 generated k-NN queries. Specifically, we record the averages of the metrics, the minimum observed values, the maximum observed values and the standard deviation of them. The coordinates of the k-NN queries are chosen randomly from existing points of the datasets.

Tables 5.18, 5.20 and 5.22 show the measured elements of m_1 for the datasets. We notice that the average values of 300 x 300 histograms are lower than their respective values of 200 x 200 histograms. This is reasonable because the calculated radiuses of the circle queries are more close to the actual radiuses when histogram is consisted of more (smaller) cells since the accuracy increases. Also, as the number of the requested neighboors (k) increases, the average value is getting lower because the formed radiuses are capable of serving more neighboors, without the need taking account adjacent cells. Furthermore, the average values of *RE* dataset are much greater than those of the other two datasets due to data skewing; some cells contain thousands of points and other cells just a few. This is reflected on the values of standard deviation.

Tables 5.19, 5.21 and 5.23 show the measured elements of m_2 for the datasets. The average values of 300 x 300 histograms are lower than their respective values of 200 x 200

histograms due to the fact that more (smaller) cells are to return less results. Also, as the number of the requested neighboors (k) increases, the average value is getting lower because cells are capable of returning more results than requested neighboors, without the need of taking account adjacent cells. Similarly to m_1, the average values of *RE* dataset are much greater than those of *SYNTH1* and *SYNTH2* due to data skewing, reflected on the high values of standard deviation.

Table 5.18 m_1 elements of k-NN queries, served by *RE* dataset histograms

200 x 200 cells				
k	**Avg**	**Min**	**Max**	**Std**
10	3444.43	24.78	33924.73	5410.30
50	1437.03	15.71	17366.66	2362.32
100	1342.30	6.19	22124.50	2714.06
300	630.93	1.66	21190.27	1803.62
800	233.36	1.19	2901.41	470.72
300 x 300 cells				
k	**Avg**	**Min**	**Max**	**Std**
10	1847.50	12.32	19722.72	2833.86
50	874.80	1.55	14953.80	1562.66
100	687.51	4.29	14952.77	1538.13
300	478.71	2.03	19880.47	1535.57
800	160.02	0.69	1324.43	249.73

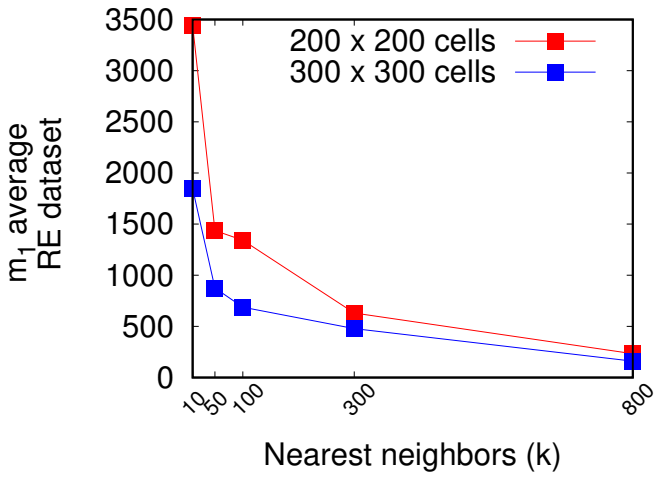

Fig. 5.13 m_1 average graph of k-NN queries, served by *RE* dataset histograms

Table 5.19 m_2 elements of k-NN queries, served by *RE* dataset histograms

200 x 200 cells				
k	**Avg**	**Min**	**Max**	**Std**
10	54082.22	134.60	265691.50	62233.96
50	13167.18	26.66	58128.70	14610.84
100	7023.66	20.95	34789.97	7471.16
300	2034.13	3.21	8919.23	2288.55
800	924.25	1.60	4129.90	965.61
300 x 300 cells				
k	**Avg**	**Min**	**Max**	**Std**
10	35962.81	101.0	161644.6	38541.95
50	7295.60	1.40	30910.36	6905.46
100	3079.17	11.23	15638.52	3287.25
300	1136.93	2.18	5426.03	1162.19
800	420.90	1.22	1984.61	428.36

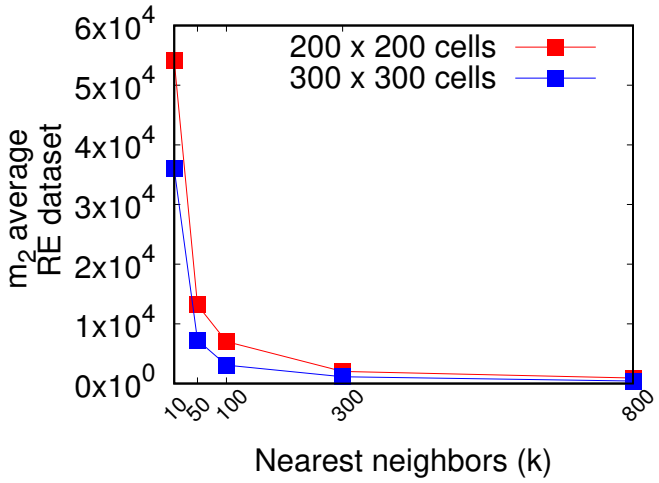

Fig. 5.14 m_2 average graph of k-NN queries, served by *RE* dataset histograms

Table 5.20 m_1 elements of k-NN queries, served by *SYNTH1* dataset histograms

200 x 200 cells				
k	**Avg**	**Min**	**Max**	**Std**
10	18.37	8.81	39.02	4.76
50	7.24	4.28	11.18	1.33
100	4.84	2.87	6.90	0.91
300	2.34	1.32	3.61	0.50
800	1.11	0.39	3.23	0.40
300 x 300 cells				
k	**Avg**	**Min**	**Max**	**Std**
10	12.17	6.87	24.10	3.02
50	4.55	2.73	6.23	0.79
100	2.87	1.54	4.15	0.59
300	1.25	0.48	2.97	0.39
800	2.16	1.14	2.63	0.24

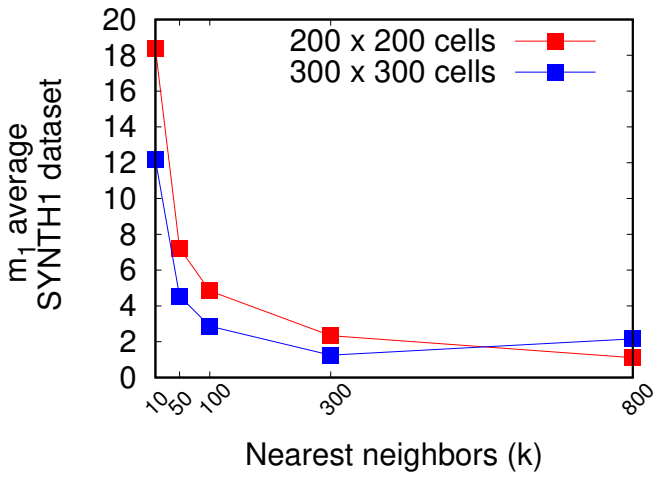

Fig. 5.15 m_1 average graph of k-NN queries, served by *SYNTH1* dataset histograms

Table 5.21 m_2 elements of k-NN queries, served by *SYNTH1* dataset histograms

200 x 200 cells				
k	**Avg**	**Min**	**Max**	**Std**
10	318.32	156.40	578.0	85.34
50	64.93	29.72	113.80	18.65
100	33.54	14.58	53.80	9.85
300	10.33	4.16	18.70	3.23
800	3.53	0.87	11.73	1.58
300 x 300 cells				
k	**Avg**	**Min**	**Max**	**Std**
10	145.97	43.00	259.7	44.16
50	29.94	12.60	48.06	7.85
100	13.79	5.14	23.26	4.24
300	4.09	1.32	10.63	1.60
800	8.97	2.97	11.86	1.55

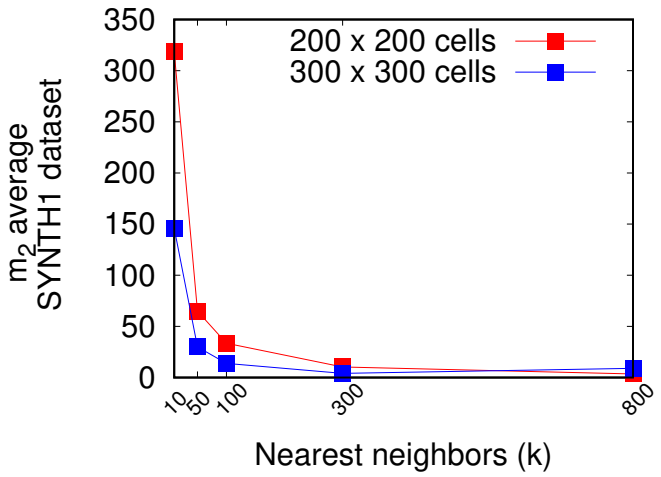

Fig. 5.16 m_2 average graph of k-NN queries, served by *SYNTH1* dataset histograms

Table 5.22 m_1 elements of k-NN queries, served by *SYNTH2* dataset histograms

200 x 200 cells				
k	**Avg**	**Min**	**Max**	**Std**
10	28.63	15.99	51.01	6.99
50	11.02	6.49	16.19	2.08
100	7.26	4.50	10.02	1.28
300	3.72	2.25	5.24	0.63
800	1.92	0.88	2.80	0.41
300 x 300 cells				
k	**Avg**	**Min**	**Max**	**Std**
10	18.43	9.57	31.66	4.51
50	6.86	3.78	10.94	1.22
100	4.60	2.75	6.52	0.87
300	2.16	1.10	3.23	0.47
800	2.54	0.32	4.05	1.23

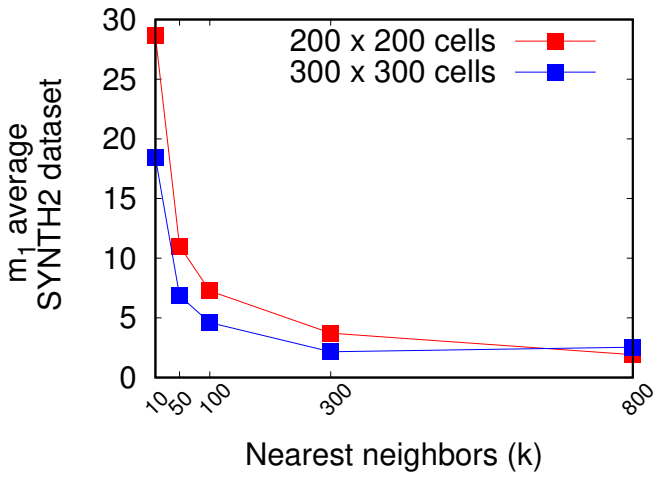

Fig. 5.17 m_1 average graph of k-NN queries, served by *SYNTH2* dataset histograms

Table 5.23 m_2 elements of k-NN queries, served by *SYNTH2* dataset histograms

200 x 200 cells				
k	**Avg**	**Min**	**Max**	**Std**
10	710.09	346.70	1184.40	194.74
50	138.05	53.36	223.48	39.98
100	66.39	30.22	110.15	19.52
300	21.41	9.37	37.35	5.80
800	7.68	2.22	13.07	2.35
300 x 300 cells				
k	**Avg**	**Min**	**Max**	**Std**
10	303.69	145.90	495.9	82.81
50	60.08	24.78	102.12	16.11
100	30.30	9.93	50.76	9.00
300	9.08	3.18	15.78	2.91
800	13.05	0.76	24.29	8.14

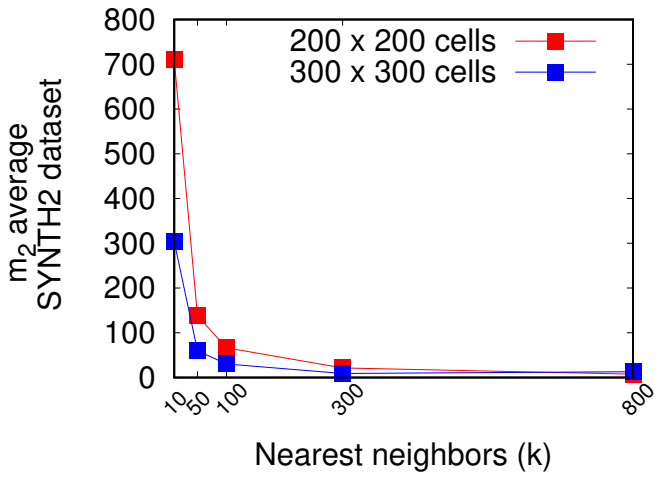

Fig. 5.18 m_2 average graph of k-NN queries, served by *SYNTH2* dataset histograms

Chapter 6

Conclusions and future work

This thesis introduces an API that allows unified access to data, stored in NoSQL databases. The API offers query primitives and a set of operators for expressing access operations over NoSQL databases. Its functionality is extended on spatio-temporal data, since spatial and spatio-temporal operators are provided, enabling thus access to mobility data. The API is implemented on top of MongoDB and its concept is demonstrated in practice.

Another objective of this thesis was the experimental evaluation of the performance of MongoDB on spatio-temporal data. According to Chapter 5, MongoDB does perform efficiently spatial and spatio-temporal queries when using the built-in spatial and compound indexes respectively. The most recent-read values of the indexes are kept in-memory, allowing efficient index use for read operations. In other words, MongoDB caches the recent-used parts of the indexes. Furthermore, MongoDB scales when performing queries by using indexes; queries that scan a large part of indexes, require more time for their execution than queries that scan just a few index values.

Concerning the adopted approach for serving k-NN queries, we notice that histograms that are composed of higher number of buckets, determine more accurately the radius of the circle queries that are to be executed. However, there is a trade-off between the accuracy of radius determination and the size of histograms; creating histograms with a higher number of buckets, requires higher storage cost. The approach demonstrates to be efficient for datasets that follow uniform distribution. It is not suitable for skewed datasets because the formed circle queries that serve k-NN types, fetch much more results than the requested (k).

Many are the directions for future work. First, the API could be extended to support other type of NoSQL stores such as column-wide and graph databases. Second, the provided geographical operators could be enriched for supporting specialized mobility operators, like trajectory similarity search. Also, the adopted approach for serving k-NN queries could be based on Quad-Trees instead of grid partitioning. This would result to space partitioning

with buckets of similar counts, being suitable for skewed data. Moreover, new techniques could be explored for performing spatio-temporal queries, as many NoSQL stores do not support indexes for such data.

References

[1] Banker, K., Bakkum, P., Verchand, S., Garrett, D., and Hawkins, T. (2014). *MongoDB in Action*. Manning.

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

[3] Cattell, R. (2010). Scalable SQL and NoSQL data stores. *SIGMOD Record*, 39(4):12–27.

[4] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. (2008). Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4.

[5] Comer, D. (1979). The ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137.

[6] Davoudian, A., Chen, L., and Liu, M. (2018). A survey on NoSQL stores. *ACM Comput. Surv.*, 51(2):40:1–40:43.

[7] Finkel, R. A. and Bentley, J. L. (1974). Quad trees a data structure for retrieval on composite keys. *Acta Inf.*, 4(1):1–9.

[8] Fox, A. D., Eichelberger, C. N., Hughes, J. N., and Lyon, S. (2013). Spatio-temporal indexing in non-relational distributed databases. In *Proceedings of the 2013 IEEE International Conference on Big Data, 6-9 October 2013, Santa Clara, CA, USA*, pages 291–299.

[9] Guan, X., Bo, C., Li, Z., and Yu, Y. (2017). ST-hash: An efficient spatiotemporal index for massive trajectory data in a nosql database. In *25th International Conference on Geoinformatics, Geoinformatics 2017, Buffalo, NY, USA, August 2-4, 2017*, pages 1–7.

[10] Guttman, A. (1984). R-trees: A dynamic index structure for spatial searching. In *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, USA, June 18-21, 1984*, pages 47–57.

[11] Iyer, A. P. and Stoica, I. (2017). A scalable distributed spatial index for the internet-of-things. In *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24-27, 2017*, pages 548–560.

[12] Nishimura, S., Das, S., Agrawal, D., and El Abbadi, A. (2011). MD-HBase: A scalable multi-dimensional data infrastructure for location aware services. In *12th IEEE International Conference on Mobile Data Management, MDM 2011, Luleå, Sweden, June 6-9, 2011, Volume 1*, pages 7–16.

[13] Paro, A. (2013). *ElasticSearch Cookbook*. Packt.

[14] Shriparv, S. (2014). *Learning HBase*. Packt.

[15] Silberschatz, A., Korth, H. F., and Sudarshan, S. (1996). Data models. *ACM Comput. Surv.*, 28(1):105–108.

[16] Zhang, N., Zheng, G., Chen, H., Chen, J., and Chen, X. (2014). Hbasespatial: A scalable spatial data storage based on hbase. In *13th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2014, Beijing, China, September 24-26, 2014*, pages 644–651.