Big Data for Mobility Tracking Knowledge Extraction in Urban Areas

D3.3 Primitive Query Operators

Document Summary Information

Grant Agreement No	780754	Acronym	TRACK & KNOW
Full title	Big Data for Mobility Tracking Knowledge Extraction in Urban Areas		
Start Date	01/01/2018	Duration	36 months
Project URL	https://trackandknowproject.eu/		
Deliverable	D3.3 Primitive Query Operators		
Work Package	WP3 Big Data Processing Toolboxes Management (BDP Toolbox)		
Contractual due date	30/06/2020	Actual submission date	30/06/2020
Nature	Other	Dissemination Level	PU
Lead Beneficiary	UPRC		
Contributions From	Evaggelia Chondrodima, Christos Doulkeridis, Harris V. Georgiou, Nikolaos Koutroumanis, Panagiotis Nikitopoulos, Nikos Pelekis, Petros Petrou, Panagiotis Tampakis, Yannis Theodoridis, Akrivi Vlachou (UPRC)		
Responsible Author	Christos Doulkeridis		

HISTORY OF CHANGES

Version	Date	Changes	Author
0.1	6/9/2020	First version of table of contents	C. Doulkeridis
0.3	20/5/2020	First version of Section 2	N. Koutroumanis
0.3	27/5/2020	First version of Section 3	N. Koutroumanis
0.4	3/6/2020	First version of Section 5	P. Tampakis
0.5	5/6/2020	First version of Section 4 and Section 1	N. Koutroumanis, C. Doulkeridis
0.6	12/6/2020	Revisions of all Sections, version sent for internal review	N. Koutroumanis, C. Doulkeridis
0.7	21/6/2020	Revisions based on reviewer (Mirco Nanni, CNR) comments	C. Doulkeridis
0.8	26/6/2020	Final minor corrections in technical sections	N. Koutroumanis, C. Doulkeridis

EXECUTIVE SUMMARY

This report comprises the third and last deliverable (D3.3) of Track&Know work package 3 "Big Data Processing Toolboxes Management (BDP Toolbox)", with main objective to report on query operators for Big Data that have been implemented during the course of the project, with particular emphasis on mobility data.

Deliverable D3.3 focuses on bridging the gap between application developers and scalable, big data storage solutions, by addressing the heterogeneity of NoSQL stores that still offer ad-hoc, idiosyncratic and non-standardized programming APIs, and stalls big data application development significantly. Drawing parallels to the relational storage landscape, our work is an attempt towards a standardized API (such as ODBC/JDBC) which provides uniform access to *any* relational database management system (RDBMS).

Essentially, the first part of this deliverable targets this limitation of modern NoSQL stores. We introduce NoDA, a programming interface that serves as an access layer between the application and the underlying NoSQL store. NoDA offers a set of basic data access operators (such as filter, project, aggregation, sorting) that are implemented for different NoSQL stores, thus hiding the heterogeneity of the underlying NoSQL stores. This uniform layer offers unique advantages to the application developer, who is empowered to learn a single API (instead of multiple APIs) and whose application code becomes portable to use another NoSQL store without modifications. In brief, NoDA serves as a technology that removes barrier of entry for application developers and NoSQL stores, improves their productivity, and speeds up application development. In addition, NoDA targets explicitly mobility data, by providing additional query operators for spatial and spatio-temporal retrieval. Last, but not least, we couple the programming API provided by NoDA with a declarative, SQL-like interface, which can be used by data scientists and business analysts to test their code using a standardized query language which is ubiquitous even beyond computer scientists.

The second part of this deliverable targets complex query operators that cannot be integrated in NoDA, because they cannot be pushed-down to the underlying NoSQL store. A typical example of such an operator is a join between different data sources. In the context of Track&Know, we demonstrate the design and implementation of the distributed sub-trajectory join (DTJ), a generic join operator that is applicable on mobility data and identifies maximal portions of sub-trajectories that are close in space and time. This operator is generic in the sense that it can be used as building block for mobility analytics, e.g., for clustering trajectories. We showcase a scalable implementation of DTJ using MapReduce/Hadoop.

TABLE OF CONTENTS

TERMS & ABBREVIATIONS

ANTLR	Another Tool for Language Recognition
API	Application Programming Interface
BPs	Breaking Points
CPU	Central Processing Unit
DTJ	Distributed Sub-Trajectory Join
DTW	Dynamic Time Warping
GPS	Global Positioning System
HDD	Hard Disk Drive
HDFS	Hadoop Distributed File System
IAAS	Infrastructure as a Service
IP	Internet Protocol
JDBC	Java Database Connectivity
kNN	k Nearest Neighbours
MR	MapReduce
NJP	Non-Joining Points
NoDA	NoSQL Data Access Operators
NoSQL	Non SQL or Non Relational
ODBC	Open Database Connectivity
POI	Point-of-interest
RAM	Random Access Memory
RDBMS	Relational Database Management System
SQL	Structured Query Language
TrI	Trajectory Index

LIST OF FIGURES

LIST OF TABLES

1 Introduction

This document is the deliverable D3.3 "Primitive Query Operators" of Task T3.3 of work package 3 "Big Data Processing Toolboxes Management (BDP Toolbox)" of the Track&Know project, which is submitted on month M30 of the project.

Its objective is to report on scalable query operators implemented over NoSQL stores, thus completing the batch processing layer of the Track&Know architecture. To this end, the work presented in this report can be separated in two major categories: (a) a unified layer for querying NoSQL stores, termed NoDA [28], which hides the heterogeneity of the underlying NoSQL stores from the big data developer, and provides a uniform way to query mobility data stored in different NoSQL stores, and (b) complex query operators for mobility data that cannot be "pushed-down" to the NoSQL store, thus they must be designed and implemented as parallel data processing tasks; we focus on a generic operator, termed *distributed sub-trajectory join (DTJ)* [44], which can be used as a building block for implementing scalable trajectory analysis tasks, such as sub-trajectory clustering [45]. It should be highlighted that the distributed sub-trajectory join is exploited in Track&Know by the Big Data Analytics (BDA) Toolbox, and in particular by the distributed sub-trajectory clustering proposed therein (reported in Deliverable D4.1). This can be perceived as an example of the interoperability between the different toolboxes proposed in Track&Know.

The work presented in this deliverable has been partly already published in SSTD [28] and ACM TSAS [44], in the context of the Track&Know project, by M30:

◇ Nikolaos Koutroumanis, Panagiotis Nikitopoulos, Akrivi Vlachou, Christos Doulkeridis: NoDA: Unified NoSQL Data Access Operators for Mobility Data. SSTD 2019: 174-177.
◇ Panagiotis Tampakis, Christos Doulkeridis, Nikos Pelekis, Yannis Theodoridis: Distributed Subtrajectory Join on Massive Datasets. ACM Trans. Spatial Algorithms and Systems 6(2): 8:1-8:29 (2020).

1.1 Purpose and Scope

In the context of the Track&Know project, large volumes of historical mobility data need to be collected, pre-processed, enriched and stored (cf. Deliverable D3.1), in order to support ad-hoc analytical queries that feed mobility analytics tasks and applications. Efficient and scalable retrieval of this data is only part of the underlying challenge (cf. Deliverable D3.2).

Another major challenge is to provide tools to big data developers for easy data access and manipulation, without shifting their focus from application development to data storage. In the

current era of Big Data, this is intensified by the wide variety of scalable storage technologies, typically NoSQL stores, each targeting different data models and query access patterns. To tackle this challenge, our work offers an abstraction layer that is conceptually located between the big data developer and the underlying NoSQL stores, hiding the data heterogeneity, the different data models, and (perhaps most importantly) the different language that each NoSQL store exposes to developers.

The proposed abstraction layer, called NoDA[1], offers flexibility and ease of use to big data developers by means of a programming API that is internally implemented for different NoSQL stores, thus hiding their heterogeneity. The programming API makes application development much easier, since developers need to learn a single API (instead of different APIs), while at the same time it permits changing the choice of storage during application development, without affecting the existing code base of the application. Moreover, the programming API is coupled with an SQL-like interface that allows declarative querying of the underlying NoSQL stores. This is the core offering of the work presented in this Deliverable. In addition, we showcase how to develop more complex operators for mobility data (such as joins) that cannot be pushed-down to the NoSQL store.

Deliverable D3.3 is submitted on month M30 of the project, reporting the results of the primitive query operators for big data. Essentially, it reports on an abstraction layer for NoSQL stores that comprises a fundamental part of the "Big Data Processing Toolbox (BDP)" that is developed in the context of WP3.

1.2 Approach for the Work package and Relation to other Deliverables

Approach Work package 3 is responsible for big data management aspects in the context of Track&Know; from data acquisition and integration to scalable storage and flexible querying. Moreover, it is responsible for delivering the "Big Data Processing Toolbox" (BDP), a generic, flexible and re-usable toolbox that facilitates data management and processing for big data developers.

Relation to other deliverables The work described in this deliverable relates to previous deliverables of the project, as follows:

- D1.2 *"Corporate Big Data Requirements"*: The specification of requirements regarding storage and querying of Big Data for supporting the needs of advanced data analysis operations has guided some design decisions made in this deliverable.
- D2.1 *"Architectures for the management of structured & unstructured data streams"*: The solutions described in this deliverable are compliant with the overall architecture of Track&Know

[1] **No**SQL **D**ata **A**ccess Operators

as specified in deliverable D2.1.

- D2.2 *"Architectures for the Management of Batch and Interactive Data Sources"*: Includes the comparative overview of different NoSQL solutions and the selection of a document-oriented store (MongoDB) to be the basis of the batch storage solution in Track&Know. However, it should be noted that the work presented in D3.3 is not limited to one specific NoSQL store.
- D3.1 *"Data Acquisition and Integration Report"*: This report essentially describes the process that enriches mobility data by means of cleansing, map-matching, and associating GPS traces with weather and points-of-interest (POIs). The resulting enriched mobility data comprises the input data set for persistent and scalable storage.
- D3.2 *"Big Data Storage and Indexing Report"*: This report focuses on efficient storage and indexing for spatio-temporal data in NoSQL stores, and its findings are exploited by the primitive query operators presented here, in order to improve their performance.
- D4.1 *"Analytics for Mobility Patterns Detection and Forecasting"*: This deliverable reports on various methods for mobility data analytics and forecasting, and some of its methods exploit the proposed operators in this deliverable. In particular, the distributed sub-trajectory clustering proposed in D4.1 build upon the distributed sub-trajectory join presented in this deliverable.

1.3 Mapping Track&Know Outputs

The purpose of this section is to map the Grant Agreement commitments, both within the formal Deliverable and Task description, against the project's respective outputs and work performed. To this end, Table 1 provides a mapping of Track&Know outputs to specific sections of this deliverable, in order to improve readability.

1.4 Methodology and Structure of this Deliverable

Methodology The work methodology of the activities within D3.3 can be summarized as follows:

- Analysis of requirements for query operators over scalable NoSQL stores, leading to separation between primitive operators (that *can* be pushed-down to the underlying store) and complex operators (that *cannot* be pushed-down to the underlying store).

Table 1: Mapping Track&Know outputs

TRACK&KNOW GA Component Title	**TRACK&KNOW GA Component Outline**	**Respective Document Chapter(s)**	**Justification**
Deliverable			
D3.3	Primitive Query Operators	All	This deliverable describes the specification and prototype implementation of the primitive query operators
Tasks			
Task 3.3 - Distributed complex query toolbox	[...] that will serve as a software layer very close to the storage layer allowing effective and efficient interaction with the massive datasets stored in Track&Know	Sect. 2	
	[...] a set of primitive query operators is going to be defined that operate over highly distributed data [...]	Sect. 3	NoDA provides basic operators implemented over scalable and distributed NoSQL stores
	[...] different implementations of the same operator can be provided for different storage solutions (e.g., HBase, MongoDB, etc.) [...]	Sect. 3	NoDA is an abstraction layer that is instantiated for different NoSQL stores
	[...] easier and more flexible software integration with the Big Data processing architecture and the various Toolboxes is going to be ensured [...]	Sect. 3, 5	NoDA is compatible with Spark, while complex operators, such as joins, are implemented on top of NoSQL
	[...] as developers are relieved from the cumbersome task of writing optimized software for data manipulation and data access	Sect. 4	Showcased by practical examples of use of the NoDA API as well as a declarative SQL-like interface

- Specification of the operators that comprise the NoDA abstraction layer, which corresponds to the primitive query operators (called NoDA).
- Design and implementation of NoDA for a document-oriented NoSQL store (MongoDB).
- Design and implementation of NoDA for a wide-column NoSQL store (HBase).
- Design and implementation of NoDA for a key-value NoSQL store (Redis).

- Development of a declarative (SQL-like) interface for NoDA.
- Design and implementation of complex join operator for mobility data, called Distributed Sub-trajectory Join (DTJ), as an example of a complex query operator that cannot be pushed-down to the NoSQL store.

Structure of this deliverable The remainder of this deliverable is structured as follows:

- Section 2 presents a crisp overview of the functionality of NoDA, thus providing a brief look on its capabilities and offerings.
- Section 3 provides the technical description of NoDA in more detail, in terms of its design and implementation, as well as its instantiation over different NoSQL stores (MongoDB, HBase, and Redis).
- Section 4 is the developer's guide of NoDA, offering practical examples of its use, both in terms of programming as well as in terms of using a declarative language.
- Section 5 presents the design and implementation of complex query operators for mobility data, focusing mainly on distributed sub-trajectory join, a special case of a join operator that cannot be "pushed-down" to the underlying NoSQL store.
- Section 6 presents the conclusions of this work.

2 Overview of Functionality

This section provides a brief overview of our work on big data processing, described in this deliverable. The provided big data processing operators are separated in two distinct categories: (a) *primitive query operators* that are data access operators that can be pushed-down to the underlying NoSQL store, and (b) *complex query operators* that cannot be pushed-down to the store, and instead need to be implemented in an existing big data processing framework.

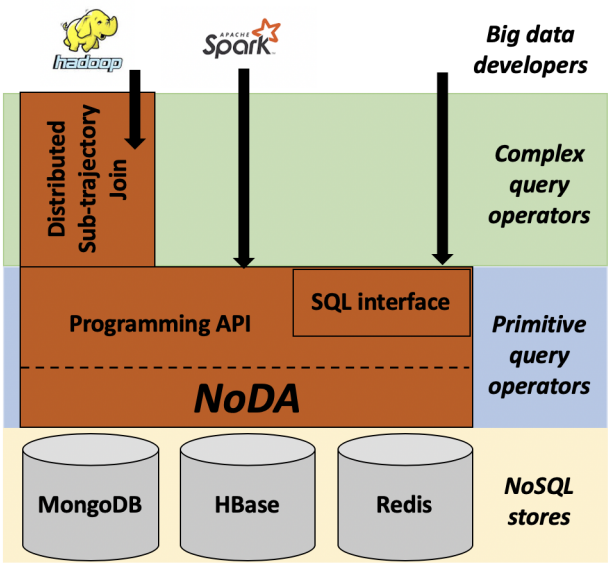

Figure 1: Broad overview of primitive and complex query operators, showing their positioning with respect to big data developers and scalable NoSQL storage. NoDA is presented in Sections 3, 4, while the distributed sub-trajectory join is presented in Section 5.

This is graphically illustrated in Figure 1 and it is further described in the subsequent subsections 2.1 and 2.2. Furthermore, it should be noted that the proposed techniques are tailored to mobility data, which is the primary data type targeted in the Track&Know project. However, we emphasize that the primitive query operators are not limited to mobility data, and can also be used for filtering, projection, aggregation, sorting, etc., of other types of data.

2.1 Primitive Query Operators

During the last fifteen years, NoSQL stores have gained ground as scalable storage solutions, as their characteristics meet the requirements of several data-intensive applications. Typically,

such applications need to handle a huge amount of data and serve countless requests within reasonable time. Also, the flexible data schema supported by NoSQL stores offers added value to the application developer, since NoSQL stores are not restricted by the rigid nature of the relational model.

However, existing NoSQL stores still have major limitations, such as the lack of standardization in data access. Every NoSQL system provides its own query language through its native libraries for handling data, whereas in the relational database world, data access is possible by using a driver (e.g., JDBC for Java programming language) and SQL. This shortcoming, also highlighted in [11], may incommode the transition to a NoSQL store, since the application developers need to learn its specific query language.

Motivated by the evident lack of a unified query language, in this deliverable, we present NoDA, an abstraction layer in the form of an API for querying NoSQL stores with native support for geospatial operations. The vision of NoDA comes to fill the existing gap between big data developers and the query languages of the NoSQL engines. Being an abstract layer, it offers common functionality over different NoSQL databases, as depicted in Figure 1. With the *functionality* term, we refer to the provided operations for data manipulation. The underlying rationale is based on the grouping of generic operations of the engines and their provision as *data access operators in a unified way*. This is offered by a developer-friendly programming API that can be used by big data applications. Alongside with this, NoDA supports a declarative, SQL-like interface that is useful for data scientists and business analysts that can easily access and manipulate data in NoSQL stores using a well-known language (SQL). This practice of providing declarative SQL-like interfaces is common in several big data processing frameworks; notable examples include Hadoop (with Hive [46] and Pig [24]), Spark (with Spark SQL [3]).

2.2 Complex Query Operators

Analysis of mobility data often requires complex data processing, such as associating data originating from different sources based on some condition (join) or even data from the same source (self-join). For instance, this is typically the case for many clustering algorithms that rely on the computation of pairwise distances between objects. Unfortunately, NoSQL stores are not designed to efficiently support join operations natively. As a result, complex query operations over big data are implemented using parallel data-processing frameworks (e.g., MapReduce/Hadoop, Spark, Flink, etc.) in order to ensure scalability.

In Track&Know, we follow the same approach and advocate the implementation of complex query operators on top of existing big data processing frameworks. To showcase this functionality, we select an operation called sub-trajectory join, which aims at identifying maximal portions of matching sub-trajectories, where "matching" refers to spatial and temporal proximity. We

demonstrate how to efficiently design and implement the *Distributed Sub-trajectory Join* (DTJ) in MapReduce, a generic operator that can be exploited by distributed sub-trajectory clustering algorithms to deliver modern applications to the end-user, including carpooling/ride-sharing and trip planning.

It should also be noted that the DTJ operator can work both independently from NoDA, for example by reading data directly from HDFS, but it can also exploit NoDA in order to fetch data from a NoSQL store (such as MongoDB, HBase, Redis).

3 Technical Description of NoDA

This section presents the technical details of NoDA, focusing on its design, the structure of its main software modules, the interfaces provided to application developers and business analysts, and finally its implementation over three NoSQL stores that belong to different categories: a document-oriented store (MongoDB), a wide-column store (HBase), and a key-value stores (REDIS).

Table 2: Supported primitive operators in NoDA

Primitives	**Arguments**	**Phase**
filter	(FilterOperator fop, FilterOperator... fops)	Definition
groupBy	(String columnName, String... columnNames)	Definition
aggregate	(AggregateOperator aggrOpr, AggregateOperator... aggrOp)	Definition
distinct	(String columnName)	Definition
max	(String columnName)	Execution
min	(String columnName)	Execution
sum	(String columnName)	Execution
avg	(String columnName)	Execution
count	()	Execution
sort	(SortOperator sop, SortOperator... sops)	Definition
limit	(int limit)	Definition
project	(String columnName, String... columnNames)	Definition
toDataframe	()	Execution

3.1 Design and Rationale of the NoDA API

The design of NoDA adopts the concept of offering generic query operators to application developers, so as to access data stored in NoSQL databases, in a simple and unified manner; *simple* because of using abstract, basic primitive query operators, hiding as much implementation details as possible, and *unified* because the primitives define specific data access functional behaviour that accommodates different NoSQL stores.

In a few words, the NoDA API groups many of the fundamental query primitives (shown in Table 2) and offers them in a comprehensive manner for performing query operations on NoSQL databases. Most of them cover the ordinary querying operations, such as fetching data that fulfills a specific condition or fetching a sorted column in ascending order. The primitive operators can be combined together, so as to form complex query expressions as well.

The functionality of the query primitives is described as follows:

- *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 (conjunctive query).
- *groupBy* – performs a group operation on a column/s, the name of which are passed as arguments. Specifically, it arranges the identical values of a specific column/s into groups. It can be optionally used in conjunction with the aggregate primitive which performs aggregations on column/columns. If the aggregate primitive is not defined, the group primitive acts as a distinct statement, finding the unique values of the column/s.
- *aggregate* – performs an aggregation/s, the type of which are passed as arguments. Specifically, it performs specific computations depending on the aggregation type. If the groupBy primitive has been preceded, then the aggregation/s are computed on the formed groups.
- *distinct* – performs a distinct operation on a column, finding the distinct (different) values. The name of the column is passed as an argument.
- *max* – calculates the maximum value of a numeric column, the name of which is passed as an argument.
- *min* – calculates the minimum value of a numeric column, the name of which is passed as an argument.
- *sum* – calculates the sum value of a numeric column, the name of which is passed as an argument.
- *avg* – calculates the average value of a numeric column, the name of which is passed as an argument.
- *count* – calculates the number of records.
- *sort* – performs sort operation/s given some (at least one) SortOperator object type arguments. If more than one SortOperator object arguments are declared, then the first determines primary sorting, the next secondary sorting, and so on.
- *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 columns, 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.

Listing 3.1 shows an example of using *max* and *count* query primitives.

Listing 3.1: Find the max value of a field and count the number of documents in a MongoDB collection using NoDA

```
import gr.ds.unipi.noda.api.client.NoSqlDbSystem;
import gr.ds.unipi.noda.api.core.nosqldb.NoSqlDbOperators;
import java.util.Optional;

public static void main(String args[]){
    NoSqlDbSystem noSqlDbSystem = NoSqlDbSystem.MongoDB()
        .Builder("username","password","database").host("192.168.1.1").port(27027).build();

    NoSqlDbOperators noSqlDbOp = noSqlDbSystem.operateOn("geoPoints");
    Optional<Double> max = noSqlDbOp.max("aField");
    int count = noSqlDbOp.count();

    noSqlDbSystem.closeConnection();
}
```

The query primitives are classified into one of the following two categories: *definition phase* and *execution phase*. The primitives that belong to the definition phase can be used in conjunction with each other as many times as needed. They are processed in a "lazy" manner, meaning that they define a sequence of operations without being actually executed. These operations are executed only 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. This separation is inspired by Apache Spark, which uses transformations and actions. Listing 3.2 illustrates the definition and execution phases. Note that using the filter primitive more than once entails that all of their arguments are operands of an AND ($\cap$) boolean operator.

Listing 3.2: Example of definition and execution phases of primitive operators

```
import gr.ds.unipi.noda.api.core.nosqldb.NoSqlDbOperators;
import org.apache.spark.sql.*;

public class DataOperationsWithSpark {
    public static void doOperations(NoSqlDbOperators noSqlDbOp) {
        Dataset<Row> dataset = noSqlDbOp.filter( ... ).filter( ... ) //definition phase
            .groupBy( ... ).sort( ... ).project( ... ) //definition phase
            .toDataframe(); //execution phase
    }
}
```

Four types of filter operators are offered from the NoDA API; comparison, boolean, geographical and geotemporal, which are explained in Tables 3, 4, 5 and 6, respectively. In the comparison operators presented in Table 3, data types T_1 and T_2 are defined as follows:

$T_1 \in$ *[short, int, long, float, double, boolean, Date, String]*
$T_2 \in T_1$ - *[boolean, String]*

Table 3: Supported comparison operators

Comparison Operators	Arguments
eq	(String columnName, T_1 columnValue)
gt	(String columnName, T_2 columnValue)
gte	(String columnName, T_2 columnValue)
lt	(String columnName, T_2 columnValue)
lte	(String columnName, T_2 columnValue)
ne	(String columnName, T_1 columnValue)

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 column (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 column (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 column (its name is passed as the first argument) is greater than or equal (≥) to a given value (passed as the second argument).
- *lt* – selects the records whose values of a specific column (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 column (its name is passed as the first argument) is less than or equal (≤) to a given value (passed as the second argument).
- *ne* – selects the records whose values of a specific column (its name is passed as the first argument) is not equal (≠) to a given value (passed as the second argument).

Listing 3.3: Example of using a comparison operator - Find all of the 5 Star hotels in Greece

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

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:

Table 4: Supported boolean operators

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

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

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

Table 5: Supported geographical operators

Geographical Operators	Arguments
inGeoPolygon	(String columnName, Coordinates c1, Coordinates c2, Coordinates c3, Coordinates... cs)
inGeoRectangle	(String columnName, Coordinates lowerBoundPoint, Coordinates upperBoundPoint)
inGeoCircleKm	(String columnName, Coordinates point, double radius)
inGeoCircleMeters	(String columnName, Coordinates point, double radius)
inGeoCircleMiles	(String columnName, Coordinates point, double radius)
geoNearestNeighbors	(String columnName, 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 column (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).

- *inGeoRectangle* – selects the records whose spatial extent that is represented by a specific column (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 column (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 column (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 column (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).
- *geoNearestNeighbors* – selects a specified number of records (passed as the third argument) whose spatial extent that is represented by a specific column (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 3.5: Example of using a geographical operator - Find the 10 nearest hotels from a location

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

The geotemporal 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;

- *inGeoTemporalPolygon* – selects the records whose spatial extent that is represented by a specific column (its name is passed as the first argument) is entirely within a polygon and

Table 6: Supported geotemporal operators

Geotemporal Operators	**Arguments**
inGeoTemporalPolygon	(String columnName, String temporalColumnName, Date lowerBoundDate, Date upperBoundDate, Coordinates c1, Coordinates c2, Coordinates c3, Coordinates... cs)
inGeoTemporalRectangle	(String columnName, Coordinates lowerBoundPoint, Coordinates upperBoundPoint, String temporalColumnName, Date lowerBoundDate, Date upperBoundDate)
inGeoTemporalCircleKm	(String columnName, Coordinates point, double radius, String temporalColumnName Date lowerBoundDate, Date upperBoundDate)
inGeoTemporalCircleMeters	(String columnName, Coordinates point, double radius, String temporalColumnName Date lowerBoundDate, Date upperBoundDate)
inGeoTemporalCircleMiles	(String columnName, Coordinates point, double radius, String temporalColumnName Date lowerBoundDate, Date upperBoundDate)

their temporal extent is in specific date range. The polygon is defined by its corner points (its coordinates are passed as arguments - at least three are needed).

- *inGeoTemporalRectangle* – selects the records whose spatial extent that is represented by a specific column (its name is passed as the first argument), is entirely within a box and their temporal extent is in specific date range. The box is defined by its lower and upper bounding points (the coordinates of which are passed as the second and third argument respectively).

- *inGeoTemporalCircleKm* – selects the records whose spatial extent that is represented by a specific column (its name is passed as the first argument), is entirely within a circle and their temporal extent is in specific date range. 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).

- *inGeoTemporalCircleMeters* – selects the records whose spatial extent that is represented by a specific column (its name is passed as the first argument), is entirely within a circle and their temporal extent is in specific date range. 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).

- *inGeoTemporalCircleMiles* – selects the records whose spatial extent that is represented by a specific column (its name is passed as the first argument), is entirely within a circle and

their temporal extent is in specific date range. 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).

Listing 3.6: Example of using a geotemporal operator - Find the spatial objects in a rectangle with temporal constraint

```
import gr.ds.unipi.noda.api.core.nosqldb.NoSqlDbOperators;
import gr.ds.unipi.noda.api.core.operators.filterOperators.geoperators.Coordinates;
import org.apache.spark.sql.*;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
import static gr.ds.unipi.noda.api.core.operators.FilterOperators.*;

public class DataOperationsWithSpark {
    public static void doOperations(NoSqlDbOperators noSqlDbOp) throws ParseException {
        SimpleDateFormat s = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss.SSS");
        Date d1 = s.parse("2017-12-01T00:00:00.000Z");
        Date d2 = s.parse("2017-12-02T23:59:59.999Z");
        Coordinates c1 = Coordinates.newCoordinates(23.65, 37.94);
        Coordinates c2 = Coordinates.newCoordinates(23.67, 37.96);

        Dataset<Row> dataset = noSqlDbOp.filter(inGeoTemporalRectangle("location",
                        c1, c2, "date",d1,d2)).toDataframe();
    }
}
```

Apart from the filter operators that are used in conjunction with the *filter* primitive, there also exist aggregate and sort operators that are used with the *aggregate* and *sort* primitives. These operators are shown in Tables 7 and 8, respectively.

Table 7: Supported aggregate operators

Aggregate Operators	Arguments
max	(String columnName)
min	(String columnName)
avg	(String columnName)
sum	(String columnName)
count	()
countNonNull	(String columnName)
countDistinct	(String columnName)

The aggregate operators can be used as arguments of the *aggregate* query primitive, performing a calculation. Their functionality is described as follows;

- *max* – finds the maximum value of a numeric column whose name is passed as an argument.
- *min* – finds the minimum value of a numeric column whose name is passed as an argument.

- *avg* – calculates the average value of a numeric column whose name is passed as an argument.
- *sum* – calculates the sum value of a numeric column whose name is passed as an argument.
- *count* – calculates the number of thr existing records.
- *countNonNull* – calculates the number of non-null values of a column whose name is passed as an argument.
- *countDistinct* – calculates the number of distinct values of a column whose name is passed as an argument.

Listing 3.7: Example of using an aggregate operator - Find the max and the average price per day of 5-Star hotels

```
import gr.ds.unipi.noda.api.core.nosqldb.NoSqlDbOperators;
import org.apache.spark.sql.*;
import static gr.ds.unipi.noda.api.core.operators.AggregateOperators.avg;
import static gr.ds.unipi.noda.api.core.operators.AggregateOperators.max;
import static gr.ds.unipi.noda.api.core.operators.FilterOperators.eq;

public class DataOperationsWithSpark {
    public static void doOperations(NoSqlDbOperators noSqlDbOp) {
        Dataset<Row> dataset = noSqlDbOp.filter(eq("star", 5))
                    .aggregate(max("approximate_price_per_day",
                        avg("approximate_price_per_day")).toDataframe();
    }
}
```

Table 8: Supported sort operators

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

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

- *asc* – sorts a given column whose name is passed as an argument, in ascending order.
- *desc* – sorts a given column whose name is passed as an argument, in descending order.

Listing 3.8: Example of using a sort operator - Sort in ascending order the 5 Star hotels by their approximate price per day

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

3.2 Technical Details

The development of NoDA was based on the Java Programming language under the *Apache Maven* project manager tool. With *Maven* usage, we achieve to organize the API into distinct parts. This modular design comes to an absolute harmony with the rationale of NoDA as it is comprised by components. Each component has its unique role such as supporting the data access operators upon a specific NoSQL engine.

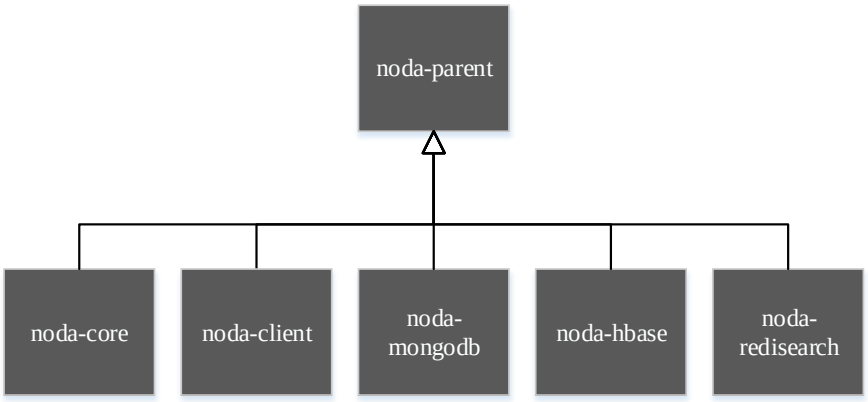

Figure 2: Structure of the NoDA project

In technical terms, *Maven* splits the software components into *modules*, aiming to project aggregation and project inheritance. This is achieved by the parent POM in which the *modules* of the project are specified, inheriting the parent's project dependencies. *Modules* can depend on each other or may be independent under a project. NoDA modules are packed by the *noda-parent*

project, as depicted in Figure 2. The project composes NoDA, aiming to group the modules under a specific place. The contained modules are the following:

- *noda-client* – this module incorporates the functionality related to the connectivity of a NoSQL engine. For all of the supported databases, there exists a respective programming package that integrates the necessary operations for opening and closing a connection. Moreover, the module contains the SQL component of NoDA for data accessing via the SQL language. The structure of this module is explained in Section 3.3.3.
- *noda-core* – this module contains the core code of NoDA where its functionality is organized into packages. The module constitutes the abstract layer where it offers to the developers a set of well-defined data access operators upon the NoSQL engines. Most of the contained Java classes are inherited from the rest (aftermentioned) modules of the NoDA project. The structure of this module will be explained in this subsection.
- *noda-mongodb* – this module contains all of the required Java classes, implementing the functionality of NoDA over MongoDB. Most of the classes inherit from the corresponding classes in the *noda-core* module. The structure of this module is explained in Section 3.4.2.
- *noda-hbase* – this module contains all of the required Java classes, implementing the functionality of NoDA over HBase. Most of the classes inherit from the corresponding classes in the *noda-core* module. The structure of this module is explained in Section 3.5.2.
- *noda-redisearch* – this module contains all of the required Java classes, implementing the functionality of NoDA over Redis (with the usage of RediSearch library). Most of the classes inherit from the corresponding classes in the *noda-core* module. The structure of this module is explained in Section 3.6.2.

The features of NoDA are imprinted in the contained Java classes in *noda-core* module. Specifically, this module contains in abstract form functional elements that should be inherited and implemented for a specific NoSQL system. This justifies the fact that the majority of Java classes are abstract, since the basic form of functional elements is offered mainly, being common for all databases. Its design adopts the bottom-up approach; the functional elements are synthesized by small units (Java classes) which operate autonomously or in combination with each other. This makes the core part of NoDA more flexible in terms of adding more features without having to refactor code in other classes.

Mainly, two types of functional elements are provided abstractly from the *noda-core* so as to be implemented from the rest of the modules; data access operators and database connectivity operations in a lesser extent, as NoDA focuses primarily to equip its users with access operators. These are modelled by the following classes, interfaces and abstract classes, grouped by packages (namespaces are shown with bold text) based on their conceptual role:

- **gr.ds.unipi.noda.api.core.constants**

- *Commons* - class which contains the API constants

- **gr.ds.unipi.noda.api.core.nosqldb**
  - *NoSqlConnectionFactory* – abstract class that contains abstract methods for accessing the query primitives and closing the established connection in a NoSQL database. These methods are implemented from the modules which provide NoDA API upon specific NoSQL databases. Particularly, this is performed through inheritance, where the connection factory classes of the modules extend this class. Also, this class initializes in a concrete method the base operator factory classes for using the supported data access operators through their facets.
  - *NoSqlDbConnectionManager* – abstract class that contains concrete and abstract methods for managing the connections to NoSQL stores. A connection is stored to a dictionary data structure in *<key, value>* form. The key is a NoSqlDbConnector object type, representing a specific type of connection with its related information. The value is an object in generic type, specified explicitly in the modules that implement the NoDA API over specific NoSQL stores. This class is inherited by the connection manager classes of the respective modules.
  - *NoSqlDbConnector* – interface that represents a specific NoSQL database connection. It is used by the NoSqlDbConnectionManager class for managing the NoSQL database connections. By implementing this interface, both *equals* and *hashCode* methods have to be implemented since they are used by the dictionary structure in the NoSqlDbConnectionManager class when checking the existence of a connection. In other words, this class represents the information associated with a unique connection to a NoSQL store. It is extended by the respective connector classes of the modules that implement the NoDA API over specific NoSQL stores.
  - *NoSqlDbOperators* – abstract class that contains abstract methods, defining the supported query primitives of the API. The classes are implemented through inheritance by the respective operators classes of the modules that provide the NoDA API over specific NoSQL stores.

- **gr.ds.unipi.noda.api.core.operators**
  - *Operator* – inteface that represents an operator through an expression whose type is generic, defined explicitly in the modules that implement the NoDA API over a specific NoSQL store.
  - *FilterOperators* – class that contains only static methods, offering the filter operators to the users of the API.
  - *AggregateOperators* – class that contains only static methods, offering the aggregate operators to the users of the API.

  - *SortOperators* – class that contains only static methods, offering the sorting operators to the users of the API.

- **gr.ds.unipi.noda.api.core.operators.filterOperators**
  - *FilterOperator* – interface that extends the *Operator* interface, representing a filter operator type.

- **gr.ds.unipi.noda.api.core.operators.filterOperators.comparisonOperators**
  - *BaseComparisonOperatorFactory* – abstract class containing only abstract methods, acting as a facet of comparison operators in order to be forwarded to the *FilterOperators* class. The abstract methods of this class are implemented through inheritance by the respective base comparison operator factory classes of the modules that provide the NoDA API over specific NoSQL stores.
  - *ComparisonOperator* – abstract class that implements the *FilterOperator*, representing a filter operator type, particularly a comparison operator. It is extended by the respective comparison operator abstract classes of the modules that implement the NoDA API over specific NoSQL stores.

- **gr.ds.unipi.noda.api.core.operators.filterOperators.logicalOperators**
  - *BaseLogicalOperatorFactory* – abstract class containing only abstract methods, acting as a facet of logical operators in order to be forwarded to the *FilterOperators* class. The abstract methods of this class are implemented through inheritance by the respective base logical operator factory classes of the modules that provide the NoDA API over specific NoSQL stores.
  - *LogicalOperator* – abstract class that implements the *FilterOperator*, representing a filter operator type, particularly a logical operator. Extended by the respective logical operator abstract classes of the modules that implement the NoDA API over specific NoSQL stores.

- **gr.ds.unipi.noda.api.core.operators.filterOperators.geoperators**
  - *Coordinates* – class that represents coordinates information (longitude, latitude).
  - *QuadTreeInstance* – class that represents a QuadTree instance, exploited by *noda-mongo* and *noda-redisearch* for kNN querying.

- **gr.ds.unipi.noda.api.core.operators.filterOperators.geoperators.geometries**
  - *Geometry* – abstract class that represents a Geometry (shape) by its coordinates. When a geometry is defined, its coordinates are checked for validity so as to define an existing location on earth.

  - *Point* – class that extends the *Geometry* abstract class, representing a point by its coordinates.
  - *Rectangle* – class that extends the *Geometry* abstract class, representing a rectangle by its lower and upper bound coordinates.
  - *Circle* – class that extends the *Geometry* abstract class, representing a circle by its center coordinates and its radius.
  - *Polygon* – class that extends the *Geometry* abstract class, representing a polygon by a set of coordinates.

- **gr.ds.unipi.noda.api.core.operators.filterOperators.geoperators .geographicalOperators**
  - *BaseGeographicalOperatorFactory* – abstract class containing only abstract methods, acting as a facet of geographical operators in order to be forwarded to the *FilterOperators* class. The abstract methods of this class are implemented through inheritance by the respective base geographical operator factory classes of the modules that provide the NoDA API over specific NoSQL stores.
  - *GeographicalOperator* – abstract class that implements the *FilterOperator*, representing a filter operator type, particularly a geographical operator. It is extended by the respective geographical operator abstract classes of the modules that implement the NoDA API over specific NoSQL stores.

- **gr.ds.unipi.noda.api.core.operators.filterOperators.geoperators .geoTemporalOperators**
  - *BaseGeoTemporalOperatorFactory* – abstract class containing only abstract methods, acting as a facet of geotemporal operators in order to be forwarded to the *FilterOperators* class. The abstract methods of this class are implemented through inheritance by the respective base geotemporal operator factory classes of the modules that provide the NoDA API over specific NoSQL stores.
  - *GeoTemporalOperator* – abstract class that implements the *FilterOperator*, representing a filter operator type, particularly a geotemporal operator. It is extended by the respective geotemporal operator abstract classes of the modules that implement the NoDA API over specific NoSQL stores.

- **gr.ds.unipi.noda.api.core.operators.filterOperators.geoperators .geoTemporalOperators.temporal**
  - *Temporal* – abstract class that represents temporal information
  - *SingleTemporalValue* – class that extends the *Temporal* abstract class, representing a specific date.

  - *TemporalBounds* – class that extends the *Temporal* abstract class, representing a specific time period by its lower and upper bound date.

- **gr.ds.unipi.noda.api.core.operators.aggregateOperators**
  - *BaseAggregateOperatorFactory* – abstract class containing only abstract methods, acting as a facet of aggregate operators in order to be forwarded to the *AggregateOperators* class. The abstract methods of this class are implemented through inheritance by the respective base aggregate operator factory classes of the modules that provide the NoDA API over specific NoSQL stores.
  - *AggregateOperator* – abstract class that implements the *Operator* interface, representing an aggregation operator type. It is extended by the respective aggregate operator abstract classes of the modules that implement the NoDA API over specific NoSQL stores.

- **gr.ds.unipi.noda.api.core.operators.sortOperators**
  - *BaseSortOperatorFactory* – abstract class containing only abstract methods, acting as a facet of sorting operators in order to be forwarded to the *SortOperators* class. The abstract methods of this class are implemented through inheritance by the respective base sort operator factory classes of the modules that provide the NoDA API upon specific NoSQL stores.
  - *SortOperator* – abstract class that implements the *Operator* interface, representing a sort operator type. Extended by the respective sort operator abstract classes of the modules that implement the NoDA API upon specific NoSQL databases.

The usage of base operator factory classes, as the facet of the provided operators, adds on them the following two features: i) to be offered uniquely under concrete classes (FilterOperators, AggregateOperators and SortOperators), thus being storage-agnostic, and ii) to have a particular functionality determined solely by the module that implements the API over a NoSQL store.

Most of the Java abstract classes and interfaces are extended and implemented correspondingly by the classes of the rest modules of the NoDA API (except the *noda-client* module). This is reasonable since the functionality of the API on a particular store is determined in the said modules, leveraging on the native libraries of the databases. In other words, implementation details upon a NoSQL store are not found in the *noda-core* module as it does not aim to provide any functionality oriented to a database; its main purpose is to have an abstract form so that any NoSQL store may embody it. The classes of the *noda-core* module are depicted using three UML class diagrams in Figs. 3, 4, 5, respectively, in different views.

Fig. 3 shows the UML class diagram of the *noda-core* module regarding the connectivity of NoDA to NoSQL database. Fig. 4 shows the UML class diagram of the *noda-core* module that is related to the operators for data accessing. The figure includes also the units that offer a

special characteristic to the operators. For example, the spatial extent of the Geographical and GeoTemporal operator is offered by the Geometry class. Fig. 5 shows the UML class diagram of the *noda-core* module that have to do with the data access operators and the classes which forward them for to the modules that implement NoDA's functionality upon a NoSQL store.

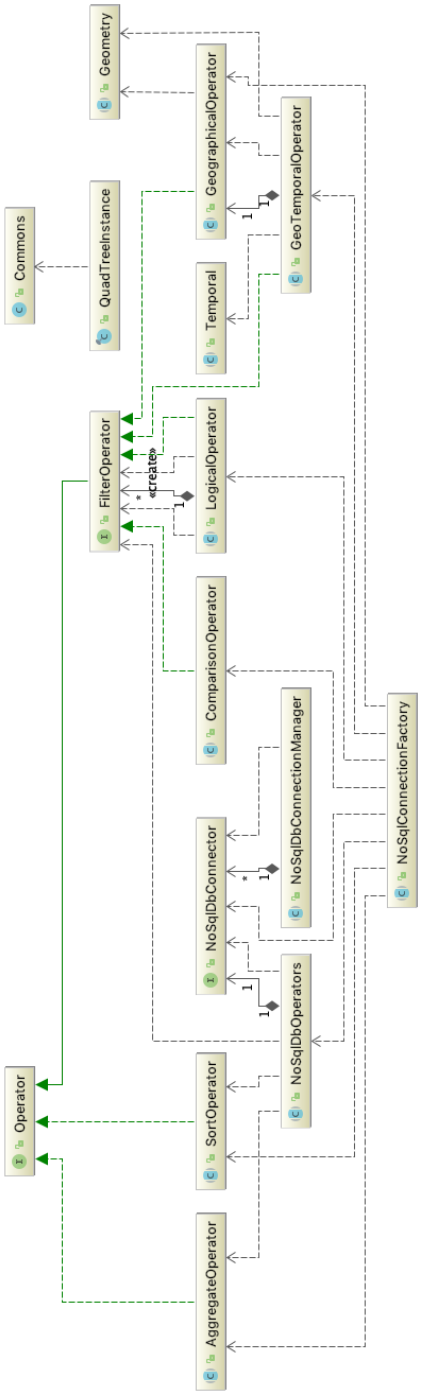

Figure 3: UML class diagram of *noda-core* module, associated with the connectivity of NoSQL System and the data access operators.

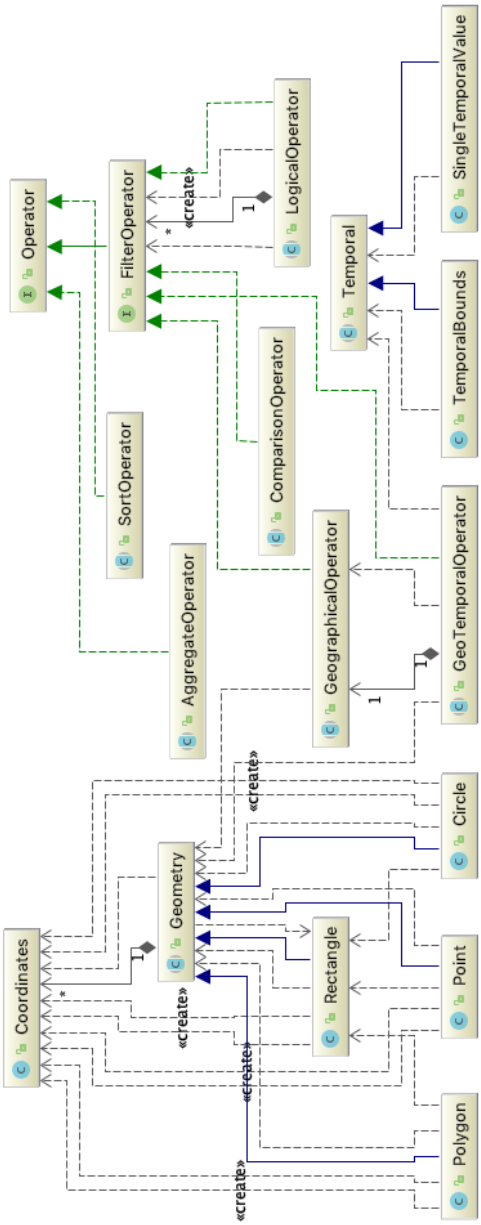

Figure 4: UML class diagram of *noda-core* module, associated with the data access operators and their units.

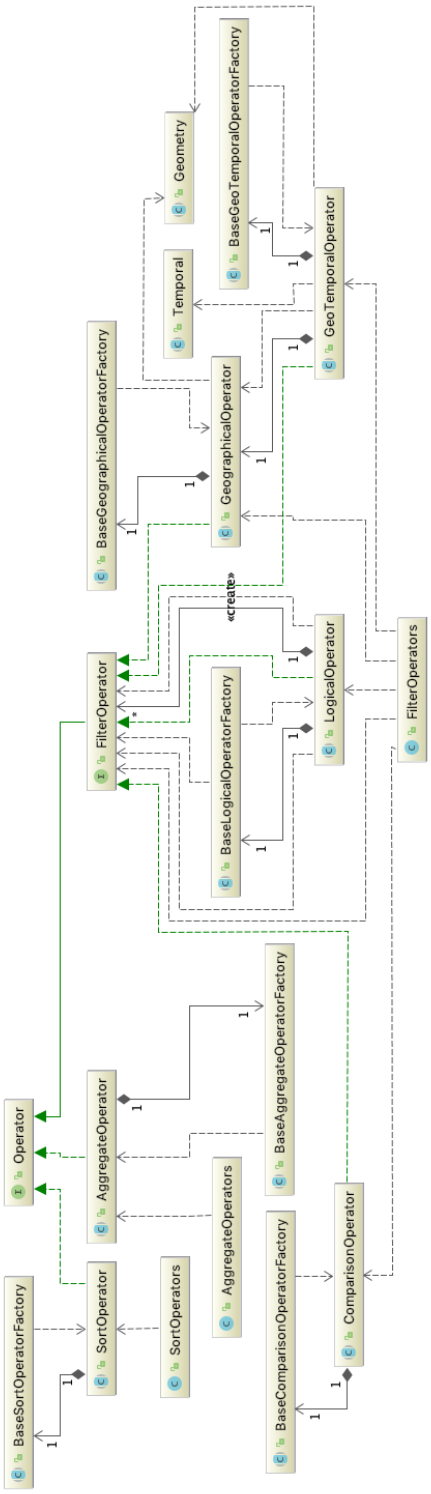

Figure 5: UML class diagram of *noda-core* module, associated with the data access operators and their facets for utilizing them

The materialization of the NoDA API takes place in the *noda-mongodb*, *noda-hbase*, *noda-redisearch* modules, for the MongoDB, HBase and Redis NoSQL stores, respectively. The majority of the contained classes extend the core classes.

3.3 Interfaces

NoDA offers two types of interfaces that can be exploited by big data developers, so as to define operations for accessing the data over a NoSQL store. These interfaces are implemented in the *noda-client* module which constitutes an integral part of the NoDA project. Concretely, the module aims to link the users/developers with the NoDA API. This is done by offering a common client interface on which every supported NoSQL store from the API, integrates the required arguments for establishing a connection to the NoSQL database. Then, after the connection has been defined, a choice is offered for either using the programming interface or the SQL interface for data access. This module depends on the other modules of the NoDA project, as it uses them to fulfil the users' requests. It also depends on the *ANTLR*[2] tool which is used for accessing the SQL expressions. *ANTLR* is a parser generator for reading, processing, executing, or translating structured text or binary files. Given a grammar, *ANTLR* generates a parser, builds parse trees, and generates a listener interface (or visitor) that makes it easy to respond to the recognition of phrases.

3.3.1 Programming interface

Assuming that we have a dataset of all of the hotels around the world with some related information (number of stars, city and price per day) stored in a NoSQL store. As shown in Listing 3.9, we can fetch via the programming interface the first 20 cities and their 5-star hotels average price per day that is greater than 500 Euros, sorted in ascending order by their average price.

[2]https://www.antlr.org/

Listing 3.9: By using the query primitives, get the first 20 cities and their their 5-star hotels average price per day that is greater than 500 Euros, sorted in ascending order by their average price

```
import gr.ds.unipi.noda.api.core.nosqldb.NoSqlDbOperators;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;

import static gr.ds.unipi.noda.api.core.operators.AggregateOperators.avg;
import static gr.ds.unipi.noda.api.core.operators.FilterOperators.eq;
import static gr.ds.unipi.noda.api.core.operators.FilterOperators.gt;
import static gr.ds.unipi.noda.api.core.operators.SortOperators.desc;

public class DataOperationsWithSpark {
    public static void doOperationsOnNoSqlDbSystem(NoSqlDbSystem noSqlDbSystem) {
        NoSqlDbOperators noSqlDbOp = noSqlDbSystem.operateOn("hotels");

        Dataset<Row> dataset = noSqlDbOp.filter(eq("star", 5))
            .groupBy("city").aggregate(avg("price_per_day"))
            .filter(gt("AVG(price_per_day)", 500))
            .sort(asc("AVG(price_per_day)"))
            .limit(20).project("city", "AVG(price_per_day)")
            .toDataframe();
    }
}
```

When utilizing the aggregations via the programming interface for computing a value given a column, its projection is named by default as *"YYY(column)"*, where YYY is the capitalized abbreviation of the aggregation. For example, by using the primitive *.aggregate(min("column"))*, its respecitve projection is named as *MIN(column)*. The same applies for the SQL interface; aggregate functions are declared as *"YYY(column)"* and projected exactly as referred.

3.3.2 SQL interface

The equivalent to Listing 3.9 but expressed in SQL, is shown in Listing 3.10

Listing 3.10: By using the SQL interface, get the first 20 cities and their their 5-star hotels average price per day that is greater than 500 Euros, sorted in ascending order by their average price

```
1  import gr.ds.unipi.noda.api.client.sql.NoSqlDbSqlStatement;
2  import org.apache.spark.sql.Dataset;
3  import org.apache.spark.sql.Row;
4
5  public class DataOperationsWithSpark {
6      public static void doOperationsOnNoSqlDbSystem(NoSqlDbSystem noSqlDbSystem) {
7          String sqlStatement = "SELECT city, AVG(price_per_day) FROM hotels " +
8              "WHERE star=5 GROUP BY city HAVING AVG(price_per_day) > 500 " +
9              "ORDER BY AVG(price_per_day) ASC LIMIT 20" ;
10         NoSqlDbSqlStatement noSqlDbSqlStmt = noSqlDbSystem.sql(sqlStatement);
11
12         Dataset<Row> dataset = noSqlDbSqlStmt.toDataframe();
13     }
14 }
```

The SQL interface is based on the programming interface, as it uses it in the background by calling its functions. Given an SQL statement, the statement is parsed and a logical tree comprised of nodes-operators is formed. Then, the nodes are accessed in a specific manner, aiming to express the SQL statement in successive query primitives. Figure 6 shows each SQL clause with a specific color and its corresponding query primitive in the same color, when expressed via the programming interface.

```
SELECT city, AVG(price_per_day)
FROM hotels
WHERE star=5
GROUP BY city
HAVING AVG(price_per_day) > 500
ORDER BY AVG(price_per_day) ASC
LIMIT 20
```

⟹

```
.operateOn("hotels")
.filter(eq("star", 5)).groupBy("city")
.aggregate(avg("price_per_date"))
.filter(gt("AVG(price_per_day)", 500))
.sort(asc("AVG(price_per_day)"))
.limit(20)
.project("city", "AVG(price_per_day)")
```

Figure 6: Example of SQL expression and NoDA query primitives mapping, depicted with colors

In order to support the spatial and spatio-temporal operators of NoDA API through the SQL Interface, we define the equivalent SQL functions listed on Table 9. The functions can be used in the *WHERE* clause of SQL.

Table 10 shows the arguments of supported SQL spatio-temporal functions. The coordinates must have the `(longitude, latitude)` form (including the parenthesis). Dates are passed as strings in the form `'dd/MM/YYYY HH:mm:ss'`. Column names should not be passed with double or single quotes. Examples are shown later in this report, in Listings 4.11, 4.12, 4.13 and 4.15.

Table 9: SQL spatio-temporal functions and the corresponding NoDA geoperators

SQL function	Equivalent NoDA Geoperator
GEO_POLYGON	inGeoPolygon
GEO_RECTANGLE	inGeoRectangle
GEO_CIRCLE_KM	inGeoCircleKm
GEO_CIRCLE_ME	inGeoCircleMeters
GEO_CIRCLE_MI	inGeoCircleMiles
GEO_NEAREST_NEIGHBORS	geoNearestNeighbors
GEO_TEMPORAL_POLYGON	inGeoTemporalPolygon
GEO_TEMPORAL_RECTANGLE	inGeoTemporalRectangle
GEO_TEMPORAL_CIRCLE_KM	inGeoTemporalCircleKm
GEO_TEMPORAL_CIRCLE_ME	inGeoTemporalCircleMeters
GEO_TEMPORAL_CIRCLE_MI	inGeoTemporalCircleMiles

3.3.3 Related Module Technical Details

The *noda-client* module consist of the following packages with their respective concrete and abstract classes:

- **gr.ds.unipi.noda.api.client**
  - *NoSqlDbSystem* – abstract class that represents the common form of the client interface. It is extended by the System classes in the other packages (except of the SQL) where each store integrates its required arguments for the connection establishment.
- **gr.ds.unipi.noda.api.mongo**
  - *MongoDBBuilderFactory* – concrete class that specifies some Builders for the required parameters when establishing a connection to MongoDB.
  - *MongoDBSystem* – concrete class that extends the NoSqlDbSystem abstract class for configuring the client interface over MongoDB.
- **gr.ds.unipi.noda.api.hbase**
  - *HBaseBuilderFactory* – concrete class that specifies a single Builder without parameters as they are defined in the next step for establishing a connection to HBase.
  - *HBaseSystem* – concrete class that extends the NoSqlDbSystem abstract class for configuring the client interface over HBase.
- **gr.ds.unipi.noda.api.redisearch**
  - *RediSearchBuilderFactory* – concrete class that specifies some Builders for the required parameters when establising a connection to RedisSearch.

Table 10: SQL spatio-temporal functions arguments

SQL function	Arguments
GEO_POLYGON	(locationColumnName, [coordinates1, coordinates2, coordinates3, ...])
GEO_RECTANGLE	(locationColumnName, [lowerCoordinates, upperCoordinates])
GEO_CIRCLE_KM	(locationColumnName, coordinates, radiusInKm)
GEO_CIRCLE_ME	(locationColumnName, coordinates, radiusInMe)
GEO_CIRCLE_MI	(locationColumnName, coordinates, radiusInMi)
GEO_NEAREST_NEIGHBORS	(locationColumnName, coordinates, neighbors)
GEO_TEMPORAL_POLYGON	(locationColumnName, [coordinates1, coordinates2, coordinates3, ...], temporalColumnName, lowerDate, upperDate)
GEO_TEMPORAL_RECTANGLE	(locationColumnName, [lowerCoordinates, upperCoordinates], temporalColumnName, lowerDate, upperDate)
GEO_TEMPORAL_CIRCLE_KM	(locationColumnName, coordinates, radiusInKm, temporalColumnName, lowerDate, upperDate)
GEO_TEMPORAL_CIRCLE_ME	(locationColumnName, coordinates, radiusInMe, temporalColumnName, lowerDate, upperDate)
GEO_TEMPORAL_CIRCLE_MI	(locationColumnName, coordinates, radiusInMi, temporalColumnName, lowerDate, upperDate)

  - *RediSearchSystem* – concrete class that extends the NoSqlDbSystem abstract class for configuring the client interface over RedisSearch.
  - *RediSearchSentinelSystem* – concrete class that extends the NoSqlDbSystem abstract class for configuring the client interface over RedisSearch when sentinels are used. Sentinel is a node that keeps track on master node and other slave nodes. Sentinels constitute a monitoring solution for Redis instances, handling automatic failover of Redis masters and service discovery.

- **gr.ds.unipi.noda.api.sql**
  - *CaseChangingCharStream* – concrete class that converts an SQL statement characters to lower or upper case (upper case conversion is used in the API).
  - *NoSqlDbSqlStatement* – concrete class that represents the operation that will occur after the definition of the SQL expression.
  - *NoSqlDbSqlStatementListener* – concrete class that extends from the SqlBaseBaseListener so as to parse the SQL statement in the form of tree whose nodes are operators.

These operators are mapped with the query primitives of NoDA and the offered operators.

- *SqlBaseBaseListener* – generated class from the *ANTLR* tool which represents the base listener of SQL base
- *SqlBaseLexer* – generated class from the *ANTLR* tool which represents the SQL base lexer
- *SqlBaseListener* – generated class from the *ANTLR* tool which represents the listener of SQL base
- *SqlBaseParser* – generated class from the *ANTLR* tool which represents the SQL base parser

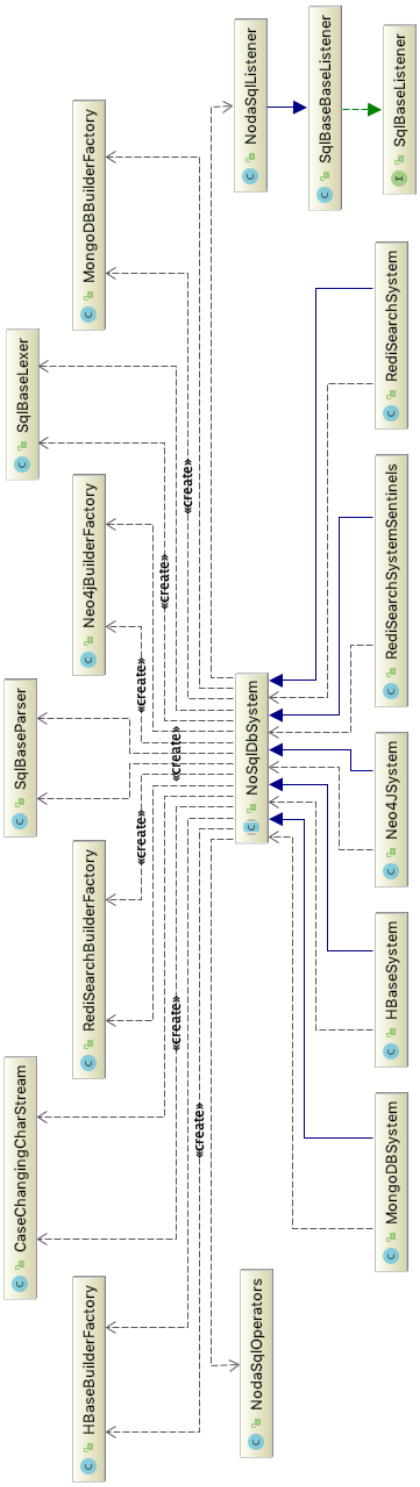

Figure 7: UML class diagram of *noda-client* module

3.4 Implementation for MongoDB

MongoDB is a document-oriented NoSQL database, storing data as documents which are BSON (binary structured object notation) objects – serialized JSON objects. A document contains pairs of fields and values, where values can be a specific data type such as string, numeric, boolean, etc. Documents are organized in collections which are namespaces that group data without enforcing a specific schema; every document may have different fields. Collections are organized in databases which constitute physical containers of them with their own set of files on the file system.

The support of schema-less data models offers the advantage of handling cases where the schema changes frequently. Also, every document can store all of the information related to it in different fields, supporting simultaneously sub-documents (nested documents). This is advantageous for queries as all of the required data is included to the document, without having to perform join operations between collections (as most NoSQL stores, MongoDB does not support joins). The replication of related data to documents leads to denormalized datasets, forming a hierarchical structure.

MongoDB supports horizontal scalability through sharding, meaning that it can exploit additional nodes/shards for handling datasets across multiple machines. A shard is a MongoDB instance which contains a subset of the data for a sharded cluster. Shards hold the entire data set of the cluster. MongoDB supports also data replication which aims to deal with automated failover via replica dataset. We refer to Deliverable D3.2 for a more detailed description of MongoDB, and we omit further details here.

3.4.1 Loading and Querying Spatio-temporal Data

In order to insert data in MongoDB, a pre-processing step is required for forming the data in JSON format if they exist in other format such as CSV which includes delimeter separated records. Then, by having the data in JSON format, we use the mongodb java driver for converting them to Documents (BSON objects) and doing bulk insertion simultaneously; every JSON record is converted to a Document and then added to a list. The list is sent to the database when having specific number of elements. This is an alternative way of inserting documents to the database (bulk insertion), more efficient than the common one which is the one-by-one insertion of documents to the database.

Since NoDA focuses on data with spatial extent, the documents should contain the spatial coordinates in particular form. Specifically, each document should contain GeoJSON object since they are supported by the 2dsphere index which is exploited for spatial and spatio-temporal querying. The GeoJSON field is added when modeling the data as Documents, at the insertion phase as mentioned previously.

MongoDB does offer many types of indexes, such as single-field, compound, geospatial, mul-

tikey, and text, which boost the performance of query execution. However, it does not provide built-in mechanisms oriented to indexing spatio-temporal data that is stored in a collection. One can exploit the compound index (2dsphere and single-value index combination) over the location and time fields for performing spatio-temporal queries. In this way, the order of the fields listed in a compound index is significant, since (for a given spatio-temporal query) the index will first scan the keys of the first field and then it will scan the keys of the second field following their declared order. Essentially, this means that performance is largely based on the selectivity of the constraint (spatial or temporal) on the first field of the compound index.

Motivated by this, we adopt the following indexing approach. We map every spatio-temporal point to to one-dimensional values by exploiting the Hilbert space-filling curve. In other words, we partition the space to disjoint 3D cells where each one covers a specific geographical area for a specific time period. Then, a space-filling curve is used in order to map cells to one-dimensional (1D) values, in a way that it preserves the locality in the 3D space also in the 1D mapping. Every cell is represented by a unique value. By adding this value to each spatio-temporal point as a field (on the insertion phase when Document is formed by a JSON record), we build a single-value index based on it. At query time, a spatio-temporal query can exploit the index by determining at first the value of the 3D cell in which it belongs. Then, having the value of the 3D cell, the index can retrieve all of the documents whose spatio-temporal part is enclosed by the cell. This requires a refinement as a final step, where the spatial and temporal fields of the documents should be checked against the query constraints.

3.4.2 Related Module Technical Details

The *noda-mongodb* module consists of the following concrete classes and abstract classes grouped by packages (depicted in Figure 8 using a class diagram).

- **gr.ds.unipi.noda.api.mongodb**
  - *MongoDBConnectionFactory* – concrete class that extends NoSqlDbConnectionFactory core abstract class so as to provide to the user of the API the query primitives and the choice of closing the established database connection to MongoDB.
  - *MongoDBConnectionManager* – concrete class that extends NoSqlDbConnectionManager core abstract class for managing the connections on MongoDB. The class manages the dictionary data structure which stores the MongoDB connections in *<key, value>* form where key is a MongoDBConnector object and value is MongoClient object offered by the mongodb-java-driver native library.
  - *MongoDBConnector* – concrete class that implements the NoSqlDbConnector core interface, so as to be used in the MongoDBConnectionManager class when checking the existence of a connection to a MongoDB store. The class retains information related to the connection, such as IP addresses and credentials, used by its equals and hashCode methods.

- *MongoDBOperators* – concrete class that extends NoSqlDbOperators, offering the functionality of query primitives for the MongoDB store.

- **gr.ds.unipi.noda.api.mongodb.filterOperators.comparisonOperators**
  - *MongoDBComparisonOperatorFactory* – concrete class that extends BaseComparisonOperatorFactory core abstract class, acting as a facet of comparison operators over MongoDB in order to be forwarded to the *FilterOperators* core concrete class.
  - *ComparisonOperator* – abstract class that extends the *ComparisonOperator* core abstract class, representing a comparison operator over MongoDB. It is extended by the concrete classes of this package that represent a particular comparison operator.
  - *OperatorEqual* – concrete class that extends ComparisonOperator abstract class, implementing the *equals* conditional operator over MongoDB.
  - *OperatorGreaterThan* – concrete class that extends ComparisonOperator abstract class, implementing the *greater than* conditional operator over MongoDB.
  - *OperatorGreaterThanEqual* – concrete class that extends ComparisonOperator abstract class, implementing the *greather than equal* conditional operator over MongoDB.
  - *OperatorLessThan* – concrete class that extends ComparisonOperator abstract class, implementing the *less than* conditional operator over MongoDB.
  - *OperatorLessThanEqual* – concrete class that extends ComparisonOperator abstract class, implementing the *less than equal* conditional operator over MongoDB.
  - *OperatorNotEqual* – concrete class that extends ComparisonOperator abstract class, implementing the *not equals* conditional operator over MongoDB.

- **gr.ds.unipi.noda.api.mongodb.filterOperators.logicalOperators**
  - *MongoDBLogicalOperatorFactory* – concrete class that extends BaseLogicalOperatorFactory core abstract class, acting as a facet of logical operators over MongoDB in order to be forwarded to the *FilterOperators* core concrete class.
  - *LogicalOperator* – abstract class that extends the *LogicalOperator* core abstract class, representing a logical operator over MongoDB. It is extended by the concrete classes of this package that represent a particular logical operator.
  - *OperatorAnd* – concrete class that extends LogicalOperator abstract class, implementing the *and* logical operator over MongoDB.
  - *OperatorOr* – concrete class that extends LogicalOperator abstract class, implementing the *or* logical operator over MongoDB.

- **gr.ds.unipi.noda.api.mongodb.filterOperators.geoperators .geographicalOperators**

- *MongoDBGeographicalOperatorFactory* – concrete class that extends BaseGeographicalOperatorFactory core abstract class, acting as a facet of geographical operators over MongoDB in order to be forwarded to the *FilterOperators* core concrete class.
- *GeographicalOperator* – abstract class that extends the *GeographicalOperator* core abstract class, representing a geographical operator over MongoDB. It is extended by the concrete classes of this package that represent a particular geographical operator.
- *OperatorInGeoRectangle* – concrete class that extends GeographicalOperator abstract class, implementing the *geospatial rectangle* filter operator over MongoDB.
- *OperatorInGeoCircle* – concrete class that extends GeographicalOperator abstract class, implementing the *geospatial circle* filter operator over MongoDB.
- *OperatorInGeoPolygon* – concrete class that extends GeographicalOperator abstract class, implementing the *geospatial polygon* filter operator over MongoDB.
- *OperatorGeoNearestNeighbors* – concrete class that extends GeographicalOperator abstract class, implementing the *geospatial nearest neighboors* operator over MongoDB.

- **gr.ds.unipi.noda.api.mongodb.filterOperators.geoperators .geoTemporalOperators**
  - *MongoDBGeoTemporalOperatorFactory* – concrete class that extends BaseGeoTemporalOperatorFactory core abstract class, acting as a facet of geotemporal operators over MongoDB in order to be forwarded to the *FilterOperators* core concrete class.
  - *GeoTemporalOperator* – abstract class that extends the *GeoTemporalOperator* core abstract class, representing a geotemporal operator over MongoDB. It is extended by the concrete classes of this package that represent a particular geotemporal operator.
  - *OperatorInGeoTemporalRectangle* – concrete class that extends GeographicalOperator abstract class, implementing the *geotemporal rectangle* filter operator over MongoDB.
  - *OperatorInGeoTemporalCircle* – concrete class that extends GeographicalOperator abstract class, implementing the *geotemporal circle* filter operator over MongoDB.
  - *OperatorInGeoTemporalPolygon* – concrete class that extends GeographicalOperator abstract class, implementing the *geotemporal polygon* filter operator over MongoDB.

- **gr.ds.unipi.noda.api.mongodb.aggregateOperators**
  - *MongoDBAggregateOperatorFactory* – concrete class that extends BaseAggregateOperatorFactory core abstract class, acting as a facet of aggregate operators over MongoDB in order to be forwarded to the *AggregateOperators* core concrete class.
  - *AggregateOperator* – abstract class that extends the *AggregateOperator* core abstract class, representing an aggregate operator over MongoDB. It is extended by the concrete classes of this package that represent a particular aggregate operator.

- *OperatorAvg* – concrete class that extends AggregateOperator abstract class, calculating the average of a set of numeric values that may result from applying a specified group expression over MongoDB.
- *OperatorMin* – concrete class that extends AggregateOperator abstract class, calculating the minimum value of a set of numeric values that may result from applying a specified group expression over MongoDB.
- *OperatorMax* – concrete class that extends AggregateOperator abstract class, calculating the maximum value of a set of numeric values that may result from applying a specified group expression over MongoDB.
- *OperatorSum* – concrete class that extends AggregateOperator abstract class, calculating the sum value of a set of numeric values that may result from applying a specified group expression over MongoDB.
- *OperatorCount* – concrete class that extends AggregateOperator abstract class, calculating the number of documents that may result from applying a specified group expression over MongoDB.
- *OperatorCountNonNull* – concrete class that extends AggregateOperator abstract class, calculating the number of non-null values in a set that may result from applying a specified group expression over MongoDB.

- **gr.ds.unipi.noda.api.mongodb.sortOperators**
  - *MongoDBSortOperatorFactory* – concrete class that extends BaseSortOperatorFactory core abstract class, acting as a facet of sort operators over MongoDB in order to be forwarded to the *SortOperators* core concrete class.
  - *SortOperator* – abstract class that extends the *SortOperator* core abstract class, representing a sort operator over MongoDB. Extended by the concrete classes of this package that represent a particular sort operator.
  - *OperatorAsc* – concrete class that extends SortOperator abstract class, sorting the documents in ascending order by a column on the MongoDB store.
  - *OperatorDesc* – concrete class that extends SortOperator abstract class, sorting the documents in descending order by a column on the MongoDB store.

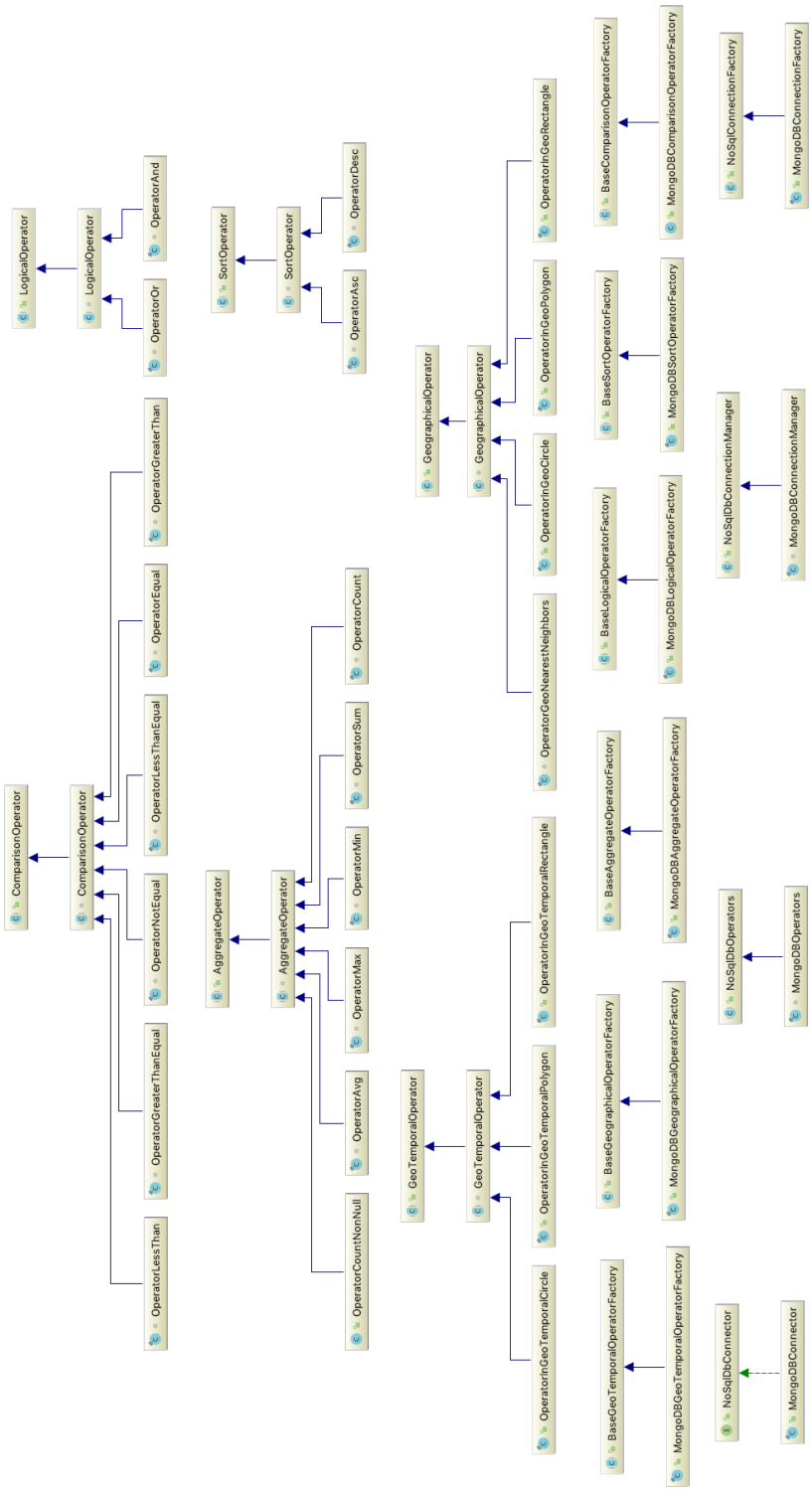

Figure 8: UML class diagram of *noda-mongdb* module

3.5 Implementation for HBase

HBase is a wide-column NoSQL database, storing data as byte arrays in HDFS. All data is stored in tables with rows and columns like in relational databases. Every row consists of cells which are the smallest basic unit of storage. A cell constitutes the actual stored value in a column in serialized form. It embodies versioning, meaning that it can store different values based on their timestamp. Each row has a unique key (*row id*) associated with it, being its identifier. This kind of identifier can be any type, as it is stored as a byte array too. Given a row id, we can access its associated columns (cells).

In HBase, columns are organized under specific groups named *column families*. Column families are stored separately on disk, offering fast column retrieval. Given a row id, a column name and its column family, we can access its actual value of it (cell). As byte arrays are accessed, they are implicitly converted to the data equivalent representation.

The underlying storage of HBase is based on HFiles which stores the row ids in lexicographic order. HFile is a block index file format where data is stored in a sequence of blocks and a separate index is maintained at the end of the file to locate the blocks.

HBase supports horizontal scalability, utilizing RegionServers which collocate with the HDFS DataNodes. A RegionServer contains a subset from a dataset where rows in a table are stored between the region's start key and end key. RegionServers are coordinating through the HBase Master which handles region assignment and operations related with create and delete tables.

3.5.1 Loading and Querying spatial data

When inserting a record to the HBase store, a row key being the unique identifier of the record is defined. This simulates the index key, as HBase does not offer directly indexes but integrates their rationale through filter operations on the row key. Two columns families are declared; one for storing columns related to the location and time (longitude, latitude, date) of the spatio-temporal point and another one for storing the remaining information related to it. This type of column separation is applied, since column families are stored in different files. This enhances the performance of a spatial query in the refinement phase as only the required information is accessed.

Since we focus on spatio-temporal querying and there exist efficient filters operating on Row Keys, the Row Key is formed so as to integrate both spatial and temporal information. This is achieved by using the geohash string value for the spatial field with a Unix timestamp for the temporal field. These fields are combined with a random string in sequence. Through the integration of a random string value on the Row key, it is possible to have distinct keys for records whose spatial and temporal information is represented by the same values.

The geohash value represents a specific region on earth, enclosed by another region which is also represented by a geohash value with fewer characters. These regions are resulted from

geohashing, a technique invented by Gustavo Niemeyer. Geohashing is a geocoding method used to encode geographic coordinates (latitude and longitude) into a short string of digits and letters. The regions are cells with varying resolutions. The more characters in the geohash string, the more precise the location it represents. The geohash technique is based on the rationale of splitting the space by using bits. Particularly, it specifies 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 as shown in Figure 9. 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, following the z-ordering.

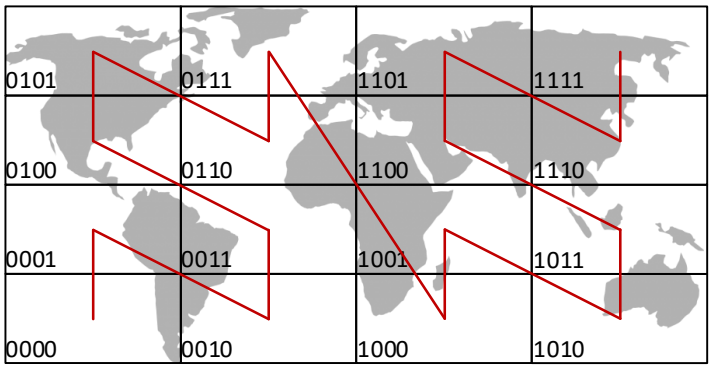

Figure 9: Geohashing space splitting

The *fuzzy row filter* is one of the built-in filters of HBase which operates on the row id. It takes as parameters a row key with fuzzy info to match row keys. The row key may contain string characters with the '?' wildcard characters. The wildcard indicates that at a specific position may match any character of the fetched keys, whereas any other character defines exact matching. Given that the adopted row key format is *geohash-timestamp_-randomstring*, an input key gbsuv7zv-??????????????-????? would fetch all the row keys that start with gbsuv7zv. Equivalently, this means that the points that exist in the area represented by this specific geohash are returned from the execution of the filter. This is the approach adopted by NoDA for querying HBase.

In NoDA, we use a geohash string for storing points that consists of 8 characters, the Unix timestamp of 13 numerical digits and the random string of 5 characters. Given a spatio-temporal box query, at first, a single geohash that represents the space (or a larger part of) is computed. If the space contains more than 8 characters, we consider the represented geohash as an 8 character string by pruning it. If fewer than 8 characters are resulted, wildcard character/s (?) are added

at the end of the string so as to consist of 8 characters. At the second phase, the common part of Unix timestamp of the lower and upper temporal bounds is found. Then, wildcard character/s are added to the common part so as to form a string with 13 characters. The formed spatial and temporal strings are concatenated with 5 wildcard characters which correspond to the random string part of the row id. The formed string is given as an input to the fuzzy row filter which fetches both the true-positive and the false-positive spatio-temporal points. Specifically, the filter retrieves the rows that match with the given expression. Then, a refinement step takes place for filtering out the points whose spatial and temporal part are not enclosed by the lower and upper bounds of the box query.

Given a spatio-temporal cylinder or polyhedron query where its spatial extent is circle and polygon respectively, the query is handled via its minimum bounding rectangle. This means that the minimum bounding box of these queries is computed and performed with the aforementioned procedure. At the refinement phase, the spatial extent of the initial query is taken into account for determining the true-positive results.

The applied string scheme can also handle spatial queries. The same procedure takes place for the spatial part of the row id. The temporal part is handled with a 13 wildcard characters string. This means that the filter will not apply any constraint on the temporal part.

3.5.2 Related Module Technical Details

The *noda-hbase* module consists of the following concrete classes and abstract classes grouped by packages (graphically illustrated in Figure 10).

- **gr.ds.unipi.noda.api.hbase**
  - *HBaseConnectionFactory* – concrete class that extends NoSqlDbConnectionFactory core abstract class so as to provide to the user of the API the query primitives and the choice of closing the established database connection to HBase.
  - *HBaseConnectionManager* – concrete class that extends NoSqlDbConnectionManager core abstract class for managing the connections on HBase database. The class manages the dictionary data structure which stores the HBase connections in *<key, value>* form where key is a HBaseConnector object and value is Connection object offered by the hbase-client native library.
  - *HBaseConnector* – concrete class that implements the NoSqlDbConnector core interface so as to be used in the HBaseConnectionManager class when checking the existence of a connection to a HBase store. The class retains information related to the connection such as IP addresses and credentials, used by its equals and hashCode methods.
  - *HBaseOperators* – concrete class that extends NoSqlDbOperators, offering the functionality of query primitives for HBase.

- **gr.ds.unipi.noda.api.hbase.filterOperators.comparisonOperators**
  - *HBaseComparisonOperatorFactory* – concrete class that extends BaseComparisonOperatorFactory core abstract class, acting as a facet of comparison operators over HBase in order to be forwarded to the *FilterOperators* core concrete class.
  - *ComparisonOperator* – abstract class that extends the *ComparisonOperator* core abstract class, representing a comparison operator over HBase. It is extended by the concrete classes of this package that represent a particular comparison operator.
  - *OperatorEqual* – concrete class that extends ComparisonOperator abstract class, implementing the *equals* conditional operator over HBase.
  - *OperatorGreaterThan* – concrete class that extends ComparisonOperator abstract class, implementing the *greater than* conditional operator over HBase.
  - *OperatorGreaterThanEqual* – concrete class that extends ComparisonOperator abstract class, implementing the *greather than equal* conditional operator over HBase.
  - *OperatorLessThan* – concrete class that extends ComparisonOperator abstract class, implementing the *less than* conditional operator over HBase.
  - *OperatorLessThanEqual* – concrete class that extends ComparisonOperator abstract class, implementing the *less than equal* conditional operator over HBase.
  - *OperatorNotEqual* – concrete class that extends ComparisonOperator abstract class, implementing the *not equals* conditional operator over HBase.
- **gr.ds.unipi.noda.api.hbase.filterOperators.logicalOperators**
  - *HBaseLogicalOperatorFactory* – concrete class that extends BaseLogicalOperatorFactory core abstract class, acting as a facet of logical operators over HBase in order to be forwarded to the *FilterOperators* core concrete class.
  - *LogicalOperator* – abstract class that extends the *LogicalOperator* core abstract class, representing a logical operator over HBase. It is extended by the concrete classes of this package that represent a particular logical operator.
  - *OperatorAnd* – concrete class that extends LogicalOperator abstract class, implementing the *and* logical operator over HBase.
  - *OperatorOr* – concrete class that extends LogicalOperator abstract class, implementing the *or* logical operator over HBase.
- **gr.ds.unipi.noda.api.hbase.filterOperators.geoperators.geographicalOperators**
  - *HBaseGeographicalOperatorFactory* – concrete class that extends BaseGeographicalOperatorFactory core abstract class, acting as a facet of geographical operators over HBase in order to be forwarded to the *FilterOperators* core concrete class.

- *GeographicalOperator* – abstract class that extends the *GeographicalOperator* core abstract class, representing a geographical operator over HBase. It is extended by the concrete classes of this package that represent a particular geographical operator.
- *OperatorInGeoRectangle* – concrete class that extends GeographicalOperator abstract class, implementing the *geospatial rectangle* filter operator over HBase.
- *OperatorInGeoCircle* – concrete class that extends GeographicalOperator abstract class, implementing the *geospatial circle* filter operator over HBase.
- *OperatorInGeoPolygon* – concrete class that extends GeographicalOperator abstract class, implementing the *geospatial polygon* filter operator over HBase.

- **gr.ds.unipi.noda.api.hbase.filterOperators.geoperators.geographicalOperators .customFilters**
  - *CircleFilter* – concrete class that extends the FilterBase class which is provided by HBase for custom filtering. The class does spatial filtering given a circle and is used in the refinement step when performing a spatial circle query.
  - *PolygonFilter* – concrete class that extends the FilterBase class which is provided by HBase for custom filtering. The class does spatial filtering given a polygon and is used in the refinement step when performing a spatial polygon query.
  - *RectangleFilter* – concrete class that extends the FilterBase class which is provided by HBase for custom filtering. The class does spatial filtering given a rectangle and is used in the refinement step when performing a spatial rectangle query.

- **gr.ds.unipi.noda.api.hbase.filterOperators.geoperators.geographicalOperators .customFilters.generated**
  - *CircleFilterProtos* – generated class by the protocol buffer compiler so as to support the customized filter of spatial circle.
  - *PolygonFilterProtos* – generated class by the protocol buffer compiler so as to support the customized filter of spatial polygon.
  - *RectangleFilterProtos* – generated class by the protocol buffer compiler so as to support the customized filter of spatial rectangle.

- **gr.ds.unipi.noda.api.hbase.filterOperators.geoperators.geoTemporalOperators**
  - *HBaseGeoTemporalOperatorFactory* – concrete class that extends BaseGeoTemporalOperatorFactory core abstract class, acting as a facet of geotemporal operators over HBase in order to be forwarded to the *FilterOperators* core concrete class.
  - *GeoTemporalOperator* – abstract class that extends the *GeoTemporalOperator* core abstract class, representing a geotemporal operator over HBase. It is extended by the concrete classes of this package that represent a particular geotemporal operator.

- *OperatorInGeoTemporalRectangle* – concrete class that extends GeographicalOperator abstract class, implementing the *geotemporal rectangle* filter operator over HBase.
- *OperatorInGeoTemporalCircle* – concrete class that extends GeographicalOperator abstract class, implementing the *geotemporal circle* filter operator over HBase.
- *OperatorInGeoTemporalPolygon* – concrete class that extends GeographicalOperator abstract class, implementing the *geotemporal polygon* filter operator over HBase.

- **gr.ds.unipi.noda.api.hbase.filterOperators.geoperators.geoTemporalOperators .customFilters**
  - *CircleTemporalFilter* – concrete class that extends the FilterBase class which is provided by HBase for custom filtering. The class does spatio-temporal filtering given a circle and time bounds. It is used in the refinement step when performing a spatio-temporal cylinder query.
  - *PolygonTemporalFilter* – concrete class that extends the FilterBase class which is provided by HBase for custom filtering. The class does spatio-temporal filtering given a polygon and time bounds. It is used in the refinement step when performing a spatio-temporal polyhedron query.
  - *RectangleTemporalFilter* – concrete class that extends the FilterBase class which is provided by HBase for custom filtering. The class does spatio-temporal filtering given a rectangle and time bounds. It is used in the refinement step when performing a spatio-temporal box query.

- **gr.ds.unipi.noda.api.hbase.filterOperators.geoperators.geoTemporalOperators .customFilters.generated**
  - *CircleTemporalFilterProtos* – generated class by the protocol buffer compiler so as to support the customized filter of spatio-temporal cylinder.
  - *PolygonTemporalFilterProtos* – generated class by the protocol buffer compiler so as to support the customized filter of spatio-temporal polyhedron.
  - *RectangleTemporalFilterProtos* – generated class by the protocol buffer compiler so as to support the customized filter of spatio-temporal box.

- **gr.ds.unipi.noda.api.hbase.aggregateOperators**
  - *HBaseAggregateOperatorFactory* – concrete class that extends BaseAggregateOperatorFactory core abstract class, acting as a facet of aggregate operators over HBase in order to be forwarded to the *AggregateOperators* core concrete class. All of the methods in the HBaseAggregateOperatorFactory class return null, as aggregations are not supported directly from HBase.

- **gr.ds.unipi.noda.api.hbase.sortOperators**

- *HBaseSortOperatorFactory* – concrete class that extends BaseSortOperatorFactory core abstract class, acting as a facet of sort operators over HBase in order to be forwarded to the *SortOperators* core concrete class.
- *SortOperator* – abstract class that extends the *SortOperator* core abstract class, representing a sort operator over HBase. It is extended by the concrete classes of this package that represent a particular sort operator.
- *OperatorAsc* – concrete class that extends SortOperator abstract class, sorting the records in ascending order by a column on HBase.
- *OperatorDesc* – concrete class that extends SortOperator abstract class, sorting the records in descending order by a column on HBase.

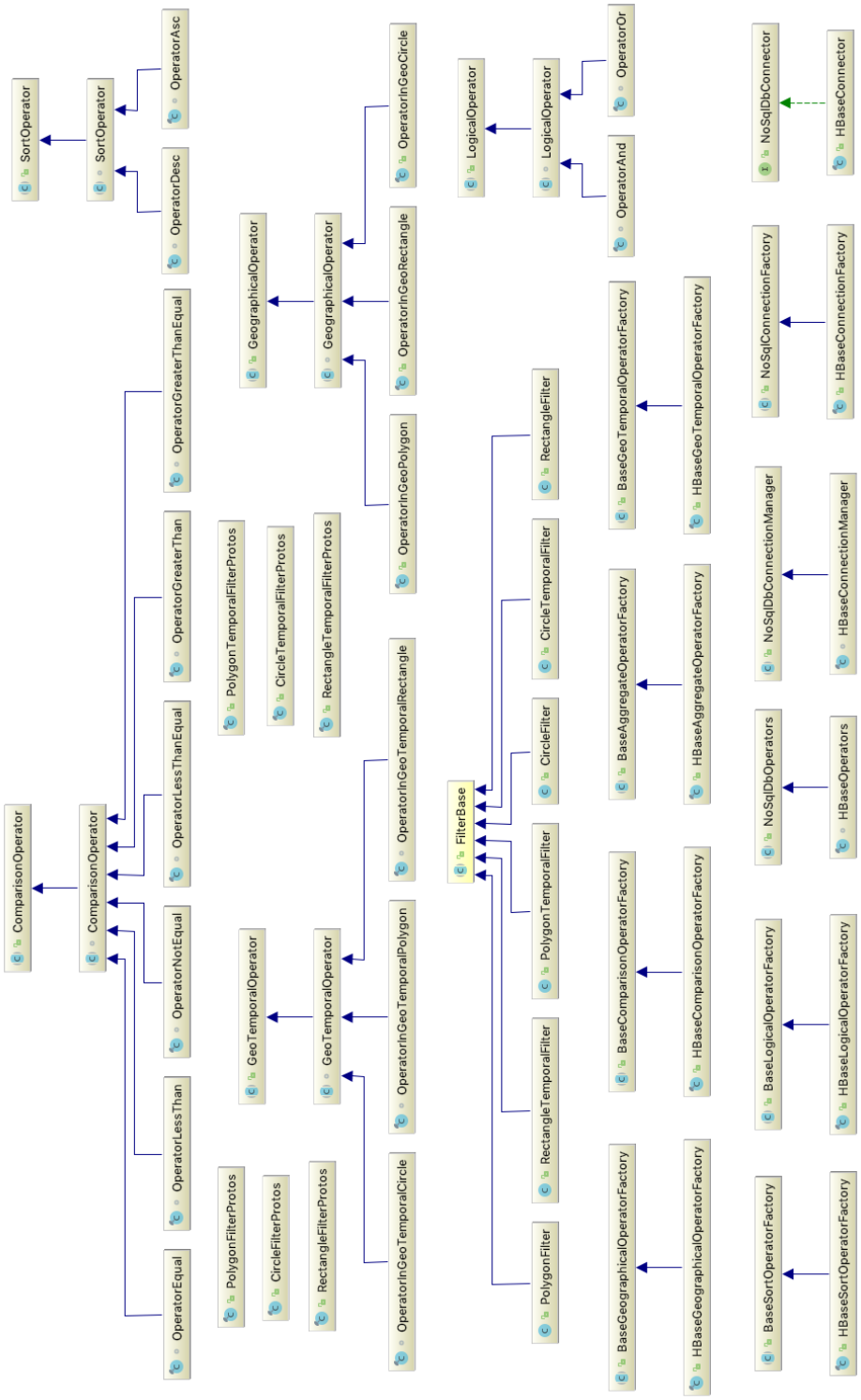

Figure 10: UML class diagram of *noda-hbase* module

3.6 Implementation for Redis

Redis is a key-value, main-memory, NoSQL store that supports many data types and structures, including strings, lists, sets, and bit arrays. A key in Redis is binary-safe, meaning that we can use any binary sequence, from a string to the content of a JPEG file. Also, an empty string is regarded as a valid key. Also, Redis provides the option to store data on hard disk for persistence.

Redis supports sharding through Redis Cluster in which data is split among Redis instances for holding a specific subset of key-values. Two types of partitioning are supported: hash-based and range-based. It adopts the master-slave architecture, non-blocking synchronization and automatic re-connection to the nodes in case of failover. Redis provides high availability via Redis Sentinel which is a distributed system for managing the failover.

At the point of this writing, the implementation over Redis is based on the RediSearch module which is a text search and secondary indexing engine on top of Redis. It does not use Redis internal structure, but it uses its own data structures and algorithms offering advanced search features. It can perform simple text search to complex structured queries, filtering by numeric properties and geographical distances.

RediSearch supports text, numeric and geographical indexes. Specifically, for the geographical fields, it uses sorted sets of Redis which are indexes by nature. Further indexes are created for the field types which have been set.

3.6.1 Loading and Querying spatio-temporal data

In order to perform efficiently search operations, RediSearch should have pre-define the schema of index which indicates how the records will be handled when added. Then, the records can be added according to the schema definition and RedisSearch handles by its own how to store and index every field by using the appropriate data structure. Also, RediSearch handles the data distribution on the Redis Cluster.

RediSearch supports filtering on geospatial data which takes place on *GEO* fields. Redis provides many commands related to geographical locations, but unlike the other available commands, they do not operate on a specific data type. These commands use sorted sets as data type under the hood. The applied technique that is used to fill the sorted set is based on geohashing, which uses z-order space-filling curve. Geohashing is described in subsection 3.5.1 Specifically, longitude and latitude are encoded so as to form a unique 52 bit integer number. The score of a sorted set can represent a 52-bit integer without accuracy loss.

Given a circle and its radius, Redis carries out a geohash search of the cell which contains its center and its 8 adjacent cells (9 cells totally). Practically, a range query is performed in each cell. Next, the distance from the center of the circle to every entry is checked, so as to be included or not in the query result.

RediSearch also supports numeric range queries for numeric data filtering. This takes place

by setting the fields on the index as *NUMERIC*. The numeric values of the records for these fields will be indexed and used for the result filtering of the query's execution. In the current version, sorted sets are used for the representation of numeric data. The bounds of the numeric filters can be exclusive or inclusive.

3.6.2 Related Module Technical Details

The *noda-redisearch* module consists of the following concrete classes and abstract classes grouped by packages (depicted in Figure 11).

- **gr.ds.unipi.noda.api.redisearch**
  - *RediSearchConnectionFactory* – concrete class that extends NoSqlDbConnectionFactory core abstract class so as to provide to the user of the API the query primitives and the choice of closing the established database connection to RediSearch.
  - *RediSearchConnectionManager* – concrete class that extends NoSqlDbConnectionManager core abstract class for managing the connections on RediSearch. The class manages the dictionary data structure which stores the RediSearch connections in *<key, value>* form where key is a RediSearchConnector object and value is Pool object offered by the jredisearch native library.
  - *RediSearchConnector* – concrete class that implements the NoSqlDbConnector core interface so as to be used in the RediSearchConnectionManager class when checking the existence of a connection to a RediSearch store. The class retains information related to the connection such as IP addresses and credentials, used by its equals and hashCode methods.
  - *RediSearchOperators* – concrete class that extends NoSqlDbOperators, offering the functionality of query primitives for RediSearch.
  - *RediSearchQueryHelper* – concrete class that retains states for the rationale structure and execution of the RediSearch query, based on the actions determined at RediSearchOperators.
- **gr.ds.unipi.noda.api.redisearch.filterOperators**
  - *RediSearchPostFilterOperator* – interface that is implemented for handling the Redis query against its filter commands. For instance, having set the primitives *.filter(X).aggregate(Y).filter(U)*, X is equivalent to query, but U is a filter command upon the aggregation.
- **gr.ds.unipi.noda.api.redisearch.filterOperators.comparisonOperators**
  - *RediSearchComparisonOperatorFactory* – concrete class that extends BaseComparisonOperatorFactory core abstract class, acting as a facet of comparison operators over RediSearch in order to be forwarded to the *FilterOperators* core concrete class.

- *ComparisonOperator* – abstract class that extends the *ComparisonOperator* core abstract class, representing a comparison operator over RediSearch. It is extended by the concrete classes of this package that represent a particular comparison operator.
- *OperatorEqual* – concrete class that extends ComparisonOperator abstract class, implementing the *equals* conditional operator over RediSearch.
- *OperatorGreaterThan* – concrete class that extends ComparisonOperator abstract class, implementing the *greater than* conditional operator over RediSearch.
- *OperatorGreaterThanEqual* – concrete class that extends ComparisonOperator abstract class, implementing the *greather than equal* conditional operator over RediSearch.
- *OperatorLessThan* – concrete class that extends ComparisonOperator abstract class, implementing the *less than* conditional operator over RediSearch.
- *OperatorLessThanEqual* – concrete class that extends ComparisonOperator abstract class, implementing the *less than equal* conditional operator over RediSearch.
- *OperatorNotEqual* – concrete class that extends ComparisonOperator abstract class, implementing the *not equals* conditional operator over RediSearch.
- *RangeValue* – concrete class for supporting numeric queries. It bypasses the current bug of the native redisearch library (1.4.0 version) in the RangeValue class for managing the double exponents.

- **gr.ds.unipi.noda.api.redisearch.filterOperators.logicalOperators**
  - *RediSearchLogicalOperatorFactory* – concrete class that extends BaseLogicalOperatorFactory core abstract class, acting as a facet of logical operators over RediSearch in order to be forwarded to the *FilterOperators* core concrete class.
  - *LogicalOperator* – abstract class that extends the *LogicalOperator* core abstract class, representing a logical operator over RediSearch. Extended by the concrete classes of this package that represent a particular logical operator.
  - *OperatorAnd* – concrete class that extends LogicalOperator abstract class, implementing the *and* logical operator over RediSearch.
  - *OperatorOr* – concrete class that extends LogicalOperator abstract class, implementing the *or* logical operator over RediSearch.

- **gr.ds.unipi.noda.api.redisearch.filterOperators.geoperators.geographicalOperators**
  - *RediSearchGeographicalOperatorFactory* – concrete class that extends BaseGeographicalOperatorFactory core abstract class, acting as a facet of geographical operators over RediSearch in order to be forwarded to the *FilterOperators* core concrete class.

- *RediSearchGeographicalOperator* – interface which incorporates the ZRangeInfo as the expression of an operator. It is implemented by the OperatorInGeoRectangle class. By default, Node object is returned from the getOperatorExpression method of the extended GeographicalOperator abstract class. This is not supported by the OperatorInGeoRectangle class.
- *GeographicalOperator* – abstract class that extends the *GeographicalOperator* core abstract class, representing a geographical operator over RediSearch. Extended by the concrete classes of this package that represent a particular geographical operator.
- *OperatorInGeoRectangle* – concrete class that extends GeographicalOperator abstract class and implements the RediSearchGeographicalOperator interface. It performs the *geospatial rectangle* filter operator over RediSearch.
- *OperatorInGeoCircle* – concrete class that extends GeographicalOperator abstract class, implementing the *geospatial circle* filter operator over RediSearch.
- *OperatorGeoNearestNeighbors* – concrete class that extends GeographicalOperator abstract class, implementing the *geospatial nearest neighboors* operator over RediSearch.
- *ZRangeInfo* – concrete class that retains the required information needed for the Zrange call for spatial querying.

- **gr.ds.unipi.noda.api.redisearch.aggregateOperators**
  - *RediSearchAggregateOperatorFactory* – concrete class that extends BaseAggregateOperatorFactory core abstract class, acting as a facet of aggregate operators over RediSearch in order to be forwarded to the *AggregateOperators* core concrete class.
  - *AggregateOperator* – abstract class that extends the *AggregateOperator* core abstract class, representing an aggregate operator over RediSearch. It is extended by the concrete classes of this package that represent a particular aggregate operator.
  - *OperatorAvg* – concrete class that extends AggregateOperator abstract class, calculating the average of a set of numeric values that may result from applying a specified group expression over RediSearch.
  - *OperatorMin* – concrete class that extends AggregateOperator abstract class, calculating the minimum value of a set of numeric values that may result from applying a specified group expression over RediSearch.
  - *OperatorMax* – concrete class that extends AggregateOperator abstract class, calculating the maximum value of a set of numeric values that may result from applying a specified group expression over RediSearch.
  - *OperatorSum* – concrete class that extends AggregateOperator abstract class, calculating the sum value of a set of numeric values that may result from applying a specified group expression over RediSearch.

- *OperatorCount* – concrete class that extends AggregateOperator abstract class, calculating the number of records that may result from applying a specified group expression over RediSearch.
- *OperatorCountNonNull* – concrete class that extends AggregateOperator abstract class, calculating the number of non-null values in a set that may result from applying a specified group expression over RediSearch.

- **gr.ds.unipi.noda.api.redisearch.sortOperators**
  - *RediSearchSortOperatorFactory* – concrete class that extends BaseSortOperatorFactory core abstract class, acting as a facet of sort operators over RediSearch in order to be forwarded to the *SortOperators* core concrete class.
  - *SortOperator* – abstract class that extends the *SortOperator* core abstract class, representing a sort operator upon RediSearch store. It is extended by the concrete classes of this package that represent a particular sort operator.
  - *OperatorAsc* – concrete class that extends SortOperator abstract class, sorting the records in ascending order by a column on RediSearch.
  - *OperatorDesc* – concrete class that extends SortOperator abstract class, sorting the records in descending order by a column on RediSearch.

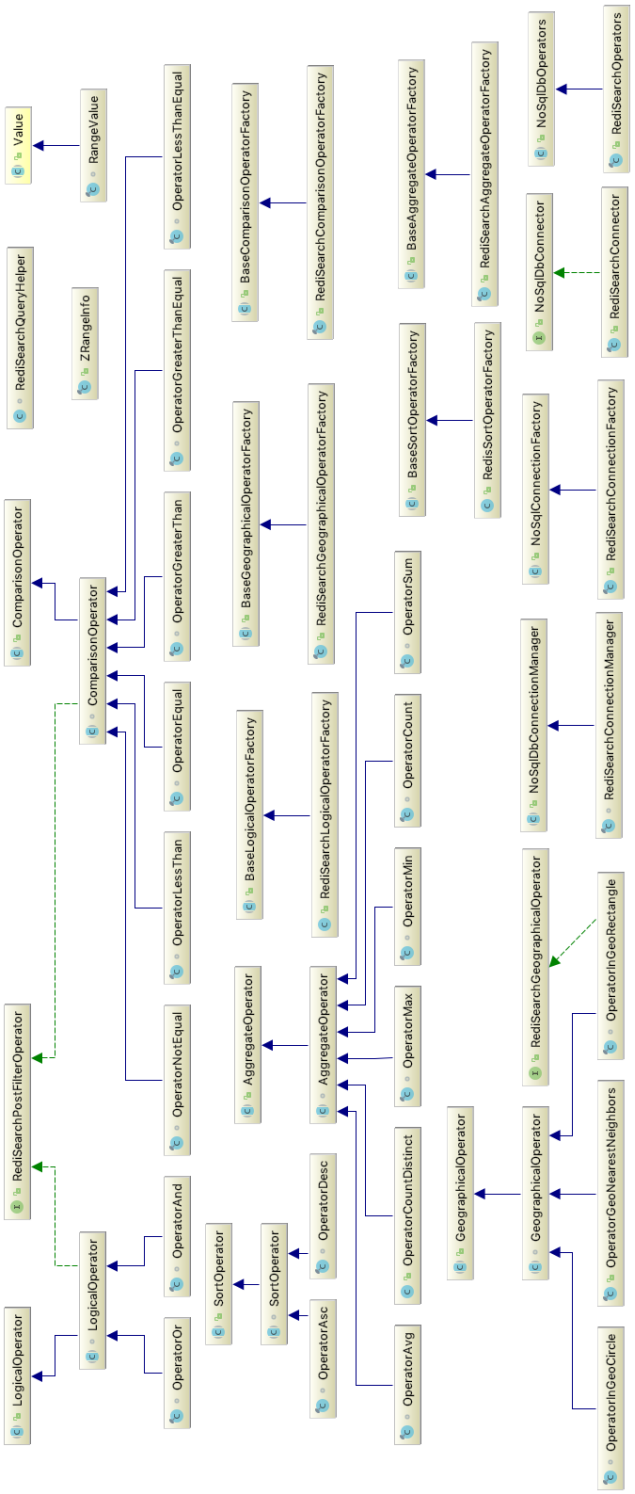

Figure 11: UML class diagram of *noda-redisearch* module

4 Developer's guide – Practical Examples of NoDA

In order to exploit the features of NoDA for accessing data, at first a connection to the database should be defined with its required arguments. Its generic form is shown in Listing 4.1 *YYY()* method represents a NoSQL database and *Builder(...)* accepts the required parameters for establishing a connection. By calling the *Builder()* method, other methods can be called, each one declaring an extra feature of a connection may have (*FEATURE1(...)* and *FEATURE2()* methods). The features are declared optionally as they are not required for the connection establishment. The *build()* method is called last for completing the declaration of the information related to the connection. The *build()* method returns a NoSqlDbSystem object. Listings 4.2, 4.3 and 4.4 show concrete connection declaration upon MongoDB, HBase and Redis stores respectively. Listing 4.5 shows an instance of combining a connection declaration to MongoDB store with a spark session. This allows to fetch queries' results in dataframe form, from which we can handle data results in distributed environment.

Listing 4.1: Generic form of declaring and closing a NoSQL database connection

```
1  import gr.ds.unipi.noda.api.client.NoSqlDbSystem;
2
3  public class GenericConnectionDeclaration {
4      public static void main(String args[]){
5
6          NoSqlDbSystem noSqlDbSystem = NoSqlDbSystem.YYY().Builder().FEATURE1()
7                  .FEATURE2().build();
8
9          //do operations ...
10
11         noSqlDbSystem.close();
12     }
13 }
```

Listing 4.2: Connection declaration on MongoDB Store

```
1  import gr.ds.unipi.noda.api.client.NoSqlDbSystem;
2  import com.mongodb.MongoClientOptions;
3
4  public class MongoDBConnectionDeclaration {
5      public static void main(String args[]){
6
7          MongoClientOptions mco = MongoClientOptions.builder().sslEnabled(true).build();
8
9          NoSqlDbSystem noSqlDbSystem = NoSqlDbSystem.MongoDB()
10                 .Builder("username","password","database")
11                 .host("127.0.0.1").port(9000).mongoClientOptions(mco).build();
12     }
13 }
```

Listing 4.3: Connection declaration on HBase Store

```
import gr.ds.unipi.noda.api.client.NoSqlDbSystem;

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

        NoSqlDbSystem noSqlDbSystem = NoSqlDbSystem.HBase().Builder()
                .host("127.0.0.1").port(9000)
                .addProperty("hbase.rpc.timeout","60000").build();
    }
}
```

Listing 4.4: Connection declaration on Redis Store, using RediSearch

```
import gr.ds.unipi.noda.api.client.NoSqlDbSystem;

public class RediSearchConnectionDeclaration {

    public static void main(String args[]){

        NoSqlDbSystem noSqlDbSystem = NoSqlDbSystem.RediSearch().Builder("masterName")
                .host("127.0.0.1").port(9000).connectionTimeout(60000).build();
    }
}
```

Listing 4.5: Declaring and closing a connection on MongoDB Store for data access, combining a Spark session

```
import gr.ds.unipi.noda.api.client.NoSqlDbSystem;
import com.mongodb.MongoClientOptions;
import org.apache.spark.sql.SparkSession;

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

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

        MongoClientOptions mco = MongoClientOptions.builder().sslEnabled(true).build();

        NoSqlDbSystem noSqlDbSystem = NoSqlDbSystem.MongoDB()
                .Builder("username","password","database").host("127.0.0.1")
                .port(9000).mongoClientOptions(mco).sparkSession(session).build();

        //do operations ...

        noSqlDbSystem.close();
    }
}
```

The *host* and *port* methods are not required for being defined. If they are not set, localhost string value will be used for the *host* and *port* will use the storage's default port when running as a standalone instance.

The *Builder* methods are unique for each store, as a NoSQL database may not require some

parameters that another database requires for connection establishment. The same applies for the feature methods. These were designed by taking account each store's peculiarities. For instance, HBase uses properties for declaring parameters where other databases do not declare parameters in such way. We have made an effort towards grouping some feature methods that have exact the same meaning for the databases (*host*, *port* and *spark* methods).

The usage of the spark method which requires a spark session as an argument, enables the query result to be fetched as a Dataframe (*toDataframe* query primitive).

Having set the information for connection establishment, we can proceed to operate on the data stored in the NoSQL database by using either the programming interface (Section 4.1) or the SQL interface (Section 4.2).

4.1 Use of the Programming Interface

Below are following 5 practical examples of using the programming interface for accessing spatio-temporal data.

Listing 4.6 shows the instance of getting in dataframe form through the programming interface, the number of spatial points inside a circle. The circle is defined by its coordinates and radius in Km.

Listing 4.6: Count the number of spatial points in a circle

```
1  import gr.ds.unipi.noda.api.core.nosqldb.NoSqlDbOperators;
2  import gr.ds.unipi.noda.api.core.operators.filterOperators.geoperators.Coordinates;
3  import org.apache.spark.sql.Dataset;
4  import org.apache.spark.sql.Row;
5  import static gr.ds.unipi.noda.api.core.operators.AggregateOperators.*;
6  import static gr.ds.unipi.noda.api.core.operators.FilterOperators.*;
7
8  public class DataOperationsByUsingTheProgrammingInterface {
9      public static void main(NoSqlDbOperators noSqlDbOp) {
10
11         //NoSQL connection definition with spark session ...
12
13         NoSqlDbOperators noSqlDbOperators = noSqlDbSystem.operateOn("geoPoints");
14
15         Coordinates coordinates = Coordinates.newCoordinates(23.7533, 37.9801);
16
17         Dataset<Row> dataset = noSqlDbOperators
18                 .filter(inGeoCircleKm("location", coordinates, 1))
19                 .aggregate(count()).toDataframe();
20
21         //close connection and spark session ...
22     }
23 }
```

Listing 4.7 shows the instance of getting in dataframe form through the programming in-

terface, the number of GPS traces that were recorded inside a spatio-temporal box for each vehicle.

Listing 4.7: Find for every vehicle that passed from a spatio-temporal box, the number of GPS traces

```
import gr.ds.unipi.noda.api.core.nosqldb.NoSqlDbOperators;
import gr.ds.unipi.noda.api.core.operators.filterOperators.geoperators.Coordinates;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
import static gr.ds.unipi.noda.api.core.operators.AggregateOperators.count;
import static gr.ds.unipi.noda.api.core.operators.FilterOperators.inGeoTemporalRectangle;

public class DataOperationsByUsingTheProgrammingInterface {
    public static void main(String args[]) throws ParseException {

        //NoSQL connection definition with spark session ...

        NoSqlDbOperators noSqlDbOperators = noSqlDbSystem.operateOn("geoPoints");

        Coordinates lowerCoordinates = Coordinates.newCoordinates(23.7533, 37.9801);
        Coordinates upperCoordinates = Coordinates.newCoordinates(23.7598, 37.9910);

        SimpleDateFormat simpleDateFormat = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");

        Date lowerDate = simpleDateFormat.parse("1/3/2020 00:00:00");
        Date upperDate = simpleDateFormat.parse("1/3/2020 23:59:50");

        Dataset<Row> dataset = noSqlDbOperators
                .filter(inGeoTemporalRectangle("location", lowerCoordinates, upperCoordinates,
                "date", lowerDate, upperDate))
                .groupBy("vehicle_id").aggregate(count()).toDataframe();

        //close connection and spark session ...
    }
}
```

Listing 4.8 shows the instance of getting in dataframe form through the programming interface, the number of vehicles that passed from a spatio-temporal cylinder.

Listing 4.8: Find the number of vehicles that passed from a spatio-temporal cylinder

```
import gr.ds.unipi.noda.api.core.nosqldb.NoSqlDbOperators;
import gr.ds.unipi.noda.api.core.operators.filterOperators.geoperators.Coordinates;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
import static gr.ds.unipi.noda.api.core.operators.AggregateOperators.countDistinct;
import static gr.ds.unipi.noda.api.core.operators.FilterOperators.inGeoTemporalCircleKm;

public class DataOperationsByUsingTheProgrammingInterface {
    public static void main(String args[]) throws ParseException {

        //NoSQL connection definition with spark session ...

        NoSqlDbOperators noSqlDbOperators = noSqlDbSystem.operateOn("geoPoints");

        Coordinates coordinates = Coordinates.newCoordinates(23.7533, 37.9801);

        SimpleDateFormat simpleDateFormat = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");

        Date lowerDate = simpleDateFormat.parse("1/3/2020 00:00:00");
        Date upperDate = simpleDateFormat.parse("1/3/2020 23:59:50");

        Dataset<Row> dataset = noSqlDbOperators
                .filter(inGeoTemporalCircleKm("location", coordinates, 1,
                "date", lowerDate, upperDate))
                .aggregate(countDistinct("vehicle_id")).toDataframe();

        //close connection and spark session ...
    }
}
```

Listing 4.9 shows the instance of getting in dataframe form through the programming interface, the start, stop time and the number of the GPS traces of a specific trajectory.

Listing 4.9: Find for a specific trajectory, its start, stop time and its number of GPS traces

```
import gr.ds.unipi.noda.api.core.nosqldb.NoSqlDbOperators;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import static gr.ds.unipi.noda.api.core.operators.AggregateOperators.count;
import static gr.ds.unipi.noda.api.core.operators.AggregateOperators.max;
import static gr.ds.unipi.noda.api.core.operators.AggregateOperators.min;
import static gr.ds.unipi.noda.api.core.operators.FilterOperators.eq;

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

        //NoSQL connection definition with spark session ...

        NoSqlDbOperators noSqlDbOperators = noSqlDbSystem.operateOn("geoPoints");

        Dataset<Row> dataset = noSqlDbOperators
                .filter(eq("trajectory_id","65p19"))
                .aggregate(min("date"), max("date"),count()).toDataframe();

        //close connection and spark session ...
    }
}
```

Listing 4.10 shows the instance of getting in dataframe form through the programming interface, the id of the trajectories in descending order, after applying a nearest neighbors filter on the GPS traces.

Listing 4.10: Get at most the 20 nearest trajectories that passed from a point, in descending order

```
import gr.ds.unipi.noda.api.core.nosqldb.NoSqlDbOperators;
import gr.ds.unipi.noda.api.core.operators.filterOperators.geoperators.Coordinates;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import java.text.ParseException;
import static gr.ds.unipi.noda.api.core.operators.FilterOperators.geoNearestNeighbors;
import static gr.ds.unipi.noda.api.core.operators.SortOperators.desc;

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

        //NoSQL connection definition with spark session ...

        NoSqlDbOperators noSqlDbOperators = noSqlDbSystem.operateOn("geoPoints");

        Coordinates coordinates = Coordinates.newCoordinates(23.7533, 37.9801);

        Dataset<Row> dataset = noSqlDbOperators
                .filter(geoNearestNeighbors("location", coordinates, 20 ))
                .groupBy("trajectory_id").sort(desc("trajectory_id")).toDataframe();

        //close connection and spark session ...
    }
}
```

4.2 Use of the SQL Interface

Below are following 5 practical examples of using the SQL interface for accessing spatio-temporal data. The instances correspond to the examples of 4.1 subsection.

Listing 4.11 shows the instance of getting in Dataframe form through the SQL interface, the number of spatial points inside a circle. The circle is defined by its coordinates and radius in Km.

Listing 4.11: Count the number of spatial points in a circle

```
1  import gr.ds.unipi.noda.api.client.sql.NoSqlDbSqlStatement;
2  import org.apache.spark.sql.Dataset;
3  import org.apache.spark.sql.Row;
4
5  public class DataOperationsByUsingTheSqlInterface {
6      public static void main(String args[]) {
7
8          //NoSQL connection definition with spark session ...
9
10         String sqlStatement = "SELECT COUNT(*) FROM geoPoints " +
11         "WHERE GEO_CIRCLE_KM(location, (23.7533, 37.9801), 1)";
12
13         NoSqlDbSqlStatement noSqlDbStmt = noSqlDbSystem.sql(sqlStatement);
14
15         Dataset<Row> dataset = noSqlDbStmt.toDataframe();
16
17         //close connection and spark session ...
18     }
19 }
```

Listing 4.12 shows the instance of getting in Dataframe form through the SQL interface, the number of GPS traces that were recorded inside a spatio-temporal box for each vehicle.

Listing 4.12: Find for every vehicle that passed from a spatio-temporal box, the number of GPS traces

```
import gr.ds.unipi.noda.api.client.sql.NoSqlDbSqlStatement;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;

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

        //NoSQL connection definition with spark session ...

        String sqlStatement = "SELECT COUNT(*) FROM geoPoints " +
                "WHERE GEO_TEMPORAL_RECTANGLE(location, [(23.7533, 37.9801), " +
                "(23.7598, 37.9910)], date, '1/3/2020 00:00:00', '1/3/2020 23:59:50') " +
                "GROUP BY vehicle_id";

        NoSqlDbSqlStatement noSqlDbStmt = noSqlDbSystem.sql(sqlStatement);

        Dataset<Row> dataset = noSqlDbStmt.toDataframe();

        //close connection and spark session ...
    }
}
```

Listing 4.13 shows the instance of getting in Dataframe form through the SQL interface, the number of vehicles that passed from a spatio-temporal cylinder.

Listing 4.13: Find the number of vehicles that passed from a spatio-temporal cylinder

```
import gr.ds.unipi.noda.api.client.sql.NoSqlDbSqlStatement;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;

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

        //NoSQL connection definition with spark session ...

        String sqlStatement = "SELECT COUNT(DISTINCT vehicle_id) FROM geoPoints " +
                "WHERE GEO_TEMPORAL_CIRCLEKM(location, (23.7533, 37.9801), 1, " +
                "date, '1/3/2020 00:00:00', '1/3/2020 23:59:50')";

        NoSqlDbSqlStatement noSqlDbStmt = noSqlDbSystem.sql(sqlStatement);

        Dataset<Row> dataset = noSqlDbStmt.toDataframe();

        //close connection and spark session ...
    }
}
```

Listing 4.14 shows the instance of getting in Dataframe form through the SQL interface, the start, stop time and the number of the GPS traces of a specific trajectory.

Listing 4.14: Find for a specific trajectory, its start, stop time and its number of GPS traces

```
import gr.ds.unipi.noda.api.client.sql.NoSqlDbSqlStatement;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;

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

        //NoSQL connection definition with spark session ...

        String sqlStatement = "SELECT MIN(date), MAX(date), COUNT(*) FROM geoPoints " +
                "WHERE trajectory_id = '65p19'";

        NoSqlDbSqlStatement noSqlDbStmt = noSqlDbSystem.sql(sqlStatement);

        Dataset<Row> dataset = noSqlDbStmt.toDataframe();

        //close connection and spark session ...
    }
}
```

Listing 4.15 shows the instance of getting in dataframe form through the programming interface, the id of the trajectories in descending order, after applying a nearest neighbors filter on the GPS traces.

Listing 4.15: Get at most the 20 nearest trajectories that passed from a point, in descending order

```
import gr.ds.unipi.noda.api.client.sql.NoSqlDbSqlStatement;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;

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

        //NoSQL connection definition with spark session ...

        String sqlStatement = "SELECT trajectory_id FROM geoPoints " +
                "WHERE GEO_NEAREST_NEIGHBORS(location, (23.7533, 37.9801), 20) "+
                "GROUP BY trajectory_id ORDER BY trajectory_id DESC";

        NoSqlDbSqlStatement noSqlDbStmt = noSqlDbSystem.sql(sqlStatement);

        Dataset<Row> dataset = noSqlDbStmt.toDataframe();

        //close connection and spark session ...
    }
}
```

5 Technical Description of Complex Query Operators

As explained in the previous sections, NoDA offers a unified layer for data access operators to NoSQL stores, implemented over different NoSQL stores (MongoDB, HBase, Redis), as well as a declarative, SQL-like interface. Essentially, this comprises a set of primitive query operators that facilitate seamless data access to scalable NoSQL stores in a unified way.

However, there exist complex query operators that their functionality as a whole cannot be pushed-down to a NoSQL store in an efficient way. An example of such a complex query is the *distributed sub-trajectory join* (DTJ) query, which is a significant operation in mobility data analytics and the cornerstone of various methods that aim to extract knowledge out of mobility data.

In more detail, concrete applications from the Track&Know Pilots that can benefit from this work include: sub-trajectory clustering (reported in Deliverable D4.1), computing representative movements in Individual Mobility Networks (IMNs) (reported in Deliverable D4.2), trip matching for carpooling (Pilot 1), and delivery optimization (Pilots 2 and 3).

5.1 Distributed Sub-trajectory Join

The so-called trajectory join problem, aims to find all pairs of "similar" (i.e. nearby in space-time) trajectories in a dataset [6, 9, 13, 43]. An even more interesting and challenging problem is the sub-trajectory join query [4], where, for each pair of trajectories, we want to retrieve all the "portions" of trajectories that are "similar". However, the sub-trajectory join is a processing-intensive operation. Centralized algorithms do not scale with the size of today's trajectory data. Moreover, such a complex query operator cannot be pushed down as a whole to a NoSQL store in an efficient way, thus custom parallel and distributed algorithms are necessary in order to provide efficient processing of sub-trajectory join, an issue largely overlooked in the related research.

Several modern applications that manage trajectory data could benefit from such an operation. For instance, in the urban traffic domain, carpooling is becoming increasingly popular. More concretely, consider a mobile application which tries to match users that can share a ride based on their past movements. Here, given a set of trajectories we want to find all the pairs of users that can share a ride for a portion of their everyday routes without significantly deviating (spatially and temporally) from their daily routine (i.e. retrieve all pairs of maximal sub-trajectories that move close in space and time). Another interesting scenario concerns the identification of suspicious movement by a governmental security agency. For instance, given a set of trajectories that depict the movement of suspicious individuals, we would like to retrieve all the pairs of moving objects that move "close" to each other for more than a threshold (moving

together for small periods of time could be considered as coincidental) as candidates for illegal activity. Moreover, such a query is in fact the building block for a number of operations than aim to identify mobility patterns, such as co-movement patterns (e.g. flocks [25], convoys [27], swarms [29]). An even more challenging problem is that of sub-trajectory clustering [33, 1]. An interesting application scenario of sub-trajectory clustering is network discovery, where given a set of trajectories (e.g from the maritime or the aviation domain) we want to identify the underlying network of movement by grouping sub-trajectories that move "close" to each other and use cluster representatives/medoids as network edges. One of the main goals of sub-trajectory clustering is to segment trajectories to sub-trajectories. Finally, trajectory segmentation techniques [31, 33], can directly benefit from the sub-trajectory join query since their input, for each trajectory, is the number of objects that were located close to it at any given time. However, the bottleneck in all these applications is the underlying processing cost of the join operation, which calls for parallel and distributed solutions that scale beyond the limitations of a single machine.

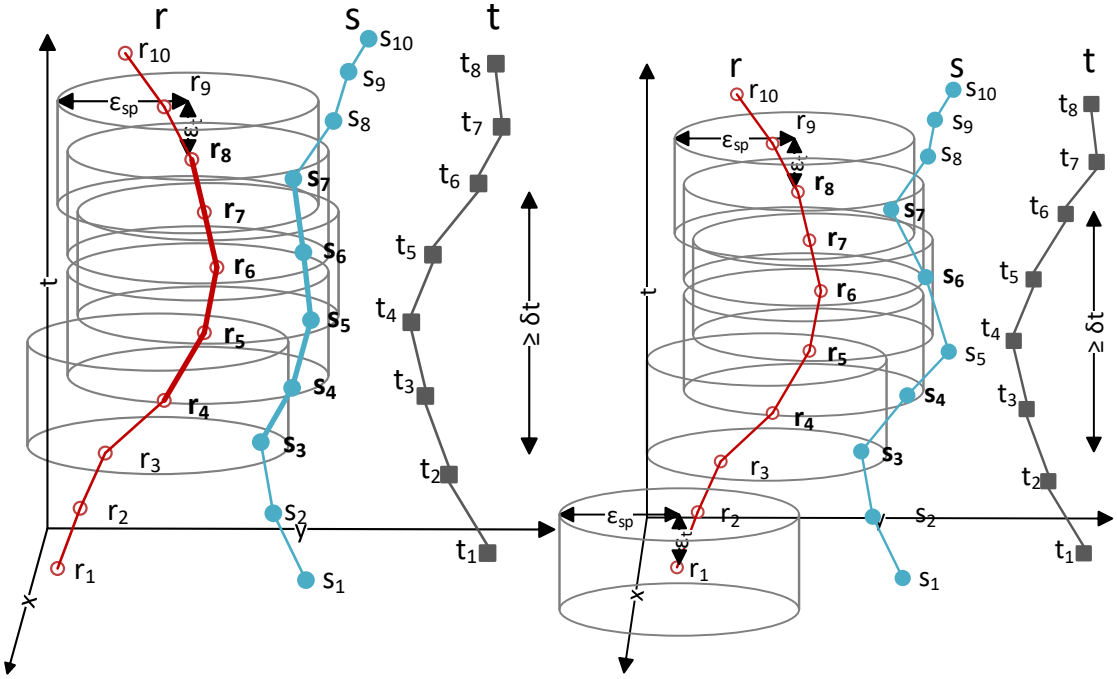

Figure 12: (a) A pair of maximally "matching" sub-trajectories and (b) a breaking point r_1 and a non-joining point s_5 w.r.t. r.

Inspired by the above application scenarios, the problem that we address in this Section is as follows: given two sets of trajectories (or a single set and its mirror in the case of self-join), identify all pairs of maximal "portions" of trajectories (or else, sub-trajectories) that move close in time and space w.r.t a spatial threshold ϵ_{sp} and a temporal tolerance ϵ_t, for at least some time duration δt. To illustrate this informal definition, as depicted in Figure 12(a), given two trajectories r and s, the pair of their maximal matching "portions" is $(\{r_4, r_5, r_6, r_7, r_8\}, \{s_3, s_4, s_5, s_6, s_7\})$. Each

point of a trajectory defines a spatio-temporal 'neighborhood' area around it, a cylinder of radius ϵ_{sp} and height ϵ_t. In order for a pair of sub-trajectories to be considered "matching", each point of a sub-trajectory must have at least one point of the other sub-trajectory in its "neighborhood", thus making the result symmetrical. A pair of matching sub-trajectories is maximal if there exists no superset of either sub-trajectories that can replace them and the pair still qualifies as a "matching" pair.

There have been some efforts to tackle variations of this problem in a centralized way [4, 6, 9]. However, these solutions discover pairs of entire trajectories and cannot identify matching sub-trajectories. In [5], all pairs of "matching" (w.r.t a spatial threshold) sub-trajectories of exactly δt duration are retrieved in contrast with the problem addressed in this Section, where the goal is to identify maximally "matching" sub-trajectories, which is vital for exploiting the output in subsequent steps, e.g. the mining operations mentioned above. Moreover, applying these centralized solutions to a parallel and distributed environment is not straightforward and is often impossible if radical changes to the methods/algorithms do not take place, since there are several non-trivial issues that arise. For instance, how to partition the data in such a way so that each partition can be processed independently and be of even size.

In a recent effort, in [37] the authors try to tackle the problem of trajectory similarity join in spatial networks in parallel. The solution proposed in [37] handles each trajectory separately and all the data have to be replicated for each trajectory and, consequently, to each node. Due to this fact, such a solution cannot scale to terabytes of data, thus making it inapplicable to Big Data. Furthermore, such an approach assumes that the underlying network is known in advance, hence it cannot support datasets of moving objects that move freely in space (e.g. from the maritime or the aviation domain). As a result, a scenario where the goal is to identify the underlying network cannot be supported. Finally, the output of [37] is pairs of trajectories and not sub-trajectories, which is significantly different than the problem addressed in this Section. More recently, in [38] the authors try to tackle the problem of trajectory similarity join. Specifically, given two sets of trajectories, a similarity function (e.g. DTW) and a similarity threshold, they aim to identify all pairs of trajectories that exceed this similarity threshold. Again, the problem addressed in [38] is to retrieve pairs of trajectories in contrast with the problem that we try to tackle in this Section, which is to retrieve all pairs of "maximally matching" sub-trajectories.

It is straightforward to claim that an integral part of any algorithm that tries to address the sub-trajectory join query is to identify all pairs of points that move "close enough" in time and space w.r.t a spatial threshold ϵ_{sp} and a temporal tolerance ϵ_t, e.g. r_4 and s_3 in Figure 12(a). In that sense, another line of research that is closely related to our problem is that of MapReduce-based spatial [49, 2, 16] and multidimensional joins [41, 30, 22], where the goal is to identify such points. A generic solution which could form the basis for any MapReduce-based spatial (or spatiotemporal) join algorithm is presented in [49], where the input data are partitioned into small, disjoint tiles at *Map* stage and get joined at the *Reduce* stage by performing a plane sweep algorithm along with a duplication avoidance technique. However, all of the above approaches try

to solve a problem that is significantly different from ours since our problem is not to join spatial or multidimensional objects but identify all pairs of "maximally matching" sub-trajectories.

In this Section, we provide efficient solutions for the *Distributed Sub-trajectory Join* processing problem, as it is formally defined in Section 5.2. To the best of our knowledge, this problem has not been addressed in the literature yet. Our main contributions are the following:

- We formally define the problem of *Distributed Sub-trajectory Join* processing, investigate its main properties, and discuss its main challenges.
- We present a well-designed algorithm, called *DTJb*, that solves the problem of *Distributed Sub-trajectory Join* processing by employing two MapReduce phases.
- We propose an improvement of *DTJb*, termed *DTJr*, which is equipped with a repartitioning mechanism that achieves load balancing and collocation of temporally adjacent data.
- To boost the performance of query processing even further, we introduce *DTJi*, which extends *DTJr* by exploiting an indexing scheme that speeds up the computation of the join.
- We compare with an appropriately modified state of the art MapReduce spatial join algorithm and show that our solution performs several times better.
- We study the performance of the proposed algorithms by using, to the best of our knowledge, the largest real trajectory dataset (56GB) used before in the relevant literature, thus demonstrating the scalability of our algorithms.

The rest of the Section is organized as follows. In Section 5.2 we introduce the problem. In Section 5.3, we present *DTJb*. Subsequently, in Section 5.4 we propose *DTJr* that utilizes a preprocessing step. In Section 5.5, we introduce *DTJi* that boosts the performance of the join processing. In Section 5.6, we provide our experimental study.

5.2 Problem Statement

Given a set R of trajectories moving in the xy-plane, a trajectory $r \in R$ is a sequence of timestamped locations $\{r_1, \ldots, r_N\}$. Each $r_i = (x_i, y_i, t_i)$ represents the i-th sampled point, $i \in 1, \ldots, N$ of trajectory r, where N denotes the length of r (i.e. the number of points it consists of). The pair (x_i, y_i) and t_i denote the 2D location in the xy-plane and the time coordinate of point r_i respectively. A sub-trajectory $r_{i,j}$ is a subsequence $\{r_i, \ldots, r_j\}$ of r which represents the movement of the object between t_i and t_j where $i < j$.

Given a pair (r, s) of trajectories (the same holds for sub-trajectories) with $r \in R$ and $s \in S$, the *common lifespan* $w_{r,s}$ is defined as the time interval $[max(r_1.t, s_1.t), min(r_N.t, s_M.t)]$, where

r_1 (s_1) is the first sample of r (s, respectively) and r_N (s_M) is the last sample of r (s, respectively). The duration of the common lifespan $w_{r,s}$ is $\Delta w_{r,s} = min(r_N.t, s_M.t)$ - $max(r_1.t, s_1.t)$

Further, let $DistS(r_i, s_j)$ denote the spatial distance between two points r_i, s_j, which is defined as the Euclidean distance, even though other distance functions are also applicable. Also, let $DistT(r_i, s_j)$ denote the temporal distance, defined as $|r_i.t - s_j.t|$.

Definition 1 ***(Matching sub-trajectories)*** *Given a spatial threshold ϵ_{sp}, a temporal tolerance ϵ_t and a time duration δt, a "match" between a pair of sub-trajectories (r', s') occurs iff $\Delta w_{r',s'} \geq \delta t - 2\epsilon_t$, and $\forall r'_i \in r'$ there exists at least one $s'_j \in s'$ so that $DistS(r'_i, s'_j) \leq \epsilon_{sp}$ and $DistT(r'_i, s'_j) \leq \epsilon_t$, and $\forall s'_j$ there exists at least one r'_i so that $DistS(s'_j, r'_i) \leq \epsilon_{sp}$ and $DistT(s'_j, r'_i) \leq \epsilon_t$.*

Definition 2 ***(Maximally matching sub-trajectories)*** *Given a pair of "matching" sub-trajectories (r', s') which belong to trajectories r, s respectively, this pair is considered a "maximal match" iff $\nexists$ superset r'' of r' or s'' of s' where the pair (r'', s') or (r', s'') or (r'', s'') would be "matching".*

At this point, we should clarify that two trajectories may have more than one "maximal matches" (i.e. pairs of sub-trajectories). Having provided the above background definitions, we can define the sub-trajectory join query between two sets of trajectories.

Definition 3 ***(sub-trajectory join)*** *Given two sets of trajectories R and S, a spatial threshold ϵ_{sp}, a temporal tolerance ϵ_t and a time duration δt, the sub-trajectory join query searches for all pairs (r', s'), $r' \in r \in R$ and $s' \in s \in S$, which are "maximally matching" sub-trajectories.*

5.2.1 A Closer Look at the Sub-trajectory Join Problem

An integral part of any algorithm addressing the sub-trajectory join query, as defined in Definition 3 above, is to identify all *pairs of joining points* (r_i,s_j), where $r_i \in r$ and $s_j \in s$, which satisfy the following property: $DistS(r_i, s_j) \leq \epsilon_{sp}$ and $DistT(r_i, s_j) \leq \epsilon_t$. In fact, the set of *joining points* is the outcome of the inner join $R \bowtie S$, where the evaluated join predicates are the ones mentioned above. However, as it will be explained next, these pairs of points do not suffice to return the correct query result.

Let $\mathcal{A}$ denote the class of correct algorithms for the sub-trajectory join problem. A naive algorithm $A \in \mathcal{A}$ would require the Cartesian product $R \times S$ to produce the correct result. We claim that $R \times S$ can be represented by three sets of points, the set of *joining points* (JP), the *breaking points* (BP) and the *non-joining points* (NJP). Formally, $R \times S = JP \cup BP \cup NJP$. The definitions of these sets follow, and the discussion is aided by Figure 12(b), which is a variation of Figure 12(a) in order to emphasize the distinction between JPs, BPs and $NJPs$.

The first set of points that needs to be identified, besides JP, contains all points $r_i \in r \forall r \in R$ that do not "match" with any other point in S. We call such points as *breaking points*.

Definition 4 ***(Breaking points)*** *A point $r_i \in r \in R$ is a breaking point iff it is not a joining point with any other point $s_j \in S$:*

$$\nexists s_j \in S\text{: } DistS(r_i, s_j) \leq \epsilon_{sp} \wedge DistT(r_i, s_j) \leq \epsilon_t.$$

As it will be shown later, the lack of information about BPs can make an algorithm $A \in \mathcal{A}$ to falsely identify a pair of sub-trajectories as "matching". The set of BP along with the set of JP is actually the outcome of the full outer join of R and S. Figure 12(b) depicts the case where r_1 is a breaking point of r (r_2, r_3 and r_{10} are also breaking points), since it does not "match" with any other point of any trajectory. Obviously, breaking points are never reported as part of the answer set and the portion of r that could possibly contribute to the result is sub-trajectory $r_{4,9}$.

The last set of points that is necessary to be identified consists of the pairs of points that do not "match", coined *non-joining points*, since some of them might indicate the start or the end of "maximally matching" sub-trajectories.

Definition 5 ***(Non-joining points)*** *A point $r_i \in r \in R$ is a non-joining point w.r.t. $s_j \in s \in S$ iff: (a) r_i and s_j are not breaking points, and (b) r_i is not a joining point with $s_j \in S$:*

$$r_i, s_j \notin BP \wedge (DistS(r_i, s_j) > \epsilon_{sp} \vee DistT(r_i, s_j) > \epsilon_t).$$

This case is illustrated in Figure 12(b), where r_5 is a non-joining point w.r.t. s_5.

Actually, if we remove condition (a) from Definition 5 then it is obvious that a *breaking point* r_i is a special case of *non-joining point* where $r_i \in R$ is a *non-joining point* with every other point $\in S$. However, we differentiate *breaking points* from *non-joining points* so as to reduce the amount of information that needs to be kept, i.e. instead of keeping multiple *non-joining points* we only keep one *breaking point*. In the section that follows, we investigate the theoretical properties of an efficient algorithm in class $\mathcal{A}$.

5.2.2 Properties of Sub-trajectory Join

In this section, we provide the theoretical properties for designing efficient algorithms for the sub-trajectory join problem. The properties shown below essentially determine which pairs of points from the sets BP and NJP are necessary for a correct algorithm in class $\mathcal{A}$.

Lemma 1 *The set of breaking points is necessary in order to produce the correct result set for the sub-trajectory Join problem.*

This result indicates that *breaking points* cannot be ignored by an algorithm, without compromising the correctness of the result. The remaining question is whether all *non-joining points* are also necessary. In the following, we define a subset of *non-joining points* points $sNJP \subseteq NJP$, and show that this subset is actually necessary.

Definition 6 ***(Subset $sNJP$ of non-joining points)*** *A non-joining point $s_j \in S$ w.r.t. a point $r_i \in R$ belongs to $sNJP$, if at least one of its adjacent points s_{j-1} or s_{j+1} is a joining*

point with any point $r_p \in r$*, with* $p \neq i$ *and* $\nexists$ *a point* r_q*, with* $q \neq i$*, such that* $DistT(r_q, s_j) \leq DistT(r_i, s_j)$.

Returning to the example of Figure 12(b), s_5 does not "match" with any point in r, even though all points of $r_{4,8}$ "match" with a point in $s_{3,7}$. Again, failure to identify pairs of points such as (s_5, r_6) would result in erroneously identifying larger "matching" sub-trajectories.

Lemma 2 *The set* $sNJP$ *of pairs of non-joining points is necessary in order to produce the correct result set for the sub-trajectory Join problem.*

In summary, our main finding is that a typical join algorithm that identifies only the set of JP is not enough in order to address the sub-trajectory join problem. Additionally to the set of JP, an algorithm needs to identify both the set of BP and the subset $sNJP$ during the join processing, in order to ensure correctness.

Distributed Sub-trajectory Join Given two sets R and S of trajectories, the typical approach for parallel join processing consists of two main phases: (a) *data repartitioning*, in order to create pairs of partitions $R_i \subset R$ and $S_j \subset S$, such that part of the join can be processed using only R_i and S_j, and (b) *join processing*, where a join algorithm is performed on partitions R_i and S_j.

Problem 1 ***(Distributed Sub-trajectory Join)*** *Given two distributed sets of trajectories,* $R = \cup R_i$ *and* $S = \cup S_j$*, compute the sub-trajectory join (Definitition. 3) in a parallel manner.*

In this setting, the main challenges are the following: (a) ensure that the created partitions are sufficient to produce parts of the total join without additional data, (b) generate even-sized partitions in order to balance the load fairly to multiple nodes, (c) handle the problem of potential duplicate existence in the join results, which may arise due to the way partitions are created, and (d) process the actual join on the partitions in an efficient way. The first challenge sets the foundations for *parallel processing*, as it identifies pairs of partitions that can be processed together, without any additional data, and produce a subset of the final join result. The second challenge is about *load balancing* and determines the efficiency of parallel processing, which is not straightforward, since processing uneven work units in parallel may lead to sub-optimal performance (as the slowest task will determine the query execution time). The third challenge, labeled *duplicate avoidance*, is about avoiding to generate duplicate results which typically occurs in parallel join processing. Finally, the fourth challenge, labeled *efficient join*, refers to the efficiency of the (centralized) algorithm used to join two partitions.

Clearly, solving the above problem is quite challenging in a distributed setting, as multiple challenges need to be addressed at the same time. In the following sections, we present a well designed solution solution to the *Distributed Sub-trajectory Join* problem along with two improved versions, following the popular MapReduce paradigm.

5.3 The Basic Sub-trajectory Join Algorithm

5.3.1 Preliminaries

One of the prevalent technologies for dealing with Big Data and offline analytics, is the MapReduce programming paradigm [12] and its open-source implementation Hadoop [40]. A lot of efforts have been made as far as it concerns join processing through this technology and a survey on limitations of MapReduce/Hadoop, also related to join processing, is conducted in [15]. In more detail, Hadoop is a distributed system created in order to process large volumes of data which are usually stored in the Hadoop Distributed File System (*HDFS*). In more detail, when running a MapReduce (MR) job, each *Mapper* processes (in parallel) an input split, which is a logical representation of data. An input split typically consists of a block of data (the default block size is 128MB) but it can be adjusted according to the users' needs by implementing a custom *FileInputFormat* along with the corresponding *FileSplitter* and *RecordReader*. Subsequently, for each record of the split the "map" function is applied. The output of the *Map* phase is sorted and grouped by the "key" and written to the local disk. Successively, the data is partitioned to *Reducers* based on a partitioning strategy (also known as *shuffling*), and each *Reducer* receives a partition (group) of data and applies the "reduce" function to the specific group. Finally, the output of the *Reduce* phase is written to *HDFS*.

5.3.2 The DTJb Algorithm

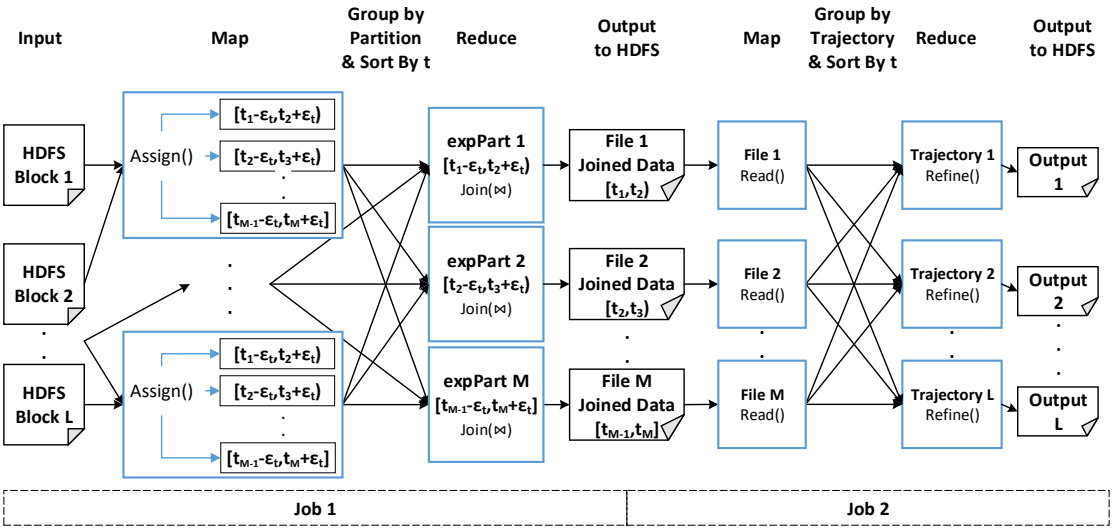

Figure 13: The DTJb algorithm in MapReduce.

Our first algorithm, named *DTJb*, consists of three phases: (a) the *Partitioning phase*, where input data is read and partitioned, (b) the *Join phase*, where the sets JP, BP and $sNJP$ are

identified in each partition, and (c) the *Refine phase*, where these sets are grouped by trajectory and sorted by time in order to identify all the pairs of "maximally matching" sub-trajectories[3].

Partitioning Phase The first challenge is how to partition the input data in order to satisfy the requirement for *parallel processing*. Partitioning the data into N disjoint temporal partitions $R = \cup_{i=1}^{N} part_i$, where R is the set of trajectories, cannot guarantee the correctness during parallel processing, due to the temporal tolerance parameter ϵ_t. Hence, we define a partitioning where each $part_i$ is expanded by ϵ_t, thus expanded partitions can be processed independently in parallel. Let $expPart_i$ denote such an expanded partition. Processing each $expPart_i$ individually guarantees correctness, but at the cost of having duplicates due to the point replication in temporally overlapping partitions. To address this *duplication avoidance* challenge, we supplement each point with a flag *partFlag* that indicates whether this point belongs to the original partition (i.e. not expanded by ϵ_t) or not.

Lemma 3 *An expanded partition $expPart_i$ is sufficient in order to produce the sets of JP and BP for $part_i$*

Unfortunately, an expanded partition $expPart_i$ is not sufficient in order to produce the set of $sNJP$ since, according to Definitition. 6, for each pair (r_j, s_k) that belongs to NJP we need to examine r_{j-1} and r_{j+1}, which may span to other partitions. However, the set of $sNJP$ can be identified at the *Refine* phase, where all the pairs concerning a trajectory are grouped together.

In more detail, we choose to partition the data into uniform temporal partitions, where for each pair of partitions $(part_i, part_j)$, with $i \neq j$ and $i, j \in [1, N]$, it holds that $DistT(t_e^{part_i}, t_s^{part_i}) = DistT(t_e^{part_j}, t_s^{part_j})$. Typically, the duration of a partition is larger than the maximum interval between two consecutive points of any trajectory. As illustrated in Figure 13, in the *Map* phase we access each data point and assign it to the expanded partition with which it intersects, essentially applying a temporal range partitioning. Then, the data is grouped by expanded partition, sorted by time and fed to the *Reduce* phase, where the *Join* procedure takes place.

Join Phase Figure 13 shows that each *Reducer* task takes as input an expanded partition and performs the *Join* operation. At this point, the duplication avoidance technique is applied, by employing the aforementioned flag and emitting only pairs where at least one point belongs to the original partition. The input of this phase is a set of tuples of the form $\langle t, x, y, trajID, partFlag \rangle$. The output of this MR job is a set of (a) JP, (b) BP and (c) candidate $sNJP$.

In more detail, we apply a plane sweep technique in order to perform the *Join*, by sweeping the temporal dimension. We choose to employ such a technique due to the fact that is much more efficient than a nested-loop join approach, since our data already arrive sorted by the temporal dimension, as illustrated in Figure 13. A typical plane sweep algorithm would emit only the set of JP, which is not enough in our case. For this reason, we devised and implemented a modified

[3]For the sake of simplicity, from now on, we are going to consider the case of self-join. The transition to the problem of joining two relations is straightforward.

Algorithm 1: Join($expPart, \epsilon_{sp}, \epsilon_t$)

```
1: Input: An expPart, ε_sp, ε_t
2: Output: All pairs of JP, BP and candidate sNJP
3: for each point i ∈ expPart do
4:    D[i] ← point
5:    TRJPlaneSweep(D[], ε_sp, ε_t)
6: TreatLastTrPoints()
7: for each point j ∈ BP[] do
8:    output((BP[j], null), True)
```

Algorithm 2: TRJPlaneSweep($D[], \epsilon_{sp}, \epsilon_t$)

```
1: Input: D[], ε_sp, ε_t
2: Output: All pairs of JP, BP and candidate sNJP
3: if D[i].partFlag=True then
4:    for each element D[j] ∈ [D[i].t − ε_t, D[i].t] do
5:       if DistS(D[i], D[j]) ≤ ε_sp then
6:          output((D[i], D[j]), True)
7:          remove D[i] from BP[]
8:          if D[j].partFlag=True then
9:             output((D[j], D[i]), True)
10:            remove D[j] from BP[]
11:         k ← getPrevTrPoint(j, D[])
12:         if FindMatch(D[], i, k, ε_sp, ε_t) = False then
13:            output((D[i], D[k]), False)
14:         k ← getPrevTrPoint(i, D[])
15:         if FindMatch(D[], j, k, ε_sp, ε_t)= False then
16:            if D[j].partFlag=True then
17:               output((D[j], D[k]), False)
18:   if there is no "match" for D[i] then
19:      BP[] ← D[i]
```

plane sweep technique, named *TRJPlaneSweep*, depicted in Figure 14, which also reports the sets of BP and candidate $sNJP$.

Algorithm 1 presents how the *Join* processing is performed. Each accessed point is inserted to an array D, which contains points sorted in increasing time. After point insertion, (Algorithm 2 is invoked for the currently accessed point (say $D[i]$) if $D[i]$ belongs to the original partition. *TRJPlaneSweep* examines the previously accessed points for the previous ϵ_t window (line 4). The role of this function is threefold. First, it identifies *joining points* with $D[i]$, e.g., point $D[j]$, and emits them in the form $((D[i], D[j]), True)$ (lines 5-10). Depending on the outcome of the duplicate avoidance technique, pairs $((D[j], D[i]), True)$ are also output. Second, it discovers points that belong to the candidate $sNJP$ set by examining whether the previous trajectory point (*getPrevTrPoint*)) of $D[j]$ (and $D[i]$), say $D[k]$, is a NJP (*FindMatch*) with each point $\in D[i].trajID$ ($D[j].trajID$, respectively) (lines 11-17). In case such points are identified, they

are output with a different flag $((D[i], D[k]), False)$ to differentiate them from JP. Third, it discovers the points that belong to BP. In more detail, in lines 18–19, a *breaking point* $D[i]$ is added to the breaking points set BP and in lines 7 and 10 is removed if a point has a "match". The remaining points in BP are reported as breaking points, using the following form: $((D[i], null), True)$ (Algorithm 1 lines 7–8).

By examining only the previous point of a JP in a trajectory, we might not examine a possible temporary adjacent point that might lie after the last JP of a trajectory in each partition. For this reason, we post-process the last JPs in order to check for candidate $sNJP$s by invoking the *TreatLastTrPoints* function (Algorithm 1 line 6).

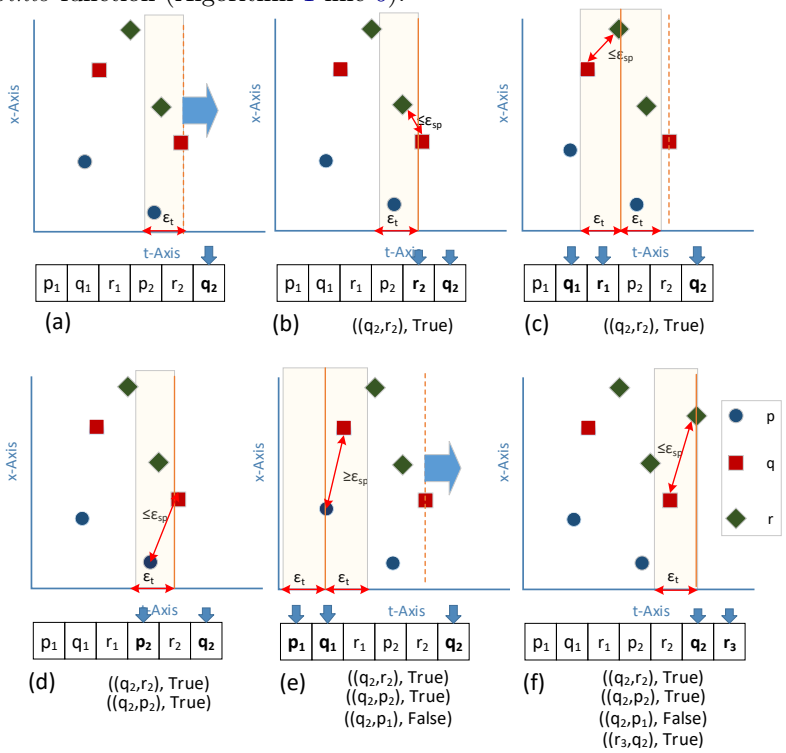

Figure 14: Join phase - The TRJPlaneSweep algorithm.

Example 1 *As illustrated in Figure 14(a), we suppose that the current point inserted into D is q_2. In Figure 14(b), assuming that $DistS(q_2, r_2) \leq \epsilon_{sp}$, we get a "match" and pair $((q_2, r_2), True)$ is reported (the symmetric pairs are omitted for simplicity). Subsequently, we need to find the previous point of r and in order to achieve this we should traverse our data backwards until we find it, as presented in Figure 14(c). When we find r_1, we need to check whether it is a NJP for each point $\in q$, as illustrated in Figure 14(c). If there exists a point $\in q$ that "matches", in our case q_1, nothing is reported and we proceed to examine whether q_2 and p_2 are JPs. If $DistS(q_2, p_2) \leq \epsilon_{sp}$ then we output the pair $((q_2, p_2), True)$, as shown in Figure 14(d). Subsequently, we need to find p_1 and check whether it is NJP for each point $\in q$. As depicted*

in Figure 14(e) there is no "match" between p_1 and any of the points of q. For this reason, we report the pair $((q_2, p_1), False)$. The same procedure is continued to the next point inserted to memory as delineated in Figure 14(f) until there are no more points inserted.

The complexity of the *Join* procedure is $O(|D| \cdot a \cdot ((1-b) \cdot |D| + b \cdot |D| \cdot (2 \cdot (L + 2 \cdot a \cdot |D|))))$, where $|D|$ is the number of points, a is the selectivity of ϵ_t and b is the selectivity of ϵ_{sp}. L is the number of points that have to be traversed in order to find the previous point of a specific trajectory. It is obvious that when a tends to reach 1 the complexity tends to reach $O(|D|^2)$. In the worst case, the complexity can be analogous to $O(|D|^3)$, when both a and b tend to reach 1. However, for a typical analysis task ϵ_t and ϵ_{sp} are much smaller than the dataset duration and the dataset diameter respectively. Roughly, we can say that the complexity is $O(a \cdot b \cdot |D|^2)$.

Refine Phase The output of the *Join* phase is actually pairs of points. From now on, let us refer to the left point of such a pair as *reference point* and the trajectory that it belongs to, *reference trajectory*. The *Refine* phase consists of a second MR job that reads the output of the *Join* step and groups points by the *reference trajectory*. Each *Reduce* task receives all pairs of points belonging to a specific trajectory, sorted first by the *reference point's* time and the by the *non-reference trajectory* ID. Figure 15 shows an example where the output pairs of points from the *Join* step are grouped, sorted and fed as input to three *Reduce* tasks (for trajectories p, q, and r respectively). The general idea here is to scan the set of JP in a sliding window fashion so as to identify "maximally" matching sub-trajectories and at the same time "consult" the sets of BP and $sNJP$ in order to avoid false identifications, as described in Section 5.2.2.

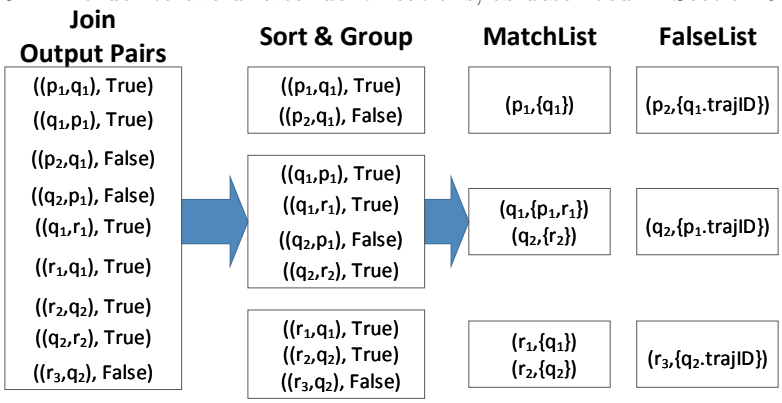

Figure 15: Output of *Join* and input of *Refine* phase.

Hence, each *Reducer* accesses all the pairs of a *reference trajectory* (say p) sorted by time, i.e., $\{p_1, p_2, \ldots, p_n\}$. Algorithm 3 describes the pseudo-code of the *Refine* phase which aims to identify all the "maximally matching" pairs of sub-trajectories of p with other sub-trajectories of any trajectory x ($x \neq p$). For each accessed pair $((p_i, x_j)$,*flag*), the algorithm assigns it in one of the two structures that it maintains: the *MatchList* and the *FalseList*. All JP and BP will be kept in the *MatchList*, whereas the candidate $sNJP$ is kept in the *FalseList* (lines 10–13).

Algorithm 3: Refine(δt, ϵ_t)

1: **Input:** Pairs of points $((p_i, x_j)$,*flag*) for a given trajectory p, sorted by time
2: **Output:** Result of *Distributed Sub-trajectory Join* for p
3: **for** each pair of points $((p_i, x_j)$,*flag*) **do**
4: **if** (p_i is encountered for the first time) **then**
5: **if** DistT(*MatchList*.lastEntry, *MatchList*.firstEntry) $\geq \delta t$ **then**
6: *resultT* $\leftarrow$ intersect lists in *MatchList* and exclude *FalseList*
7: *resultF* $\leftarrow$ apply sliding window of δt to *resultT*
8: *resultFinal* $\leftarrow$ *resultFinal* $\bigcup$ *resultF*
9: remove *MatchList*.firstEntry
10: **if** (*flag* = True) **then**
11: addToMatchList(p_i, x_j)
12: **else**
13: addToFalseList(p_i, x_j)
14: output(*resultFinal*)

Again, this is more clearly depicted in the example of Figure 15. Also, notice that for each *reference point* in the *MatchList*, we maintain a list of points sorted by trajectory ID.

Lemma 4 *The set of candidate sNJP is sufficient so as to identify the set of sNJP at the* Refine *phase.*

The algorithm proceeds as follows: as soon as all pairs of points of a specific *reference point* p_i have been accessed, it initiates processing on the *MatchList*. The processing takes place only if the first and last point of p in *MatchList* have temporal distance greater than or equal to δt (line 5). The processing essentially identifies points of other trajectories that join with points of p in the whole temporal window. This is performed by intersecting the lists in *MatchList* and excluding points existing in the *FalseList* (line 6). List intersection is efficiently performed in linear time to the length of the lists, since the lists are sorted by trajectory ID. Figure 16 depicts the result of this processing as *resultT*.

Subsequently, the points in *resultT* are processed as follows. We start from the first point and take into consideration all points with temporal distance at most $\delta t - 2\epsilon_t$ from the first point. From this set of points, we derive the sub-trajectories that "match" for the entire $\delta t - 2\epsilon_t$ window, and insert them in *resultF* (line 7). The temporary results of the *resultF* structure are added to the final result structure *resultFinal*, if not already contained in it (line 8). Then, a new set of points is considered, of temporal distance at most $\delta t - 2\epsilon_t$ from the second point of *resultT* and the process is repeated, similarly to a sliding a window of duration $\delta t - 2\epsilon_t$ on *resultT*. In the end, the first entry of the *MatchList* $(p_1, \{q_1, r_1, s_1\})$ is removed (line 9), as all potential results containing p_1 have already been produced. The algorithm terminates when the entire trajectory is traversed, the *resultFinal* is returned and each element of this list is emitted.

Example 2 *Figure 16 presents a working example of the Refine algorithm given the specific* MatchList *and* FalseList *of trajectory p.Assuming that* $DistT(p_1.t, p_7.t) \geq \delta t$*, we intersect all*

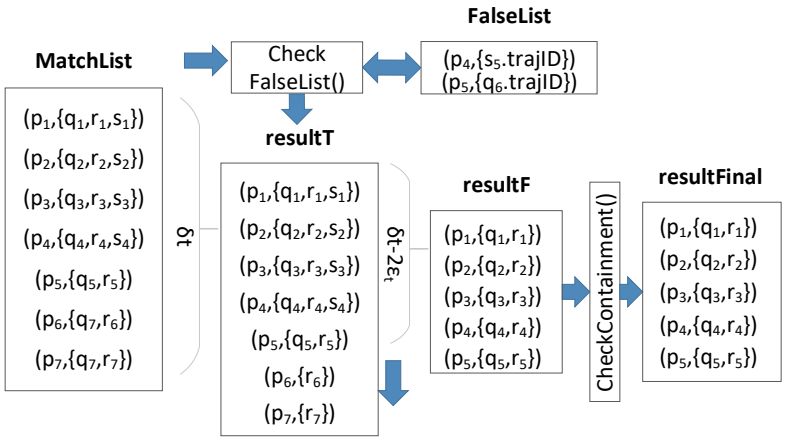

Figure 16: Refine procedure.

the lists contained in the specific window of the MatchList *and we pass the result to* resultT. *In this way, the list of the last entry of* resultT *will contain only the points that belong to the sub-trajectories that move "close" enough with p for the whole δt window. During list intersection, we take into account the* FalseList *structure in order to deal with points that belong to sNJP. Specifically, even though for each* p_i, *with* $i \in [1,7] \exists$ *a "match" with q, however* q_6 *has no "match" with p, as depicted in the* FalseList. *For this reason, q should be excluded from* resultT *after* p_5. *Then, a sliding* $\delta t - 2\epsilon_t$ *window is created that traverses* resultT, *and for each such window we intersect all lists and the result is stored in* resultF. *For the first* $\delta t - 2\epsilon_t$ *window, as depicted in Figure 16, sub-trajectories* $r_{1,5}$ *and* $q_{1,5}$ *are identified. The reason for this is to discover the sub-trajectories that move "close" enough, with p for the whole* $\delta t - 2\epsilon_t$ *window. Subsequently, before proceeding to the next δt window, the contents of* resultF *are inserted to the final result, if not already contained.*

The complexity of the *Refine* procedure is $O(T \cdot SW \cdot dt \cdot l)$, where T is the average number of points in a trajectory, SW is the number of points contained in the δt window, dt is the number of points contained in the $\delta t - 2\epsilon_t$ window and l is the size of the list. The complexity, here, clearly depends on the average number of points per trajectory, the ϵ_t and δt parameter, and the number of pairs emitted by the join phase which in turn depends on ϵ_t and ϵ_{sp}.

5.4 Sub-trajectory Join with Repartitioning

Even though the *DTJb* algorithm provides a correct solution to the *Distributed Sub-trajectory Join* problem, it has some limitations. In particular, it does not address the *load balancing* challenge, since it does not handle the case of temporally skewed data. Also, due to the two

chained MR jobs, the intermediate output of the first job is written to HDFS and must be read again by the second job, which imposes a significant overhead as its size is comparable and can be even bigger than the original dataset.

Motivated by these limitations, we propose an improved two-step algorithm ($DTJr$), which consists of the *repartitioning* and the *query* step. Each step is implemented as a MR job. However, the repartitioning step is considered a preprocessing step, since it is performed once and is independent of the actual parameters of our problem, namely ϵ_{sp}, ϵ_t, and δt.

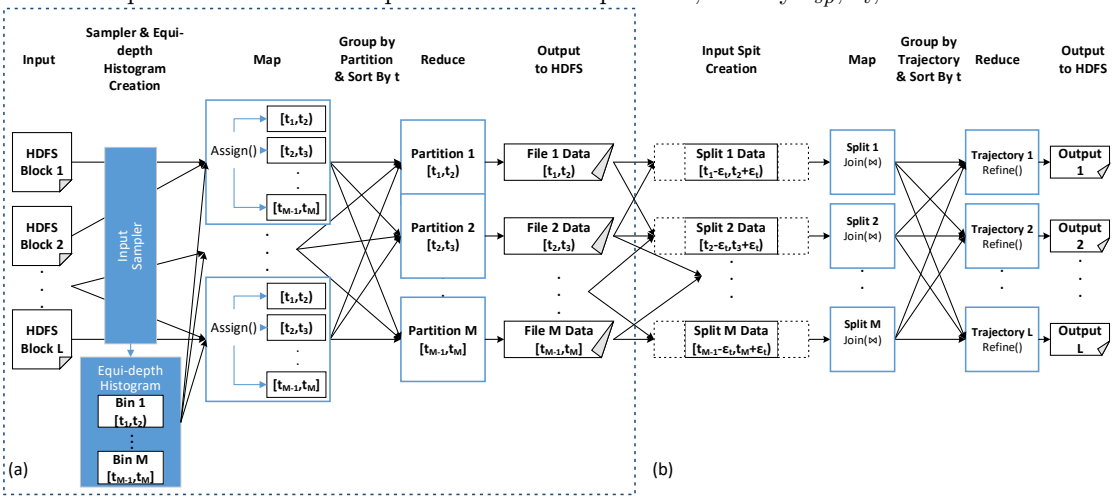

Figure 17: The DTJb algorithm in MapReduce: (a) Repartitioning step and (b) Query step.

5.4.1 Repartitioning

The aim of the repartitioning step is to split the input dataset in M equi-sized, temporally-sorted partitions (files), which are going to be used as input for the join algorithm. This is essential for two reasons: (a) it will provide the basis for load balancing, by addressing the issue of temporal skewness in the input data, and (b) it will result in temporal collocation of data, thus drastically reducing processing and network communication costs.

The repartitioning step is performed by means of a MR job as follows. We sample the input data, using Hadoop's *InputSampler*, and construct an equi-depth histogram on the temporal dimension. The histogram contains M equi-sized bins, i.e. the numbers of points in any two bins are equal, where the borders of each bin correspond to a temporal interval $[t_i, t_j)$.

The equi-depth histogram is exploited by the *Map* phase in order to assign each incoming data object in the corresponding histogram bin, based on the value of its temporal dimension. Each "map" function outputs each data object using as key a value $[1, M]$ that corresponds to the bin that the object belongs to. During shuffling, all data objects that belong to a specific bin are going to be sorted in time and will be collected by a single "reduce" function (thus having M "reduce" functions). As a result, each "reduce" function writes an output file to *HDFS* that contains all data objects in a specific temporal interval $[t_i, t_j)$ sorted by increasing time. A

graphical view of the MR job is provided in Figure 17(a).

A subtle issue is how to determine the number M of bins (and, consequently, output files). A small value of M, smaller than the number of nodes in the cluster, would be opposed to the collocation property because data would have to be transferred through the network. On the other hand, a large value of M would result to many small files, smaller than the *HDFS* block size, and would lead to inefficient use of resources as well as increasing the management cost of these *HDFS* files. A good compromise is to have files of equal size to the *HDFS* block. Hence, the number of files can be calculated as $M = \lceil \frac{InputTotalSize}{hdfsblocksize} \rceil$. Collocation can be further improved by placing temporally adjacent files to the same nodes. This can be achieved by grouping together k consecutive files, where $k = \lceil \frac{M}{N} \rceil$, with N being the number of nodes, and using their group id as the partition key.

5.4.2 The DTJr Algorithm

In order to minimize the I/O cost, the MR job that implements the proposed algorithm performs the *Join* procedure in the *Map* phase, and the *Refine* in the *Reduce* phase. To achieve this, we need to provide to a *Map* task as input, a data partition that contains all necessary data in order to perform part of the *Join* procedure *independently* from other *Map* tasks. Thus, an *HDFS* block produced by the repartitioning phase is expanded with additional points that exist at time (+/-)ϵ_t, and this is the process of *InputSplits* creation. In this way, points are duplicated to other *HDFS* blocks, which means that the same point may be output by two different *Map* tasks. To avoid this pitfall, a different *duplicate avoidance mechanism* is introduced which practically determines that a point is going to be output only by a single *Map* task; the *Map* task processing the *HDFS* block where the point belongs to.

As already mentioned, each data partition (*InputSplit*) that is fed to a *Map* task should contain all the data needed to perform the join of points for the specific partition, i.e. data for the period $[t_s^{part} - \epsilon_t, t_s^{part} + \epsilon_t]$. However, an output file produced by the repartitioning step is not sufficient due to the temporal tolerance ϵ_t, thus we need to augment these output files with extra data points, so that they form independent data partitions. At technical level, we devised and implemented a new *FileInputFormat* called *BloatFileInputFormat*, along with the corresponding *FileSplitter* and *RecordReader*, which selectively combines different files in order to create splits that carry all the necessary data points. Furthermore, during the creation of input splits we augment (as metadata) each split with the starting and ending time of the original partition of each split, termed t_s^{base} and t_e^{base}. The utility is to provide us with a simple way to perform duplicate avoidance at the *Join* phase.

Figure 17(b) shows that each *Map* task takes as input a split and performs the join at the level of point for a specific data partition. The input of this phase is a set of tuples of the form $\langle t, x, y, trajID \rangle$ sorted in ascending time t order. Since the data are already sorted w.r.t. the temporal dimension, we can apply the *Join* procedure, presented in Section 5.3.2. The output of the *Map* phase will be the JP, BP and $sNJP$ sets. Finally, the *Refine* procedure presented

in Section 5.3.2 can be performed at the *Reduce* phase.

5.5 Index-based Sub-trajectory Join with Repartitioning

The *Join* step of the previous algorithms is common and operates on the array D that contains temporally sorted points. However, it can be improved in two ways. First, by employing spatial filtering in order to avoid attempting to join points that are far away. Second, by having an index structure that given a point p_i can efficiently locate the (temporally) previous point p_{i-1} of p. Motivated by these observations, we devised and implemented an indexing scheme in order to speed up the processing of the join.

5.5.1 Indexing Scheme

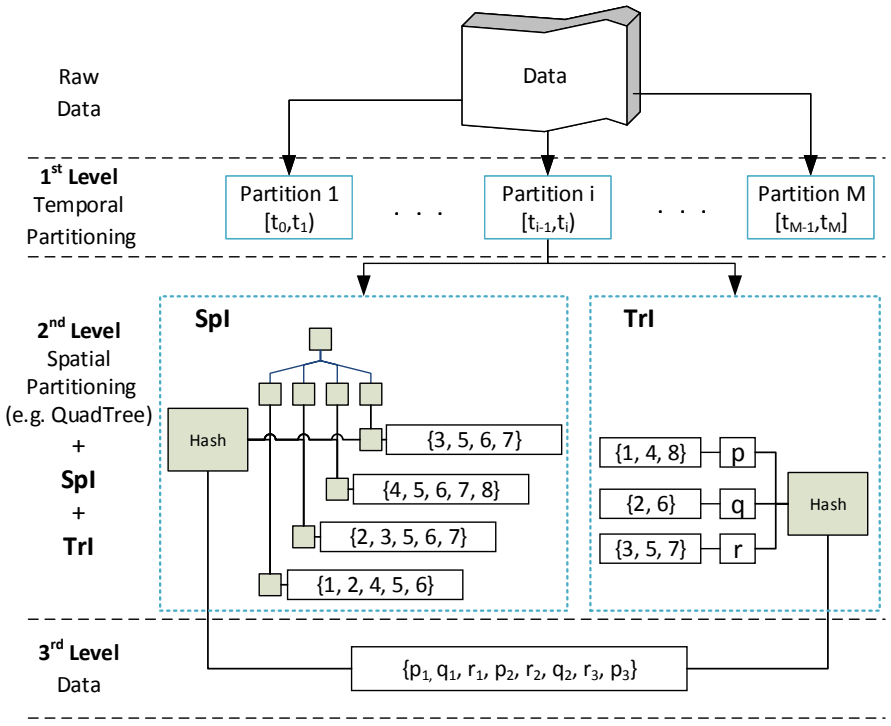

Figure 18: Indexing Scheme of $DTJi$ algorithm

As illustrated in Figure 18, this scheme consists of 3 levels. We already covered the first level in Section 5.4.1, where the initial data are partitioned to equi-sized temporal partitions (Section 5.4). At the second level, we partition the space. In order to have load balanced partitions we utilize the spatial partitioning provided by QuadTrees. More specifically, an "empty" QuadTree is created once, by sampling the original data, as in [16], and is written to *HDFS*. It is important to mention

here that the QuadTree contains only the spatial partitions and not the actual points. Then, when a new query is posed, the QuadTree is loaded into Hadoop's *distributed cache* in order to be accessible by all the nodes. Moreover, at the same level, we employ two indexes. The first index is a spatial index (SpI) which enables pruning of points based on their spatial distance, thus decreasing significantly the number of points that need to be examined within the ϵ_t window. The second index is an index that keeps track of the representation of each individual trajectory within the temporally sorted structure D (TrI), thus providing an efficient way to access the previous trajectory point. The two indexes are created gradually, as the data are read from HDFS. Finally, at the third level, we have the temporally sorted data that correspond to the specific temporal partition.

Spatial Index (SpI) The spatial index, called SpI, utilizes a given space partitioning, in our case QuadTrees. For each spatial partition of the QuadTree, SpI keeps a temporally sorted array where each entry is the position of a point that is contained in the given partition expanded by ϵ_{sp}. SpI is implemented as a HashMap with key the partition id and value the sorted array. Thus, a partition can be accessed in O(1), while a point in a partition can be accessed in $O(logP_i)$, where P_i here is the number of points in the corresponding sorted array. The construction of SpI has $O(|D| \cdot h)$ complexity, where $|D|$ is the number of points in the specific temporal partition and h is the height of the QuadTree, since for each point we need to traverse the QuadTree in order to find out in which expanded partition it is contained. Note that each point is enriched with the id of its original (i.e. not expanded) spatial partition, thus consisting of $\langle trajID, x, y, t, PartitionID \rangle$.

Trajectory Index (TrI) The TrI index keeps track of each individual trajectory within D. TrI is also implemented as a HashMap with key the trajectory id. For each trajectory, the value is a temporally sorted array, where each entry corresponds to a point of a trajectory, and the value of the entry is an integer indicating the point's position in D. Thus, a trajectory point can be efficiently accessed in $O(logT)$, where T is the number of points of a trajectory. To exemplify, the first element of the array holds the position of the first point of the trajectory inside D and so on. The construction of this index has $O(T)$ time complexity since the data is already sorted in time.

5.5.2 The DTJi Algorithm

Having these two indexes at hand we can utilize them in order to perform the join operation in an efficient way. Algorithm 4, presents the index-enhanced plane sweep procedure. Initially, the QuadTree is loaded into memory from the *distributed cache* (line 3) and then, each accessed point is inserted not only to an array D, which contains points sorted in increasing time, but also to the SpI and TrI indexes. Finally, the TRJPlaneSweep^I() algorithm is invoked for each accessed point (lines 4–7).

Algorithm 4: Join^I (Split, ϵ_{sp}, ϵ_t, t_s^{base}, t_e^{base})

1: **Input:** A split, ϵ_{sp}, ϵ_t, t_s^{base}, t_e^{base}
2: **Output:** All pairs of JP, BP and candidate $sNJP$
3: $QT \leftarrow LoadQuadTree()$
4: **for** each *point* $i \in Split$ **do**
5: **if** $point.t \in [t_s^{base} - \epsilon_t, t_e^{base} + \epsilon_t]$ **then**
6: $D[i], TrI, SpI \leftarrow point$
7: TRJPlaneSweep$^I(D[], TrI, SpI, \epsilon_{sp}, \epsilon_t, t_s^{base}, t_e^{base})$
8: TreatLastTrPoints()
9: **for** each *point* $j \in BP[]$ **do**
10: output($(BP[j], null)$, True)

Algorithm 5, presents the TRJPlaneSweep^I() algorithm. Here, given a point $p_i \in p$, instead of scanning the whole ϵ_t window before it, in order to find "matches", we perform a search in SpI and get only the points that belong to the same partition as p_i by invoking the getCandidatePoint() method (line 4). The partition id is retrieved in $O(1)$ and then binary search is performed in the temporally sorted list of points in order to find the position of p_i inside it. Having that, we can get the previous element, which will be the previous point in time that lies within the same partition, and check if the temporal and spatial constraint are satisfied. If they are satisfied, we have a "match", we proceed to the previous element of SpI and so on and so forth. Assuming that we have a "match" with q_j that belongs to trajectory q we need to find the previous point of q. This is achieved by invoking getPrevTrPoint^I, which performs a search in TrI in order to retrieve in O(1) the entry of q (lines 11, 14). Then, by performing binary search in the temporally sorted list, we can find the position of q_j and can easily get q_{j-1}. Having that, we need to find if it "matches" with any point that belongs to p. Here, instead of scanning the whole $2\epsilon_t$ window of q_{j-1} in order to check for "matches" with p, we perform a search in TrI in order to get the points of p that exist "close" to the time of q_{j-1} (lines 12, 15). Then, if the spatial and temporal constraints are satisfied we have a "match" and the *FindMatch^I()* method returns True. Otherwise, the whole procedure continues, until the temporal constraint is not satisfied anymore.

The complexity of the index-based solution is $O(|D| \cdot h \cdot (log_2 P_i \cdot a \cdot P_i((1-b) \cdot P_i + b \cdot P_i \cdot (2 \cdot (log_2 T + (log_2 T + a \cdot T)))))))$, with $|D|$ being the number of points, h the height of the QuadTree, a and b the selectivity of ϵ_t and ϵ_{sp} respectively. P_i is the number of points within the i-th partition expanded by ϵ_{sp}, where $P_i \ll |D|$, and T is the number points per trajectory. In the worst case, where a and b tend to 1, the complexity can reach $O(|D| \cdot log_2 P_i \cdot P_i^2)$. However, again this only occurs for values of ϵ_t and ϵ_{sp} that are comparable to the dataset's duration and diameter respectively. Roughly speaking, the complexity drops to $O(|D| \cdot (log_2 P_i \cdot a \cdot b \cdot P_i^2))$, which clearly shows the benefit attained when employing the proposed indexing scheme.

Algorithm 5: TRJPlaneSweep$^I(D[], TrI, SpI, \epsilon_{sp}, \epsilon_t, t_s^{base}, t_e^{base})$

```
1: Input: D[], ε_sp, ε_t, t_s^base, t_e^base
2: Output: All pairs of JP, BP and candidate sNJP
3: if DuplCheck(D[i].t, t_s^base, t_e^base)=True then
4:   for each element D[j] returned by getCandidatePoint(i, SpI, D[]) do
5:     if DistS(D[i], D[j]) ≤ ε_sp then
6:       output((D[i], D[j]), True)
7:       remove D[i] from BP[]
8:       if DuplCheck(D[j].t, t_s^base, t_e^base)=True then
9:         output((D[j], D[i]), True)
10:        remove D[j] from BP[]
11:      k ← getPrevTrPoint^I(j, D[], TrI);
12:      if FindMatch^I(D[], i, k, ε_sp, ε_t, TrI)= False then
13:        output((D[i], D[k]), False)
14:      k ← getPrevTrPoint^I(i, D[], TrI);
15:      if FindMatch^I(D[], j, k, ε_sp, ε_t, TrI) = False then
16:        if DuplCheck(D[j].t, t_s^base, t_e^base)=True then
17:          output((D[j], D[k]), False)
18:  if there is no "match" for D[i] then
19:    BP[] ← D[i]
```

5.6 Experimental Study

In this section, we provide our experimental study on the comparative performance of the three variations of our solution, namely (1) *DTJb* that uses two MR jobs (Section 5.3), (2) *DTJr* that employs repartitioning and a single job to perform the join (Section 5.4), and (3) *DTJi* that additionally uses the SpI and TrI indexes for more efficient join processing (Section 5.5). Furthermore, we compare our solution with the work presented in [49].

The experiments were conducted in a 49 node Hadoop 2.7.2 cluster, provided by *okeanos*[4], an IAAS service for the Greek Research and Academic Community. The master node consists of 8 CPU cores, 8 GB of RAM and 60 GB of HDD while each slave node is comprised of 4 CPU cores, 4 GB of RAM and 60 GB of HDD. Our configuration enables each slave node to launch 4 containers, thus resulting that at a given time the cluster can run up to 192 jobs (*Map* or *Reduce*).

For our experimental study, we employed two real datasets from two different domains (urban and the maritime). In more detail, the first one (named SIS) is a 27GB proprietary insurance dataset of moving objects around Rome and Tuscany area, that contains approximately 2.2×10^7 trajectories that correspond to 7.2×10^8 points. This dataset belongs to Pilot 1 of the Track&Know project. The second one, coined IMIS[5], consists of 699,031 trajectories of ships

[4] https://okeanos.grnet.gr/home/

[5] The IMIS dataset has been kindly provided by IMIS Hellas for research and educational purposes. It is available for downloading at http://chorochronos.datastories.org

moving in the Eastern Mediterranean for a period of 3 years. This dataset contains approximately 1.5 billion records, 56GB in total size.

Our experimental methodology is as follows: Initially, we verify the scalability of our algorithms by varying (a) the dataset size, and (b) the number of cluster nodes (Section 5.6.1). Finally, we compare our solution with the work presented in [49] (Section 5.6.2).

5.6.1 Scalability

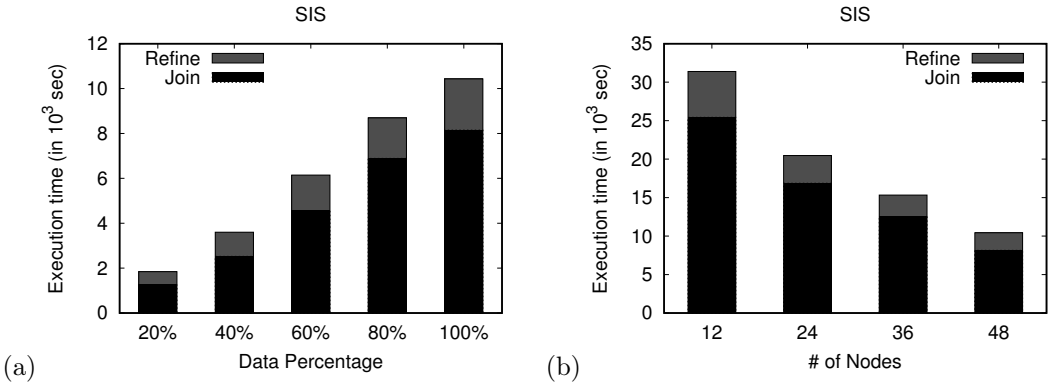

Figure 19: Scalability analysis varying (a) the size of the dataset and (b) the number of nodes

Initially, we vary the size of our dataset and measure the execution time of our algorithm. To study the effect of dataset size, we created 4 portions (20%, 40%, 60%, 80%) of the original dataset. As the dataset size increases and the number of nodes remains the same, it is expected that the execution time will increase. In order to measure this, for each portion D_i of the dataset with $i \in [1, 5]$, we calculate $SlowDown = \frac{T_{D_i}}{T_{D_1}}$, where T_{D_1} is the execution time of the first portion (i.e. 20%) and T_{D_i} the execution time of the current one. As shown in Figure 19(a), as the size of the dataset increases, our solutions appears to have linear behaviour. To investigate further the performance of our solution, we measure separately the execution time of the *Join* and *Refine* phases. Concerning the *Join* phase, as illustrated in Figure 19(a), it appears that it is the most "expensive" part of the procedure in comparison with the *Refine* phase, as expected. Subsequently, we keep the size of the dataset fixed (at 100%) and vary the number of nodes. As illustrated in Figure 19(b), we observe that our approach presents linear scaling.

5.6.2 Comparative Evaluation

As already mentioned, the problem of *Distributed Sub-trajectory Join* has not been addressed yet in the literature and it is not straightforward (if and) how state of the art solutions to trajectory similarity search and trajectory join can be adapted to solve the problem. However, if we utilize only a specific instance of our problem, when $\delta t = 0$, then we only need to identify the set of JP during the *Join* phase. Based on this observation, we select to compare with the work presented in [49], called *SJMR*, a state of the art MapReduce-based spatial join algorithm, which is able

to identify efficiently the set of JP that will be passed to the *Refine* procedure and produce the desired result. The reason why *SJMR* was chosen is that it is a generic solution which could form the basis for any distributed spatial join algorithm and thus required the minimum amount of modifications so as to match with our problem specification.

More specifically, *SJMR* repartitions the data at the Map phase and Joins them at the Reduce phase by performing a plane sweep join. For the sake of comparison, we modified *SJMR* by injecting time as a third dimension and introducing parameters ϵ_{sp} and ϵ_t. In more detail. at the Map phase the spatiotemporal space is divided to tiles using a fine grained grid. Then, each data point is expanded by ϵ_{sp} and ϵ_t and is assigned to the tiles with which it intersects. Subsequently, the tiles are mapped to partitions using the method described in [49]. At the Reduce phase, the points are grouped by partition and sorted by one of the dimensions (we chose the temporal dimension so as to be aligned with our solution). Finally, we sweep through the time dimension and report the set of JP.

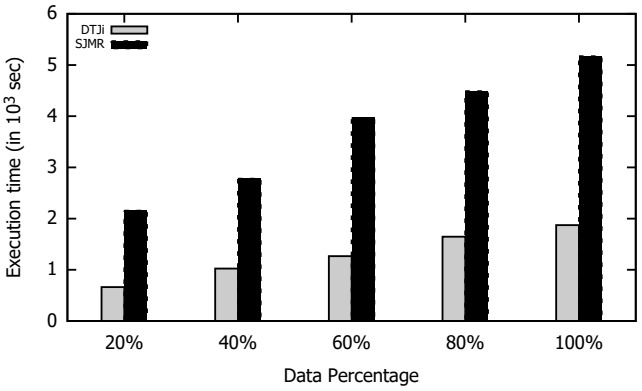

Figure 20: Comparative evaluation between $DTJi$ and *SJMR*

So, in this set of experiments we compare $DTJi$-Join, which outperforms $DTJb$-Join and $DTJr$-Join, with *SJMR*. In more detail, we vary the size of our dataset and measure the execution time of the two algorithms. The results, as illustrated in Figure 20 show that $DTJi$-Join not only performs significantly better than *SJMR* but more importantly, the gain of $DTJi$-Join over *SJMR* increases for larger data sets. The reason for this behaviour lies mainly due to the utilization of the indexing structure of $DTJi$ ([49] uses no indexes) and the fact that $DTJi$-Join is a Map-only job where the repartitioning cost is "paid" only once (as a preprocessing step), unlike *SJMR*, where this cost is "paid" every time at the Map phase, as explained earlier. For a more thorough experimental evaluation, please refer to [44].

6 Conclusions

In this deliverable, we presented the query operators developed in the context of Work package 3 in the Track&Know project. The work targets batch processing of big data stored using a scalable NoSQL storage system.

Our main contribution, in line with the work description in the Grant Agreement, is an abstraction layer in the form of a programming interface (API), which resides between applications and scalable NoSQL stores. This layer hides the heterogeneity of data models and languages of NoSQL stores from the application developer, thus offering a unified way to access the underlying NoSQL stores. Data access is provided by means of basic query operators (filter, project, aggregation, sorting, etc.) as well as a set of mobility operators (range and k-NN queries). This is an important step forward towards providing a standardized data access mechanism over different NoSQL stores, in the same spirit of ODBC/JDBC in relational data management systems. The benefits for application and big data developers are manifold: avoid learning different APIs and languages for each individual NoSQL store, ease of programming and flexibility through a single API, portability of the application code from one NoSQL storage system to another, etc. To demonstrate our proposal, we provide three prototype implementations over a document-oriented store (MongoDB), a wide-column store (HBase), and a key-value store (REDIS). Moreover, we couple the programming API that we offer with a declarative, SQL-like interface, which allows data scientists and business analysts to interact with different NoSQL stores using a (variant of a) standardized query language such as SQL.

The second contribution relates to complex query operators for mobility data, which is the main type of data stored and analyzed in Track&Know. Complex query operators cannot be efficiently supported by the underlying NoSQL store. A join between data sources is a typical example of such a complex operator. Therefore, we introduce a generic, distributed join operator that target mobility data, called distributed sub-trajectory join (DTJ), which identifies maximal portions of trajectories that move close in space and time. This operator is fundamental because it is the main building block for different scalable data analysis tasks, for example it can be exploited for implementing scalable trajectory clustering [45]. We describe the design and implementation of DTJ in MapReduce/Hadoop and we evaluate its performance against a state-of-the-art algorithm using real-life data from diverse domains, including the Pilots of the Track&Know project.

Regarding future work, we feel confident that big data practitioners will find interest in our work, in order to move towards the vision of standardized data access and efficient querying for scalable NoSQL stores. We believe that in the next years, researchers both from the industry and academia are going to focus on the problem of unified access to different scalable storage solutions, an issue that already has attracted attention in different contexts, including polyglot persistence and polystores [18].

References

[1] Pankaj K. Agarwal, Kyle Fox, Kamesh Munagala, Abhinandan Nath, Jiangwei Pan, and Erin Taylor. Subtrajectory clustering: Models and algorithms. In *PODS*, pages 75–87, 2018.

[2] Ablimit Aji, Fusheng Wang, Hoang Vo, Rubao Lee, Qiaoling Liu, Xiaodong Zhang, and Joel H. Saltz. Hadoop-gis: A high performance spatial data warehousing system over mapreduce. *PVLDB*, 6(11):1009–1020, 2013.

[3] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark SQL: relational data processing in spark. In Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1383–1394. ACM, 2015.

[4] Petko Bakalov, Marios Hadjieleftheriou, Eamonn J. Keogh, and Vassilis J. Tsotras. Efficient trajectory joins using symbolic representations. In *Proceedings of MDM*, pages 86–93, 2005.

[5] Petko Bakalov, Marios Hadjieleftheriou, and Vassilis J. Tsotras. Time relaxed spatiotemporal trajectory joins. In *Proceedings of ACM-GIS*, pages 182–191, 2005.

[6] Petko Bakalov and Vassilis J. Tsotras. Continuous spatiotemporal trajectory joins. In *GSN*, pages 109–128, 2006.

[7] Christian Böhm, Bernhard Braunmüller, Markus M. Breunig, and Hans-Peter Kriegel. High performance clustering based on the similarity join. In *Proceedings of CIKM*, pages 298–305, 2000.

[8] Kevin Buchin, Maike Buchin, Marc J. van Kreveld, and Jun Luo. Finding long and similar parts of trajectories. *Comput. Geom.*, 44(9):465–476, 2011.

[9] Yun Chen and Jignesh M. Patel. Design and evaluation of trajectory join algorithms. In *Proceedings of SIGSPATIAL*, pages 266–275, 2009.

[10] Christophe Claramunt, Cyril Ray, Elena Camossi, Anne-Laure Jousselme, Melita Hadzagic, Gennady L. Andrienko, Natalia V. Andrienko, Yannis Theodoridis, George A. Vouros, and Loïc Salmon. Maritime data integration and analysis: recent progress and research challenges. In *Proceedings of EDBT*, pages 192–197, 2017.

[11] Ali Davoudian, Liu Chen, and Mengchi Liu. A survey on nosql stores. *ACM Comput. Surv.*, 51(2):40:1–40:43, 2018.

[12] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: a flexible data processing tool. *Commun. ACM*, 53(1):72–77, 2010.

[13] Hui Ding, Goce Trajcevski, and Peter Scheuermann. Efficient similarity join of large sets of moving object trajectories. In *TIME*, pages 79–87, 2008.

[14] Xin Ding, Lu Chen, Yunjun Gao, Christian S. Jensen, and Hujun Bao. Ultraman: A unified platform for big trajectory data management and analytics. *PVLDB*, 11(7):787–799, 2018.

[15] Christos Doulkeridis and Kjetil Nørvåg. A survey of large-scale analytical query processing in mapreduce. *VLDB J.*, 23(3):355–380, 2014.

[16] Ahmed Eldawy and Mohamed F. Mokbel. Spatialhadoop: A mapreduce framework for spatial data. In *Proceedings of ICDE*, pages 1352–1363, 2015.

[17] Ahmed Eldawy and Mohamed F. Mokbel. The era of big spatial data: A survey. *Foundations and Trends in Databases*, 6(3-4):163–273, 2016.

[18] Aaron J. Elmore, Jennie Duggan, Mike Stonebraker, Magdalena Balazinska, Ugur Çetintemel, Vijay Gadepally, Jeffrey Heer, Bill Howe, Jeremy Kepner, Tim Kraska, Samuel Madden, David Maier, Timothy G. Mattson, Stavros Papadopoulos, Jeff Parkhurst, Nesime Tatbul, Manasi Vartak, and Stan Zdonik. A demonstration of the bigdawg polystore system. *Proc. VLDB Endow.*, 8(12):1908–1911, 2015.

[19] EMSA. Automated behaviour monitoring (abm) algorithms – operational use at emsa. In *Proceedings of MKDAD Workshop*, pages 12–16, 2016.

[20] Qi Fan, Dongxiang Zhang, Huayu Wu, and Kian-Lee Tan. A general and parallel platform for mining co-movement patterns over large-scale trajectories. *PVLDB*, 10(4):313–324, 2016.

[21] Yixiang Fang, Reynold Cheng, Wenbin Tang, Silviu Maniu, and Xuan S. Yang. Scalable algorithms for nearest-neighbor joins on big trajectory data. In *Proceedings of ICDE*, pages 1528–1529, 2016.

[22] Sergej Fries, Brigitte Boden, Grzegorz Stepien, and Thomas Seidl. Phidj: Parallel similarity self-join for high-dimensional vector data with mapreduce. In *Proceedings of ICDE*, pages 796–807, 2014.

[23] Francisco García-García, Antonio Corral, Luis Iribarne, Michael Vassilakopoulos, and Yannis Manolopoulos. Enhancing spatialhadoop with closest pair queries. In *Proceedings of ADBIS*, pages 212–225, 2016.

[24] Alan Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan Narayanam, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava. Building a highlevel dataflow system on top of mapreduce: The pig experience. *Proc. VLDB Endow.*, 2(2):1414–1425, 2009.

[25] Joachim Gudmundsson and Marc J. van Kreveld. Computing longest duration flocks in trajectory data. In *14th ACM International Symposium on Geographic Information Systems, ACM-GIS 2006, November 10-11, 2006, Arlington, Virginia, USA, Proceedings*, pages 35–42, 2006.

[26] Edwin H. Jacox and Hanan Samet. Metric space similarity joins. *ACM Trans. Database Syst.*, 33(2):7:1–7:38, 2008.

[27] Hoyoung Jeung, Man Lung Yiu, Xiaofang Zhou, Christian S. Jensen, and Heng Tao Shen. Discovery of convoys in trajectory databases. *PVLDB*, 1(1):1068–1080, 2008.

[28] Nikolaos Koutroumanis, Panagiotis Nikitopoulos, Akrivi Vlachou, and Christos Doulkeridis. Noda: Unified nosql data access operators for mobility data. In Walid G. Aref, Michela Bertolotto, Panagiotis Bouros, Christian S. Jensen, Ahmed Mahmood, Kjetil Nørvåg, Dimitris Sacharidis, and Mohamed Sarwat, editors, *Proceedings of the 16th International Symposium on Spatial and Temporal Databases, SSTD 2019, Vienna, Austria, August 19-21, 2019*, pages 174–177. ACM, 2019.

[29] Zhenhui Li, Bolin Ding, Jiawei Han, and Roland Kays. Swarm: Mining relaxed temporal moving object clusters. *PVLDB*, 3(1):723–734, 2010.

[30] Wuman Luo, Haoyu Tan, Huajian Mao, and Lionel M. Ni. Efficient similarity joins on massive high-dimensional datasets using mapreduce. In *Proceedings of MDM*, pages 1–10, 2012.

[31] Costas Panagiotakis, Nikos Pelekis, Ioannis Kopanakis, Emmanuel Ramasso, and Yannis Theodoridis. Segmentation and sampling of moving object trajectories based on representativeness. *IEEE Trans. Knowl. Data Eng.*, 24(7):1328–1343, 2012.

[32] Jignesh M. Patel and David J. DeWitt. Partition based spatial-merge join. In *Proceedings of SIGMOD*, pages 259–270, 1996.

[33] Nikos Pelekis, Panagiotis Tampakis, Marios Vodas, Costas Panagiotakis, and Yannis Theodoridis. In-dbms sampling-based sub-trajectory clustering. In *Proceedings of EDBT*, pages 632–643, 2017.

[34] Suprio Ray, Bogdan Simion, Angela Demke Brown, and Ryan Johnson. Skew-resistant parallel in-memory spatial join. In *Proceedings of SSDBM*, pages 6:1–6:12, 2014.

[35] Thomas Seidl, Sergej Fries, and Brigitte Boden. MR-DSJ: distance-based self-join for large-scale vector data analysis with mapreduce. In *Proceedings of DBIS*, pages 37–56, 2013.

[36] Shuo Shang, Lisi Chen, Zhewei Wei, Christian S. Jensen, Kai Zheng, and Panos Kalnis. Trajectory similarity join in spatial networks. *PVLDB*, 10(11):1178–1189, 2017.

[37] Shuo Shang, Lisi Chen, Zhewei Wei, Christian S. Jensen, Kai Zheng, and Panos Kalnis. Parallel trajectory similarity joins in spatial networks. *VLDB J.*, 27(3):395–420, 2018.

[38] Zeyuan Shang, Guoliang Li, and Zhifeng Bao. DITA: distributed in-memory trajectory analytics. In *SIGMOD*, pages 725–740, 2018.

[39] Sameh Shohdy, Yu Su, and Gagan Agrawal. Load balancing and accelerating parallel spatial join operations using bitmap indexing. In *Proceedings of International Conference on High Performance Computing*, pages 396–405, 2015.

[40] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of MSST*, pages 1–10, 2010.

[41] Yasin N. Silva and Jason M. Reed. Exploiting mapreduce-based similarity joins. In *Proceedings of SIGMOD*, pages 693–696, 2012.

[42] Yasin N. Silva, Jason M. Reed, and Lisa M. Tsosie. Mapreduce-based similarity join for metric spaces. In *Proceedings of Cloud-I*, page 3, 2012.

[43] Na Ta, Guoliang Li, Yongqing Xie, Changqi Li, Shuang Hao, and Jianhua Feng. Signature-based trajectory similarity join. *IEEE Trans. Knowl. Data Eng.*, 29(4):870–883, 2017.

[44] Panagiotis Tampakis, Christos Doulkeridis, Nikos Pelekis, and Yannis Theodoridis. Distributed subtrajectory join on massive datasets. *ACM Trans. Spatial Algorithms and Systems*, 6(2):8:1–8:29, 2020.

[45] Panagiotis Tampakis, Nikos Pelekis, Christos Doulkeridis, and Yannis Theodoridis. Scalable distributed subtrajectory clustering. In *2019 IEEE International Conference on Big Data (Big Data), Los Angeles, CA, USA, December 9-12, 2019*, pages 950–959. IEEE, 2019.

[46] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Anthony, Hao Liu, and Raghotham Murthy. Hive - a petabyte scale data warehouse using hadoop. In Feifei Li, Mirella M. Moro, Shahram Ghandeharizadeh, Jayant R. Haritsa, Gerhard Weikum, Michael J. Carey, Fabio Casati, Edward Y. Chang, Ioana Manolescu, Sharad Mehrotra, Umeshwar Dayal, and Vassilis J. Tsotras, editors, *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*, pages 996–1005. IEEE Computer Society, 2010.

[47] Dong Xie, Feifei Li, and Jeff M. Phillips. Distributed trajectory similarity search. *PVLDB*, 10(11):1478–1489, 2017.

[48] Demetrios Zeinalipour-Yazti, Song Lin, and Dimitrios Gunopulos. Distributed spatio-temporal similarity search. In *Proceedings of CIKM*, pages 14–23, 2006.

[49] Shubin Zhang, Jizhong Han, Zhiyong Liu, Kai Wang, and Zhiyong Xu. SJMR: parallelizing spatial join with mapreduce on clusters. In *Proceedings of CLUSTER*, pages 1–8, 2009.

A Ethics Proforma

A. PERSONAL DATA

1. Is personal data going to be processed for the completion of this deliverable?
   (a) If "yes", do they refer only to individuals connected to project partners? Or to third parties as well?

   We have used anonymized data from the data providers in the project, corresponding to the three pilots. The anonymization process took place prior to data release for usage in Track&Know.

2. Are "special categories of personal data" going to be processed for this deliverable? (whereby these include personal data revealing racial or ethnic origin, political opinions, religious or philosophical beliefs, and trade union membership, as well as, genetic data, biometric data, data concerning health or data concerning a natural person's sex life or sexual orientation) **No.**

3. Has the consent of the individuals concerned been acquired prior to the processing of their personal data?
   (a) If "yes", based on the Project's Consent Form? On a different legal basis?

   Yes, the partners that act as data providers in the project have acquired consent from individuals prior to data processing. In the case of personal data, a consent form has been used. In case of data coming from one of the clients of a data provider, appropriate agreements have been made about the usage of such data, always appropriately anonymized.

4. In the event of processing of personal data, is the processing:
   (a) "Fair and lawful", meaning executed in a fair manner and following consent of the individuals concerned? **Yes.**
   (b) Performed for a specific (project-related) cause only? **The usage of any data within the Task 3.3 and Deliverable D3.3 concerned only implementation and testing purposes of the developed tools. For the fulfilment of these tasks appropriately anonymised and deidentified data samples were provided by partners.**
   (c) Executed on the basis of the principle of proportionality (meaning that only data that are necessary for the processing purposes are being processed)? **Yes.**
   (d) Based on high-quality personal data? **Yes.**

5. Are all lawful requirements for the processing of the data (for example, notification of the competent Data Protection Authority(s), if applicable) adhered to? **Not applicable.**

6. Have individuals been made aware of their rights (particularly the rights to access, rectify and delete the data)? **Yes, the partners that act as data providers in the project have acquired consent from individuals prior to data processing, and took care of informing them of their rights.**

B. DATA SECURITY

1. Have proportionate security measures been undertaken for protection of the data, taking into account project requirements and the nature of the data? **Yes.**

   (a) Brief description of such measures (including physical-world measures, if any)

   There have been a number of security measures in place including: (a) remote access to the machines used only by SSH, (b) data resides on the platform only during the execution of experiments, (c) access to the NoSQL store requires authentication.

2. Is there a data breach notification policy in place within your organisation? **Yes.**

C. DATA TRANSFERS

1. Are personal data transfers beyond project partners going to take place for this deliverable? **No.**

   (a) If "yes", do these include transfers to third (non-EU) countries?

2. Are personal data transfers to public authorities going to take place for this deliverable? **No.**

   (a) Do any state authorities have direct or indirect access to personal data processed for this deliverable?

3. Taking into account that the Project Coordinator is the "controller" of the processing and that all other project partners involved in this deliverable are "processors" within the same contexts, are there any other personal data processing roles attributed to any third parties for this deliverable? **No.**

D. ETHICS AND RELATED ISSUES

1. Are personal data of children going to be processed for this deliverable? **No.**

2. Is profiling in any way enabled or facilitated for this deliverable? **Yes, however GDPR compliance measures are applied.**

3. Are automated-decisions made or enabled for this deliverable? **No.**

4. Have partners for this deliverable taken into consideration system architectures of privacy by design and/or privacy by default, as appropriate? **Yes.**

5. Have partners for this deliverable taken into consideration gender equality policies? **Not applicable.**

6. Have partners for this deliverable taken into consideration confidentiality of the data requirements? **Yes.**

B NoDA

B.1 NoDA Maven Modules Info

NoDA is a multi-module Java project based on *Apache Maven* and it is available at:

https://github.com/nkoutroumanis/NoSQL-Operators

Particularly, all of its modules are located under the *noda-parent* project which serves as the parent project. The project includes a pom.xml file where its configuration regarding to the packaging of the modules and its external dependencies is defined. The modules of the project have also their own pom.xml file where their external dependencies are declared. The declared dependencies in the parent pom.xml file are inherited from the child modules, meaning that can be shared and thus be utilized directly; without having to be defined in children pom.xml files. The advantage of this is that duplication is reduced.

B.1.1 Project: noda-parent

In the *noda-parent* project, we have declared only the dependencies that are utilized directly from all of the rest modules. Being the parent project, it does not contain any programming packages or implementation components. Its external dependencies are the following:

1. *junit* v. 4.12 – library used for testing through which we check parts of the implementation code while developing.
2. *slf4j-api* v. 1.7.26 – Simple Logging Facade for Java API used in conjunction with the logback-classic dependency.
3. *logback-classic* v. 1.2.3 – library which is considered as the reference implementation of SLF4J, used for logging.

B.1.2 Module: noda-core

The *noda-core* module consists of 13 packages with 32 concrete/abstract classes and interfaces. Approximately, 1000 code lines composite its functionality. The module is used as an external dependency by all the rest modules. Specifically, its classes and interfaces are inherited and implemented from the *noda-mongodb*, *noda-hbase* and *noda-redisearch* modules. The *noda-client* module uses the *noda-core* dependency just to have access on the abstract form of NoDA when defining a connection.

The declared dependencies in the *noda-core* pom.xml file are inherited from the modules that use it as an external dependency. For this reason, we do not only declare the dependencies that are used directly from the *noda-core* module, but also the common ones we want to make

available to any module that implements NoDA functionality upon a NoSQL database. The declared dependencies in the *noda-core*, are the following:

1. *spark-core* v. 2.4.0 – the fundamental (core) library for using the Spark framework.
2. *spark-sql* v. 2.4.0 – the library through which we can utilize Dataframes (Datasets) from Spark when fetching the results from a NoSQL database.
3. *quadtree* – a library[6] which implements a QuadTree in order to form an equi-depth histogram for kNN querying.

B.1.3 Module: noda-client

The *noda-client* module consists of 5 packages with 15 concrete/abstract classes and interfaces. Approximately, 15,500 code lines (including the generated classes from ANTLR) composite its functionality. It uses the following external dependencies:

1. *noda-core* – the fundamental (core) library for using the NoDA abstraction layer upon a NoSQL database.
2. *noda-mongo* – the library that implements NoDA upon MongoDB.
3. *noda-hbase* – the library that implements NoDA upon HBase.
4. *noda-redisearch* – the library that implements NoDA upon Redis database by using the RediSearch search engine.

The *noda-mongo*, *noda-hbase* and *noda-redisearch* dependencies are declared as optional. This means that when utilizing *noda-client* for accessing a NoSQL store, the optional dependencies are not included. The user has to define explicitly the *module* that implements NoDA upon the NoSQL store that is going to be accessed.

B.1.4 Module: noda-mongodb

The *noda-mongodb* module consists of 7 packages with 39 concrete/abstract classes and interfaces. Approximately, 1300 code lines composite its functionality. It uses the following external dependencies:

1. *noda-core* – the fundamental (core) library for inheriting from its classes in order to implement NoDA functionality upon MongoDB.
2. *mongo-java-driver* v. 3.8.2 – the MongoDB driver in Java programming language for synchronous and asynchronous interaction.
3. *mongo-spark-connector* v.2.4.0 – the MongoDB Connector for Spark library which provides integration between MongoDB and Apache Spark.

[6] https://github.com/nkoutroumanis/QuadTree

B.1.5 Module: noda-hbase

The *noda-hbase* module consists of 11 packages with 43 concrete/abstract classes and interfaces. Approximately, 9100 code lines (including the generated classes from Protocol Buffers) composite its functionality. It uses the following external dependencies;

1. *noda-core* – the fundamental (core) library for inheriting from its classes in order to implement NoDA functionality upon HBase.
2. *hbase-client* v. 2.2.4 – the HBase client library in Java programming language for accessing the database.
3. *geo* v. 0.7.1 – a library[7] which provides utility methods for geohashing. It is used for spatial and spatio-temporal querying upon HBase.

B.1.6 Module: noda-redisearch

The *noda-redisearch* module consists of 7 packages with 38 concrete/abstract classes and interfaces. Approximately, 1400 code lines composite its functionality. It uses the following external dependencies;

1. *noda-core* – the fundamental (core) library for inheriting from its classes in order to implement NoDA functionality upon RediSearch.
2. *jredisearch* v. 1.4.0 – the Java client library for RediSearch.

[7] https://github.com/davidmoten/geo