ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
"ΔΙΑΧΕΙΡΙΣΗ ΠΛΗΡΟΦΟΡΙΑΣ ΚΑΙ ΔΕΔΟΜΕΝΩΝ"

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

BigSQLTraj: A SQL-extended framework for storing & querying big mobility data

Πέτρος Θ. Πέτρου

Επιβλέπων: **Μανόλης Κουμπαράκης,** Καθηγητής

ΑΘΗΝΑ

ΜΑΙΟΣ 2019

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

BigSQLTraj: A SQL-extended framework for storing & querying big mobility data

Πέτρος Θ. Πέτρου
Α.Μ.: 1424

ΕΠΙΒΛΕΠΩΝ: **Μανόλης Κουμπαράκης,** Καθηγητής

ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ: **Δημήτριος Γουνόπουλος,** Καθηγητής

Μάιος 2019

ΠΕΡΙΛΗΨΗ

Τα τελευταία χρόνια, λόγω της ευρείας χρήση αισθητήρων και έξυπνων συσκευών, παρατηρείται μια εκθετική παραγωγή δεδομένων κίνησης, που εντάσσονται στην κατηγορία δεδομένα μεγάλης κλίμακας (big data). Για παράδειγμα εφαρμογές δρομολόγησης, παρακολούθηση κυκλοφοριακής ροής, έλεγχος στόλου ακόμη και προβλέψεις ή αποφυγή κινδύνων βασίζονται στην επεξεργασία χωρικών και χωροχρονικών δεδομένων. Τα δεδομένα αυτά πρέπει να αποθηκεύονται και να επεξεργάζονται κατάλληλα ώστε στη συνέχεια να αποτελέσουν γνώση για τους οργανισμούς. Προφανώς η διαδικασία αυτή απαιτεί συστήματα και τεχνολογίες κατάλληλες για τον μεγάλο όγκο δεδομένων εισόδου. Στην παρούσα διπλωματική εργασία χρησιμοποιήσαμε δεδομένων από κινήσεις πλοίων και πιο συγκεκριμένα δεδομένα που παράγονται από το automatic identification system (AIS).

Για τους σκοπούς της συγκεκριμένης διπλωματικής εργασίας αναπτύχθηκε το σύστημα BigSQLTraj: Ένα πλαίσιο βασισμένο σε SQL για την αποθήκευση και επερώτηση μεγάλων δεδομένων από κινούμενα αντικείμενα. Οι εφαρμογές μεγάλων δεδομένων περιλαμβάνουν τα επίπεδα διαχείρισης, επεξεργασίας, αναλυτικές και οπτικοποίησης δεδομένων από ετερογενής πηγές ή σε ιστορικά δεδομένα ή σε δεδομένα ροών. Στην παρούσα διπλωματική εργασία εξετάζουμε τα επίπεδα διαχείρισης και επεξεργασίας μεγάλων ιστορικών δεδομένων. Στόχος του συστήματος είναι να παρέχει την δυνατότητα σε χρήστες να αποθηκεύουν και να επεξεργάζονται με αποδοτικό τρόπο μεγάλα γεωχωρικά και χωροχρονικά δεδομένα πάνω από ένα κατανεμημένο σύστημα επεκτείνοντας ή αναπαράγοντας μεθόδους και αλγορίθμους από ήδη υπάρχοντα συστήματα. Πρώτος στόχος της εργασίας είναι να επιλεχθούν εργαλεία που θα μπορούν να επικοινωνούν μεταξύ τους και θα παρουσιάζουν μια ενιαία εικόνα στους εξωτερικούς χρήστες. Οι καινοτομίες που παρέχει το σύστημα είναι η δημιουργία μεθόδων για ισοκατανεμημένη, αλλά ταυτόχρονα βασισμένη στην ομοιότητα, διαμέριση των δεδομένων στους κόμβους της συστάδας υπολογιστών, η δημιουργία μιας SQL διεπαφής στο κατανεμημένο σύστημα που θα παρέχει εξελιγμένες μεθόδους για την επεξεργασία των αποθηκευμένων δεδομένων και θα επιτρέπει σε συστήματα που ήδη αλληλεπιδρούν με συστήματα βασισμένα σε SQL να μεταφερθούν σε τεχνολογίες μεγάλων δεδομένων με τις ελάχιστες δυνατές αλλαγές.

Πρώτος στόχος της παρούσας διπλωματικής εργασίας είναι η ενσωμάτωση (integration) διάφορων τεχνολογιών. Η υλοποίηση της παρούσας διπλωματικής βασίζεται σε βιβλιοθήκες ανοιχτού κώδικα για επεξεργασία μεγάλων δεδομένων. Οι βιβλιοθήκες αυτές είναι: Apache Hadoop, Apache Spark, Apache Hive και Apache Tez. Οι βασικότερες λειτουργίες που παρέχει η βιβλιοθήκη Apache Hadoop είναι το κατανεμημένο σύστημα αρχείων (Hadoop Distributed File System) που γράφονται και διαβάζονται τα δεδομένα. Επιπλέον ο διαχειριστής πόρων του Apache Hadoop (Yarn - resource manager) που ελέγχει το φόρτο εργασίας των υπολογιστών της συστάδας και αναθέτει τις διεργασίες που πρέπει να εκτελεστούν. Τα δύο αυτά εργαλεία είναι αποτελούν τον πυλώνα τις ενσωμάτωσης μεταξύ των υπολογιστών της συστάδας αλλά και των βιβλιοθηκών που τρέχουν στη συστάδα. Η βιβλιοθήκη Apache Spark, μέσω του προγραμματιστικού πλαισίου MapReduce, παρέχει την λειτουργία την επεξεργασίας είτε σε ιστορικά δεδομένα είτε σε ροές δεδομένων και την αποθήκευσή τους στο κατανεμημένο σύστημα αρχείων του Hadoop. Στη συνέχεια το Apache Hive μας δίνει την δυνατότητα για εκτέλεση ερωτημάτων σε αρχεία που βρίσκονται στο κατανεμημένο σύστημα αρχείων του Hadoop μέσω της HiveQL γλώσσας που είναι ισοδύναμη με της παραδοσιακή SQL, ενώ οι βιβλιοθήκες Apache Spark και Apache Tez αποτελούν την μηχανή εκτέλεσης (execution engine) ενός HiveQL ερωτήματος και μεταφράζουν την επερώτηση σε MapReduce διαδικασία.

Κανένα από τα παραπάνω συστήματα δεν έχει την δυνατότητα επεξεργασίας γεωχωρικών ή δεδομένων κίνησης στην βασική του εκδοχή. Οι προθήκες που έγιναν περιλαμβάνουν:

- Δημιουργία συναρτήσεων για τον καθαρισμό χωροχρονικών σημείων και δημιουργία τροχιών κινούμενων αντικειμένων από τα σημεία αυτά με την βιβλιοθήκη Apache Spark

- Χωροχρονικός καταμερισμός των τροχιών στους υπολογιστές της συστάδας, δημιουργία ευρετηρίων. Τα ευρετήρια περιλαμβάνουν την χωροχρονική έκταση της διαμοιρασμένης πληροφορίας και μια κωδικοποίηση βασισμένη σε τρισδιάστατα τοπικά ευρετήρια βάσει της πληροφορίας που έχει κάθε υπολογιστής με χρήση των βιβλιοθηκών Apache Spark και Apache Hadoop.

- Δημιουργία κατάλληλων μεθόδων, για την αξιοποίηση της αποθήκευσης τους προηγούμενου βήματος, για επερωτήσεις διαστήματος (range queries) και επερωτήσεων ομοιότητας (kNN queries).

Η σύγκριση που πραγματοποιήσαμε αφορά τη χρονική απόδοση των επερωτήσεων διαστήματος (range queries) και επερωτήσεων ομοιότητας (kNN queries), βάσει του τρόπου αποθήκευσης των δεδομένων όπως αναφέρθηκε προηγουμένως. Σε πρώτη φάση συγκρίναμε την χρονική διάρκεια ολοκλήρωσης των παραπάνω ερωτημάτων για τους διαθέσιμους τρόπους αποθήκευσης και για τους διαθέσιμους μηχανισμούς εκτέλεσης συναρτήσει του αριθμού των υπολογιστών που τρέχουν στο κατανεμημένο σύστημα (scalability). Στη συνέχεια συγκρίναμε την χρονική διάρκεια ολοκλήρωσης των παραπάνω ερωτημάτων για τους διαθέσιμους τρόπους αποθήκευσης και για τους διαθέσιμους μηχανισμούς εκτέλεσης συναρτήσει του όγκου δεδομένων (speed-up), αυξάνοντας σε κάθε βήμα των όγκο δεδομένων. Τα αποτελέσματα μας έδειξαν ότι ο πιο αποδοτικός τρόπος εκτέλεσης των ερωτημάτων με τη χρήση ενός ευρετηρίου για την διαμοιρασμένη πληροφορία και στην συνέχεια η χρήση μιας κωδικοποίησης βασισμένη σε τοπικά ευρετήρια για την ανάκτηση του τελικού αποτελέσματος με μηχανισμό εκτέλεσης τη βιβλιοθήκη Apache Spark.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Βελτιστοποίηση Επερωτήσεων

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Διαχείριση Δεδομένων, Χωροχρονικά Δεδομένα, Μεγάλα Δεδομένα, Ευρετήρια, Κατανεμημένα Συστήματα

ABSTRACT

Last decades, the need for performing advanced queries over massively produced data, such as mobility traces, in efficient and scalable ways is particularly important. This thesis describes BigSQLTraj a framework that supports efficient storing, partitioning, indexing and querying on spatial and spatio-temporal (i.e. mobility) data over a distributed engine. Every big data end-to-end application consists of four layers, data management, data processing, data analytics and data visualization for heterogeneous data sources for batch or streaming data. This thesis focuses on data management and data processing for historical data.

The first goal is finding systems that offers ready-to-use integration pipelines to take advantage of the best operation of each tool. For our implementation we chose open source big data frameworks such as Apache Hadoop, Apache Spark, Apache Hive and Apache Tez. Apache Hadoop and especially its distributed file system (HDFS) allowed all the other libraries to have a common read and write layer. On the other hand, Hadoop's Resource Manager (Yarn) exploits the all the available computer resource. BigSQLTraj extending the functionality of existing spatial or spatio-temporal systems, centralized or distributed, to create two core and independent components. The first component is responsible for storing, spatiotemporal partitioning and indexing the data into a distributed file system and it is implemented on-top of Apache Spark. Many spatio-temporal partitioners and a 3D-STRtree index are implemented to support a collection of operators apart from existing partitioners and indexing methods that inherit from state-of-the-art distributed spatial and spatiotemporal systems.

The second component is a distributed sql engine. We extend the functionality of HiveQL in order to achieve rapid access in such kind of data (i.e. geospatial and mobility data) and storing. Our final goal is optimizing Hive's join procedure that is required for both query types using the data structures from the first toolbox. We demonstrate the functionality of our approach and we conduct an extensive experimental study based on state-of-the-art benchmarks for mobility data. Our benchmark focuses on the total execution time of range queries and kNN queries based on the data storing model. At first we compare the temporal performance of different storing alternatives and execution engines for the entire dataset and vary the number of workers in order to review the systems scalability. Furthermore, we vary the size of our dataset and measure the execution time of the queries. To study the effect of dataset size, we split the original dataset into 5 chunks (20%, 40%, 60%, 80%, 100%). Based on the results we come to the conclusion that the best workflow includes a global index structure for workers metadata and a local index-based encoding for storing the entire trajectories of a partition into a single column and the execution time seems to follow linear behaviour.

SUBJECT AREA: Query Prcessing

KEYWORDS: Data Management, Mobility Data, Big Data, Indexing, Distributed Systems

To my beloved mother - Sophia

ACKNOWLEDGEMENTS

I would like to thank Konstantina Bereta and Manolis Koubarakis for their understanding and supervision during the writing of my thesis. I would also like to thank Datastories Lab from University of Piraeus for providing computer resources for clustering setup. Last but not least, I would like to thank Nancy Koukouri, my family and my friends for all their support and understanding. Finally, my acknowledgements to IMIS Hellas (www.imishellas.gr) for kindly providing the AIS dataset which allowed me to test and demonstrate my work.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

PREFACE

Nowadays data is growing with remarkable speed, and this vast amount data must be processed properly if we want to have control over it, transform it to information and gain knowledge. GPS-equipped devices are everywhere and tracking positions. This leads in a imperative need for developing new techniques, technologies and services for these kind of data and their producers. Sometimes hardware resources they are not enough in itself. Last decades, management and processing algorithms have been proposed in the literature. Location-based services, object's future position, traffic prediction, safety and privacy are only some example of mobility data applications in the real world. All these applications are based on some primitive spatiotemporal queries. This thesis aims to conduct an empirical evaluation and benchmarking of the state-of-the-art techniques over big processing systems under different setups.

1. INTRODUCTION

The continuous and significant growth of data together with improved access to data and the availability of powerful Information and Communication Technologies (ICT) systems have led to intensified activities around Big Data Value. Powerful data techniques and tools allow collecting, storing, analysing, processing and visualising vast amounts of data. Open data initiatives are gaining momentum, providing broad access to data from the public sector, business and science. The exploitation of Big Data in various sectors has a potential socio-economic impact far beyond the specific Big Data market. Therefore, it is essential to embrace new technology, applications, use cases and business models within and across various sectors and domains. This will ensure the rapid adoption of Big Data by organisations and individuals, and provide major returns in terms of growth and competitiveness. In particular, the efficiency gains made possible by Big Data will also have a profound societal impact. The issue for organizations is not storing or retrieving data, but finding useful pieces. Information is power, and those who can distinguish between raw data (garbage) and knowledge can create extra profits.

Lately, there is an explosion in the usage of smart devices (smartphones, smart stations (weather, transportation), internet of things, etc.) in every moment in our life. Humans are used to publishish their location (with text or images) via social media in daily base, transportation companies (i.e. shipping companies) and generally moving entities must publish their positions for security or other reasons. Spatial and spatiotemporal data is a special category of data sources. Every human, car, even vessels and airplanes can register their position in customized databases. Data types from the above use cases are more complicated and there is a need for non-traditional database application such as location-based services, WWW repositories. Location-based services are IT services for providing information based on the current state of the object (current or recent spatial(-temporal) history) [5]. For example, by using the information about the location it is possible to fetch relevant information such as hot paths, nearby points of interests(POIs) and available services (e.g. traffic management, carpooling, navigational services) [10]. At this point it is important to clarify two different, but with common features, concepts. Spatiotemporal data in general and spatiotemporal trajectories (or mobility data) are two different data types, which both combine the space and time.

Koubarakis et al. [2] was one of the earliest efforts to study STDB, from modeling to implementation aspects, as the final result of the pioneering ‘‘ChoroChronos” EU funded research project. Partially resulted by the same project, Güting & Schneider [8] was the first (and still remains a ‘‘must-read”) monograph in the field of MOD. A short introduction to the concept of spatio-temporal (trajectory) data is presented in [20].

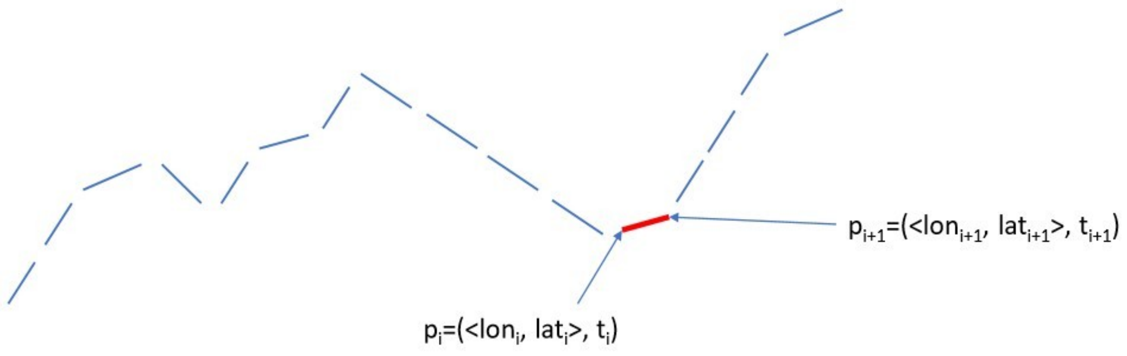
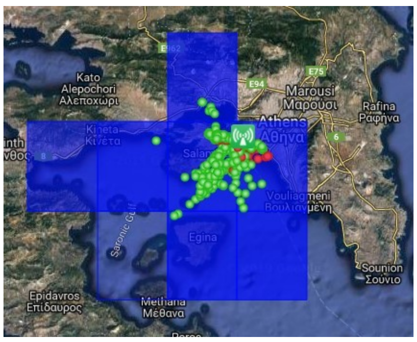
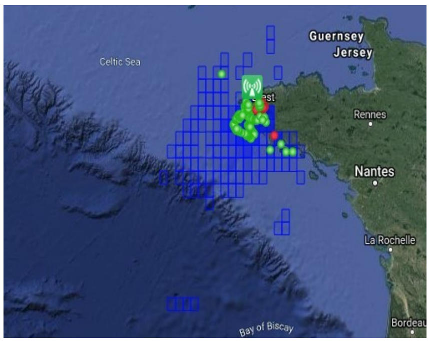

p_{i+1}=(<lon_{i+1}, lat_{i+1}>, t_{i+1})
p_i=(<lon_i, lat_i>, t_i)

Figure 1: A trajectory as a sequence of time-stamped locations of a moving object

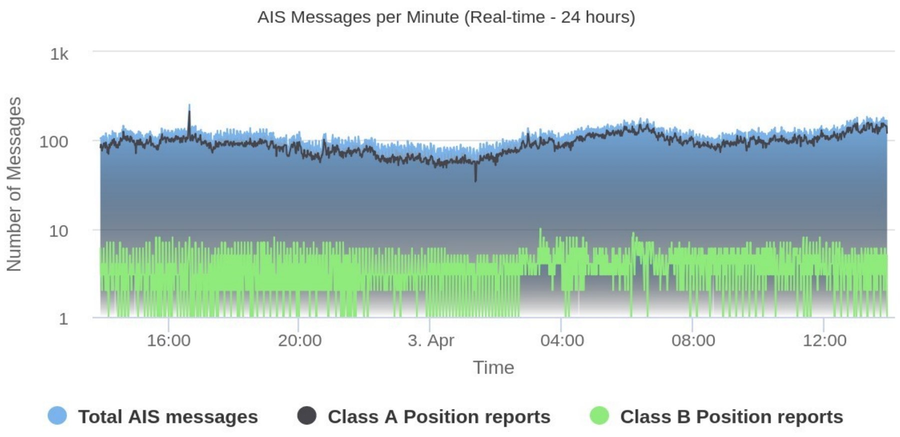
The next question is whether the mobility data and its applications in real life could be characterized as big data and follow the 4V. MarineTraffic[1] is an organization that collects mobility message from 3971 stations. Its station cover radius is almost 30km. The two figures below show examples of two such stations and their input messages rate.



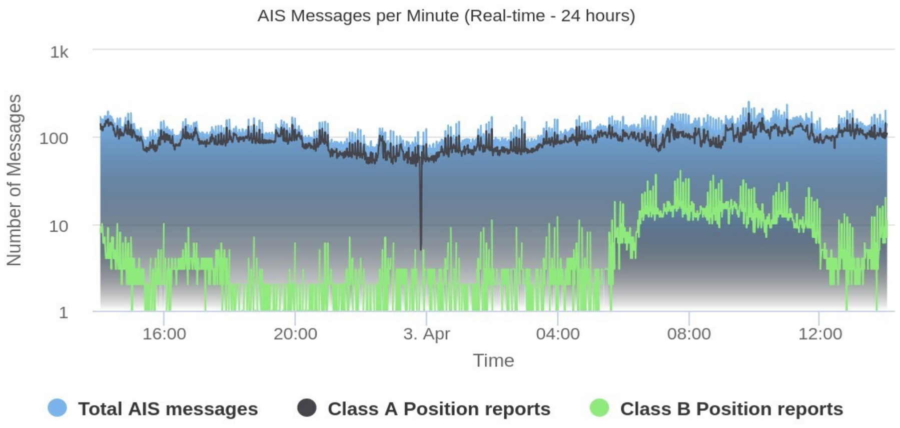


(a) (b)

Figure 2: Piraeus & Brest AIS station location and are coverage



(a)



(b)

Figure 3: Incoming messages rate

[1] www.marinetraffic.com

Figure 3 shows that each station receives 100 messages per minute. This means 397100 per minute for all station around the world and 23826000 messages per hour! In better of our knowledge it would impossible for a system to handle and execute query over this amount of data only with the resources of the memory. Systems that can give rapid disk access would be necessary, in order to give the ability to end-users to handle such data and join them with points of interest.

In this thesis we proposed BigSQLTraj a framework for managing querying and indexing spatial and spatiotemporal (mobility) data over a distributed (file) system. Our implementation consists of two components. The first toolbox extends or reproduces current state of the art Spatial(-Temporal) Big Data Engine that adds spatial & spatiotemporal support (partitioning, indexing and storing). The second one is an extension of HiveQL that adds spatial and spatiotemporal functions (UDFs) to the Hive Language, thus we can take advantage of the data management pipeline from the first toolbox.

This document contains eight chapters. At first, we begin with giving generic definitions, such as data types, the purpose of this master thesis, and the algorithms we have implemented to achieve the final results. In the following section, we provide the necessary background knowledge about the technologies we have used. All of them are big data frameworks. Later on, in the third section, a reader can find previous works related to our research area. We have briefly represented benchmark, indices and systems that dealing with problems related to modility data. In Section 4, we have described the proposed systems architecture principles and compare it with a typical big data architecture. In Section 5, we describe the data managent workflow. We discuss problems about the data quality, filters that cleansed data and how our proposed partitioner framework works. In Section 6, we describe the query processing framework. We present the implement UDFs for mobility data that improve system’s performance and we give example with full queries examples. Finally, we conclude the achieved results, gains and the aim of this master thesis, and add some words about future works can be done in the related area.

2. BACKGROUND

In this section we introduce the state-of-the-art big data engines and their components that are used to implement this thesis. Moreover, we briefly present vessel messages' structure because part of these messages are our data source.

2.1 Apache Hadoop

Hadoop[2] [33] is an open source distributed processing framework that administers data processing and storage for big data applications running in computer cluster. It is the first open source framework that provides MapReduce API, with a resource manager and a distributed storage layer. It is a unified platform that is the core element of a growing ecosystem of massive management and processing technologies. Hadoop is more flexible than traditional database systems and data warehouses, enabling users to gather, process, analyze structure and (especially) unstructured data more flexibly.

Hadoop runs on clustered computers and can horizontal scale up in order to support new nodes and massive amounts of data. It uses a distributed file system (a.k.a. HDFS) that's designed to provide quick access over distributed data, plus fault-tolerant capabilities so applications can handle individual failures.

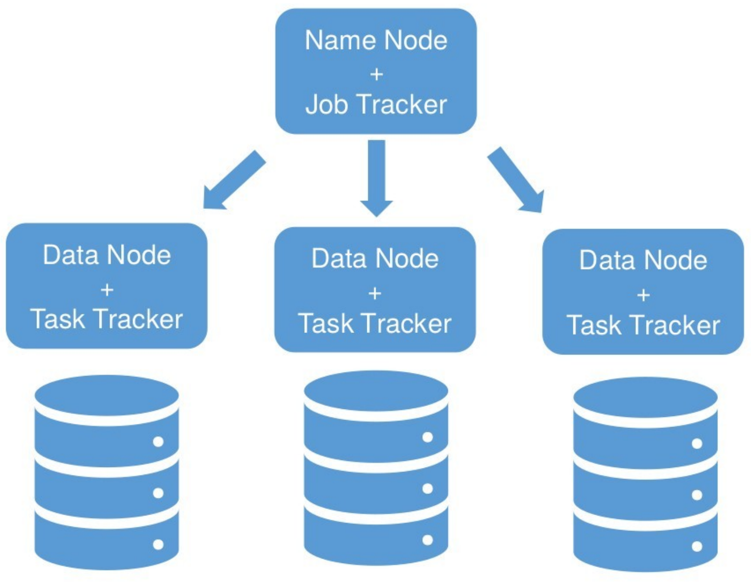


Figure 4: Hadoop Architecture

2.1.1 Apache Hadoop Yarn

YARN's basic idea is to divide resource management and job scheduling/monitoring functionalities into separate daemons. The idea is to have a global ApplicationMaster (AM) and ResourceManager (RM). A submitted application is either a single job or a job DAG.

[2] http://hadoop.apache.org/

The data-computation framework is formed by ResourceManager and NodeManager. ResourceManager is the ultimate authority that arbitrates resources across all of the system's applications. NodeManager is the per-machine framework agent responsible for containers, supervision their use of resources (cpu, memory, disk, network) and reporting to ResourceManager / Scheduler the same thing.

The per-application ApplicationMaster is, in effect, a framework specific library and is tasked with negotiating resources from the ResourceManager and working with the NodeManager(s) to execute and monitor the tasks.

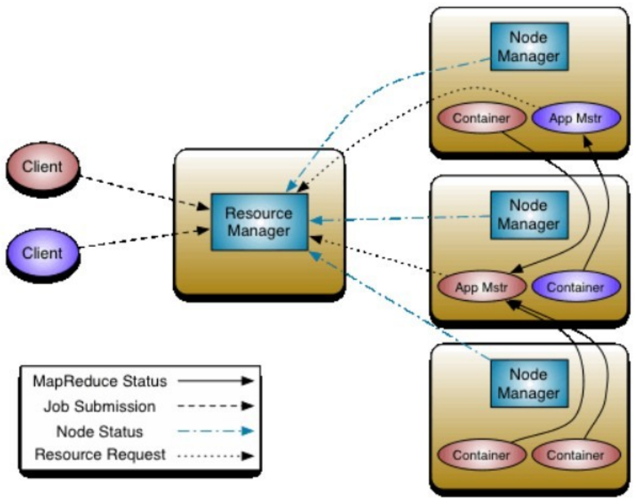


Figure 5: Yarn Workflow

The ResourceManager has two main components: Scheduler and ApplicationsManager.

The Scheduler is responsible for allocating resources to the different running applications subject to familiar capacity constraints, queues etc. Scheduler is pure planner in the sense that it does not monitor or track the application status. Moreover, it cannot guarantee the restart of failed tasks either due to application or hardware failures. The Scheduler performs its scheduling function depend on the application's resource requirements; it does so based on the abstract notion of a resource container that incorporates elements such as memory, cpu, disk, network etc. The Scheduler has a pluggable policy that is liable for splitting the cluster computing power among the various queues, applications etc.

The ApplicationsManager is responsible for accepting job-submissions, negotiating the first container for executing the application specific ApplicationMaster and provides the service for restarting the ApplicationMaster container on failure. The per-application ApplicationMaster has the responsibility of negotiating appropriate resource containers from the Scheduler, tracking their status and monitoring for progress.

2.1.2 Apache Hadoop HDFS

HDFS has a master/slave architecture. An HDFS cluster consists of a single NameNode, a master server that manages the file system namespace and regulates access to files by clients. In addition, there are a number of DataNodes, usually one per node in the cluster, which manage storage attached to the nodes that they run on.

HDFS exposes a file system namespace and allows user data to be stored in files. Internally, a file is split into one or more blocks and these blocks are stored in a set of DataNodes. The NameNode executes file system namespace operations like opening, closing, and renaming files and directories. It also determines the mapping of blocks to DataNodes. The DataNodes are responsible for serving read and write requests from the file system's clients. The DataNodes also perform block creation, deletion, and replication upon instruction from the NameNode.

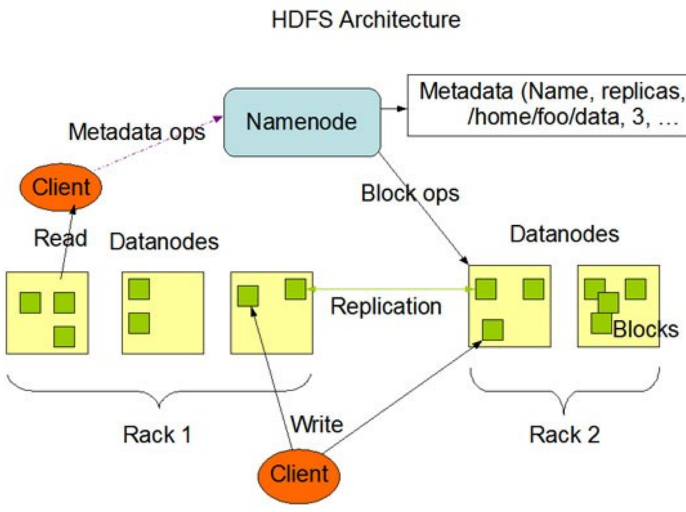


Figure 6: HDFS Architecture

The NameNode and DataNode are pieces of software designed to run on commodity machines. These machines typically run a GNU/Linux operating system (OS). HDFS is built using the Java language; any machine that supports Java can run the NameNode or the DataNode software. Usage of the highly portable Java language means that HDFS can be deployed on a wide range of machines. A typical deployment has a dedicated machine that runs only the NameNode software. Each of the other machines in the cluster runs one instance of the DataNode software. The architecture does not preclude running multiple DataNodes on the same machine but in a real deployment that is rarely the case.

The existence of a single NameNode in a cluster greatly simplifies the architecture of the system. The NameNode is the arbitrator and repository for all HDFS metadata. The system is designed in such a way that user data never flows through the NameNode.

2.2 Apache Hive

HiveQ[3] [34] provides users with a SQL-like declarative language that is the core contribution of Hive system. HiveQL compiles queries into map-reduce jobs executed on Hadoop ecosystem framework such as Spark or Tez. In addition, HiveQL supports custom mapreduce scripts to be plugged into queries.

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

The Hive query language (HiveQL) follows most of the SQL standard capabilities and some extensions that we have found useful. Traditional SQL features like from clause sub-queries, various types of joins – inner, left outer, right outer and outer joins, cartesian products, group bys and aggregations, union all, create table as select and many useful functions on primitive and complex types make the language very SQL like. In fact, for many of the constructs mentioned before it is exactly like SQL. This enables anyone familiar with SQL to start a hive cli (command line interface) and begin querying the system right away. Useful metadata browsing capabilities like show tables and describe are also present and so are explain plan capabilities to inspect query plans (though the plans look very different from what you would see in a traditional RDBMS).

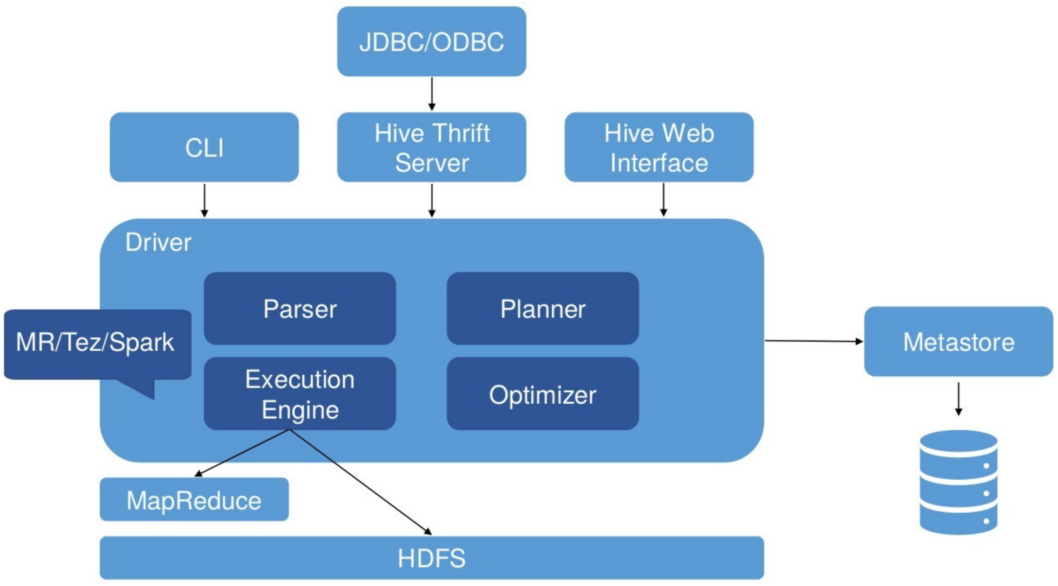


Figure 7: Hive Architecture

2.3 Apache Spark

Spark[4] [35] is one of the biggest alternatives to Hadoop. On its website [9], developers claim that sometimes it is 100 times faster than Hadoop’s MapReduce regarding memory processes. This lets us say that users can use Hadoop (HDFS) as storage of old data but processing them via Spark will be easier and faster. The key point of Spark programming is Resilient Distributed Dataset (RDD). RDDs are lazily evaluated, and it lets Spark to find an efficient plan for computations. Since results of RDD operations are RDDs too, these transformations are not computed immediately. Instead, when an action is being performed, Spark checks all the transformations introduced and creates an optimized execution plan which sometimes builds up better modularity than the programmer thought of. The execution is performed only once for the whole graph of transformations. It is worthy to emphasize that RDDs shares the data amongst computation nodes and they are only called when there is an action taking place.

Spark SQL is a new module in Apache Spark that integrates relational processing with Spark’s functional programming API. Spark SQL is compatible with Hive and lets Spark programmers leverage the benefits of relational processing (e.g., Declarative queries,

[4]https://spark.apache.org/

optimized storage, creating UDFs), and lets SQL users execute map-reduce tasks via SQL. Spark SQL makes two main additions. First, it offers much tighter integration between relational and procedural processing, through a declarative DataFrame API that integrates with procedural Spark code. Second, it includes a highly extensible optimizer, Catalyst, built using features of the Scala programming language, that makes it easy to add composable rules, control code generation, and define extension points. Using Catalyst, we have built a variety of features (e.g., schema inference for JSON, machine learning types, and query federation to external databases) tailored for the complex needs of modern data analysis. We see Spark SQL as an evolution of both SQL-on-Spark and of Spark itself, offering richer APIs and optimizations while keeping the benefits of the Spark programming model.

2.4 Apache Tez

Tez[5] [36] is a framework for YARN-based, Data Processing Applications in Hadoop. Apache Tez is an extensible framework for building high performance batch and interactive data processing applications, coordinated by YARN in Apache Hadoop. Tez improves the MapReduce paradigm by dramatically improving its speed, while maintaining MapReduce's ability to scale to petabytes of data. Important Hadoop ecosystem projects like Apache Hive and Apache Pig use Apache Tez, as do a growing number of third party data access applications developed for the broader Hadoop ecosystem.

Apache Tez provides a developer API and framework to write native YARN applications that bridge the spectrum of interactive and batch workloads. It allows those data access applications to work with petabytes of data over thousands nodes. The Apache Tez component library allows developers to create Hadoop applications that integrate natively with Apache Hadoop YARN and perform well within mixed workload clusters.

Since Tez is extensible and embeddable, it provides the fit-to-purpose freedom to express highly optimized data processing applications, giving them an advantage over end-userfacing engines such as MapReduce and Apache Spark. Tez also offers a customizable execution architecture that allows users to express complex computations as dataflow graphs, permitting dynamic performance optimizations based on real information about the data and the resources required to process it.

- ➢ Execution Performance
  - Performance gains over Map Reduce
  - Optimal resource management
  - Plan reconfiguration at runtime
  - Dynamic physical data flow decisions

By allowing projects like Apache Hive and Apache Pig to run a complex DAG of tasks, Tez can be used to process data, that earlier took multiple MR jobs, now in a single Tez job as shown below.

[5] https://tez.apache.org/

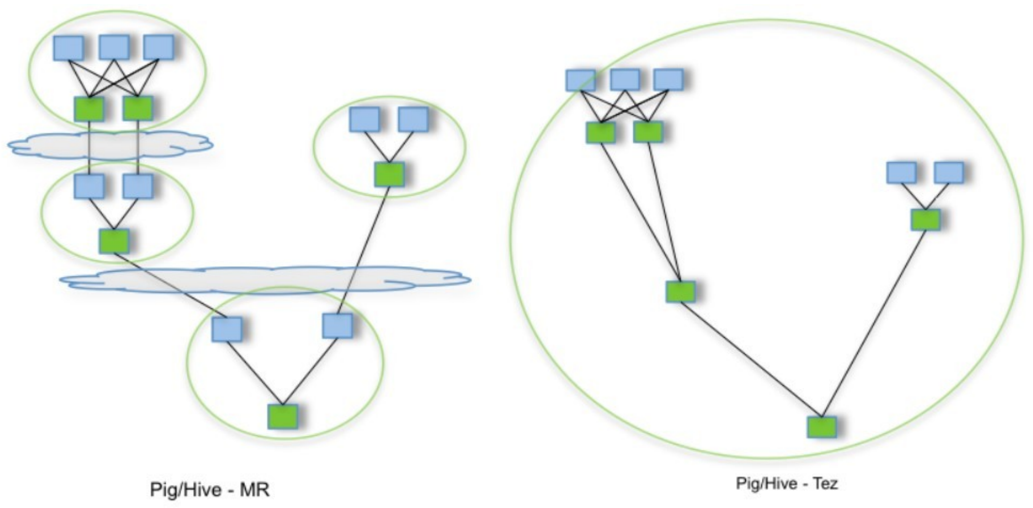


Figure 8: MR vs Tez data flow

2.5 Apache ORC

The Optimized Row Columnar (ORC)[6] [37] file format provides a highly efficient way to store Hive data. It was designed to overcome limitations of the other Hive file formats. Using ORC files improves performance when Hive is reading, writing, and processing data.

ORC files as part of the initiative to massively speed up Apache Hive and improve the storage efficiency of data stored in Apache Hadoop. The focus was on enabling high-speed processing and reducing file sizes.

ORC is a self-describing type-aware columnar file format designed for Hadoop workloads. It is optimized for large streaming reads, but with integrated support for finding required rows quickly. Storing data in a columnar format lets the reader read, decompress, and process only the values that are required for the current query. Because ORC files are type-aware, the writer chooses the most appropriate encoding for the type and builds an internal index as the file is written. ORC supports the complete set of types in Hive, including the complex types: structs, lists, maps, and unions.

ORC files are divided into stripes that are roughly 64MB by default. The stripes in a file are independent of each other and form the natural unit of distributed work. Within each stripe, the columns are separated from each other so the reader can read just the columns that are required. An ORC file contains groups of row data called stripes, along with auxiliary information in a file footer.

This diagram illustrates the ORC file structure:

[6] https://orc.apache.org/

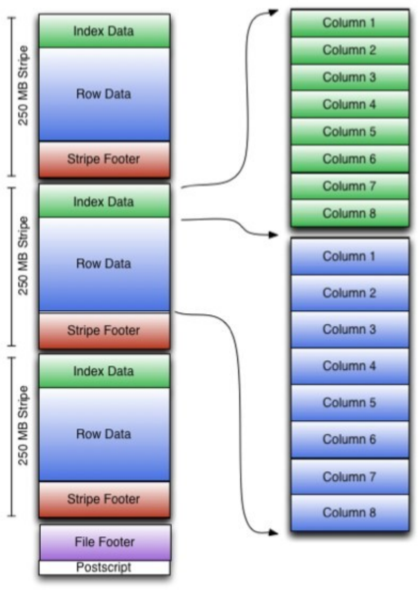


Figure 9: ORC file structure

As shown in the diagram, each stripe in an ORC file holds index data, row data, and a stripe footer. The stripe footer contains a directory of stream locations. Row data is used in table scans. Index data includes min and max values for each column and the row positions within each column. Row index entries provide offsets that enable seeking to the right compression block and byte within a decompressed block. Note that ORC indexes are used only for the selection of stripes and row groups and not for answering queries.

2.6 AIS Messages

The **Automatic Identification system (AIS)** [1] is an automatic tracking system that uses transponders on ships and is used by vessel traffic services (VTS). When satellites are used to detect AIS signatures, the term Satellite-AIS (S-AIS) is used. AIS information supplements marine radar, which continues to be the primary method of collision avoidance for water transport.

Information provided by AIS equipment, such as unique identification, position, course, and speed, can be displayed on a screen or an ECDIS. AIS is intended to assist a vessel's watchstanding officers and allow maritime authorities to track and monitor vessel movements. AIS integrates a standardized VHF transceiver with a positioning system such as a GPS receiver, with other electronic navigation sensors, such as a gyrocompass or rate of turn indicator. Vessels fitted with AIS transceivers can be tracked by AIS base stations located along coastlines or, when out of range of terrestrial networks, through a growing number of satellites that are fitted with special AIS receivers which are capable of deconfliction a large number of signatures.

The information contained in each AIS-data packet (or message) can be divided into the following two main categories:

- ➢ Dynamic Information, transmitted every 2 to 10 seconds depending on the vessel's speed and course while underway and every 6 minutes while anchored. A dynamic AIS messages includes:
  - Maritime Mobile Service Identity number (MMSI) - a unique identification number for each vessel station
  - AIS Navigational Status (e.g. 1=at anchor, 7=engaged in fishing, etc.)
  - Rate of Turn - right or left (0 to 720 degrees per minute)
  - Speed over Ground - 0 to 102 knots (0.1-knot resolution)
  - Position Coordinates (latitude/longitude - up to 0.0001 minutes accuracy)
  - Course over Ground - up to 0.1° relative to true north
  - Heading - 0 to 359 degrees
  - Bearing at own position - 0 to 359 degrees
  - UTC seconds - the seconds field of the UTC time when the subject data-packet was generated.

- ➢ Static & Voyage related Information is provided by the subject vessel's crew and is transmitted every 6 minutes regardless of the vessel's movement status:
  - International Maritime Organization number (IMO) - note that this number remains the same upon transfer of the subject vessel's registration to another country (flag)
  - Call Sign - international radio call sign assigned to the vessel by her country of registry
  - Name
  - Type (or cargo type) - the AIS ID of the subject vessel's ship type
  - Dimensions - approximated to the nearest metre (based on the position of the AIS Station on the vessel)
  - Location of the positioning system's antenna on board the vessel
  - Type of positioning system (e.g., GPS)
  - Draught - 0.1 to 25.5 meters
  - Destination
  - Estimated time of arrival - UTC month/date hours:minutes

3. RELATED WORK

In this section we are going to briefly describe other techniques or systems that are related to our work and explain their similarities and differences. We present indices for mobility data which is the basic structure for optimizing query performance, benchmarks that introduces the evaluation and functionalities of mobility databases and the principles of systems in big spatial(-temporal) era.

The quadtree [38] is a data structure appropriate for storing information to be retrieved on composite keys. We discuss the specific case of two-dimensional retrieval, although the structure is easily generalized to arbitrary dimensions. Algorithms are given both for straightforward insertion and for a type of balanced insertion into quad trees. Empirical analyses show that the average time for insertion is logarithmic with the tree size. An algorithm for retrieval within regions is presented along with data from empirical studies which imply that searching is reasonably efficient. We define an optimized tree and present an algorithm to accomplish optimization in n log n time. Searching is guaranteed to be fast in optimized trees. Remaining problems include those of deletion from quad trees and merging of quad trees, which seem to be inherently difficult operations.

R-tree [39] is the most efficient balanced index in spatial databases like PostGIS [40]. The 3D R-tree is a straightforward extension of the R-tree, treats time as an extra “spatial” dimension and it is the earliest method for indexing trajectories of moving objects. The default insertation of entire trajectories could lead to excessive dead space, most common solution for this issue is inserting segments of a trajectory in the index.

TB-tree [12] (Trajectory Bundle tree) maintains the ‘trajectory’ concept. TB-tree is a heightbalanced tree with the index records in its leaf nodes; leaf nodes are of the form (MBB , Orientation), where MBB is the 3-dimensional bounding box of the 3-dimensional line segment belonging to an object’s trajectory, a leaf node contains entries of a single trajectory only and Orientation is a flag used to reconstruct the actual 3-dimensional line segment inside the MBB among four different alternatives that could exist. Since, by definition, each leaf node contains entries of a single trajectory, the object identifier (id) needs to be stored only once, in the leaf node header. Like R-tree, internal and leaf node MBBs belonging to the same tree level are allowed to overlap. Each internal or leaf node in the tree corresponds to a physical disk page (or disk block, which is the fundamental element on which the actual disk storage is organized) and contains between m and M entries (M is the node capacity—fanout—and m in the case of TB-tree is set to 1). For each trajectory, a double linked list connects leaf nodes together to reconstruct the entire trajectory.

In [5], Theodoridis proposes a database schema and a set of ten benchmark database queries (current & past positions of moving objects) regarding the support of location-based services (LBS). No benchmark data or an experimental study is presented. The focus is on requirements of databases for location-based services and mobility data. The benchmark includes point and range selection queries on stationary and moving reference objects distance, k-NN, similarity-based queries and join queries. Benchmarks’s queries can be divided in three categories:

- Queries on stationary reference objects
- Queries on moving reference objects
- Join queries

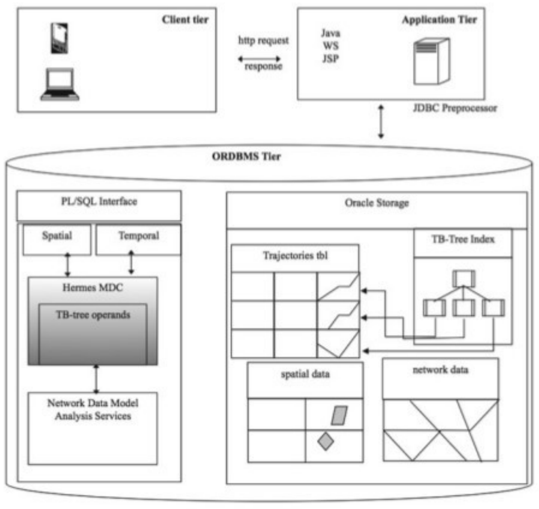
On the other hand, BerlinMOD [30] is more realistic since it is based on real MOD system, called SECONDO (to be overviewed later in this section). A data generator is implemented in order to create persons' trip (e.g. home-work) on road network. In this setting, an extensive set of various types of queries, non-spatial, simple spatial, sophisticated range, NN and other trajectory-oriented queries is proposed, which evaluate the functionality of a MOD engine. BerlinMOD queries (rephrased) with spatiotemporal characteristics are:

- Return objects that have ever been as close as 10 m or less to each other?
- Return objects that meet other objects based on spatiotemporal thresholds
- Return objects passed given points at a specific time
- Return objects travelled within a region during the time periods

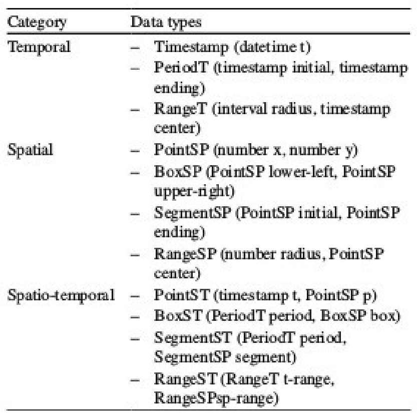
BerlinMOD experimental study it is focuses on range queries execution time with different setup.

We are going to briefly present systems with similar functionality with our proposed framework(approach). The systems can be categorized based on three criteria architecture, datatypes and query interface. Basically, the architecture of the system can be either centralized or distributed. Similarly, what datatype they can handle (store, index, query) spatial or spatiotemporal, as well as the query interface (language) can be varied in SQL or a classic programming language (i.e. Java or Scala).

Hermes [19] is a prototype system based on a powerful query language for trajectory databases, which enables the support of aggregative Location-Based Services (LBS). Hermes provides an SQL interface comprised of types, functions and operators that the user can combine in order to construct data and perform calculations on them.



(a)



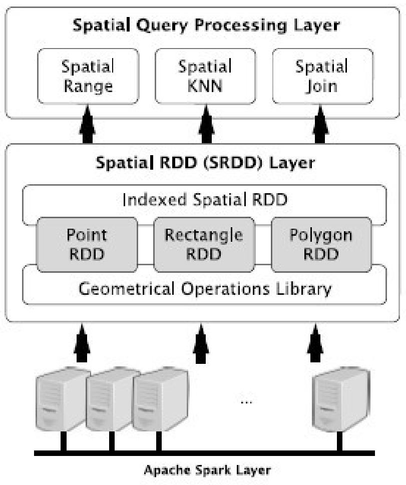
Category	Data types
Temporal	– Timestamp (datetime t)
	– PeriodT (timestamp initial, timestamp ending)
	– RangeT (interval radius, timestamp center)
Spatial	– PointSP (number x, number y)
	– BoxSP (PointSP lower-left, PointSP upper-right)
	– SegmentSP (PointSP initial, PointSP ending)
	– RangeSP (number radius, PointSP center)
Spatio-temporal	– PointST (timestamp t, PointSP p)
	– BoxST (PeriodT period, BoxSP box)
	– SegmentST (PeriodT period, SegmentSP segment)
	– RangeST (RangeT t-range, RangeSPsp-range)

(b)

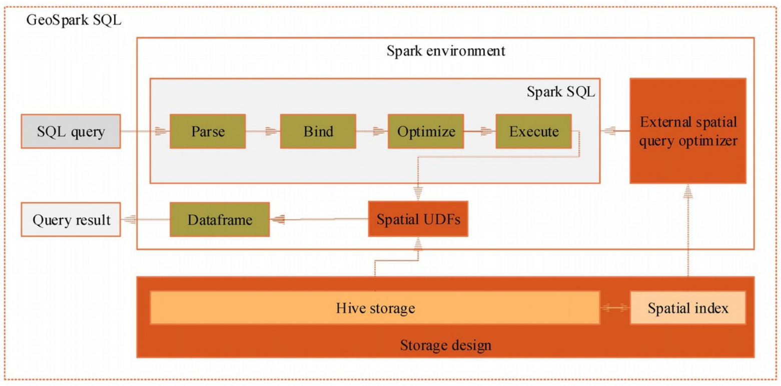
Figure 10: Hermes architecture and supported datatypes

SECONDO [32] is RDBMS behind BerlinMOD, especially adjusted to be extended by algebra modules. SECONDO is one of the first database system prototypes that can handle moving objects, that is, continuously changing time-dependent geometries. Its architecture consists of (i) a kernel, which offers query processing over a set of implemented type system algebras, (ii) an optimizer, which implements the essential part of an SQL-like language. Parallel Secondo [31] is constructed by coupling the Hadoop framework with a set of single-computer SECONDO distributed on the cluster. Its components work independently, nodes in Hadoop communicate with each other through its HDFS (Hadoop Distributed File System), whereas each SECONDO database processes database queries and exchanges intermediate data with the others through Parallel SECONDO File System. Through various PQC (Parallel Query Convertor) operators provided in the master database, a custom parallel query can be converted into a sequence of Hadoop jobs, being processed with a set of independent tasks running on all nodes simultaneously. These tasks are processed by data servers in parallel, complying with the runtime scheduling of the Hadoop framework.

Geospark [24] is an in-memory cluster computing framework for processing large-scale spatial data. GeoSpark is built on top of the Apache Spark and extends it with two layers: a) Spatial RDD (Resilient Distributed Datasets, the basic data model for Spark) Layer which is responsible for loading and supporting geospatial object, b) Spatial Query Processing Layer efficiently execute spatial query processing algorithms (e.g. Spatial Range, Join, KNN query) on Spatial RDDs. Geospark also allows users to create a spatial index (e.g. R-tree, Quadtree) that boosts spatial data processing performance in each SRDD partition. Geospark SQL [25] is the SQL extention of Geospark and is able to achieve real-time query processing. GeoSpark SQL provides a convenient SQL interface and achieves both efficient storage management and high-performance parallel computing through integrating Hive and Spark. GeoSpark SQL performs better when dealing with compute-intensive spatial queries such as the kNN query and the spatial join query



(a) GeoSpark



(b) GeoSpark SQL

Figure 11: GeoSpark frameworks architecture

STARK [26] is one of the first attempts that adds, more or less, spatiotemporal support in Spark. It offers spatio-temporal framework that aims to optimize queries for data sets with spatial and temporal components that is built on top of SPARK. We have to emphasize the fact that STARK supports spatio(-temporal) data modeling, but all its operators, its partitioners and indexing mechanism work with the spatial part of the data. Moreover, the BSP partitioner works only with the spatial length to create boxes/partitions inside the dataset MBB and the variable “max partition per cost” is useless because in fact was needed in case the data should be written on disk [9].

In [3] authors describe a spatial extension based on Exareme system [18] (master-slave model), to support large-scale spatial SQL queries and evaluate system's performance using the Jackpine Benchmark [11]. System is flexible and can support all spatial operators (e.g. intersects, contains, etc.) for all spatial data types. In order to achieve better performance, they implement a grid partitioner in the system. Moreover, each partition has a R-Tree index structure for achieving better performance. System's performance is quite stable and is comparable with STARK. On the other hand, this approach doesn't support operators for mobility data.

UlTraMan [27] is a flexible and scalable distributed platform for trajectory data management and analytics. UlTraMan is a Spark extension that provides a unified engine for both efficient data management and distributed computing and offering an enhanced computing paradigm in a highly modular architecture to enable both pipeline customization and module extension. UlTraMan uses Simba [23] spatial partitioner and create local indices (i.e. for each partition) and a global index that contains partitions MBR. UlTraMan integrates Chronicle Map [22] (an in-memory, embedded key-value store designed for low-latency) with Spark in order to provide efficient data access. Furthermore, data is stored in off-heap memory to relieve GC pressure, and the data is persisted at runtime through the support of simultaneous access from multiple processes.

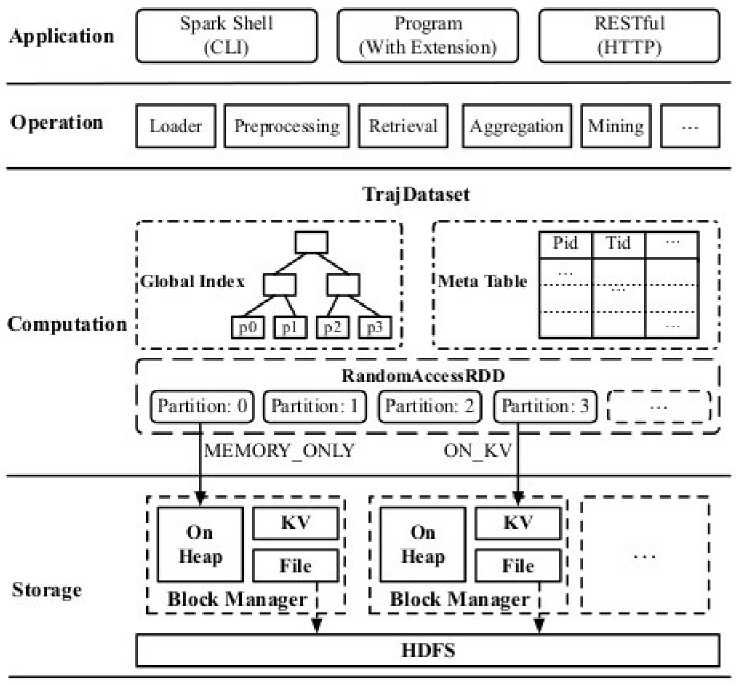


Figure 12: UlTraMan architecture

TrajSpark [28] offers a framework for processing moving objects data. TrajSpark introduce IndexTRDD, an RDD of trajectory segments, to support efficient data storage and management by incorporating a global and local indexing strategy. Two-level index layer with a global layer consisting of a three-level index layer and lower local hash indices. The global index is composed of a time range index where each range is indexed via a grid index and each grid is further indexed via a B+-tree. TrajSpark monitor the change of data distribution by importing a time decay model which alleviates the repartitioning overhead occurred in existing Spark-based systems and gets a good partition result at the same time.

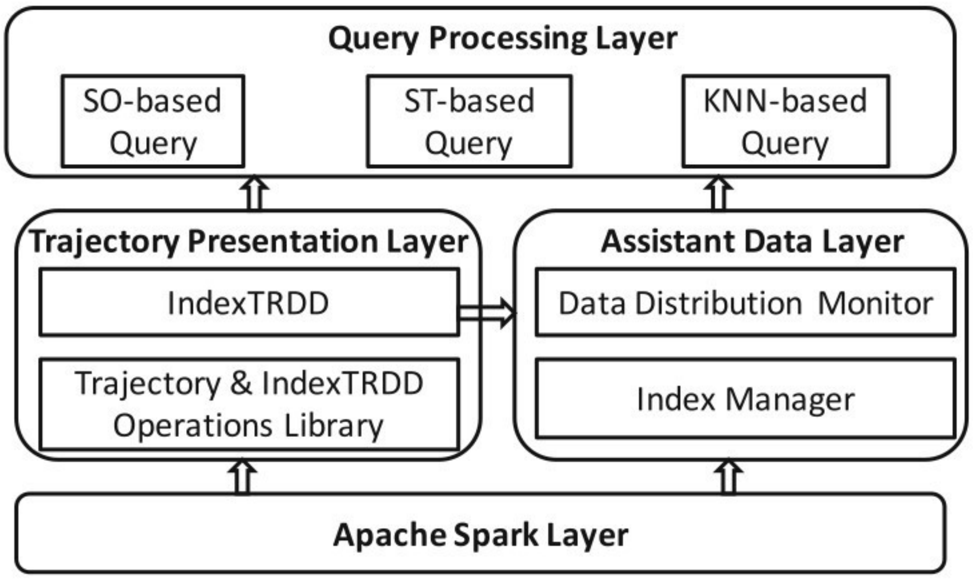


Figure 13: TrajSpark architecture

Tables below (1, 2, 3) classifies the above systems based on their main characteristics. Our implemetation will provide a big data SQL with JDBC connectivity solution for mobility data, which is a functionality none of the above systems provide.

Table 1: Systems Architecture

	Hermes	Secondo	TrajStore	GeoSpark	STARK	Exareme Spatial	UlTraMan	TrajSpark
Centralized	✓	✓	✓					
Distributed/ Parallel		✓		✓	✓	✓	✓	✓

Table 2: Data Types

	Hermes	Secondo	TrajStore	GeoSpark	STARK	Exareme Spatial	UlTraMan	TrajSpark
Spatial	✓	✓		✓	✓	✓		
Mobility	✓	✓	✓				✓	✓

Table 3: Query Language

	Hermes	Secondo	TrajStore	GeoSpark	STARK	Exareme Spatial	UlTraMan	TrajSpark
SQL	✓	✓		✓		✓		
Other		✓	✓	✓			✓	✓

4. THE BIGSQLTRAJ ARCHITECTURE PRINCIPLES

In this section we describe the architecture principles for our proposed system. At first, we introduce an end-to-end big data application architecture and then we explain how our implementation fits these standards. Our implementation, as the most big data engines, adopts the master-slave architecture.

A big data architecture is designed to handle the ingestion, processing, and analysis of data that is too large or complex for traditional database systems. Big data solutions involve one or more workload types:

- Batch processing
- Streaming processing
- Interactive exploration of big data.
- Analytics tasks

The following diagram shows the logical components that fit into a big data architecture.

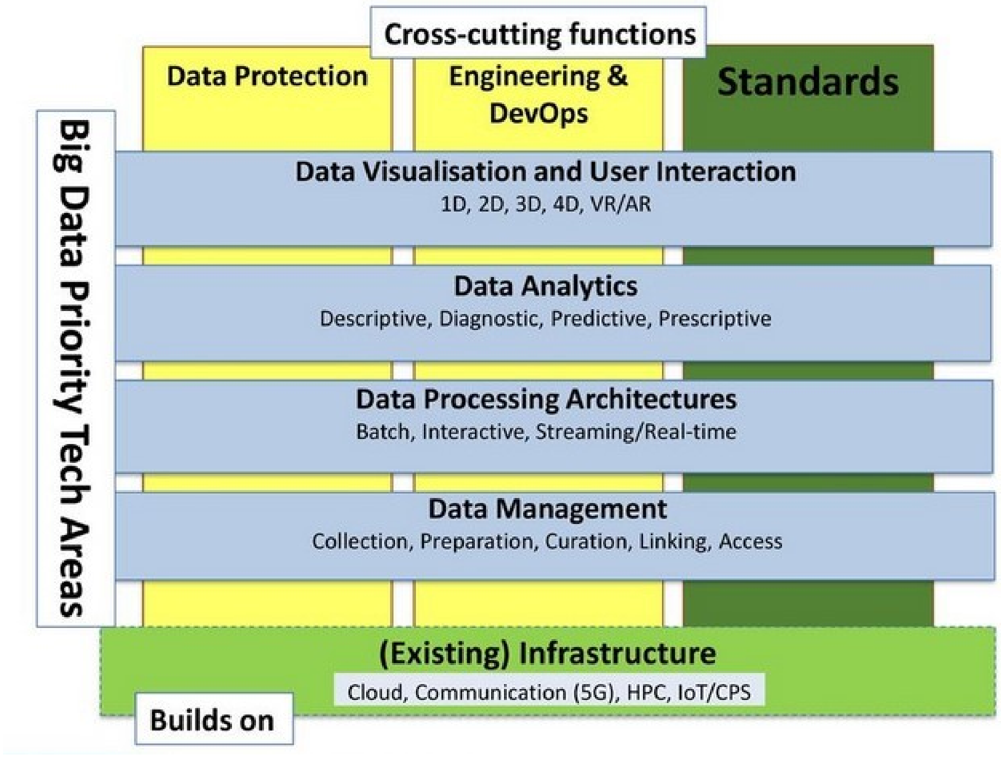

Cross-cutting functions
Data Protection
Engineering & DevOps
Standards
Big Data Priority Tech Areas
Data Visualisation and User Interaction
1D, 2D, 3D, 4D, VR/AR
Data Analytics
Descriptive, Diagnostic, Predictive, Prescriptive
Data Processing Architectures
Batch, Interactive, Streaming/Real-time
Data Management
Collection, Preparation, Curation, Linking, Access
(Existing) Infrastructure
Cloud, Communication (5G), HPC, IoT/CPS
Builds on

Figure 14: Big Data Value Reference Model (source: www.bdva.eu)

Data Management Layer: batch processing operations is typically stored in a distributed file store that can hold high volumes of large files in various formats. This kind of store is often called a data lake. A typical big data environment should convert data as needed and send it to the correct storage(partition) in the right format.

Data Processing Layer: datasets are so large, often a big data solution must process data files using long-running batch jobs to filter, aggregate, and otherwise prepare the data for analysis. Most of the times a MapReduce task is applied on this layer.

The features proposed by this thesis rely on these layers. The lack of mature storage data management layer is the main drawback of the Spark-based systems that are introduced in the previous section. UlTraMan and TrajSpark use extremely custom data loading workflows for parsing input data into a useful format, but they cannot easily upload new data if there was a streaming component. It is also important to notice that simple operators like join could double the program's structures and provoke memories issues. For example, a range query pruning the useless partitions, Spark will create a new data structure with the valid data and it will need extra resources. On the other hand, our proposed architecture is more flexible in the previous problems. Incoming data can be processed by Spark modules and then stored in the right format for the data processing layer. It would be easier for a streaming module to update partition index add new data to the storage layer rather than execute the complete workflow from the scratch. Moreover disk-based approach allowed us to retrieve and keep into memory only data that is necessary for process. We decided to create two different toolboxes because in this way we could select the best pipelines from each library.

We decided to create two different toolboxes because in this way we could select the best pipelines from each library and we can support different operations and functionalities. We used 3 Apache libraries, Hadoop, Spark and Hive. At first, Hadoop guarantees a mature storage system with Hadoop Distributed File System Component (HDFS) and through HDFS integration is a trivial task. Spark and Hive can read and write data on the HDFS using existing API and common file format like ORC. Furthermore, Apache YARN which is the Hadoop's resource manager, is responsible for task scheduling.

Figure 15 describes the data management and storing procedure. We used Apache Spark for this task because it is the most mature big data engine for handling batch and streaming data and store them on HDFS. In general, we can read data from multiple sources and then we use Spark's MapReduce framework in order to repartition mobility data based on spatiotemporal similarity and creating files that are accessed by Hive. Our implementation guaranteed load balancing.

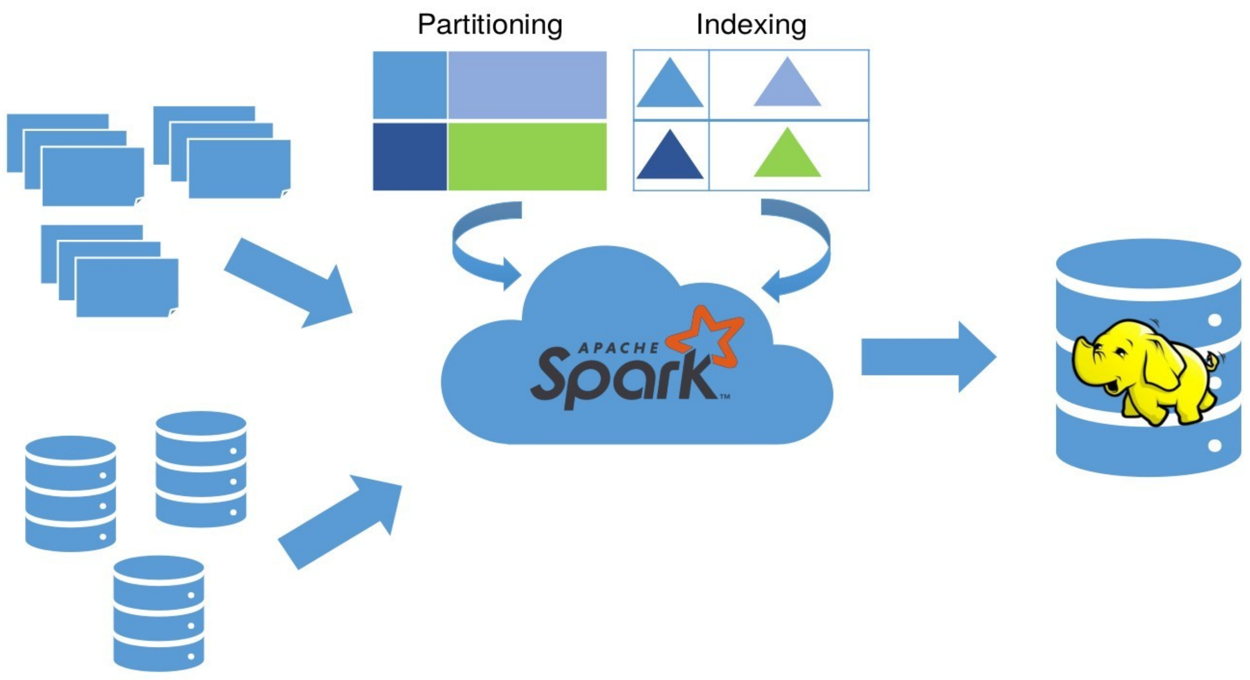


Figure 15: Spark SpatioTemporal Component

Figure 16 shows the architecture of the second toolbox that is built on-top of Hive and it is the querying mechanism of this thesis. We used Hive because it follows the SQL-standards with JDBC APPI and offers distributed SQL querying over HDFS by translating SQL queries into

MapReduce jobs. We implemented several Hive UDFS, that are described in the section 6 in order to support advanced data processing techniques for mobility data like bucket loading, filtering (range querying), index scan and kNN queries. Every application that supports JDBC API can interact with our platform.

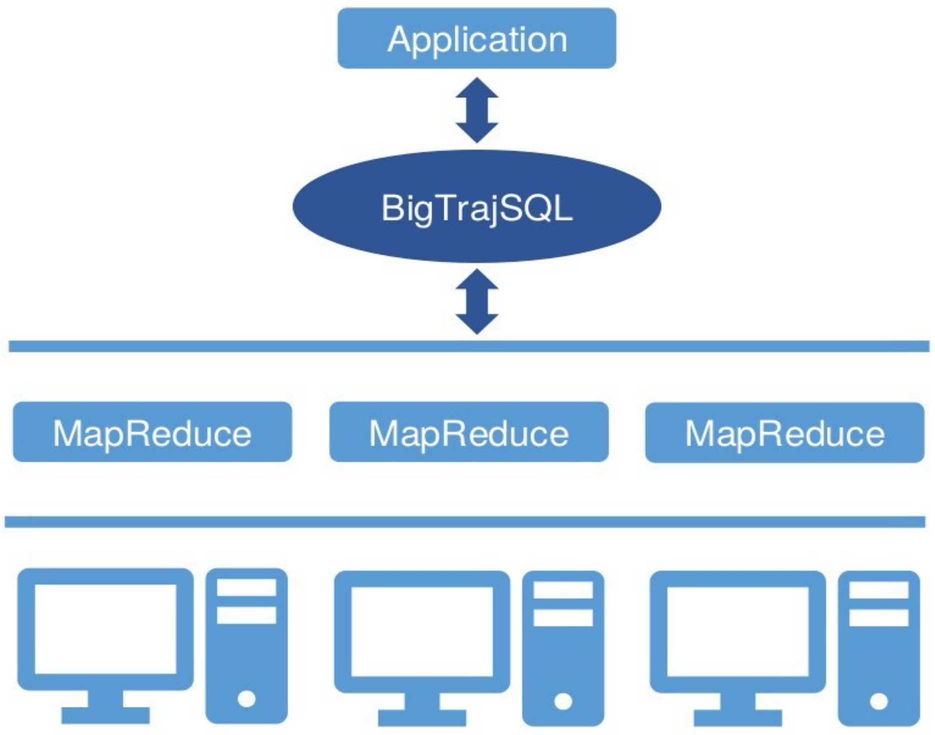


Figure 16: HiveQL SpatioTemporal Component

5. PREPROCESSING

Quality of data is an important aspect for evaluating the performance of spatiotemporal algorithms and data structures. Noisy data can lead to pointless results on the quality of the proposed techniques. One of the most recent noise elimination algorithms has been based on the spatiotemporal attributes of the objects in order to remove outliers. We focus on these filters and we describe our implementation in the next section

The first job of this thesis is to apply all the necessary steps in order to cleansed spatiotemporal points and create trajectories. A common strategy used to treat raw mobility data is described in the next figure. The first two steps are almost mutual for every preprocessing workflow and every type of data.

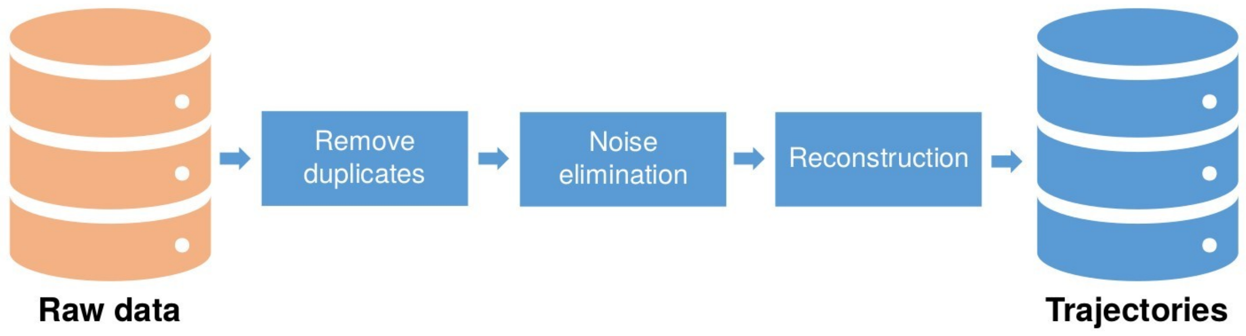


Figure 17: Preprocessing & Storage Workflow

Due to network failures, deserialization of the information, station coverage overlapping or no coverage common issues for raw mobility data include:

- ➢ more than one messages for the same vessel, time and position (duplicates)
- ➢ outlier positions (‘‘off-course position”)
- ➢ messages that received with delay (‘‘out-of-sequence”)
- ➢ gap in reporting

Definitely this a stream kind of procedure, but data cleansing is not part of this thesis. For this reason, cleansing filters are applied in a batch manner with Spark. Getting distinct messages is an easy task and the functionality provided by Spark. For noise elimination, adapted some filters based on spatiotemporal criteria such as speed and heading of the moving object. For example, positions with instantaneous velocity over 120 knots or with significant heading changes are deleted. Also, other important attributes are the sampling rate and the distance of the messages/positions, because based on the sampling rate we divided(reconstruct) object’s history into trajectories.

A spatiotemporal positions sequence <p_i, t_i> could be divided into independent trajectories based on spatial or temporal gaps that appear in the raw data (note that the sequence is clean in term of noise). We use two attributes in order to identify trajectories:

- ➢ Spatial gap: the upper limit distance between two successive time−stamped positions of the same object so that they could be treated as the same trajectory.

➢ Temporal gap: the upper limit elapsed time between two successive time−stamped positions of the same object so that they could be treated as the same trajectory.

Having set S gap and T gap threshold, we have to iterate over the objects history in order to find those points $<p_{i-1}, t_{i-1}>$ and $<p_i, t_i>$ that they overcome the defined thresholds. Then, we identify $<p_{i-1}, t_{i-1}>$ as the ending point of the existing trajectory and $<p_i, t_i>$ as the starting point of a new trajectory of that moving object.

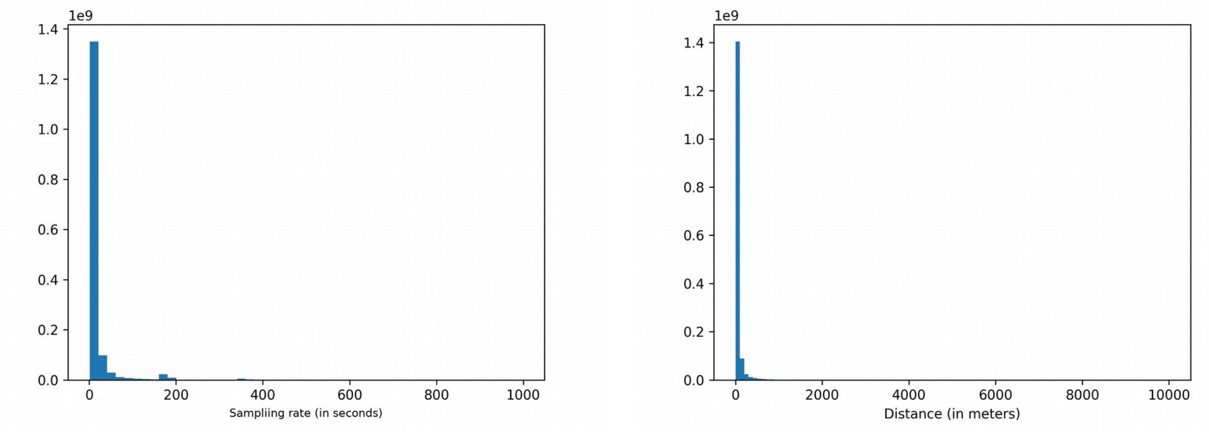
Obviously, setting these parameters is application-dependent and they should produce reasonable results.

For this task, we used off-the-shelf functions that are implemented by Spark. In particular:

1) partitioning data based on the objects' id and order messages by timestamp
2) for each message find its lead message and calculate speed, heading, sampling and distance
3) filtering outliers and splitting trajectory if it's necessary

Apart from the fact that these steps are executed in batch manner their migration to a streaming engine (thus they fit in a big data architecture) is brute force because they just need rely on the previous point, so typical windows operation can reproduce the exactly same procedure.

The histograms below are occurred from the thesis' dataset (IMIS dataset) and helped us define the spatial and temporal thresholds.



(a) Time elasped (b) Points Distance

Figure 18: IMIS dataset Statistics

We set spatial threshold 2000m and elapsed time threshold 400s. The partitioning algorithms and querying algorithms from the next section take as input trajectories that are created/constructed based on these thresholds. 1934322863

We set spatial threshold 2000m and elapsed time threshold 400s. The partitioning algorithms and querying algorithms from the next section take as input trajectories that are created/constructed based on these thresholds.

We conducted experiments against a real AIS dataset containing almost 100GB of AIS messages (i.e. 1934322863 positions) spanning from 1 March 2007 to 31 March 2010 for N = 20206 distinct vessels in the Aegean, the Ionian, and part of the Mediterranean Sea. Most vessels were frequently sailing, e.g., passenger ships or ferries to the islands. The cleansed output contains 1536743282 points. We create 5271228 trajectories based on the defined thresholds and we remove trajectories with a single point Table 4 summarizes some basic statistics about the input dataset.

Table 4: Dataset Description

Raw points	1934322863
Cleansed points	1536743282
Trajectories	5271228
Average Length	10821.57 meters
Average Duration	3200 seconds
Average Sampling(trajectory)	92 seconds
Average Points(trajectory)	174

6. IMPLEMENTATION

In this section, we represent all the steps for creating the trajectories database. It is the full implementation of the data management layer that is the most crucial factor for querying effectiveness and next the query processing toolbox pipelines.

6.1 Partitioning and Indexing Mechanism

The implementation workflow starts from data storage and management based on Apache Spark library. As described in Section 4 we have created methods for spatio-temporal partitioning and indexing mobility data. At first, we present spatiotemporal partitioner steps and then the features of the local indices.

6.1.1 Spatio-temporal Partitioners

After creating trajectories for all the dataset, we have to store them in the proper way thus we take advantage of the distributed storage and processing API mechanism in order to accomplish further optimizations. Achieving this task, we should rely on mobility attributes of the trajectories and extending current spatial partitioners into spatiotemporal ones because all the system described in Section 2 are implement or apply indexing, and thus partitioning, only in spatial dimension and ignoring time. Moreover, we need our local indices, index for each partitioned data, supporting mobility data. So, we extend geospatial STRtree provided from JTS library. To describe better the changes, we have introduced at the pure geospatial indices, we deem it appropriate to briefly describe the default (spatial) algorithms.

As already mentioned, our proposed partitioned are based on spatiotemporal characteristics. At first, we extended STARK spatial partitioner and create a *3D equal grid partitioner*, which takes as input the dataset's MBR and split it into 3D cells based on user defined ranges for each dimension. The amount of 3D cells that has to broadcast to all workers and the execution time and the complexity O(N) for assign a trajectory into a partitioner are the main drawback of this solution. Subsequently, we extend quad-tree algorithm in order to handle 3D data. This is so called *Octree* and it's a straight forward extension from quad tree. The main differences can be found in node data type that is 3D minimum bounding box and in subdividing algorithm that produces eight children, but in other ways the fundamentals of the algorithms are the same. Octree algorithms takes as input the dataset'ss MBR and a subset of the trajectories database. Obviously, it is important the sample dataset follow the same statistics (e.g. length, duration) as the full dataset. Getting a representative subset of a dataset it is not a part of this thesis, but the off-the-shelf Spark sampling algorithm returns quite good results and as shown in figures(histograms) below we are able to partition the full dataset with almost equally partitions (fit normal distribution - load balancing). This is a crucial assumption that of our implementation because we need near trajectories, in spatial and temporal dimensions, to be stored in the same partition. Systems that were presented in the section 2 are using pure spatial partitioners. Only TrajSpark uses quadtree with trajectory id or time as extra feature for partitioning, but also split a trajectory into its points.

More or less both partitioners have the same functionality and the main difference is detected in sharing candidate partitioners' MBR and searching for partitioner.

We used the mean point of each trajectory to assign it to partition, because especially in maritime data the middle point of the trip (mean point) is the most representative of the trajectory. Obviously, someone could use more than one point (e.g. start point, mean point and end point) to assign a trajectory to a partition or could replicate trajectories to every partition rectangle that trajectory crosses for better distribution, but this strategy could occur an explosion for the needed resources. So, due to the hardware limitation and because our current implementation fit to the normal distribution, we use only the mean point of trajectories to partition the dataset.

Algorithm 1 describes the full pseudocode of our Octree partitioner implementation, including sampling, partitioning, creating local and global indices. Moreover, figure 19 shows a working example of the partitioning step only and figure 20 show a running example and the difference between quadtree and the octree implementation based on three days sample of the full dataset.

Algorithm 1 Repartition Algorithm

```
Input: max_level, max_item_per_node ,trajectories_sample
Output: repartitioned trajectories, local indices, global index
1: traj_dataset ← load(trajectories)
2: mbr ← trajectories_dataset.getMBR()
3: subset ← traj_dataset.getSample(sample)
4: partitions_mbr ← octree (mbr, subset, max_level, max_item_per_node)  //could be 3dGrid
5: repartition ← traj_dataset.map(assign(trajectory, partitions_mbr)).repartition()
6: local_indices ← repartition.createPartitionIndex()  //fit partitioner in order to contains the full trajectories of the partition
7: global_indices ← createIndex(local_indices.getRoot())  //create index from each partition MBR
8: store(repartition)
9: store(indices)
```

Algorithm 2 Octree Algorithm

```
Input: MBR, trajectories, max_level, max_item_per_node
Output: octree
```

```
1:  octree.root ← MBR
2:  while trajectories.hasNext() do
3:      traj ← trajectories.nect()
4:      traj_mbr ←traj.getMBR()
5:      leafNode ←octree.insert(traj_mbr)
6:      if leafNode.items > max_item_per_node & leafNode.level < max_level then
7:          octree.split(leaf_node)
8:      end if
9:  end while
10: return octree
```

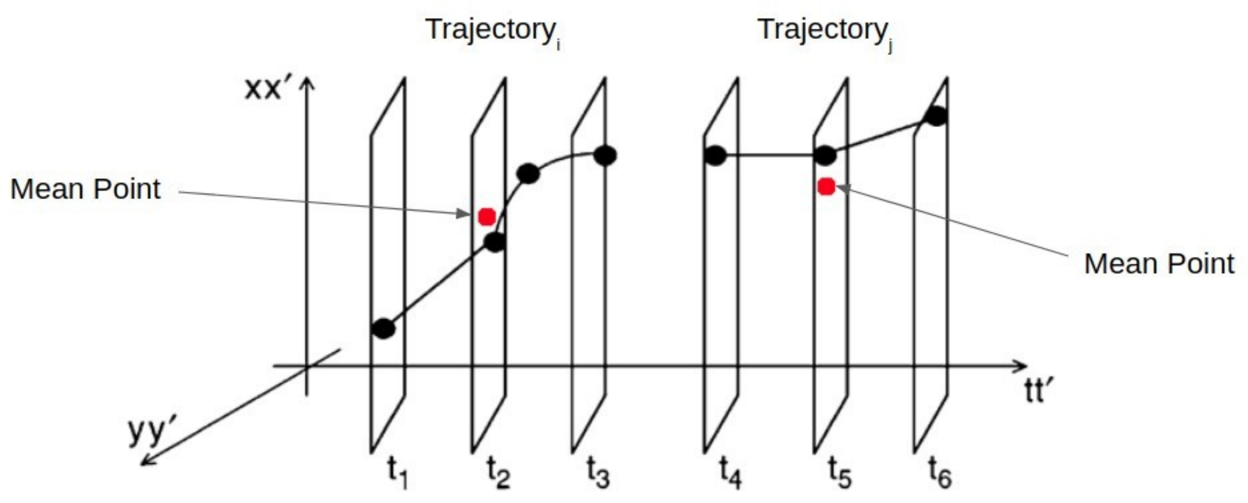
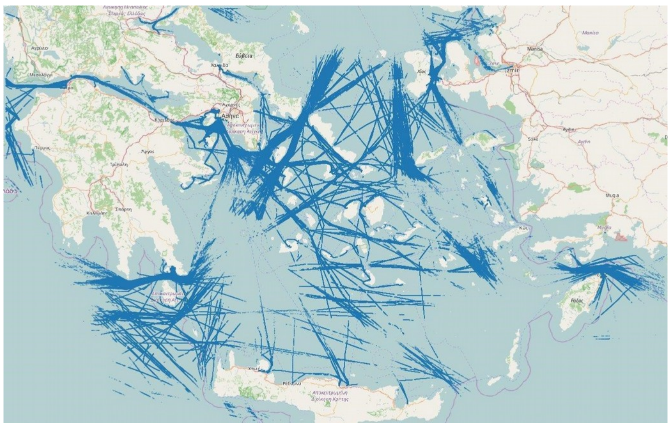
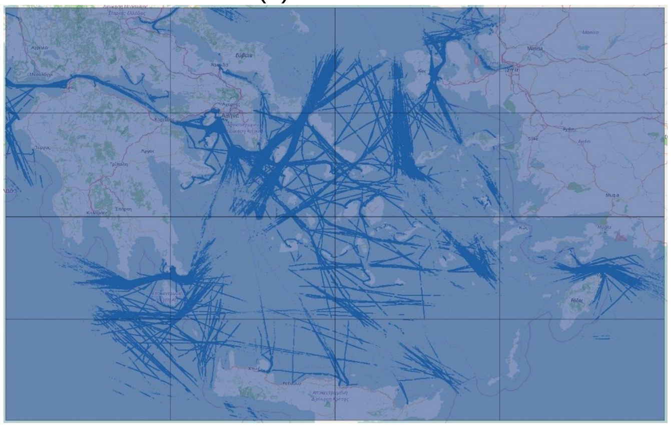


Figure 19: Trajectory Partition

The next figures show that our proposed implementation achieve load balancing among partitions, even the different parameter setup. It is noteworthy that we were forced to implement spatiotemporal partitioners due to memory failures of the spatial partitioners. Pure repartitioning based on only spatial characteristics lead to huge volume of data for each worker and inbalanced partitions that were over cluster resources. In better of our knowledge our implementation is the first proposing spatiotemporal partitioners since UlTraMan used SIMBA’s spatial partitioner and TrajSpark used two-phase pruning based on B-tree for time dimension and a spatial tree for partitioning the data. Furthermore, our preliminaries results showed small number of partitioners gave us better performance, but cluster resources constraints have compelled us to run our benchmark with almost 4000 partitioners (Figure 21)


(a) Raw Data


(b) QuadTree Result

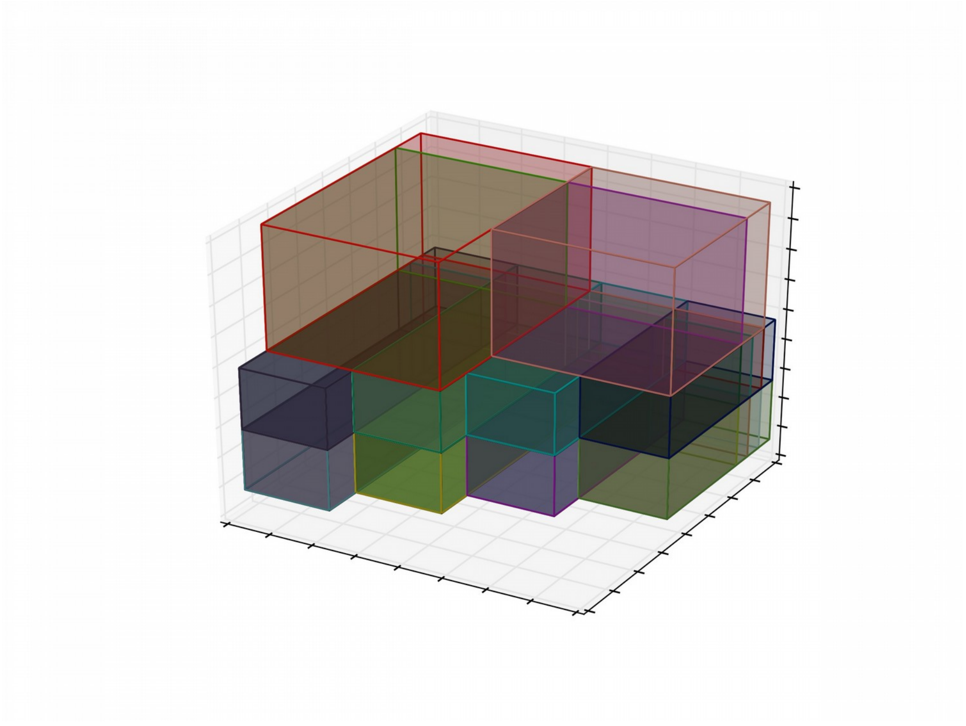
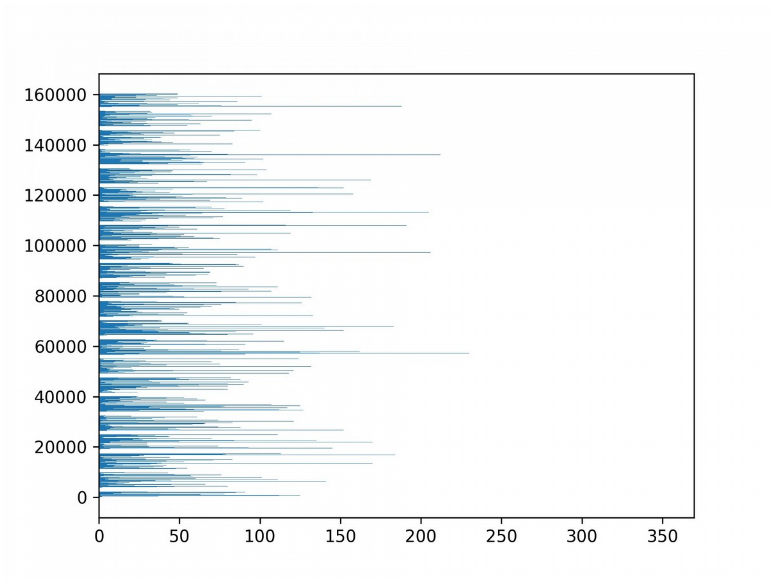
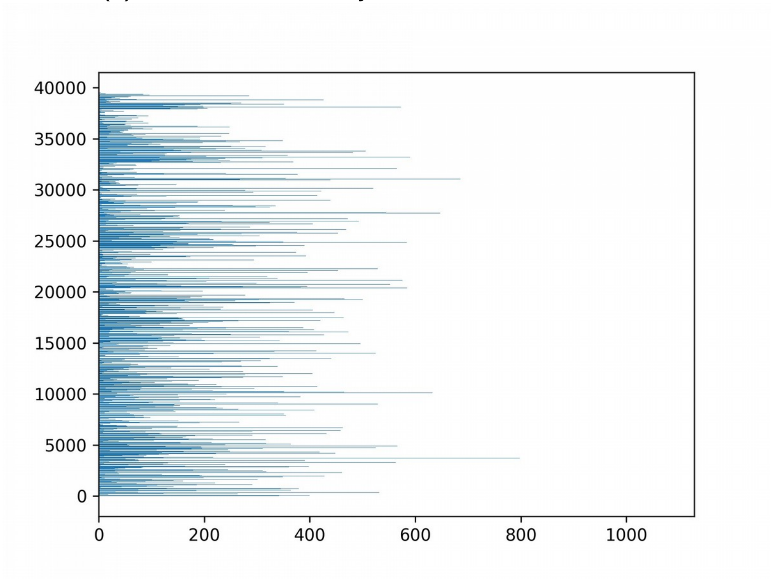

(c) Octree Result

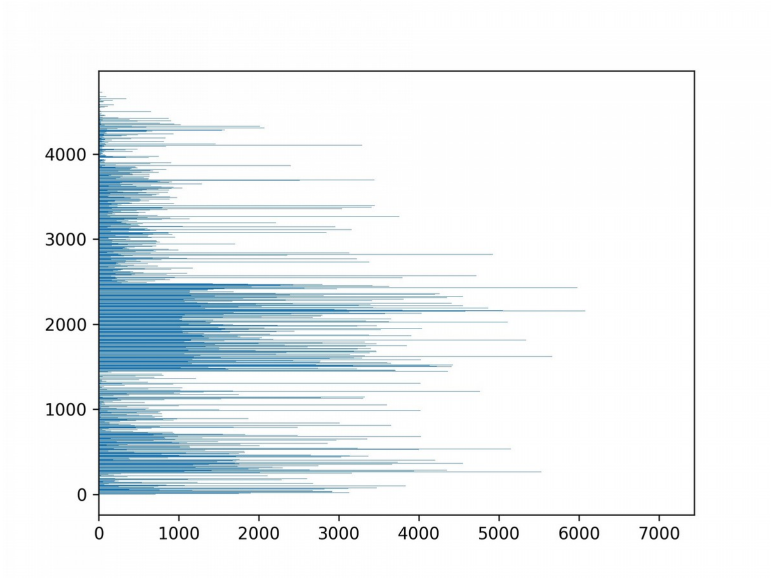
Figure 20: IMIS 3 Days Spatiotemporal Partitioner Running Example



(a) Octree max item by node: 20 & max level: 30



(b) Octree max item by node: 40 & max level: 15



(c) Octree max item by node: 360 & max level: 30

Figure 21: IMIS Dataset Spatiotemporal Partition Results

6.1.2 Spatio-temporal Index

The next step after partitioning is the indexing of the data. In order to achieve local optimization, namely speed-up query execution time when a worker processing a subset of the data is required a local index mechanism. In our implementation we extend spatial STR-tree from the JTS library in order to use time for sorting. Actually, we use time and spatial characteristics when build STR-tree and create its nodes. We decided to extend this library because we wanted to support spatial and spatiotemporal data and simultaneously be compatible with most of the existing systems and using one implementation. JTS library and the features we added achieve this goal. Apart from adding time-spatial sorting, we added method in order to support mobility query processing. More specifically, we implemented algorithms for range queries with 3D MBR intersects over tree structure and also implemented MINDIST tree search between trajectories and 3D MBR so that we can execute kNN queries. MINDIST is equal with zero if entities overlap else is the distance between the closest points. These methods allowed us to search a partitioner's data with logN complexity. Algorithm 3 and Algorithm 4 describes in pseudocode the steps for range queries and nn queries taking advantage of a tree structure. Both algorthims using a depth-first search mechanism in order to explore possible candidate for the final result. Algorithm 3 use only overlap for continuing the depth first search (non-leaf node) or adding result (entry in leaf node) for output. On the other hand, Algorithm 4 use MINDIST measurement for continuing the depth-search (non-leaf node) and call a different distance function (i.e. DTW or LCSS for mobility data) for assigning result (entry in leaf node) to output. In our implementation we use these algorithms with minimal changes.

Algorithm 3 Index Range Query Algorithm

```
    Input: tree, query
    Output: candidate_set
 1: stack.add(tree.getRoot())
 2: candidate_set ← ∅
 3: while stack.nonEmpty() do
 4:     node ← stack.pop()
 5:     if node is not leaf node then
 6:         if node overlaps q then
 7:             stack.push(node)
 8:         end if
 9:     else
10: for each entry ∈ node do
11:             if entry overlaps q then
```

```
12:             candidate_set.add(e)
13:         end if
14:     end for
15:   end if
16: end while
17: return candidate_set
```

Algorithm 4 Index NN Query Algorithm

Input: tree, query, mindist_threshold

Output: nn_result

```
1: stack.add(tree.getRoot())
2: nn_result ←∅
3: nearest ← INFINITY
4: while stack.nonEmpty() do
5:   node ← stack.pop()
6:   if node is not leaf node
then
7:     if node.distance(query) < mindist_thershold then
8:       stack.push(node)
9:     end if
10:  else
11:    for each entry ∈ node do
12:      if entry.distance(query) < nearest then
13:        nearest ← entry.distance(query)
14:        nn_result ← entry
15:      end if
16:    end for
17:  end if
```

```
18: end while
19: return nn_result
```

Creating Hive tables is our final goal. Partitioning and indexing mechanism is the key factor for having optimized tables that could be efficient processed by execution engine. After finishing the data management workflow, we ended up with the following schema:

Table 5: Hive tables

Name	Columns	Description
Global Index	tree binary	one row table, easily broadcast
TrajectoryIndex	partitionID int, index[trajectories] binary	bucketed by partitionID, each row contains trajectories for the partition in tree encoding
Trajectories	objID bigint, trajectory array, rowID bigint, partitionID int	cleansed and rebucketed trajectories, bucketed by partitionID or rowID, each row contains a trajectory,
MBRIndex	partitionID int, index[trajectoriesMBR] binary	bucketed by partitionID, each row contains mbrs of trajectories for the partition in tree encoding

6.2 Mobility Queries Toolbox

The Scalable Big Data Operation Component - Big Data Processing (BDP) Toolbox provide parallel and distributed query processing techniques for spatiotemporal query languages (i.e. the Distributed Complex Query Toolbox). The efficient indexing techniques using sophisticated partitioning and access algorithms enable the operators to perform complex and demanding operations over big amount of data in spatiotemporal dimension.

In order to support access to the distributed storage layer, a set of primitive query operators (UDF) have been implemented that operate over highly distributed data. Examples of such operators include: scan, index-scan, filter(range-queries), distance-join, kNN queries.

BigSQLTraj operators have been designed in an abstract way, thus achieving two goals at the same time:

1) easily executed via pure SQL commands and JDBC technologies without required any knowledge of MapReduce programming

2) every system that read data from HDFS can reproduce the same functionality (epsecially index scan) by only translate existing UDFs (pure Java that use custom serializers and deserializers for indices)

The most important queries for every mobility data application are range queries and similarity queries and its alternatives. Location-based services, map matching even predictive analytics algorithms have steps that need a fast pruning processing or find distance between trajectories, that's why this thesis focus and optimize these types of queries. Our core optimization relies on the index and partition that are described in the previous section. Custom deserializers have been implemented in order to access global and local indices. Each query (range or kNN) firstly scan the global index with the proper way and then process the full dataset. Algorithm 3 and Algorithm 4 present the step for processing index-based queries respectively.

6.2.1 Extension of Hive with trajectory operators

In this section we describe implemented Hive UDFs their input parameters, a briefly description and the output result. These UDFs extend Hive with support for mobility data. It is important to note that Hive doesn't provide any API for creating new data types or indexing structures. Every optimization in this thesis is achieved based on the below operators. Especially index-based operators achieving effectively pruning for search space.

- **ST_Intersects3D**
  - Input: trajectory and MBR or MBR and MBR, tolerance variables used to extent MBR
  - Returns true if objects spatiotemporal intersects

- **ST_IndexIntersects**
  - Input: STmbr, tree structure, tolerance variables used to extent MBR.
  - Using Algorithm 3, scan Global Index or MBRIndex table 5
  - Returns id for trajectories that intersects with STmbr
- **IndexIntersectsTraj** ◦ Input: STmbr, tree structure, tolerance variables used to extent MBR.
  - Using Algorithm 3, scan TrajectoryIndex table 5
  - Returns id & trajectories that intersects with STmbr
- **MbbConstructor**
  - Input: trajectory or user defined mbr, tolerance variables used to extent MBR.
  - Returns MBR as Hive data type
- **TrajBoxDist**
  - Input: trajectory and Stmbr
  - Returns zero if trajectory overlaps STmbr else the distance between closest points
- **DTW**
  - Input: trajectory, trajectory, w (algorithm parameter for pruning distance matrix), distance function (i.e. Euclidean, Manhattan, Haversine), parameters for spatio-temporal tolerance
  - Returns distance between the input trajectories if they temporal overlap
- **LCSS**

- Input: trajectory, trajectory, distance function (i.e. Euclidean, Manhattan, Haversine) parameters for spatiotemporal tolerance
- Returns distance (dissimilarity) between the input trajectories

➢**IndexTrajKNN**

- Input: query_trajectory, tree, mindist therhold, temporal tolerance
- Using Algorithm 4, scan Global Index or MBRIndex table 5
- Returns id and trajectory which are candidate NN with the query_trajectory

➢**IndexStoreTrajKNN**

- Input: query_trajectory, tree, mindist therhold, temporal tolerance
- Using Algorithm 4, scan TrajectoryIndex table 5
- Returns id trajectory and distance which are candidate NN with the query_trajectory

➢**ToOrderedList**

- Input: id_i, $traj_i$, k, distance (i.e calculated DTW or LCSS result) input based on IndexTrajKNN, IndexStoreTrajKNN

- Returns sorted by distance kNN trajectories

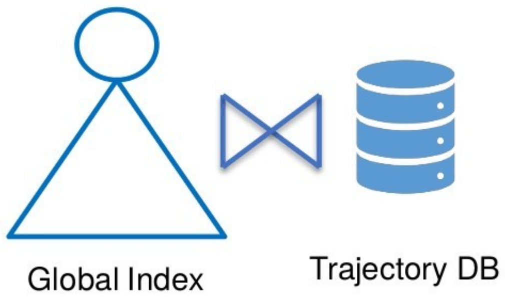
6.2.2 BigSQLTraj supported queries:

In this section we describe SQL queries of our implementation and how a user could execute range and kNN queries with our proposed platform. Hive table are already described table 5. There are three alternatives for each query how processing the full dataset except brute force scan:

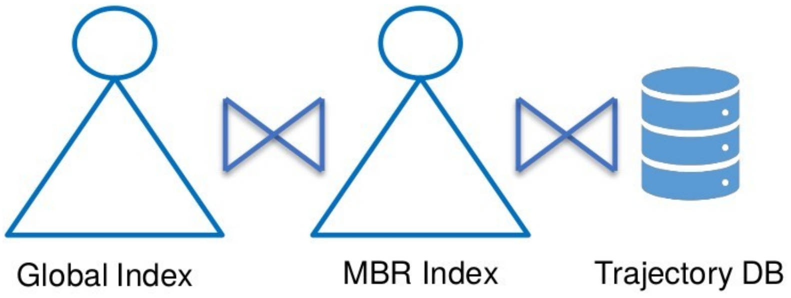
1) data have been bucketed by partition id, thus a simple search in the global index is required and then a join with the trajectories table
2) data have been bucketed by trajectory id, thus a search in the global index and a search in the local index that only contain trajectories id are required and then a join with the trajectories table

3) data have encoding via STR-Tree, thus a simple search in the global index is required and then a join with the local indices that have stored with the entire trajectories (not only with their MBRs).

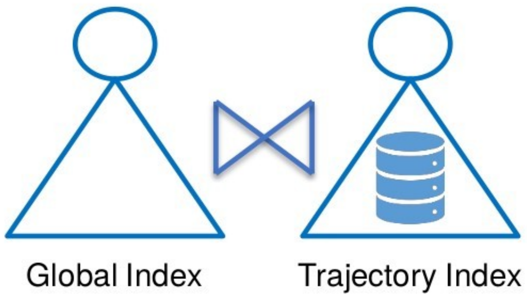
Figure 22 describes the pipelines for each execution plan. Each transition from one table to another is achieved vi join operator. It is significant remarkable that our partitioning algorithm and the management plan combined with Hive's of the self tables and joins optimizations gave us extra speed up for querying execution time.



(a)



(b)



(c)

Figure 22: BigSQLTraj Execution Plans

Methods for accessing each data structure based on the required algorithm are presented in the previous section 6.2.1. More or less execution plans are the same for both queries (range and kNN) because they depend in the same structures and almost same algorithms 3, 4.

Tables 6 and 7 present the queries that an end-user can execute in our proposed system. A short description is provided in order to link queries with data management layer and execution plans 22. In case of range queries Q is always a 3D MBR and Intersects=[ST_Intersects3D, ST_IndexIntersects, IndexIntersectsTraj] depends on querying table. In case of range queries Q is always a trajectory, knn=[IndexTrajKNN, IndexStoreTrajKNN] depends on querying table, distance=[DTW, LCSS] and knn_result=ToOrderedList.

Table 6: Range Queries

Query	Description
SELECT * FROM trajectories WHERE Intersects(**Q**, trajectory)	Brute force range query. Scan all database
SELECT * FROM trajectories INNER JOIN (SELECT Intersects(**Q**, tree) FROM global_index) AS t ON (partitionID) WHERE Intersects(**Q**, trajectory)	First query Global Index for finding partitions that intersects with range query, then execute the range query on trajectories table that is bucketed by partition id [5]. Join global index with trajectories table. Execution plan (a).
SELECT * FROM trajectories INNER JOIN (SELECT Intersects(**Q**, tree) FROM mbrindex INNER JOIN (SELECT Intersects(**Q**, tree) FROM global_index) as p ON (partitionID)) AS t ON (rowID) WHERE Intersects(**Q**, trajectory)	First query Global Index for finding partitions that intersects with range query, then execute the range query on MBRIndex table for candidate intersects with Trajectories tables. Finally execute range query only for candidate trajectories [5]. Join Global Index, MBRIndex, Trajectories tables. Execution plan (b).
SELECT Intersects(**Q**, tree) FROM TrajectoriesIndex INNER JOIN (SELECT Intersects (**Q**, trajectory) FROM global_index) AS t ON (partitionID)	First query Global Index for finding partitions that intersects with range query, then execute the range query on TrajectoriesIndex table [5]. Join Global Index with TrajectoriesIndex table. Execution plan (c).

Table 7: kNN Queries

Query	Description

SELECT *
 FROM (SELECT rowID$_i$, t$_i$, **Q**, distance(t$_i$, **Q**)
 FROM trajectories) AS temp
 ORDER BY distance LIMIT k
	Brute force kNN query. Scan all database.

SELECT rowID$_i$, knn_result FROM (SELECT rowID$_i$, t$_i$, **Q**, distance(t$_i$, **Q**) FROM trajectories INNER JOIN (SELECT knn(**Q** , tree) FROM global_index) AS t) AS final ON (partitionID) GROUP BY rowID$_i$	First query Global Index based on MINDIST for finding partitions with candidate similar trajectories [5], then execute similarity function for all candidates and order the result and return kNN. Execution plan (a).
SELECT rowID$_i$, knn_result FROM (SELECT rowID$_i$, t$_i$, **Q**, distance(t$_i$, **Q**) FROM (SELECT knn(**Q**, tree) FROM index INNER JOIN (SELECT knn(**Q**, tree) FROM partition_index) AS p) AS t ON (partitionID)) AS temp)) AS final GROUP BY rowID$_i$	First query Global Index based on MINDIST for finding partitions that has candidate kNN trajectories, then execute the kNN search on MBRIndex table for candidate intersects with Trajectories tables [5]. Finally calculate distance between query and candidate kNN trajectories, order the result and return kNN. Join Global Index, MBRIndex, Trajectories tables. Execution plan (b)

SELECT rowID$_i$, knn_result
 FROM (SELECT knn(**Q**, tree)
 FROM indexTrajectories INNER JOIN
 (SELECT knn(**Q**, tree) FROM partition_index) AS t
 ON (partitionID)) AS final GROUP BY rowID$_i$
	First query Global Index based on MINDIST for finding partitions that has candidate NN trajectories, then execute the kNN query on TrajectoriesIndex table and order the result and return kNN [5].
	 Join Global Index with TrajectoriesIndex table.
	 Execution plan (c).

7. EXPERIMENTAL EVALUATION

In this section, we present the results of our experimental study. After creating our cluster in ∼Okeanos[7], we were going to test our benchmark scenario with several different configurations. One by one we installed and ran all clustered engines. While running benchmark used Yarn as the resource management platform. Spark, Tez and Hive have been installed over Hadoop ecosystem. We configured hadoop so that it could used all the available resources from the cluster. All results in the next paragraph are conducted with the following resources:

- ∼Okeanos cluster with 15 slave nodes and one master node. Each node has 60gb disk, 8gb memory and 8 CPU
- Hive stable version 2.3
- Spark version 2.3
- Hadoop version 2.7.3
- Tez version 0.9

A real world dataset is employed to study the query performance of BigSQLTraj. Datasets description and statistics are presented on table 4 in section 5. The main characteristics are:

- 1536743282 points almost 100GB
- 5271228 trajectories

7.1 Experimental Setup

Partitioned trajectories dataset and its indices are stored as ORC files and they were bucketing into 30 buckets. The bucketing key is different per case, because as shown in the previous section the join key is different, so sometimes the bucketing key is the trajectory id and other the partition id. Partitioned tables can optimize filtering queries, but in our case, we need fast joins. Hive provides join optimization for bucketing tables and especially with bucketing and sorting by bucket key tables (sort bucket join). Moreover, it is crucial to clarify that buckets number (i.e 30 buckets) should be some dozens for better performance. Hive can improve its performance (faster execution) for join queries when someone increase buckets number, but there is an upper limit. If this limit is overcome performance getting worse. So, our empirical experiments shown that a range between 15-60 buckets achieve the best performance.

TrajSpark evaluates performance of the system on a 12-node cluster running standalone Spark 1.5.2, each node is equipped with 8-cores Intel E5335 2GHz processor and 16GB memory. TrajSpark compare its performance with GeoSpark and SIMBA with two datasets (real data 190GB and synthetic 1,4TB). Benchmarking examine data loading, single object queries (i.e. input is object id and temporal range), spatiotemporal range quries based on STR-Tree and kNN queries. Experiments are conducted with different input batch size.

UlTraMan conduct experiments on a 12-node cluster running Spark 2.1.1, Hadoop 2.7.1 and Chronicle Map 3.14. Each node is equipped with 12-cores Intel Xeon E5-2620 2.4GHz and 40GB memory. Benchmarking examine query performance for id queries, range queries and

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

kNN queries with different input batch size. UlTraMan compare the performance difference using Chronicle Map or of the self Spark/Hadoop pipelines and at the same time the benefits of local and global indices.

Table 8: Experimental setup parameters

Parameter	Values
Encoding scheme	**STRtree encoding**, default encoding
Logical plans	**global index & bucketing table**, global index-local index & bucketing table, default
Physical plans	**Sort Merge Bucket Join**, Brute Force
Execution Engine	**Spark**, Tez

Type of queries. We focused our experiments on range and knn queries. All of our experiments were conducted using almost the same query parameters.

Algorithms. We have implemented logical and physical plans as described in the previous sections for spatio-temporal queries. More specifically, we experimented with (a) global and local index logical plans, (b) Sort Merge Bucket Join, Brute Force physical plans and (C) default, STRtree encoding scheme. Table 8 summarizes the algorithms used during the experimental evaluation process

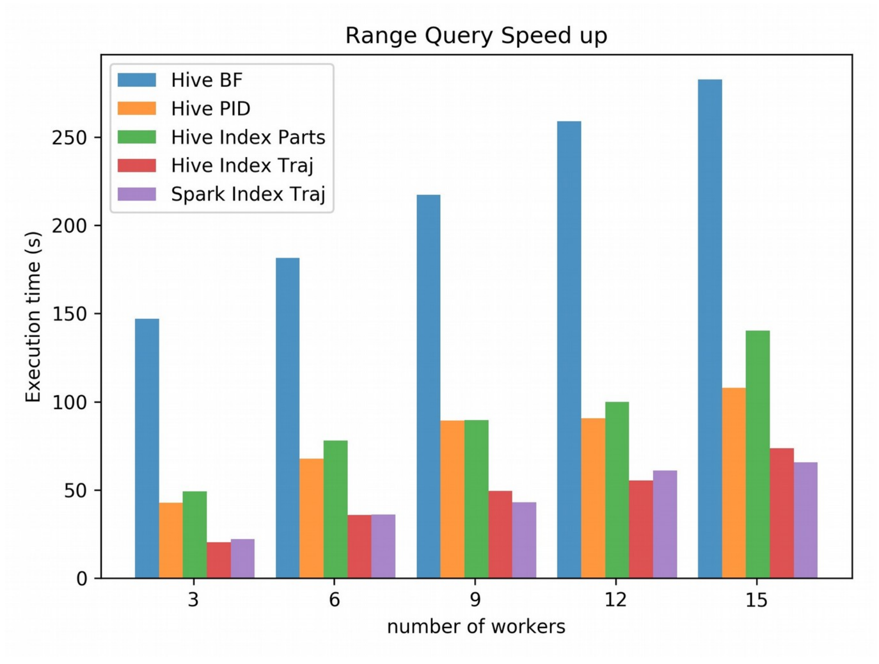
Metrics. Our main evaluation metric was the total execution time of each experiment on the cluster. The actual execution time of our algorithms is presented here. Each experiment was run 3 times, and the average execution time is depicted in the charts.

7.2 Results

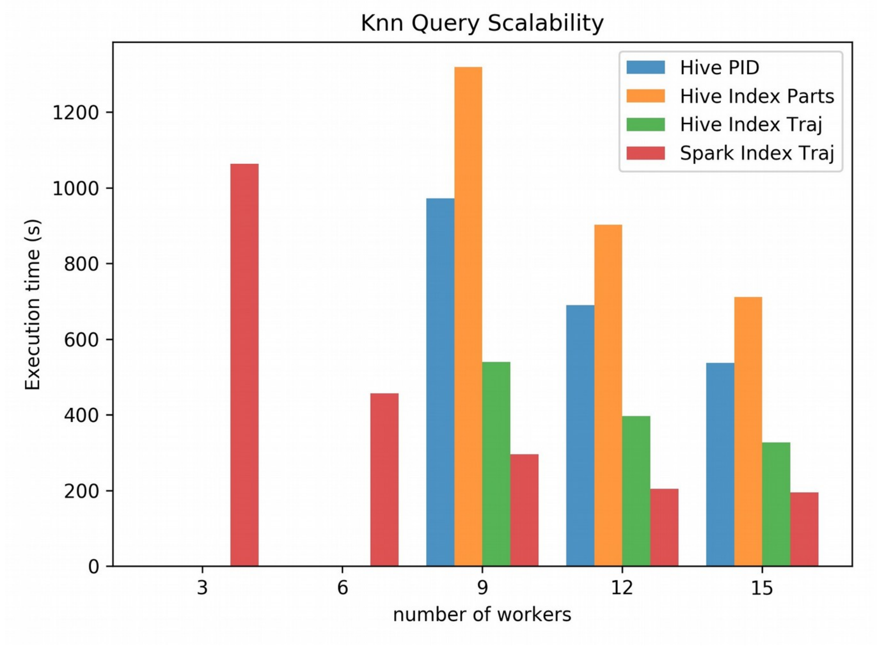
Figures 23 depicts the execution time comparing the available storage strategies with different number of workers and both execution engines for range and knn queries. Figures 24 depicts the execution time comparing the available storage strategies with different size dataset size for range and kNN queries. Clearly, by using partition(global) index we are able to prune early rows which do not satisfy the query. This improves performance by at least 50% comparatively with brute force. It is also important to note index contains only trajectories MBR and points to a bucketized by trajectory identifier table have worse performance than table that bucketized by partition identifier. This performance difference is quite notable especially for kNN queries which is the most challenging query. Also, STRtree encoding outperforms all the other data management strategies.

For range queries the input was a random partitioner's MBR. We made this decision in order to stress the system and examine the query performance with a big number of returning rows. Obviously, the the input for kNN query was a random trajectory from the dataset. Range and kNN queries have quite similar executions time. This occurs because Hive as engine has some overhead for system's metadata, planning and execute query especially with join operator. Moreover, the total execution time could be affected by the hardware limitation. Our proposed system achieve is scalable, extra workers speed up systems performance and increasing data size linearly decrease performance speed.

Both figures conclude the combination of global index with the STRtree encoding is the most best execution plan and improves performance over 80% comparatively with brute force and almost 50% comparatively with the other plan. This means that tree deserialization does not charge the system performance and every delay is because of Hive's semantics. Obviously, Spark outperforms Tez because the execution plans does not require huge joins or a need to write intermediate results on disk. Spark takes advantage of only-in-memory execution. Especially kNN queries has almost the same execution with range queries despite the fact that need extra calculation.

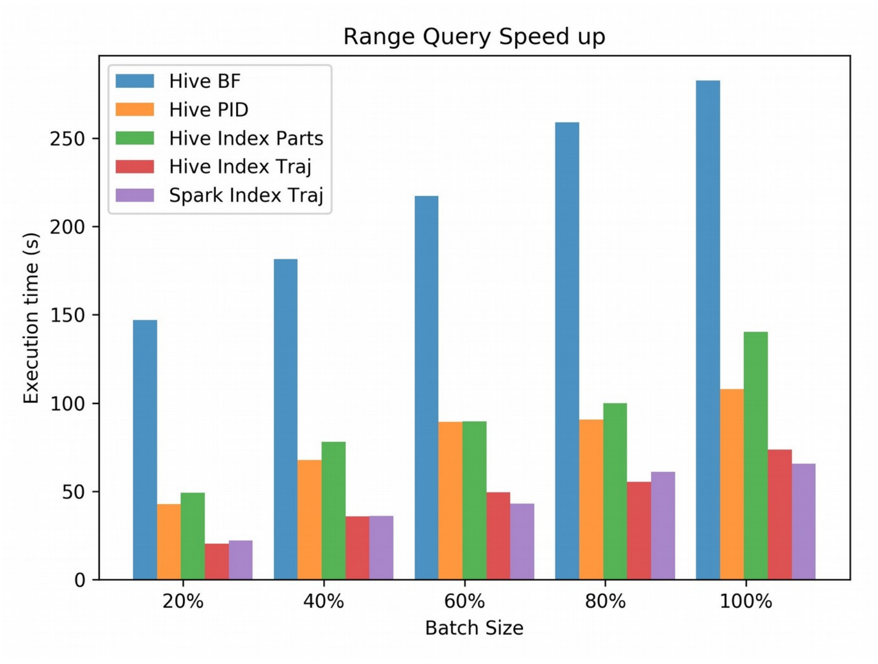


(a) Range Query

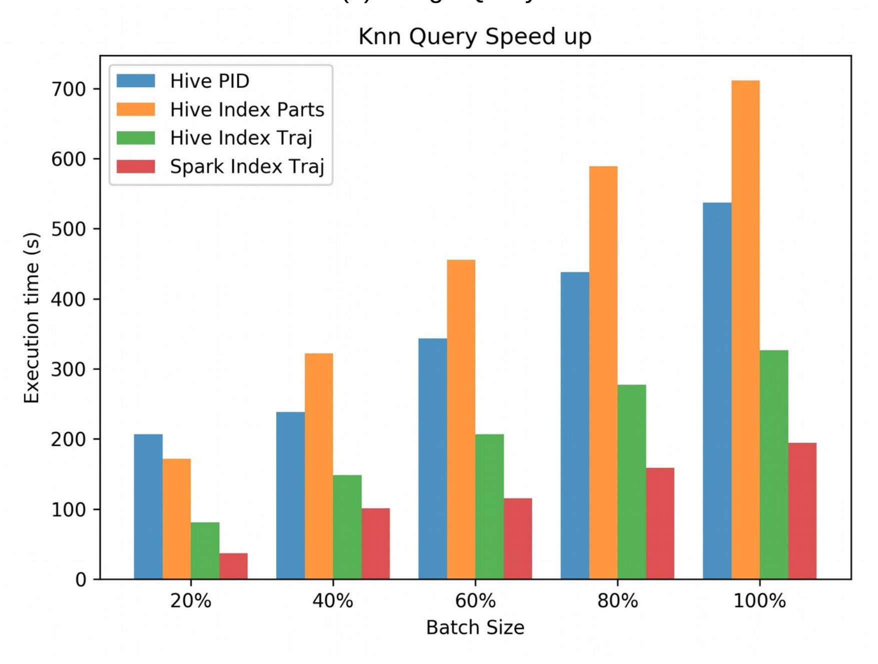


(b) Knn Query

Figure 23: BigSQLTraj Scalability Experiments



(a) Range Query



(b) Knn Query

Figure 24: BigSQLTraj Speed up Experiments

8. CONCLUSION & FUTURE WORK

In this thesis, we present the first flexible, distributed and scalable big data solution to address the problem of big mobility data management and processing with using SQL. The spread of the concept of big mobility data to basic users in conjunction with the lack of software frameworks that would be able to handle spatio-temporal data and methods lead to the development of this thesis. Our proposed BigSQLTraj system, which comprise of a Processing and a Storage layer, is designed to benefit by the tools and best practices for handling vast sizes of data. Our experiments demonstrate the performance of our system, which is able to efficiently process range and kNN spatio-temporal queries, in a few seconds. Achieving this goal, it was necessary the extension of existing tree structures that speed up query execution. The combination of global and local indices, which is more of a matter architecture despite implementation or technologies, is the key and the final result of this thesis. Splitting big datasets based on similarity and create global and local metadata or indices structures must be supported by every big data engine, so that the advantages of space pruning lead to effective query processing

BigSQLTraj provides a clear SQL interface to its data types, functions and operators that make it easy to learn, use and integrate with existing JDBC systems when it comes to managing spatio-temporal data. We explained its components and demonstrated its capabilities on a real world dataset.

There is always room for improvement on a framework like BigSQLTraj and some of the areas this can be done are:

- ➢ evaluate the functionality of the system under real demands, such as a streaming tool always inserts new data in order to measure resposiveness
- ➢ advanced spatio-temporal processing (e.g. computational geometry algorithms)
- ➢ semantic trajectories management and processing: integrating text and spatio-temporal data (semantic or annotated trajectories). Also, there is a need for more complex indices and processing workflows
- ➢ sophisticated partitioners, creating methods in order to minimize partitions overlap and increase compactness

REFERENCES

[1] International Maritime Organization. International Convention for the Safety of Life at Sea [Internet]. 1974. Available: http://www.imo.org/blast/contents.asp?topic_id=257 & doc_id=647

[2] Koubarakis, M., Sellis, T., Frank, A. U., Grumbach, S., Güting, R. H., Jensen, C. S., ... & Schek, H. J. (Eds.). (2003). Spatio-temporal databases: The CHOROCHRONOS approach (Vol. 2520). Springer.

[3] Giannousis, K., Bereta, K., Karalis, N., & Koubarakis, M. (2018, December). Distributed Execution of Spatial SQL Queries. In 2018 IEEE International Conference on Big Data (Big Data) (pp. 528-533). IEEE.

[4] Patroumpas, K., Alevizos, E., Artikis, A., Vodas, M., Pelekis, N., & Theodoridis, Y. (2017). Online event recognition from moving vessel trajectories. GeoInformatica, 21(2), 389-427.

[5] Theodoridis, Y. (2003). Ten benchmark database queries for location-based services. The Computer Journal, 46(6), 713-725.

[6] Pfoser, D., Jensen, C. S., & Theodoridis, Y. (2000, September). Novel approaches to the indexing of moving object trajectories. In VLDB (pp. 395-406).

[7] Frentzos, E., Gratsias, K., & Theodoridis, Y. (2007, April). Index-based most similar trajectory search. In 2007 IEEE 23rd International Conference on Data Engineering (pp. 816-825). IEEE.

[8] Güting, R. H., & Schneider, M. (2005). Moving objects databases. Elsevier.

[9] He, Y., Tan, H., Luo, W., Feng, S., & Fan, J. (2014). MR-DBSCAN: a scalable MapReduce-based DBSCAN algorithm for heavily skewed data. Frontiers of Computer Science, 8(1), 83-99.

[10] Trasarti, R., Guidotti, R., Monreale, A., & Giannotti, F. (2017). Myway: Location prediction via mobility profiling. Information Systems, 64, 350-367.

[11] Ray, S., Simion, B., & Brown, A. D. (2011, April). Jackpine: A benchmark to evaluate spatial database performance. In 2011 IEEE 27th International Conference on Data Engineering (pp. 1139-1150). IEEE.

[12] Pfoser, D., Jensen, C. S., & Theodoridis, Y. (2000, September). Novel approaches to the indexing of moving object trajectories. In VLDB (pp. 395-406).

[13] Vlachos, M., Kollios, G., & Gunopulos, D. (2002). Discovering similar multidimensional trajectories. In Proceedings 18th international conference on data engineering (pp. 673-684). IEEE.

[14] Maguerra, S., Boulmakoul, A., Karim, L., & Badir, H. A Survey on Solutions for Big Spatio-Temporal Data Processing and Analytics.

[15] Pelekis, Nikos, and Yannis Theodoridis. “The case of big mobility data.” Mobility Data Management and Exploration. Springer, New York, NY, 2014. 211-231.

[16] Pelekis, Nikos, and Yannis Theodoridis. “Modeling and acquiring mobility data.” Mobility data management and exploration. Springer, New York, NY, 2014. 51-73.

[17] Pelekis, Nikos, and Yannis Theodoridis. “Mobility Database Management.” Mobility Data Management and Exploration. Springer, New York, NY, 2014. 75-99.

[18] Chronis, Y., Foufoulas, Y., Nikolopoulos, V., Papadopoulos, A., Stamatogiannakis, L., Svingos, C., & Ioannidis, Y. E. (2016, March). A Relational Approach to Complex Dataflows. In EDBT/ICDT Workshops.

[19] Pelekis, Nikos, and Yannis Theodoridis. “Moving Object Database Engines.” Mobility Data Management and Exploration. Springer, New York, NY, 2014. 101-117.

[20] Frentzos, Elias, Yannis Theodoridis, and Apostolos N. Papadopoulos. “Spatio-temporal trajectories.” Encyclopedia of Database Systems. Springer US, 2009. 2742-2746.

[21] Meagher, Donald. (1980). Octree Encoding: A New Technique for the Representation, Manipulation and Display of Arbitrary 3-D Objects by Computer.

[22] Chronicle map. https://github.com/OpenHFT/Chronicle-Map, 2017.

[23] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo. Simba: Efficient in-memory spatial analytics. In SIGMOD, pages 1071–1085, 2016.

[24] Yu, Jia, Jinxuan Wu, and Mohamed Sarwat. ”Geospark: A cluster computing framework for processing large-scale spatial data.” Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems. ACM, 2015.

[25] Huang, Zhou, et al. “GeoSpark SQL: An Effective Framework Enabling Spatial Queries on Spark.” ISPRS International Journal of Geo-Information 6.9 (2017): 285.

[26] Hagedorn, Stefan, Philipp Götze, and Kai-Uwe Sattler. The STARK Framework for Spatio-Temporal Data Analytics on Spark. Datenbanksysteme für Business, Technologie und Web (BTW 2017) (2017).

[27] Ding, X., Chen, L., Gao, Y., Jensen, C. S., & Bao, H. (2018). UlTraMan: A unified platform for big trajectory data management and analytics. Proceedings of the VLDB Endowment, 11(7), 787-799.

[28] Zhang, Z., Jin, C., Mao, J., Yang, X., & Zhou, A. (2017, July). Trajspark: A scalable and efficient in-memory management system for big trajectory data. In Asia-Pacific Web (APWeb) and Web-Age Information Management (WAIM) Joint Conference on Web and Big Data (pp. 11-26). Springer, Cham.

[29] Cudre-Mauroux, P., Wu, E., & Madden, S. (2010, March). Trajstore: An adaptive storage system for very large trajectory data sets. In 2010 IEEE 26th International Conference on Data Engineering (ICDE 2010) (pp. 109-120). IEEE.

[30] Düntgen, C., Behr, T., & Güting, R. H. (2009). BerlinMOD: a benchmark for moving object databases. The VLDB Journal—The International Journal on Very Large Data Bases, 18(6), 1335-1368.

[31] Lu, J., & Güting, R. H. (2012, December). Parallel secondo: boosting database engines with hadoop. In 2012 IEEE 18th International Conference on Parallel and Distributed Systems (pp. 738-743). IEEE.

[32] Güting, R. H., Behr, T., & Düntgen, C. (2010). SECONDO: A Platform for Moving Objects Database Research and for Publishing and Integrating Research Implementations. IEEE Data Eng. Bull., 33(2), 56-63.

[33] White, T. (2012). Hadoop: The definitive guide. ” O’Reilly Media, Inc.”.

[34] Thusoo, A., Sarma, J. S., Jain, N., Shao, Z., Chakka, P., Zhang, N., ... & Murthy, R. (2010, March). Hive-a petabyte scale data warehouse using hadoop. In 2010 IEEE 26th international conference on data engineering (ICDE 2010) (pp. 996-1005). IEEE.

[35] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., & Stoica, I. (2010). Spark: Cluster computing with working sets. HotCloud, 10(10-10), 95.

[36] Saha, B., Shah, H., Seth, S., Vijayaraghavan, G., Murthy, A., & Curino, C. (2015, May). Apache tez: A unifying framework for modeling and building data processing applications. In Proceedings of the 2015 ACM SIGMOD international conference on Management of Data (pp. 1357-1369). ACM.

[37] Kumar, R., & Kumar, N. (2016). Improved join operations using ORC in HIVE. CSI transactions on ICT, 4(2-4), 209-215.

[38] Samet, H. (1984). The quadtree and related hierarchical data structures. ACM Computing Surveys (CSUR), 16(2), 187-260.

[39] Guttman, A. (1984). R-trees: a dynamic index structure for spatial searching (Vol. 14, No. 2, pp. 47-57). ACM.

[40] Obe, R. O., & Hsu, L. S. (2011). PostGIS in action (Vol. 2). Greenwich: Manning.

[41] Frentzos, E., Gratsias, K., Pelekis, N., & Theodoridis, Y. (2007). Algorithms for nearest neighbor search on moving object trajectories. Geoinformatica, 11(2), 159-193.