ST-Hadoop: A MapReduce Framework for Big Spatio-temporal Data Management

A DISSERTATION
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Louai Alarabi

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
Doctor of Philosophy

Mohamed F. Mokbel

May, 2019

Acknowledgements

There are many people have earned my gratitude for their contribution and involvement to my success in graduate school, including my parents, siblings, spouse, children, doctoral adviser, examination committees, friends, colleagues, and sponsor.

I am incredibly thankful to Prof. Mohamed F. Mokbel for his guidance throughout my stay in graduate school. I want to express my sincere gratitude for his generous support. I consider myself highly fortunate to work with him closely. I always have been inspired by his passion and commitment toward research and building systems. I would not have become the scientist that I am today without his wisdom and assistance, I have grasped from him critical thinking and hard working.

Besides the primary adviser, I am grateful to Prof. Shashi Shekhar, Prof. Steven M Manson, Prof. Jon Weissman, and Prof. Abhishek Chandra for serving on my preliminary and thesis committee. Very appreciative for their essential and constructive feedback that shaped my final dissertation. I am thankful to Prof. Shashi Shekhar for his insightful comments and for sharing with me his tremendous experience in the spatial data management field. I am grateful to Prof. Steven Manson for his insightful comments and for sharing his expertise in geographic information science all the way through my Master and Ph.d program. I am thankful to Prof. Jon Weissman, and Prof. Abhishek Chandra for their profound insight and knowledge in distributed frameworks. I am grateful to Dr. Shana Watters for sharing with me her teaching experience and being a role model. I am very thankful for her encouragement. I sincerely appreciate all skills they imparted to my academic and professional life.

I am very thankful to Prof. Saleh M. Basalamah for his strong support and recommendation to join Prof. Mokbels research group. I want to express my sincere appreciation to (UQU) UMM AL-Qura University for their generous sponsorship during the

past eight years in grad school. I would also thank (SACM) the Saudi Cultural Mission, (NSF) the National Science Foundation, and Microsoft for their generous grants and financial support that otherwise I would not be able to share my scientific explorations and be actively engage in conferences.

I am so grateful for my supportive and motivating family. I am honored to be a son of a great man Mishal Alarabi, and wonderful mother Hanan Aqlan. I am very thankful to my parents for their unconditional support and believing in me. I am a proud brother to amazing siblings Lena, Wael, Walaa, Raied, and Raneem. I am genuinely indebted to my cheerleader wife, Mada Sabbah, for her unlimited support and countless sacrifices. Together, we celebrated every acceptance and moderated every rejection. Mada and my children Aseel, Zain and the little unborn ones are the joy of my life and the special blessing.

I would like to thank all members of Mokbel's Lab, including alumni and current ones for making my experience in data management lab and grad school fun and exciting. In particular, Mohamed Sarwat, Abdeltawab Hendawi, Jie Bao, Ahmed Eldawy, Rami Alghamdi, Saif Al-Harthi, Christopher Jonathan, Mashaal Musleh, Ibrahim Sabek, Harshada Chavan, Bin Cao, Prof. Kwang Woo Nam, Xiaochuang Yao, and Rana Forsati.

Dedication

To those who held me up over the years. Thank you.

Abstract

Apache Hadoop, employing the MapReduce programming paradigm, that has been widely accepted as the standard framework for analyzing big data in distributed environments. Unfortunately, this rich framework was not genuinely exploited towards processing large-scale spatio-temporal data, especially with the emergence and popularity of applications that create them in large-scale. The huge volumes of spatio-temporal data come from applications, like Taxi fleet in urban computing, Asteroids in astronomy research studies, animal movements in habitat studies, neuron analysis in neuroscience research studies, and contents of social networks (e.g., Twitter or Facebook). Managing space and time are two fundamental characteristics that raised the demand for processing spatio-temporal data created by these applications. Besides the massive size of data, the complexity of shapes and formats associated with these data raised many challenges in managing spatio-temporal data.

The goal of dissertation is centered on establishing a full-fledged big spatio-temporal data management system that serves the need for a wide range of spatio-temporal applications. This involves indexing, querying, and analyzing spatio-temporal data. We propose ST-Hadoop; the first full-fledged open-source system with a native support for big spatio-temporal data, available to download `http://st-hadoop.cs.umn.edu/`. ST-Hadoop injects spatio-temporal data awareness inside the highly popular Hadoop system that is considered the state-of-the-art for off-line analysis of big data systems. Considering a distributed environment, we focus on the following: (1) indexing spatio-temporal data and (2) Supporting various fundamental spatio-temporal operations, such as range, kNN, and join. Throughout this document, we will touch base on the background and related work, motivate for the proposed system, and highlight our contributions.

Contents

Acknowledgements **i**

Dedication **iii**

Abstract **iv**

List of Tables **viii**

List of Figures **ix**

1 Introduction **1**

1.1 Contribution and Organization 4

2 Background and Related work **7**

2.1 Architecture 8

2.1.1 MapReduce 8

2.1.2 Resilient Distributed Dataset (RDD) 9

2.1.3 Big Distributed Stream Platforms 9

2.1.4 Key-Value Store 10

2.1.5 Parallel Database Systems 11

2.2 Implementation Approach 11

2.2.1 On-Top of the Framework 12

2.2.2 Built inside the Framework 12

2.2.3 Ad-hoc on Big Spatial System 13

2.2.4 Built From-Scratch 13

2.3 Language . . . 14
2.4 Indexing Technique . . . 14
2.5 Spatio-temporal Operation . . . 15
2.6 ST-Hadoop Contribution . . . 16

3 ST-Hadoop Architecture Overview 20

4 Spatio-temporal Indexing in MapReduce Layer 23
4.1 Concept of Hierarchy . . . 25
4.2 Index Construction . . . 26
4.2.1 Phase I Sampling . . . 26
4.2.2 Phase II Temporal Slicing . . . 28
4.2.3 Phase III Spatial Indexing . . . 31
4.2.4 Phase IV Physical Writing . . . 31
4.3 Index Maintenance . . . 31
4.4 Experiments . . . 33
4.4.1 Experimental Settings . . . 33
4.4.2 Index Construction . . . 34

5 Spatio-temporal Operations on MapReduce 36
5.1 Spatio-temporal Range Query . . . 36
5.2 Spatio-temporal kNN Query . . . 37
5.3 Spatio-temporal Join . . . 46
5.4 Experiments . . . 47
5.4.1 Spatiotemporal Range Query . . . 47
5.4.2 K-Nearest-Neighbor Queries (kNN) . . . 48
5.4.3 Spatiotemporal Join . . . 52

6 Spatio-temporal Query Optimizer in MapReduce 54
6.1 Heuristic Query Optimization . . . 55
6.2 Cost-based Optimization . . . 57
6.3 Experiments . . . 58

7 Summit Trajectory library in ST-Hadoop **61**
7.1 introduction . . . 62
7.2 background and related work . . . 64
7.3 System Overview . . . 68
7.4 Trajectory Indexing . . . 69
7.5 Trajectory Operations . . . 72
7.6 Trajectory Range Query (TRQ) . . . 75
7.7 Trajectory nearest neighbor Query (TKNN) . . . 76
7.7.1 (TKNN) Point-based . . . 76
7.7.2 (TKNN) Trajectory-based . . . 81
7.8 Trajectory Similarity Query (TSQ) . . . 86
7.9 Experiments . . . 91
7.9.1 Experiments Settings . . . 92
7.9.2 Range Query . . . 93
7.9.3 Summit Stability . . . 96
7.9.4 Trajectory Nearest Neighbor query . . . 98
7.9.5 Trajectory Similarity query . . . 101

8 Language Layer **104**
8.1 Basic Spatio-temporal Data types . . . 104
8.2 Basic Functions and Operations . . . 105
8.3 Trajectory Spatio-temporal Data types . . . 106
8.4 Trajectory Functions and Operations . . . 107

9 Conclusion **109**

References **111**

List of Tables

2.1 Existing Work in the area of Big Spatial data 18
2.2 Existing Work in the area of Big Spatio-temporal data 19
4.1 Twitter Datasets 33
4.2 ST-Hadoop Experiments Parameters 34
7.1 New York Taxi and Limousine Dataset 92
7.2 Summit Experiments Parameters Settings 92

List of Figures

1.1 Range query in SpatialHadoop vs. ST-Hadoop 2
3.1 ST-Hadoop system architecture 21
4.1 HDFSs in ST-Hadoop VS SpatialHadoop 24
4.2 Indexing in ST-Hadoop 27
4.3 Data-Slice 29
4.4 Time-Slice 30
4.5 Temporal Hierarchy Index 32
5.1 Landscape of spatio-temporal kNN operation 40
5.2 Correctness check Final Answer 41
5.3 Correctness Check when $0 \leq \alpha \leq 1$ 42
5.4 Refinement Final Answer 44
5.5 Spatio-temporal Join 45
5.6 Range Query VS Input files (TB) 48
5.7 Range Query VS Block size (MB) 49
5.8 Range query with Block size VS Slicing ratio (α) 50
5.9 The execution of kNN query on different input files 51
5.10 kNN query with various k 51
5.11 kNN throughput while varying (α) of Ranking Function 52
5.12 Spatio-temporal Join 53
6.1 Conceptual representation of ST-Hadoop meta-data index 55
6.2 Spatio-temporal Range Query Interval Window 58
6.3 The effect of the spatio-temporal query ranges on the best value of $\mathcal{M}$. 59
6.4 ST-Hadoop Greedy query optimizer VS heuristic $\mathcal{M}$ 60
7.1 Similarity query in ST-Hadoop vs. Summit 63

7.2 Summit Architecture 68
7.3 Temporal Slicing 70
7.4 Trajectory Indexing 71
7.5 Summit Trajectory Operations 73
7.6 Local Computation: initial k answer 76
7.7 Correctness Check Cylinder 79
7.8 MBR Trajectory distance 81
7.9 Nearest Neighbor Trajectory-based in Summit 83
7.10 Nearest Neighbor Trajectory-based correctness check 84
7.11 Abstract idea of similarity computation in Summit 88
7.12 Summit global computation of similarity query 89
7.13 Range Query 94
7.14 Summit stability 96
7.15 kNN Query 99
7.16 DTW similarity Query 101

Chapter 1

Introduction

The importance of processing spatio-temporal data has gained much interest in the last few years, especially with the emergence and popularity of applications that create them in large-scale. For example, Taxi trajectory of New York city archive over 1.1 Billion trajectories [1], social network data (e.g., Twitter has over 500 Million new tweets every day) [2], NASA Satellite daily produces 4TB of data [3, 4], and European X-Ray Free-Electron Laser Facility produce large collection of spatio-temporal series at a rate of 40GB per second, that collectively form 50PB of data yearly [5]. Beside the huge achieved volume of the data, space and time are two fundamental characteristics that raise the demand for processing spatio-temporal data.

The current efforts to process big spatio-temporal data on MapReduce environment either use: (a) *General purpose* distributed frameworks such as Hadoop [6] or Spark [7], or (b) *Big spatial data systems* such as ESRI tools on Hadoop [8], Parallel-Secondo [9], $\mathcal{MD}$-HBase [10], Hadoop-GIS [11], GeoTrellis [12], GeoSpark [13], or SpatialHadoop [14]. The former has been acceptable for typical analysis tasks as they organize data as non-indexed heap files. However, using these systems as-is will results in sub-performance for spatio-temporal applications that need indexing [15, 16, 17]. The latter reveal their inefficiency for supporting time-varying of spatial objects because their indexes are mainly geared toward processing spatial queries, e.g., SHAHED system [18] is built on top of SpatialHadoop [14].

Even though existing big spatial systems are efficient for spatial operations, nonetheless, they suffer when they are processing spatio-temporal queries, e.g., "*find geo-tagged*

```
Objects = LOAD 'points' AS (id:int, Location:POINT, Time:t);
Result  = FILTER Objects BY
          Overlaps (Location, Rectangle(x1, y1, x2, y2))
           AND t < t2 AND t > t1;
```

(a) Range query in SpatialHadoop

```
Objects = LOAD 'points' AS (id:int, STPoint:(Location,Time));
Result  = FILTER Objects BY
          Overlaps (STPoint, Rectangle(x1, y1, x2, y2), Interval (t1, t2) );
```

(b) Range query in ST-Hadoop

Figure 1.1: Range query in SpatialHadoop vs. ST-Hadoop

news in California area during the last three months". Adopting any big spatial systems to execute common types of spatio-temporal queries, e.g., *range query*, will suffer from the following: (1) The spatial index is still illsuited to efficiently support time-varying of spatial objects, mainly because the index are geared toward supporting spatial queries, in which result in scanning through irrelevant data to the query answer. (2) The system internal is unaware of the spatio-temporal properties of the objects, especially when they are routinely achieved in large-scale. Such aspect enforces the spatial index to be reconstructed from scratch with every batch update to accommodate new data, and thus the space division of regions in the spatial-index will be jammed, in which require more processing time for spatio-temporal queries. One possible way to recognize spatio-temporal data is to add one more dimension to the spatial index. Yet, such choice is incapable of accommodating new batch update without reconstruction the whole index from scratch.

This paper introduces ST-Hadoop; the first full-fledged open-source MapReduce framework with a native support for spatio-temporal data, available to download from [19]. ST-Hadoop is a comprehensive extension to Hadoop and SpatialHadoop that injects spatio-temporal data awareness inside each of their layers, mainly, indexing, operations, and language layers. ST-Hadoop is compatible with SpatialHadoop and Hadoop, where programs are coded as *map* and *reduce* functions. However, running a

program that deals with spatio-temporal data using ST-Hadoop will have orders of magnitude better performance than Hadoop and SpatialHadoop. Figures 1.1(a) and 1.1(b) show how to express a spatio-temporal range query in SpatialHadoop and ST-Hadoop, respectively. The query finds all points within a certain rectangular area represented by two corner points $\langle x1, y1 \rangle$, $\langle x2, y2 \rangle$, and a within a time interval $\langle t1,t2 \rangle$. Running this query on a dataset of 10TB and a cluster of 24 nodes takes 200 seconds on SpatialHadoop as opposed to only one second on ST-Hadoop. The main reason of the sub-performance of SpatialHadoop is that it needs to scan all the entries in its spatial index that overlap with the spatial predicate, and then check the temporal predicate of each entry individually. Meanwhile, ST-Hadoop exploits its built-in spatio-temporal index to only retrieve the data entries that overlap with *both* the spatial and temporal predicates, and hence achieves two orders of magnitude improvement over SpatialHadoop.

ST-Hadoop is a comprehensive extension of Hadoop that injects spatio-temporal awareness inside each layers of SpatialHadoop, mainly, *language*, *indexing*, *MapReduce*, and *operations* layers. In the *language* layer, ST-Hadoop extends Pigeon language [20] to supports spatio-temporal data types and operations. The *indexing* layer, ST-Hadoop spatiotemporally loads and divides data across computation nodes in the Hadoop distributed file system. In this layer ST-Hadoop scans a random sample obtained from the whole dataset, bulk loads its spatio-temporal index in-memory, and then uses the spatio-temporal boundaries of its index structure to assign data records with its overlap partitions. ST-Hadoop sacrifices storage to achieve more efficient performance in supporting spatio-temporal operations, by replicating its index into *temporal hierarchy index structure* that consists of two-layer indexing of temporal and then spatial. The *MapReduce* layer introduces two new components of *SpatioTemporalFileSplitter*, and *SpatioTemporalRecordReader*, that exploit the spatio-temporal index structures to speed up spatio-temporal operations. Finally, the *operations* layer encapsulates the spatio-temporal operations that take advantage of the ST-Hadoop *temporal hierarchy index structure* in the indexing layer, such as spatio-temporal range, nearest neighbor, and join queries.

The key idea behind the performance gain of ST-Hadoop is its ability to load the data in Hadoop Distributed File System (HDFS) in a way that mimics spatio-temporal index structures. Hence, incoming spatio-temporal queries can have minimal data access

to retrieve the query answer. ST-Hadoop is shipped with support for three fundamental spatio-temporal queries, namely, spatio-temporal range, nearest neighbor, and join queries. However, ST-Hadoop is extensible to support a myriad of other spatio-temporal operations. We envision that ST-Hadoop will act as a research vehicle where developers, practitioners, and researchers worldwide, can either use it directly or enrich the system by contributing their operations and analysis techniques.

1.1 Contribution and Organization

In this section, we will highlight our contribution in the area of scaling big spatio-temporal data on MapReduce framework and outline a road map for this dissertation. Our research in building a big spatio-temporal system has accomplished several notable milestones. First, in February 2017 we released ST-Hadoop on a website for the public: `http://st-hadoop.cs.umn.edu/`. Second, we received a certificate of recognition from ACM SIGMOD (top-tier conference) for being selected as a finalist for the student research competition [21]. Third, we published ST-Hadoop research study with all technical details in SSTD 2017 [22]. Our paper was selected among the best papers in the conference and invited for a special issue of GeoInformatica journal [23]. Both SSTD and GeoInformatica are top-tier conference and journal in the area of spatial and spatio-temporal data management. Forth, we have demonstrated the capability and the efficiency of ST-Hadoop to VLDB 2017 conference attendees [24]. In this demonstration, ST-Hadoop successfully managed to index and query billions of spatio-temporal records. Fifth, we extend the infrastructure and operations of ST-Hadoop to support trajectory data, and we received a certificate of recognition from ACM SIGSPATIAL for being the first place winner of graduate student research competition [25]. Sixth, our contribution in supporting trajectory data won a gold medal from Microsoft and invited for a special issue of SIGSPATIAL newsletter [26]. Finally, our research got invited to participate in ACM grand final graduate students research competition, where winners will be invited to ACM Banquet, i.e., "Turing Award Banquet" along with their adviser, where they receive formal recognition.

The overreaching goal of this dissertation is to conduct research, develop the

required knowledge to advance the state-of-the-art of Big Spatio-temporal data management on the MapReduce framework. This thesis is the first of its kind that provides ST-Hadoop; a full-fledged open-source framework mainly to support a workload of batch spatio-temporal analytic on MapReduce. The proposed system is extensible to support a myriad of spatio-temporal indexing and operations. We envision that open-source nature of the proposed framework will act as a research vehicle where experts developers, domain practitioners, and researchers worldwide, can either use the proposed framework directly or enrich it by contributing their operations and analysis techniques. Given this general overview, this document is organized as follows:

• **Chapter 2** gives a comprehensive survey of existing research studies from both academia and industry in the area of supporting big spatio-temporal data.

• **Chapter 3** presents the architecture design of our proposed framework ST-Hadoop; as the first full-fledged open-source MapReduce framework with built-in support for spatio-temporal data.

• **Chapter 4** investigates the structural design of indexing in MapReduce that supports spatio-temporal data. The proposed indexing approaches incorporate the functionality of various big spatio-temporal batch workloads.

• **Chapter 5** presents the implementation of three fundamental spatio-temporal operations, namely, spatio-temporal range, nearest neighbor, and join queries. We envision more operations can be added by professional developers, domain experts, and researchers.

• **Chapter 6** investigates the spatio-temporal query optimization. In particular, this chapter we examined two standard query optimization model of heuristic and cost-based models.

• **Chapter 7** introduces an extension library to support analytic operation for a large scale trajectory data in MapReduce. This extension is driven by the ubiquity

of location-based services, that produce a massive amount of trajectories. Querying and analyzing trajectory data become a must for a wide range of applications. The proposed extension is well-suited to efficiently support several basic trajectory queries, such as range, kNN, and similarity queries. These queries and the architectural design of the proposed library are extendable, in a way that it enables users to build various applications on trajectories.

• **Chapter 8** describes how casual users can interact with ST-Hadoop through its language layer. We discussed basic spatio-temporal and trajectory data types, functions, and operations. ST-Hadoop extends the state-of-the-art spatial SQL like language that makes our system more usable.

• **Chapter 9** concludes and summarizes our contributions.

Chapter 2

Background and Related work

Triggered by the needs to process large-scale spatio-temporal data, there is an increasing recent interest in using big distributed frameworks, such as Hadoop [6], Spark [7], or Flink [27] to support spatio-temporal operations. Classification is essential for the study of any subject. Thus, this chapter classifies existing research studies by considering five aspects of big spatio-temporal data systems. (1) The system architecture, which identifies the storage paradigm designed for storing and retrieving data, such as RDBMS, columnar DBMS, parallel DBMS, MapReduce, key-value store, RDD, or Discretized Stream. (2) The implementation approach, which defines whether it's implemented on the top of the distributed framework, add-hoc using big systems, built inside the base core of a distributed framework, or entirely developed from scratch. (3) The support of a high-level scripting language. (4) The type of indexes employed by the system, if any exists. (5) The supported spatio-temporal operations by the research study.

Table 2.1 depicts the existing work from both academia and industry in the area of supporting big spatio-temporal data. Each row represents a system or a body of work related to big spatio-temporal data, while each column represents one of the main characteristic of big spatio-temporal data system. The rest of this chapter details over each one of the five characteristic, namely, architecture, approach, language, indexing, and operation.

2.1 Architecture

In this section we will briefly describe existing systems architecture design, that typically follow one of the standard approaches used in big distributed frameworks, such as, relational DBMS [34, 36, 44], columnar DBMS [43], parallel DBMS [9, 37], MapReduce [9, 11, 14, 15, 16, 17, 18, 23, 31, 32, 47, 49, 50, 60], key-value store [10, 41, 42], resilient distributed datasets (RDD) [12, 13, 45, 46, 59], Azure [52], or Discretized Stream (DStream) [54], as described in the first column of Table 2.1 and Table 2.2 Some of these systems modify the underlying system to better support spatial or spatio-temporal data but they still keep the central architecture. In the meantime, some use the underlying architecture as-is [47, 54]. In our proposed system ST-Hadoop we follow the MapReduce architecture. However, it is the only system that modifies the MapReduce query processing engine and modifies the file organization of the Hadoop Distributed File System (HDFS) to better support indexing spatio-temporal data.

2.1.1 MapReduce

MapReduce [61] is one of the most popular distributed architecture commonly used for processing and analyzing big analytic data. MapReduce is a shared-nothing distributed system. It is has been widely adopted for various applications from both industry and academia. The main idea of the MapReduce platform is to push the computation to data, unlike other paradigms where data are brought to the main-memory platform for computations.

Apache Hadoop [6] is an open source framework developed in Java based adopting MapReduce paradigm, and it is used for distributed storage and processing massive data. Hadoop platform can be viewed as a cluster of machines that consist of one Master node and several worker nodes. Hadoop platform offers various tools, such as Hadoop Distributed File System (HDFS), job scheduling and resource management capabilities. The most critical component of Hadoop is its HDFS storage layer. Before processing data in Hadoop, data need to be into the HDFS. Initially, Hadoop splits data files into large blocs. Then, it distributes them through nodes in the cluster. After, it transfers the packaged map-reduce tasks into nodes in parallel to process data. This method aims to push the computation to data where they reside, which offers fast and

efficient processing for the massive scale of data.

2.1.2 Resilient Distributed Dataset (RDD)

UC Berkeley introduced and implemented a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner [62]. This abstraction named resilient distributed datasets (RDD) that enables clusters to stores intermediate results in-memory for scale-out complex computations, such as iterative algorithms in Machine learning, graph processing, and interactive data mining algorithms of clustering and regressions.

Apache Spark [7] released in 2010 as an open source frameworks. It has a similar programming model to MapReduce, but it extends it with in-memory data sharing abstraction (RDD). Hence, Spark is classified as shared memory parallel distributed system. When loading data into spark the same programming paradigm in MapReduce is applied, where data is partitioned across a cluster and manipulated in-memory in a parallel fashion by *map*, *filter*, and *groupBy* operations. Spark used in wide range of applications, including graph processing [63], Machine learning [64], spatial computing [12, 13, 65], and spatio-temporal computing [46, 57, 58, 59].

To distinguish RDD and MapReduce, the power of MapReduce resides in the fact that it enables parallel computations without directly share intermediate data. This let MapReduce inefficient for the applications that reuse intermediate results, such as iterative machine learning algorithms (e.g., as K-means clustering and logistic regression). In the meantime, RDD enables its users to store intermediate results inside the RAM. In contrast to MapReduce where it would write intermediate data to the HDFS, in which incur overhead disk I/O and serialization. Thus, RDD is up to 20X faster than MapReduce for iterative applications.

2.1.3 Big Distributed Stream Platforms

In this section, we will discuss two frameworks that follows stream processing model, namely, Storm [66] and Flink [27].

Apache Storm is an open source distributed real-time stream computation platform. Initially developed by Twitter and now it is available on Apache projects. The

architecture design of Storm is similar to Spark, such that it is a shared-memory distributed system. However unlike Spark, Storm does not follow a MapReduce programming paradigm; instead, Storm provides topology programming model. A topology in Storm is defined as a directed graph where the vertices represent computation, and the edges represent the data flow between computation components. This model distributes stream partitions between processing nodes. Each node processes its input stream as if its entire stream. In particular, Storm has two distinct vertices in its topology model recognized as Spouts and Bolts. The Spout represents a source of stream tuples that are used within the topology. Meanwhile, Bolts are serving as a processing components for incoming data. The output of Bolts can be passed to a set of other bolts for further computation, or stored in the cluster Storage. DSI [55] and DITIR [56] supports spatio-temporal data on-top of Strom framework. The DSI [55] implemented Horizontal and vertical Strip index to support nearest neighbor query on moving datasets. In the meantime, DITIR [56] employs B+tree and R-tree to support spatio-temporal range query.

Apache Flink is an open-source framework for real-time stream and batch processing. Unlike Storm which only supports stream processing. In Flink's parallel streaming tasks are similar to Storm's bolts. Storm and Flink have in common that they aim for low latency stream processing by pipelined data transfers. Mobility Streaming [54] implemented on-top of Flink Spatio-temporal Range query to support analyzing trajectory data.

2.1.4 Key-Value Store

Several Systems adopted a simple data model, where data are stored corresponding to key-value. In this model, each key is a unique, and value can be of any types and sizes, unlike traditional relational databases where data types and number of attributes are unified across all tuples in a table. This model enables big systems to store and retrieve of a schema-less data by keys. Examples of popular key-value store systems, include Accumulo [67], HBase [68], Dynamo DB [69], Cassandra [70]. Apache Accumulo and HBase operate on-top of the Hadoop Distributed File System (HDFS). In the meantime, Cassandra uses a storage structure similar to a log and stores directly into the file system. Several research studies extends key-value stores to support spatial data, and

spatio-temporal data, such as $\mathcal{MD}$-HBase [10] extended HBase, and GeoMesa [41] and GeoWave [42] extended Accumulo.

2.1.5 Parallel Database Systems

Apache AsterixDB distinguished itself as the first open-source parallel Big Data Management System (**BDMS**) [71]. Before being incubated by the Apache Software Foundation, AsterixDB has initially been developed by a team of faculty, staff, and students at UC Irvine and UC Riverside [38, 39]. The project was initiated as NSF-sponsored project in 2009, the goal of which was to combine the best ideas from the parallel database world, the MapReduce paradigm, and the semi-structured (e.g., XML/JSON) data world to create a next-generation BDMS.

AstrixDB aims to support ingesting, storing, indexing, querying and analyzing massive amounts of data efficiently. It uses Log-Structured Merge (LSM) trees [72] as the primary underlying technology for all of its internal data storage and indexing. Also, it adds several secondary indexing techniques with LSM, such as B+-tree, R-tree, and inverted index. Entries inserted into an LSM-tree are temporary resides in the main memory, and when in-memory data exceeds a specific capacity, data are flushed into the index that resides on disk. In the meantime, computations and queries execution is handled by Hyracks runtime. Jobs submitted to AstrixDB in the form of DAGs that consists of operators and connectors. The operator is a computation component in the DAG; meanwhile, the connector is similar to a pipeline that feeds the output of one operator to the next operator.

2.2 Implementation Approach

In this section we will classify the existing works in the area of processing spatio-temporal data based on the implementation approach used to build the system. The second column of Table 2.1 shows four categories of the implementation approach: (1) on-top of existing framework [15, 16, 17, 47, 48, 49, 50, 51, 54], (2) ad-hoc on big spatial framework [18, 35, 60], (3) built inside existing system [9, 10, 11, 13, 14, 31, 32, 34, 41, 42, 43, 44, 45, 46], or (4) entirely built from-scratch [36, 37].

2.2.1 On-Top of the Framework

Existing work in this category has mainly focused on addressing a specific spatio-temporal operation. The idea of on-top of MapReduce framework is to develop map and reduce functions for the required operation, which will be executed on-top of existing Hadoop cluster. The same idea applies to other frameworks, such Spark or Flink. Examples of these operations includes spatio-temporal range query [15, 16, 17, 49, 50, 54], spatio-temporal join [47, 48, 51]. However, using Hadoop as-is results in a poor performance for spatio-temporal applications that need indexing.

2.2.2 Built inside the Framework

The research studies in this category build their indexes or the operations inside the core of the distributed framework. The main benefit of following such an approach is to make the internal of the distributed framework is more aware of the nature and the characteristic of the operations and structure of the data organization; and thus, achieves better efficiency. This approach has been adopted by several systems including, GeoMesa [41], GeoSpark [13], SpatialHadoop [14], SharkDB [43], Hippo [34], GeoWave [42], DITA [45], ESRI tools on Hadoop [31], Parallel-Secondo [9], $\mathcal{MD}$-HBase [10], Hadoop-GIS [11], PITS [44], ScalaGiST [32], and Spatio-temporal Join [46].

Several spatio-temporal System in this category has mainly focused on combining the three spatio-temporal dimensions (i.e., x, y, and time) into a single-dimensional lexicographic key. For example, GeoMesa [41] and GeoWave [73] both are built upon Accumulo platform [67] and implemented a space filling curve to combine the three dimensions of geometry and time. Yet, these systems do not attempt to enhance the spatial locality of data; instead they rely on time load balancing inherited by Accumulo. Hence, they will have a sup-performance for spatio-temporal operations on highly skewed data. Meanwhile, ST-Hadoop extends Hadoop framework to support spatio-temporal data. ST-Hadoop temporally loads spatio-temporal data across computation nodes, in a way that enables ST-Hadoop to utilize the spatial and temporal locality of data.

2.2.3 Ad-hoc on Big Spatial System

Several big spatial systems in this category are still ill-suited to perform spatio-temporal operations, mainly because their indexes are only geared toward processing spatial operations, and their internals are unaware of the spatio-temporal data properties [8, 9, 10, 11, 13, 14, 28, 30, 32, 74]. For example, SHAHED, TAGHREED and GISQF systems, where all run spatio-temporal operations as an ad-hoc using SpatialHadoop [18, 35, 60].

2.2.4 Built From-Scratch

Existing works in this category has mainly focused on building a complete infrastructure to leverage and advance the functionality to meet the need for a specific application. Although, constructing a system from scratch gives the flexibility to gain a solid performance, yet, the usability of that system is still only narrowed and limited to support particular applications. This approach has mostly accepted for supporting trajectory data, such as in BRACE [28], SciDB [29, 30], TrajStore [36], Elite [37], and AstrixDB [38, 39]

The proposed ST-Hadoop is designed as a generic MapReduce system to support spatio-temporal queries, and assist developers in implementing a wide selection of spatio-temporal operations. In particular, ST-Hadoop leverages the design of Hadoop and SpatialHadoop to loads and partitions data records according to their time and spatial dimension across computations nodes, which allow the parallelism of processing spatio-temporal queries when accessing its index. In this paper, we present two case study of operations that utilize the ST-Hadoop indexing, namely, spatio-temporal range and join queries. ST-Hadoop operations achieve two or more orders of magnitude better performance, mainly because ST-Hadoop is sufficiently aware of both temporal and spatial locality of data records.

2.3 Language

Several big systems ship a SQL-like language with their system, allowing non-technical users to interact directly with their operations without any requirement of adding significant code. As shown in the third column of Table 2.1, there are several SQL-like high level languages were developed to interact with big distributed systems, such as SQL [34, 36, 44], SQL-like [9, 11], HiveQL [31, 49, 50], Scala-based [12, 13], Pigeon [14, 18, 23], and CQL [41]. Since Pigeon language is compliant with GCC standard [75], ST-Hadoop will not provide an entirely new language. Instead, it extends Pigeon language [20] by adding spatio-temporal data types, functions, and operations.

2.4 Indexing Technique

General purpose distributed systems have been acceptable for typical analysis tasks as they organize data as non-indexed heap files, where files are partitioned into chunks, each of a specific size. This is typically done on any in-memory or the persistent storage of any distributed frameworks, such as in RDD and HDFS. However, using these systems as-is will result in sub-performance for applications that need indexing. The existing works of supporting spatio-temporal data are either:

- Construct a three dimensional index, such as in CloST [16], SharkDB [43], Elite [37], SciHive [49, 50]. The major problem with this approach is that reconstructing the index is required with every batch update.
- Combine the three spatio-temporal dimensions (i.e., x, y, and time) into a single-dimensional lexicographic key. For example, CoPST [48], GeoMesa [41], and GeoWave [73]. GeoMesa and GeoWave both are built upon Accumulo platform [67] and implemented a space-filling curve to combine the three dimensions of geometry and time. Yet, these systems do not attempt to enhance the spatial locality of data; instead, they rely on time load balancing inherited by Accumulo. Hence, they will have a sup-performance for spatio-temporal operations on highly skewed data.
- Utilize a spatial index and filter on the fly the temporal query predicate. This

approach has been adopted by several systems including, PRADASE [15], TrajStore [36], cloud-based [52], UlTraMan [59], SHAHED [18], TAGHREED [60], DITA [45], PITS [44], Spatio-temporal Join [46], Big Climate [17], and PHiDJ [51].

- Maintain a Hierarchical indexing structure. ST-Hadoop [23] index consists of two-layer indexing of a temporal and spatial. The temporal index disjoint the time interval, meanwhile, the second layer preserve the spatial locality of the spatio-temporal data. The two-layer in ST-Hadoop is replicated in a Temporal Hierarchy. ST-Hadoop trade-off storage to achieve more efficient performance through its index replication. Thus, ST-Hadoop temporally loads spatio-temporal data across computation nodes, in a way that enables ST-Hadoop to utilize the spatial and temporal locality of data.

2.5 Spatio-temporal Operation

In this section we will discuss various queries supported in both area of big spatial and spatio-temporal systems. The main reason of listing spatial queries in this section is that because domain experts who need to process spatio-temporal data tends to either utilize big spatial or spatio-temporal systems for processing their spatio-temporal data. Typically, when big spatial system used then additional temporal filter is added on the top before retrieve the final answer.

As shown on the fifth column of Table 2.1, various fundamental queries supported by the listed systems, namely spatial Range query (SRQ) [9, 10, 11, 13, 14, 31, 32, 34, 35], spatial nearest neighbor queries (SkNN) [10, 11, 13, 14, 31, 32], spatial join query (SJ) [9, 11, 13, 14], temporal range query (TRQ) [23, 52], spatio-temporal range query (STRQ) [15, 16, 17, 18, 23, 36, 37, 41, 42, 43, 44, 49, 50, 52, 54, 59, 60], spatio-temporal nearest neighbor queries (STkNN) [23, 37, 43, 59, 76], spatio-temporal nearest neighbor join queries (STkNNJ) [47], spatio-temporal join query (STJ) [23, 46, 48, 51], and spatio-temporal similarity join query (STSimilarity Join) [45].

ST-Hadoop with its architecture design is the only system that allows researcher, developers, and domain experts to extends its functionality. Currently, ST-Hadoop supports three main functionality of spatio-temporal range query, and spatio-temporal nearest neighbor query, and spatio-temporal join query. More sophisticated operations,

such as similarity queries, or clustering can be realized following the same techniques discussed later in this paper.

2.6 ST-Hadoop Contribution

In our proposed system ST-Hadoop we follow the MapReduce architecture, mainly because most big data systems use the Hadoop Distributed File System (HDFS) as backbone storage in their framework. ST-Hadoop distinguishes itself from other systems discussed earlier in this chapter by the fact that ST-Hadoop is the only system that modifies the MapReduce query processing engine and modifies the file organization of the Hadoop Distributed File System (HDFS) to better support indexing spatio-temporal data.

ST-Hadoop implementation approach is designed as built-in on a generic MapReduce system to support spatio-temporal applications, and assist developers in implementing a wide selection of spatio-temporal operations and indexing techniques. In particular, ST-Hadoop leverages the design of Hadoop and SpatialHadoop to loads and partitions data records according to their time and spatial dimension across computations nodes, which allow the parallelism of processing spatio-temporal queries when accessing its index. In this dissertation, we present several case study of operations that utilize the ST-Hadoop indexing, namely, spatio-temporal range, nearest neighbor, join queries. Besides, we have extended our design to support trajectory data (i.e., a particular type of spatio-temporal data). Our proposed system operations achieve two or more orders of magnitude better performance, mainly because ST-Hadoop is sufficiently aware of both temporal and spatial locality of data records.

Research studies follows on-top approach previously mentioned in tables 2.1 and 2.2. These studies use Hadoop as-is, in which their implementation incurred in poor performance for spatio-temporal applications that need indexing. Meanwhile, ad-hoc technique on big spatial systems reveals its inefficiency for spatio-temporal operations, mainly because spatial system indexes are geared toward supporting spatial queries. In contrast, ST-Hadoop loads and partitions spatio-temporal data across computation nodes, in a way that enables ST-Hadoop to utilize the spatial and temporal locality of data on HDFS, which is not the case with other spatio-temporal systems that only rely

on time load balancing.

There are several SQL like language for Big data systems, such as [14, 18, 23], and CQL [41]. Since Pigeon [20] language is compliant with spatial GCC standard [75], ST-Hadoop will not provide an entirely new language. Instead, it extends the Pigeon language by adding spatio-temporal data types, functions, and operations.

ST-Hadoop index is classified as a Hierarchical multi-version indexing structure. ST-Hadoop [23] index consists of two-layer indexing of a temporal and spatial. The temporal index disjoints the time interval; meanwhile, the second layer preserves the spatial locality of the spatio-temporal data. The two-layer in ST-Hadoop is replicated in a Temporal Hierarchy. ST-Hadoop trade-off storage to achieve more efficient performance through its index replication. Thus, ST-Hadoop temporally loads spatio-temporal data across computation nodes, in a way that mimic and utilize the spatial and temporal locality of data, and achieve spatio-temporal load balancing to their partitions.

ST-Hadoop with its architecture layered design is the only system that allows researcher, developers, and domain experts to extends its functionality. Currently, ST-Hadoop supports several fundamental operations, namely, spatio-temporal range query, spatio-temporal nearest neighbor query, spatio-temporal join query, nearest neighbor trajectory queries, and trajectory similarity query. More sophisticated operations, such as pattern-mining queries, or clustering can be realized following the same techniques discussed later in this thesis.

Table 2.1: Existing Work in the area of Big Spatial data

	Architecture	**Approach**	**Language**	**Index**	**Operation(s)**
BRACE [28]	MapReduce	From-scratch	BRASIL	Grid	SJ
SciDB [29, 30]	Array DB	From-scratch	AQL, AFL	Kd-tree	SRQ, SkNN
SpatialHadoop [14]	MapReduce	Built-in	Pigeon	R-tree,Quad tree, other	SRQ, SkNN, SJ
ESRI tools on Hadoop [31]	MapReduce	Built-in	HiveQL	PMR Quad Tree	SRQ, SkNN
Hadoop-GIS [11]	MapReduce	Built-in	SQL-like	Grid	SRQ, SkNN, SJ
ScalaGiST [32]	MapReduce	Built-in	-	GiST	SRQ, SkNN
Parallel-Secondo [9]	Parallel DB + MapReduce	Built-in	SQL-Like	Local R-tree	SRQ, SJ
Sphinx [33]	Parallel DB	Built-in	SQL	R-tree, Quad-tree	SRQ, SJ
Hippo [34]	Relational DBMS	Built-in	SQL	Bitmap	SRQ
$\mathcal{MD}$-HBase [10]	Key-Value store	Built-in	-	Quad-tree, Kd-tree	SRQ, SkNN
GeoSpark [13]	RDD	Built-in	Scala-based	R-tree, Quad-tree	SRQ, SkNN, SJ
GeoTrellis [12]	RDD	On-top	Scala-based	-	Map Algebra
GISQF [35]	MapReduce	Ad-hoc	-	Grid	SRQ

Table 2.2: Existing Work in the area of Big Spatio-temporal data

	Architecture	**Approach**	**Language**	**Index**	**Operation(s)**
TrajStore [36]	Relational DBMS	From-scratch	SQL	Quad-tree	STRQ
Elite [37]	Parallel DB	From-scratch	-	Oct-tree	STRQ, STkNN
AsterixDB [38, 39]	BDMS	From-scratch	AQL	R-tree	SRQ,Temporal
ST-Hadoop [23]	MapReduce	Built-in	Pigeon	Hierarchical two-Level of temporal (e.g., equi-width or equi-depth), and Spatial indexes (e.g., R-tree, Quad-tree, other)	STRQ,STkNN,STJ
QaDR-tree [40]	MapReduce	Built-in	-	3D quad-tree	STRQ
GeoMesa [41]	Key-Value store	Built-in	CQL	Geohash,SFC	STRQ
GeoWave [42]	Key-Value store	Built-in	-	Geohash,SFC	STRQ
SharkDB [43]	column-oriented DBMS	Built-in	-	Hierarchical Frame based	STRQ,STkNN
PITS [44]	Relational DBMS	Built-in	SQL	Grid + temporal B+tree	STRQ
DITA [45]	RDD	Built-in	-	R-tree	STSimilarity Join
Spatio-temporal Join [46]	RDD	Built-in	-	Partial Local Quad-tree	STJ
CloST [16]	MapReduce	On-top	-	Hierarchical quad-tree	STRQ
PRADASE [15]	MapReduce	On-top	-	Quad-tree + Inverted Index	STRQ
Distributed KNN-Join [47]	MapReduce	On-top	-	-	STkNNJ
Big Climate [17]	MapReduce	On-top	-	Grid	STRQ
CoPST [48]	MapReduce	On-top	-	SFC	STJ
SciHive [49, 50]	MapReduce	On-top	HiveQL	Array	STRQ
PHiDJ [51]	MapReduce	On-top	-	Grid	STJ
cloud-based [52]	Azure [53]	On-top	Extended API	Grid	TRQ, STRQ
Mobility Streaming [54]	Flink (DStream) [27]	On-top	-	-	STRQ
DSI [55]	Storm	On-top	-	Strip Index	KNN
DITIR [56]	Storm	On-top	-	B+tree and R-tree	STRQ
DTR-tree [57]	RDD	On-top	-	2D R-tree	RQ
DMTR-tree [58]	RDD	On-top	-	2D R-tree + Inverted Index	Skyline Query
UlTraMan [59]	RDD	On-top	-	R-tree	STRQ,STkNN
SHAHED [18]	MapReduce	Ad-hoc	Pigeon	Quad-tree	STRQ
TAGHREED [60]	MapReduce	Ad-hoc	-	Quad-tree	STRQ

Chapter 3

ST-Hadoop Architecture Overview

Figure 3.1 gives the high level architecture of our ST-Hadoop system; as the first full-fledged open-source MapReduce framework with a built-in support for spatio-temporal data. ST-Hadoop cluster contains one master node that breaks a map-reduce job into smaller tasks, carried out by slave nodes. Three types of users interact with ST-Hadoop: (1) *Casual users* who access ST-Hadoop through its spatio-temporal language to process their datasets. (2) *Developers*, who have a deeper understanding of the system internals and can implement new spatio-temporal operations, and (3) *Administrators*, who can tune up the system through adjusting system parameters in the configuration files provided with the ST-Hadoop installation. ST-Hadoop adopts a layered design of four main layers, namely, *language*, *Indexing*, *MapReduce*, and *operations* layers, described briefly below:

Language Layer: This layer extends Pigeon language [20] to supports spatio-temporal data types (i.e., STPoint, time and interval) and spatio-temporal operations (e.g., overlap, and join). The Pigeon Language is a high level language complaint with OGC standards. Details are given in chapter 8.

Indexing Layer: ST-Hadoop spatiotemporally loads and partitions data across

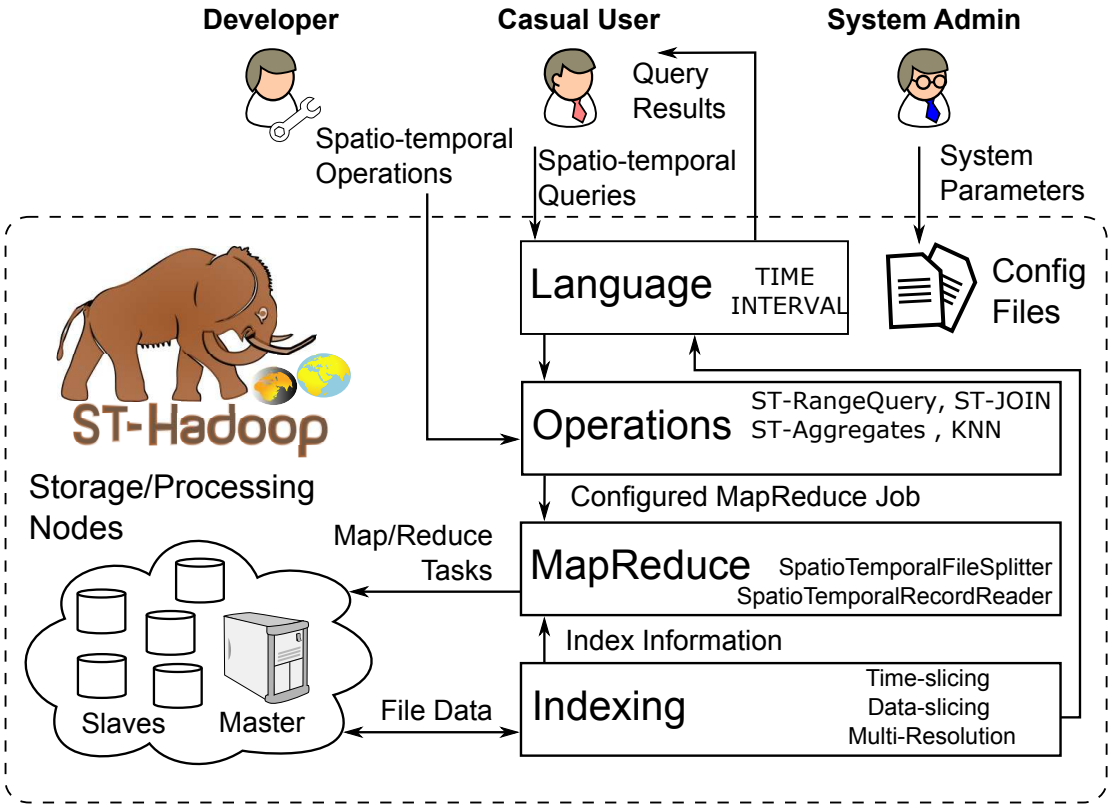


Figure 3.1: ST-Hadoop system architecture

computation nodes. In this layer ST-Hadoop scans a random sample obtained from the input dataset, bulk-loads its spatio-temporal index that consists of two-layer indexing of temporal and then spatial. Finally ST-Hadoop replicates its index into *temporal hierarchy index structure* to achieve more efficient performance for processing spatio-temporal queries. ST-Hadoop introduces two techniques for partitioning temporal dimension of space and data partitioning namely, Time-slicing and Data-slicing. As for the spatial level of partitioning, ST-Hadoop partitions the spatial dimensions using any of the implemented spatial bull-loading partitioning techniques in ST-Hadoop, such as R-tree, R+-tree, Z-Curve, Grid, Quad-tree, or KD-tree. Details of the index layer are given in chapter 4.

MapReduce Layer: In this layer, new implementations added inside SpatialHadoop

MapReduce layer to enables ST-Hadoop to exploits its spatio-temporal indexes and realizes spatio-temporal predicates. In particular, *SpatioTemporalFileSplitter* and *SpatioTemporalRecordReader*, which allows to access the two-level of spatio-temporal indexes in ST-Hadoop and reads spatio-temporal within partitions, respectively. We are not going to discuss this layer any further, mainly because few changes made to inject time awareness in this layer. The implementation of MapReduce layer was already discussed in great details [14].

Operations Layer: This layer encapsulates the implementation of three common spatio-temporal operations, namely, spatio-temporal range, nearest neighbor, and join queries. More operations can be added to this layer by ST-Hadoop *developers*. Details of the operations layer are discussed in chapter 5.

The goal of this thesis is to describe how Hadoop as big distributed Map-Reduce systems can be modified in its internal components to support spatio-temporal data and applications. In a nutshell, ST-Hadoop Cluster contains one master node and several worker nodes. The Master node in ST-Hadoop triggers and manages the spatio-temporal operations. Meanwhile, the worker nodes carry the computations as map-reduce tasks. Through this document, we will describe the full stack of ST-Hadoop starting from storage, indexing, operations, and language. In our architecture design, we allow experts users and system administrators to tune ST-Hadoop configuration files, to guide how spatio-temporal data partition, along with other basic cluster configurations. We envision the layered design of ST-Hadoop will act as a research engine for domain experts and research to extends its capability and builds applications on the top-of ST-Hadoop.

Chapter 4

Spatio-temporal Indexing in MapReduce Layer

Input files in Hadoop Distributed File System (HDFS) are organized as a heap structure, where the input is partitioned into chunks, each of size 64MB. Given a file, the first 64MB is loaded to one partition, then the second 64MB is loaded in a second partition, and so on. While that was acceptable for typical Hadoop applications (e.g., analysis tasks), it will not support spatio-temporal applications where there is always a need to filter input data with spatial and temporal predicates. Meanwhile, spatially indexed HDFSs, as in SpatialHadoop [14] and ScalaGiST [32], are geared towards queries with spatial predicates only. This means that a temporal query to these systems will need to scan the whole dataset. Also, a spatio-temporal query with a small temporal predicate may end up scanning large amounts of data. For example, consider an input file that includes all social media contents in the whole world for the last five years or so. A query that asks about contents in the USA in a certain hour may end up in scanning all the five years contents of the USA to find out the answer.

ST-Hadoop HDFS organizes input files as spatio-temporal partitions that satisfy one main goal of supporting spatio-temporal queries. ST-Hadoop imposes temporal slicing, where input files are spatiotemporally loaded into intervals of a specific time granularity, e.g., days, weeks, or months. Each granularity is represented as a level in ST-Hadoop index. Data records in each level are spatiotemporally partitioned, such

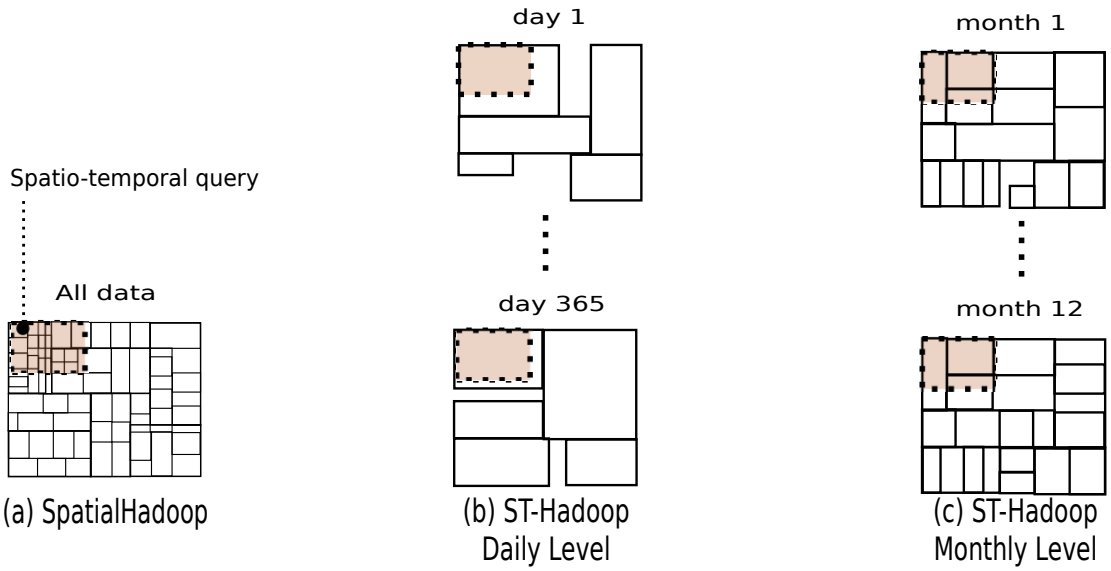


Figure 4.1: HDFSs in ST-Hadoop VS SpatialHadoop

that the boundary of a partition is defined by a spatial region and time interval.

Figures 4.1(a) and 4.1(b) show the HDFS organization in SpatialHadoop and ST-Hadoop frameworks, respectively. Rectangular shapes represent boundaries of the HDFS partitions within their framework, where each partition maintains a 64MB of nearby objects. The dotted square is an example of a spatio-temporal range query. For simplicity, let's consider a one year of spatio-temporal records loaded to both frameworks. As shown in Figure 4.1(a), SpatialHadoop is unaware of the temporal locality of the data, and thus, all records will be loaded once and partitioned according to their existence in the space. Meanwhile in Figure 4.1(b), ST-Hadoop loads and partitions data records for each day of the year individually, such that each partition maintains a 64MB of objects that are close to each other in both space and time. Note that HDFS partitions in both frameworks vary in their boundaries, mainly because spatial and temporal locality of objects are not the same over time. Let's assume the spatio-temporal query in the dotted square "*find objects in a certain spatial region during a specific month*" in Figures 4.1(a), and 4.1(b). SpatialHadoop needs to access all partitions overlapped with query region, and hence SpatialHadoop is required to scan one year of records to get the final answer. In the meantime, ST-Hadoop reports the query answer by accessing

few partitions from its daily level without the need to scan a huge number of records.

4.1 Concept of Hierarchy

ST-Hadoop imposes a replication of data to support spatio-temporal queries with different granularities. The data replication is reasonable as the storage in ST-Hadoop cluster is inexpensive, and thus, sacrificing storage to gain more efficient performance is not a drawback. Updates are not a problem with replication, mainly because ST-Hadoop extends MapReduce framework that is essentially designed for batch processing, thereby ST-Hadoop utilizes incremental batch accommodation for new updates.

The key idea behind the performance gain of ST-Hadoop is its ability to load the data in Hadoop Distributed File System (HDFS) in a way that mimics spatio-temporal index structures. To support all spatio-temporal operations including more sophisticated queries over time, ST-Hadoop replicates spatio-temporal data into a *Temporal Hierarchy Index*. Figures 4.1(b) and 4.1(c) depict two levels of days and months in ST-Hadoop index structure. The same data is replicated on both levels, but with different spatio-temporal granularities. For example, a spatio-temporal query asks for objects in one month could be reported from any level in ST-Hadoop index. However, rather than hitting 30 days' partitions from the daily-level, it will be much faster to access less number of partitions by obtaining the answer from one month in the monthly-level.

A system parameter can be tuned by ST-Hadoop administrator to choose the number of levels in the *Temporal Hierarchy index*. By default, ST-Hadoop set its index structure to four levels of days, weeks, months and years granularities. However, ST-Hadoop users can easily change the granularity of any level. For example, the following code loads taxi trajectory dataset from "NYC" file using one-hour granularity, Where the `Level` and `Granularity` are two parameters that indicate which level and the desired granularity, respectively.

```
trajectory = LOAD 'NYC' as
           (id:int, STPoint(loc:point, time:timestamp))
           Level:1  Granularity:1-hour;
```

4.2 Index Construction

Figure 4.2 illustrates the indexing construction in ST-Hadoop, which involves two scanning processes. The first process starts by scanning input files to get a random sample, and this is essential because the size of input files is beyond memory capacity, and thus, ST-Hadoop obtains a set of records to a sample that can fit in memory. Next, ST-Hadoop processes the sample n times, where n is the number of levels in ST-Hadoop index structure. The temporal slicing in each level splits the sample into m number of slice (e.g., $slice_{1.m}$). ST-Hadoop finds the spatio-temporal boundaries by applying a spatial indexing on each temporal slice individually. As a result, outputs from temporal slicing and spatial indexing collectively represent the spatio-temporal boundaries of ST-Hadoop index structure. These boundaries will be stored as meta-data on the master node to guide the next process. The second scanning process physically assigns data records in the input files with its overlapping spatio-temporal boundaries. Note that each record in the dataset will be assigned n times, according to the number of levels.

ST-Hadoop index consists of two-layer indexing of a temporal and spatial. The conceptual visualization of the index is shown in the right of Figure 4.2, where lines signify how the temporal index divided the sample into a set of disjoint time intervals, and triangles symbolize the spatial indexing. This two-layer indexing is replicated in all levels, where in each level the sample is partitioned using different granularity. ST-Hadoop trade-off storage to achieve more efficient performance through its index replication. In general, the index creation of a single level in the *Temporal Hierarchy* goes through four consecutive phases, namely sampling, temporal slicing, spatial indexing, and physical writing.

4.2.1 Phase I Sampling

The objective of this phase is to approximate the spatial distribution of objects and how that distribution evolves over time, to ensure the quality of indexing; and thus, enhance the query performance. This phase is necessary, mainly because the input files are too large to fit in memory. ST-Hadoop employs a map-reduce job to efficiently read a sample through scanning all data records. We fit the sample into an in-memory simple

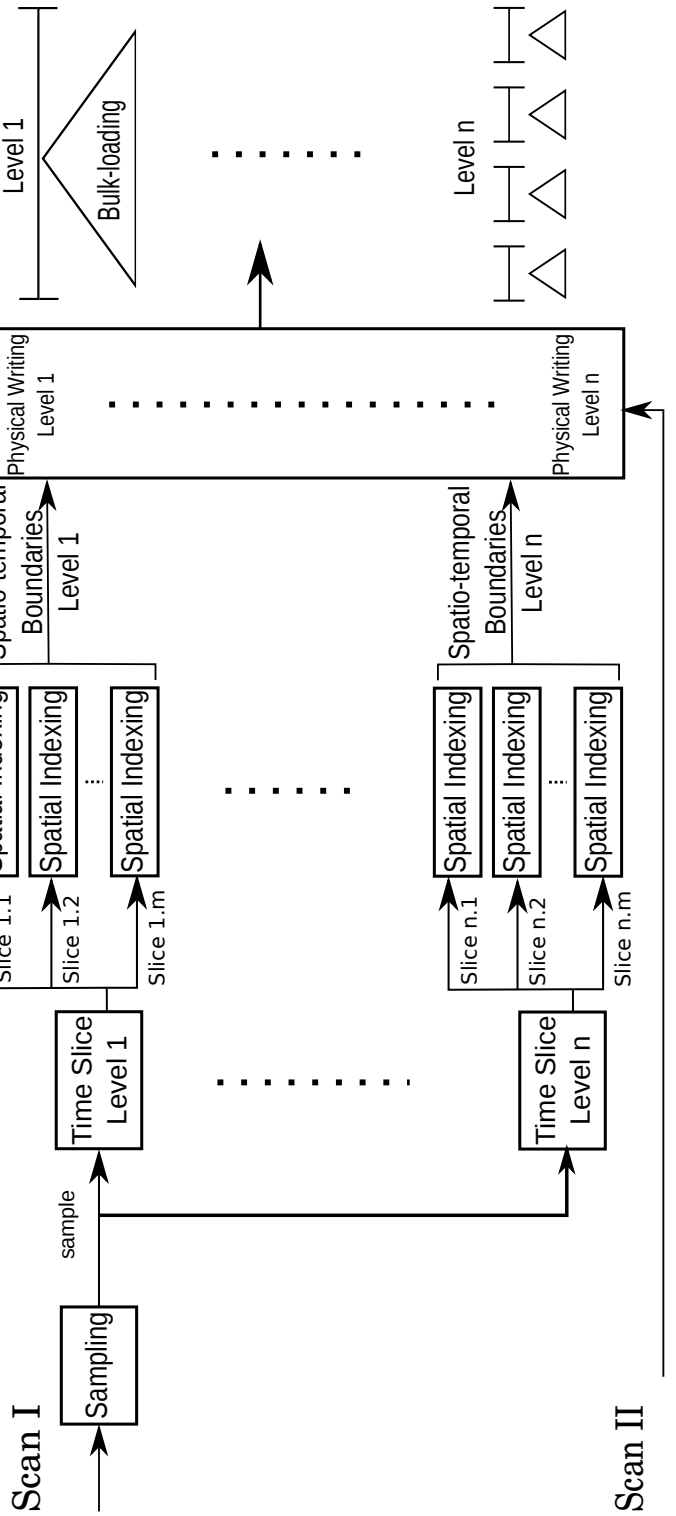


Figure 4.2: Indexing in ST-Hadoop

data structure of a length (L), that is an equal to the number of HDFS blocks, which can be directly calculated from the equation $L = (Z/B)$, where Z is the total size of input files, and B is the HDFS block capacity (e.g., 64MB). The size of the random sample is set to a default ratio of 1% of input files, with a maximum size that fits in the memory of the master node. This simple data structure represented as a collection of elements; each element consist of a time instance and a space sampling that describe the time interval and the spatial distribution of spatio-temporal objects, respectively. Once the sample is scanned, we sort the sample elements in chronological order to their time instance, and thus the sample approximates the spatio-temporal distribution of input files.

4.2.2 Phase II Temporal Slicing

In this phase ST-Hadoop determines the temporal boundaries by slicing the in-memory sample into multiple time intervals, to efficiently support a fast random access to a sequence of objects bounded by the same time interval. ST-Hadoop employs two temporal slicing techniques, where each manipulates the sample according to specific slicing characteristics: (1) *Time-partition*, slices the sample into multiple splits that are uniformly on their time intervals, and (2) *Data-partition* where the sample is sliced to the degree that all sub-splits are uniformly in their data size. The output of this phase finds the temporal boundary of each split, that collectively cover the whole time domain.

The rational reason behind ST-Hadoop two temporal slicing techniques is that for some spatio-temporal archive the data spans a long time-interval such as decades, but their size is moderated compared to other archives that are daily collect terabytes or petabytes of spatio-temporal records. ST-Hadoop proposed the two techniques to slice the time dimension of input files based on either time-partition or data-partition, to improve the indexing quality, and thus gain efficient query performance. The time-partition slicing technique serves best in a situation where data records are uniformly distributed in time. Meanwhile, data-partition slicing best suited with data that are sparse in their time dimension.

∗ **Data-partition Slicing.** The goal of this approach is to slice the sample to

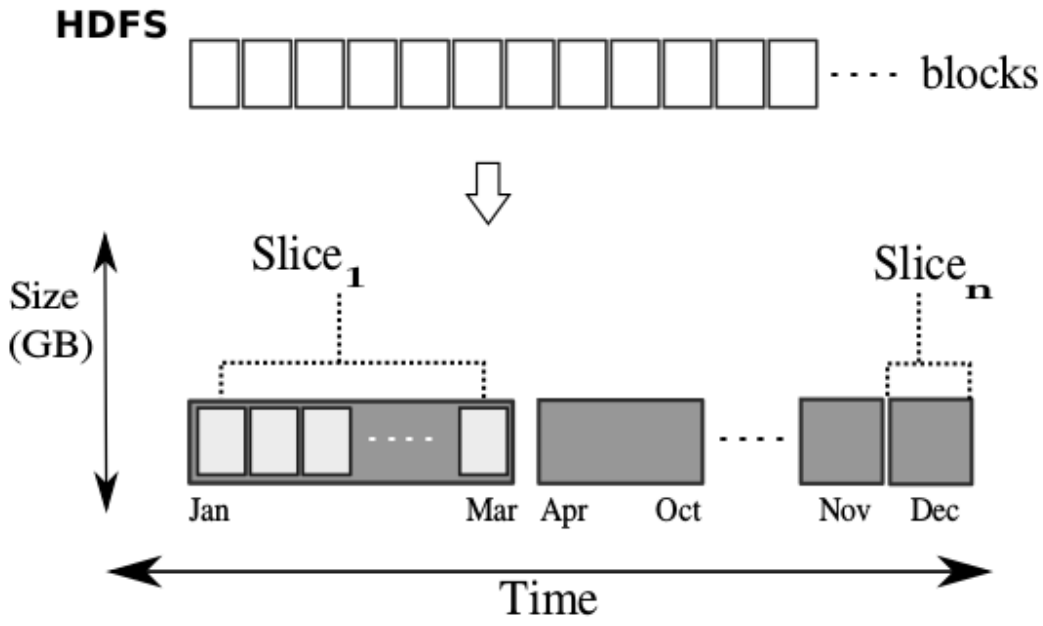


Figure 4.3: Data-Slice

the degree that all sub-splits are equally in their size. Figure 4.3 depicts the key concept of this slicing technique, such that a $\texttt{slice}_1$ and $\texttt{slice}_n$ are equally in size, while they differ in their interval coverage. In particular, the temporal boundary of $\texttt{slice}_1$ spans more time interval than $\texttt{slice}_n$. For example, consider 128MB as the size of HDFS block and input files of 1 TB. Typically, the data will be loaded into 8 thousand blocks. To load these blocks into ten equally balanced slices, ST-Hadoop first reads a sample, then sort the sample, and apply `Data-partition` technique that slices data into multiple splits. Each split contains around 800 blocks, which hold roughly a 100 GB of spatio-temporal records. There might be a small variance in size between slices, which is expectable. Similarly, another level in ST-Hadoop temporal hierarchy index could loads the 1 TB into 20 equally balanced slices, where each slice contains around 400 HDFS blocks. ST-Hadoop users are allowed to specify the granularity of data slicing by tuning α parameter. By default four ratios of α is set to 1%, 10%, 25%, and 50% that create the four levels in ST-Hadoop index structure.

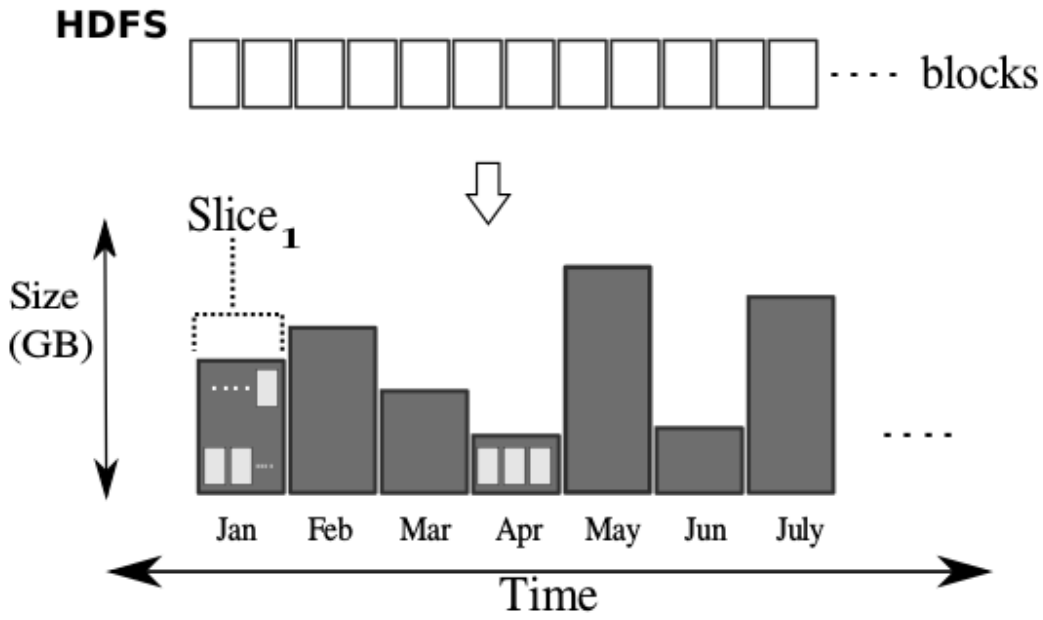


Figure 4.4: Time-Slice

* **Time-partition Slicing.** The ultimate goal of this approach is to slices the input files into multiple HDFS chunks with a specified interval. Figure 4.4 shows the general idea, where ST-Hadoop splits the input files into an interval of one-month granularity. While the time interval of the slices is fixed, the size of data within slices might vary. For example, as shown in Figure 4.4 Jan slice has more HDFS blocks than April.

ST-Hadoop users are allowed to specify the granularity of this slicing technique, which specified the time boundaries of all splits. By default, ST-Hadoop finer granularity level is set to one-day. Since the granularity of the slicing is known, then a straightforward solution is to find the minimum and maximum time instance of the sample, and then based on the intervals between the both times ST-Hadoop hashes elements in the sample to the desired granularity. The number of slices generated by the time-partition technique will highly depend on the intervals between the minimum and the maximum times obtained from the sample. By default, ST-Hadoop set its index structure to four levels of days, weeks, months and years granularities.

4.2.3 Phase III Spatial Indexing

This phase ST-Hadoop determines the spatial boundaries of the data records within each temporal slice. ST-Hadoop spatially index each temporal slice independently; such decision handles a case where there is a significant disparity in the spatial distribution between slices, and also to preserve the spatial locality of data records. Using the same sample from the previous phase, ST-Hadoop takes the advantages of applying different types of spatial bulk loading techniques in HDFS that are already implemented in SpatialHadoop such as Grid, R-tree, Quad-tree, and Kd-tree. The output of this phase is the spatio-temporal boundaries of each temporal slice. These boundaries stored as a meta-data in a file on the master node of ST-Hadoop cluster. Each entry in the meta-data represents a partition, such as $< id, MBR, interval, level >$. Where id is a unique identifier number of a partition on the HDFS, MBR is the spatial minimum boundary rectangle, $interval$ is the time boundary, and the level is the number that indicates which level in ST-Hadoop temporal hierarchy index.

4.2.4 Phase IV Physical Writing

Given the spatio-temporal boundaries that represent all HDFS partitions, we initiate a map-reduce job that scans through the input files and physically partitions HDFS block, by assign data records to overlapping partitions according to the spatio-temporal boundaries in the meta-data stored on the master node of ST-Hadoop cluster. For each record r assigned to a partition p, the map function writes an intermediate pair $\langle p, r \rangle$ Such pairs are then grouped by p and sent to the reduce function to write the physical partition to the HDFS. Note that for a record r will be assigned n times, depends on the number of levels in ST-Hadoop index.

4.3 Index Maintenance

This index structure can be described as a temporal hierarchy for spatio-temporal indices as shown in Figure 4.5. ST-Hadoop merges a set of temporal slices from the lower most layer to create a slice with a larger time interval. For simplicity let's assume the lowermost layer was sliced into days, then ST-Hadoop combines a set days to create a

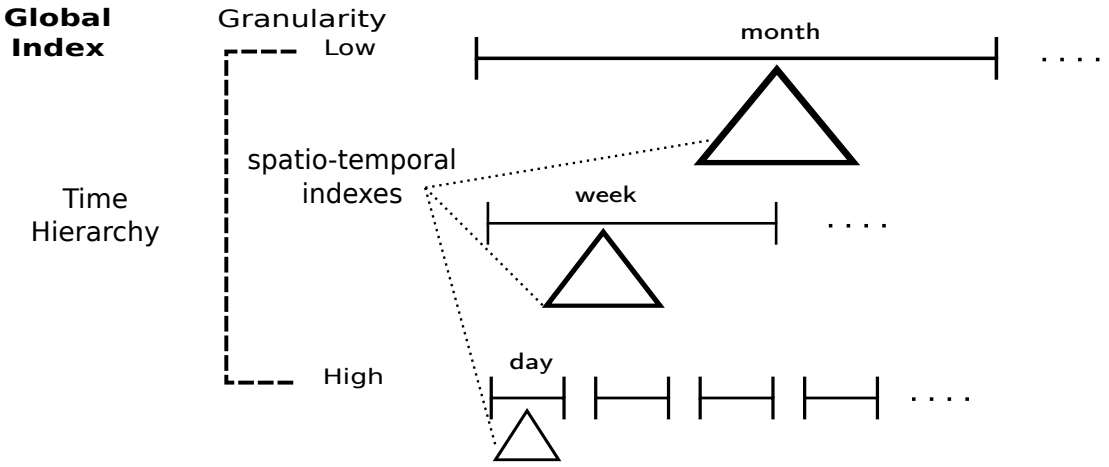


Figure 4.5: Temporal Hierarchy Index

week-slice in the layer above. Likewise, ST-Hadoop reads a sample from the merged set to bulk load its spatio-temporal index. Note that this step is necessary as the size and the distribution of objects vary from a lower (i.e., day) to the above layer (i.e., week). For each layer in the hierarchical index, ST-Hadoop iterates two-level bulk loading techniques of a temporal and then spatial, with different time granularity. A system parameter can be tuned by ST-Hadoop administrator to choose the number of layers and their granularity. By default, ST-Hadoop set its *temporal hierarchy index* to four layers with a resolution of days, weeks, months and years, respectively. Similarly, the granularity of the four layers in *Data-based* slicing will have different slicing ratios (α), such as 1%, 10%, 25%, and 50%.

ST-Hadoop in a regular base such as every day maintains its *Temporal Hierarchy Index*, to reflects updates on the index with the incoming data. First, it creates a new two-level indexing in the lowest layer using one MapReduce job to index spatio-temporal records similar to the same granularity of that layer. Then check if the newly created index will help to create an index in the above layer, if not then it will be carried out for a next maintenance call. During the maintenance, if there is any indices contribute to the above layer, then data of these indices will be merged, and a new two-level indexing will be created with a bigger granularity.

Table 4.1: Twitter Datasets

Twitter Data	Size	Num-Records	Time window
`Large`	10TB	> 1 Billion	> 3 years
`Average-Large`	6.7TB	692 Million	1 years
`Medium-Large`	3TB	152 Million	9 months
`Moderate-Large`	(1TB)	115 Million	3 months

4.4 Experiments

This section provides an extensive experimental performance study of ST-Hadoop compared to SpatialHadoop and Hadoop. We decided to compare with this two frameworks and not other spatio-temporal DBMSs for two reasons. First, as our contributions are all about spatio-temporal data support in Hadoop. Second, the different architectures of spatio-temporal DBMSs have great influence on their respective performance, which is out of the scope of this paper. Interested readers can refer to a previous study [77] which has been established to compare different large-scale data analysis architectures. In other words, ST-Hadoop is targeted for Hadoop users who would like to process large-scale spatio-temporal data but are not satisfied with its performance. The experiments are designed to show the effect of ST-Hadoop indexing and the overhead imposed by its new features compared to SpatialHadoop. However, ST-Hadoop achieves two orders of magnitude improvement over SpatialHadoop and Hadoop.

4.4.1 Experimental Settings

Cluster Setup. All experiments are conducted on a dedicated internal cluster of 24 nodes. Each has 64GB memory, 2TB storage, and Intel(R) Xeon(R) CPU 3GHz of 8 core processor. We use Hadoop 2.7.2 running on Java 1.7 and Ubuntu 14.04.5 LTS. Table 4.2 summarizes the configuration parameters used in our experiments. Default parameters (in parentheses) are used unless mentioned.

Datasets. To test the performance of ST-Hadoop we use the Twitter archived dataset [2]. The dataset collected using the public Twitter API for more than three years, which contains over 1 Billion spatio-temporal records with a total size of 10TB. To scale out time in our experiments we divided the dataset into different time intervals

Table 4.2: ST-Hadoop Experiments Parameters

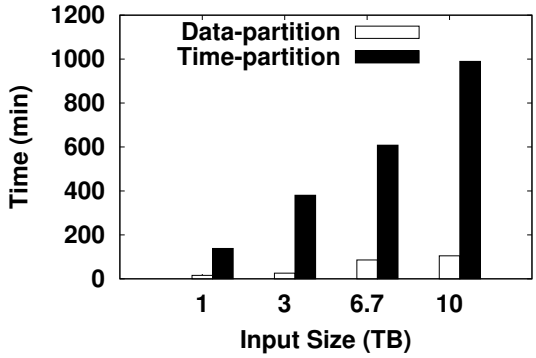
Parameter	Values (default)
HDFS block capacity (B)	32, 64, (128), 256 MB
Cluster size (N)	5, 10, 15, 20, (23)
Selection ratio (ρ)	(0.01), 0.02, 0.05, 0.1, 0.2, 0.5, 1.0
Data-partition slicing ratio(α)	0.01, 0.02, 0.025, 0.05, (0.1), 1
Time-partition slicing granularity(σ)	(days), weeks, months, years
Spatio-temporal proximity (α)	0,0.2, (0.5), 0.6, 0.8, 1.0

and sizes, respectively as shown in Table 4.1. The default size used is 1TB which is big enough for our extensive experiments unless mentioned.

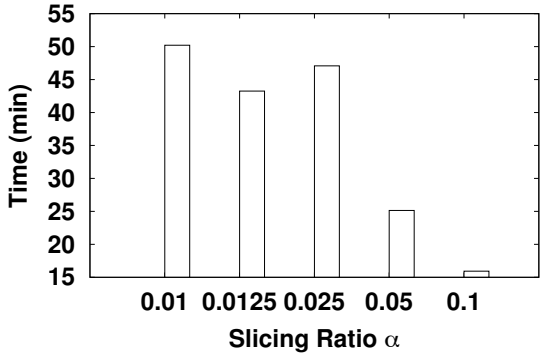
4.4.2 Index Construction

Figure 4.6(a) gives the total time for building the spatio-temporal index in ST-Hadoop. This is a one time job done for input files. In general, the figure shows excellent scalability of the index creation algorithm, where it builds its index using data-partition slicing for a 1TB file with more than 115 Million records in less than 15 minutes. The data-partition technique turns out to be the fastest as it contains fewer slices than time-partition. Meanwhile, the time-partition technique takes more time, mainly because the number of partitions are increased, and thus increases the time in physical writing phase.

In Figure 4.6(b), we configure the temporal hierarchy indexing in ST-Hadoop to construct five levels of the two-layer indexing. The temporal indexing uses `Data-partition` slicing technique with different slicing ratio α. We evaluate the indexing time of each level individually. Because the input files are sliced into splits according to the slicing ratio, which directly effects on the number of partitions. In general with stretching the slicing ratio, the indexing time decreases, mainly because the number of partitions will be much less. However, note that in some cases the spatial distribution of the slice might produce more partitions as in shown with 0.25% ratio.



(a) Input Files)



(b) Data-partition

Chapter 5

Spatio-temporal Operations on MapReduce

The combination of the spatiotemporally load balancing with the temporal hierarchy index structure gives the core of ST-Hadoop, that enables the possibility of efficient and practical realization of spatio-temporal operations, and hence provides orders of magnitude better performance over Hadoop and SpatialHadoop. In this section, we discuss several fundamental spatio-temporal operations, namely, range (Section 5.1), kNN (Section 5.2), and join (Sections 5.3) as case studies of how to exploit the spatio-temporal indexing in ST-Hadoop. Other operations can also be realized following similar approaches.

5.1 Spatio-temporal Range Query

A range query is specified by two predicates of a spatial area and a temporal interval, A and T, respectively. The query finds a set of records R that overlap with both a region A and a time interval T, such as *"finding geotagged news in California area during the last three months"* . ST-Hadoop employs its spatio-temporal index described in Section 4 to provide an efficient algorithm that runs in three steps, *temporal filtering*, *spatial search*, and *spatio-temporal refinement*, described below.

In the **temporal filtering** step, the hierarchy index is examined to select a subset of partitions that cover the temporal interval T. The main challenge in this step is

that the partitions in each granularity cover the whole time and space, which means the query can be answered from any level individually or we can mix and match partitions from different level to cover the query interval T. Depending on which granularities are used to cover T, there is a tradeoff between the number of matched partitions and the amount of processing needed to process each partition. To decide whether a partition P is selected or not, ST-Hadoop computes the coverage ratio along with the number of partitions needed to be processed and then selects the granularity based on the minimum number of partitions.

In the **spatial search** step, Once the temporal partitions are selected, the *spatial search* step applies the spatial range query against each matched partition to select records that spatially match the query range A. Keep in mind that each partition is spatiotemporally indexed which makes queries run very efficiently. Since these partitions are indexed independently, they can all be processed simultaneously across computation nodes in ST-Hadoop, and thus maximizes the computing utilization of the machines.

Finally in the **spatio-temporal refinement** step, compares individual records returned by the *spatial search* step against the query interval T, to select the exact matching records. This step is required as some of the selected temporal partitions might partially overlap the query interval T and they need to be refined to remove records that are outside T. Similarly, there is a chance that selected partitions might partially overlap with the query area A, and thus records outside the A need to be excluded from the final answer.

5.2 Spatio-temporal *k*NN Query

The spatio-temporal nearest neighbor query takes a spatio-temporal point Q, a spatio-temporal predicates θ, a spatio-temporal ranking function F_α, and an integer k as an input, and returns the k spatiotemporally closest points to Q such that: (1) The k points are within the temporal distance θ_{time}. (2) The k points are not far from the spatial distance θ_{space}. (3) The top k points are ranked according to the spatio-temporal ranking function F_α that combines the spatial proximity and the temporal closeness of $p \in P$ to the query point Q. For example, a crime analyst might be interested to find the relationship between crimes, which can be described as "*find the top 10*

closest crimes to a given crime Q in downtown that took place on the 2^{nd} *during last year*". With the spatio-temporal information of the query point Q, ST-Hadoop adds a spatio-temporal ranking function F_α to the kNN query. The ranking function allows ST-Hadoop to compromise between spatial proximity and temporal closeness of its top-k points to the the query point.

Definition Spatio-temporal Ranking Function.
The ranking function F_α indicates whether a user query leans toward spatial proximity or temporal concurrency. If $\alpha = 1$, then the user cares about spatial closeness, i.e., the top-k results will be spatially closest to the query point. If $\alpha = 0$, then the user cares about temporal recency, i.e., the top-k results will be temporally recent to query point. Meanwhile, if α value is between zero and one, then the user cares about spatio-temporal proximity. The spatio-temporal proximity can be computed with the following mathematical equation.

$$\begin{aligned} F_\alpha(Q,p) = \quad & \alpha \times SpatialDist(Q.loction, p.location) \\ + \quad & (1-\alpha) \times TemporalDist(Q.timestamp, p.timestamp) \end{aligned}$$

The spatio-temporal ranking function F_α dependents on both *SpatialDist* and the *TemporalDist* functions, which they are normalized and monotonic. Each has a value range from zero to one. The *SpatialDist* is the Euclidean distance between two points' locations divided by the maximum spatial distance θ_{space}, where θ_{space} is the distance from a query point Q to the k^{th} furthest location. Meanwhile, the *TemporalDist* is that ratio of delta times of Q and p to the total temporal interval θ_{time}. The temporal interval θ_{time} is the time distance from the query point Q to the k^{th} furthest point in time.

Figure 5.1 gives a landscape of all possible ways to process the kNN operation in ST-Hadoop. Without loss of generality, let's suppose that the ST-Hadoop indexes input files into intervals of days, and a user is interested in discovering the top k points to a given query point Q during the last year. As shown in the top of the Figure, One extreme when α is equal to one, which indicates that the user cares about spatial proximity in

their k result. Hence, all partitions overlap with query point Q needs to be processed from the last year. On the opposite side if α is equal to zero, then the user cares about temporal closeness in their k result. This means that at most we are going to process partitions at the same time interval. Between those two extremes reside the challenge, such that to what extent we need to process partitions from other time intervals to find the top-k points. ST-Hadoop applies a simple and efficient technique that capable of pruning the search space to process only n number of partitions, which guarantee that those partitions will have the final k answers.

In Hadoop, a kNN query scans entire points in input files, calculates their spatio-temporal similarity distance to the query point Q, and provides the top-k to Q [78, 79]. Meanwhile, in spatially indexed HDFS's [8, 14, 32, 80], only spatial kNN is supported. This means that the kNN operation on a spatially indexed HDFS also needs to scan all points in input files to search for the temporal closeness. ST-Hadoop considers both space and time; and thus, in its spatio-temporal kNN query exploits simple pruning techniques to achieve orders of magnitude better performance. ST-Hadoop kNN algorithm runs in three phases, kNN initial answer, correctness check, and kNN refinement.

In the ***k*NN initial answer phase**, we come up with an initial answer of the k closest points to Q within a single partition in the HDFS. ST-Hadoop first locates the partition that includes Q, by feeding the *SpatioTmeporalFileSplitter* with a filter function that selects only the overlapping partition from the temporal interval. ST-Hadoop exploits a *SpatioTmeporalRecordReader* to reads the selected partition, then executes a traditional kNN algorithm to produce the initial k answers. The function F_α computes the spatio-temporal distance between any points and the query point Q.

In the **correctness check phase**, we check if the initial k answer can be considered final. The main idea of this phase is to draw a test Cylinder centered at Q with radius r equal to the spatial distance from Q to its k^{th} furthest neighbor in space. The height l of the cylinder is equal to the temporal distance from Q to its k^{th} furthest neighbor in time. The radius and the height of the cylinder change only if there is potential point dominate the score of the ranking functions of the furthest k^{th} point in the initial

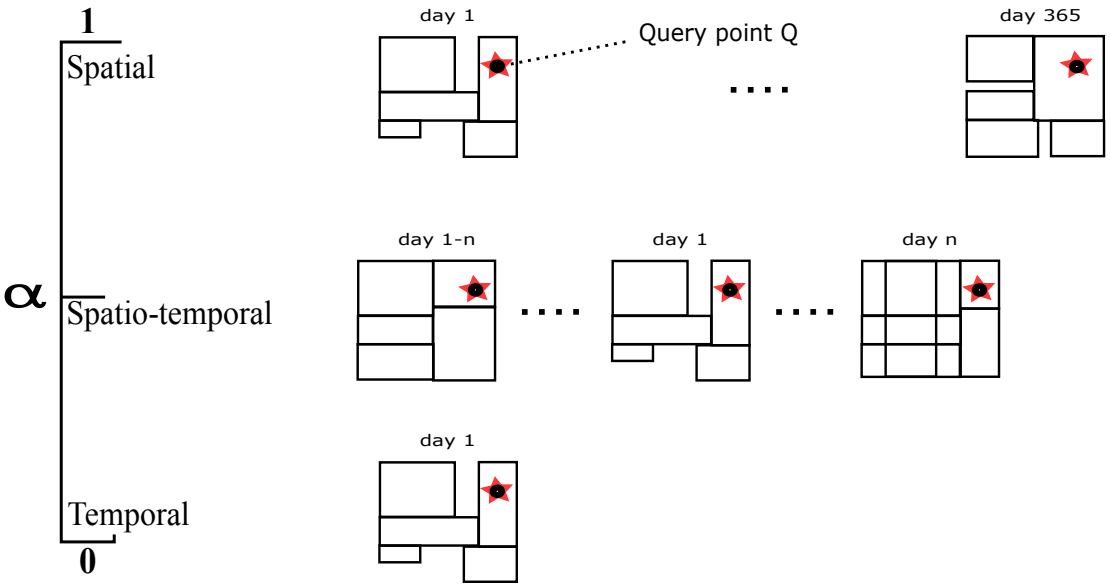


Figure 5.1: Landscape of spatio-temporal kNN operation

results in any dimension, i.e., space or time. If the cylinder does not overlap with any partitions spatially or temporally, then we terminate the process, and the initial answer is considered final. Otherwise, we proceed to the next phase.

Three cases encounter when we draw the test cylinder to check for correctness in this phase as follows.

• **Case 1 ($\alpha = 1$):** If a user specifies α with 1, then the user cares more about spatial proximity than temporal. This means that we need to check the correctness of all time intervals. As shown in the top of Figure 5.1, the query point Q overlap with all year partitions. If input files only indexed in one level, then the cylinder height is equal to the whole θ_{time}. On the other hand, if the input files indexed into a temporal hierarchy, then rather than accessing a huge number of partitions, ST-Hadoop feeds the temporal query predicate θ_{time} to its query optimizer. The query optimizer will generate an execution plan that selects the overlap partition with Q from a lower granularity level, i.e., yearly level. Next, we execute a traditional kNN algorithm to produce new initial k answers again. The new height of the cylinder is going to be equal to zero. Next, we draw a cylinder centered at Q with a radius equal to the furthest k^{th} neighbor. If the

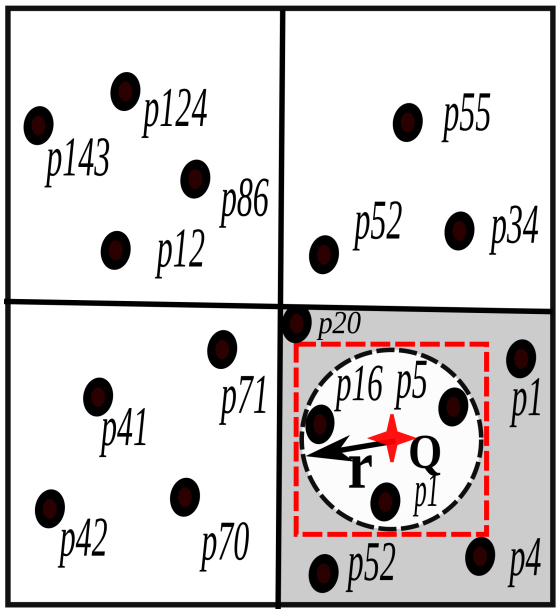


Figure 5.2: Correctness check Final Answer

cylinder does not overlap with any partition other than Q, then we terminate, and the new initial answer considered final. Otherwise, we processed to the next phase.

• **Case 2 ($\alpha = 0$):** If a user specifies α with zero, then the user cares more about temporal proximity. This means that we need to check the correctness from the same time interval. First, we draw a cylinder centered at Q with a radius equal to the spatial distance from Q to its k^{th} furthest neighbor, obtained from the initial answer. The height of the cylinder is equal to zero. If the cylinder does not overlap with any partition other than Q, then we terminate the process, and the initial answer considered final. Otherwise, we processed to the next phase.

Figure 5.2 gives an example of a kNN query for point Q with a $k = 3$ and α is equal to zero. The shaded partitions are the one considered in the processing. The dotted test cylinder has a height equal to zero, composed from the initial answer p1, p5, p16. The cylinder does not overlap with any other partitions than Q; thus, the initial answer is considered final.

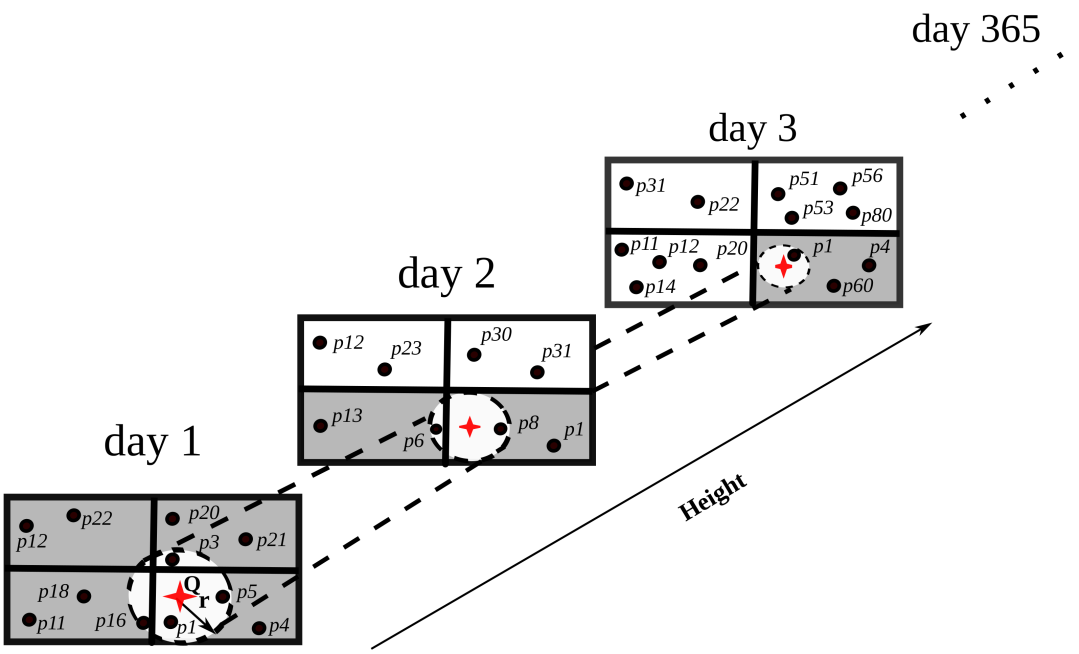


Figure 5.3: Correctness Check when $0 \leq \alpha \leq 1$

• **Case 3** ($0 \leq \alpha \leq 1$)**:** If a user specifies any α value between zero and one, then this means that the user cares about the spatio-temporal proximity. The main idea is to gradually draw the cylinder and make sure that the k^{th} furthest neighbor point is not dominated by any other points in both dimensions, i.e., space and time. A point dominates the k^{th} point if it is as good or better in ranking score, and better at least in one dimension of either spatial or temporal.

Figure 5.3 illustrates the idea of test cylinder. The query point Q initially overlap with a single time interval, e.g., day 1. We check if some points either in the next or previous interval can dominate the score of the k^{th} furthest neighbor from the same initial interval, e.g., day 1. If a dominance point exists, then we modify the cylinder height and radius accordingly in the next interval. Notice that the radius of the cylinder in next time interval is getting smaller, this is because we gradually draw the test cylinder. We continue this process until we reach a time interval that has no dominance point that can dominate the k^{th} furthest point.

The cautiously drawing of the test cylinder algorithm has two consecutive steps as

follows.

Step I: In this step, we check if there is a point from a different partition(s) exist within the same temporal interval, such that it can dominate the ranking score of the initial furthest k^{th} neighbor. In other words, in this step, we determine the radius of the cylinder within the same temporal interval. First, Starting from the partition that overlap with Q from the same time interval, we draw the circle of the cylinder centered at Q with a radius equal to the spatial distance of the k^{th} furthest neighbor. If partitions overlap with the circle's MBR, then we check if the nearest point from the overlapped partition(s) can dominate the score of the initial furthest k^{th}. If a dominance exists, then we consider processing this partition, update our furthest k^{th} in the initial answer with the dominating point, and subsequently, we proceed to the next step.

Step II: In this step, we modify the height of the test cylinder by checking if there is a point from the next or the previous temporal interval can dominate the furthest k^{th} neighbor. For the sake of simplicity, let's consider the next temporal interval in our discussion. However, the presented technique is operated to examine interval in both directions. First, we find the partition that overlaps with Q from the next time interval. Then, we check if the temporal distance along with the minimum spatial distance between Q and the new partition can dominate the furthest k^{th}. If the score beats the k^{th}, then a dominance might exist in that partition. Thus, we consider processing this partition by modifying the height of the cylinder. Recursively we repeat the processing of the two steps with every new interval appended to the cylinder height until no further dominance exist. Finally, if no dominance point exists, then we can proceed to the next phase.

Figure 5.3 gives an example of a kNN query that finds the top-4 points for point Q with α value equal to 0.2 over the last year. ST-Hadoop starts from the Q partition on day 1. First, we find the top-4 neighbor from day 1, and then we insert the score of the furthest 4^{th} neighbor from day 1 to a priority queue, e.g., p_{16}. Iterate over the other overlap partitions from next temporal interval, e.g., days 2. In each iteration ST-Hadoop checks if the ranking score of the minimum distance point of the new partition can beat the ranking score of the p_{16}. If it beats, then this new partition is

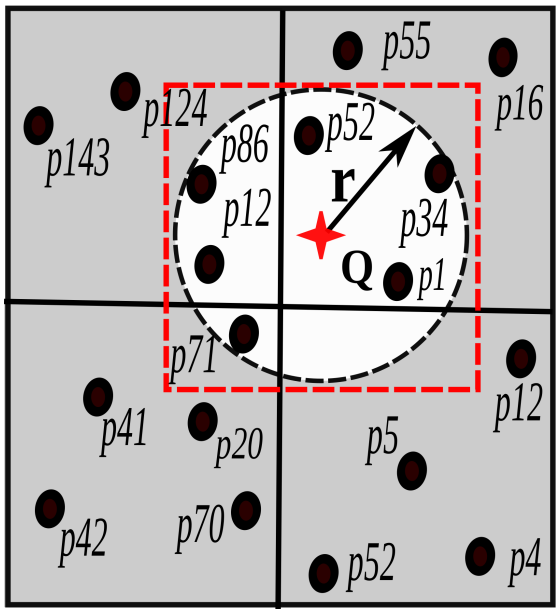


Figure 5.4: Refinement Final Answer

considered, and we modify the cylinder height and radius respectively. As depicted in Figure 5.3, p_6 in day 2 dominates p_{16}. We repeat this process until no dominance point can be found in the next temporal interval. As shown in the example after the third day, we do not need to modify the height of the cylinder, since there is no dominance point exist any further. Henceforward, other partitions will be ignored and no further computation required, and we can proceed to the next phase.

In the ***k*NN refinement phase**, We check if there are points in the overlap partitions might contribute to the final answer. If α is equal to zero or one, then we run a spatial range query to get all points inside the MBR of the test circle, as the cylinder height is equal to zero. Meanwhile, if $0 \leq \alpha \leq 1$, then we run we run a spatio-temporal range query to get all points inside the MBR of the test cylinder. The cylinder radius and height are obtained from the previous phase. Finally, we scan over the range query result and process it with the traditional kNN algorithm to find the final answer.

Figure 5.4 and 5.3 gives two examples of refinement phase. The shaded partitions are the ones that are considered in the range query. In Figure 5.4, the circle of the

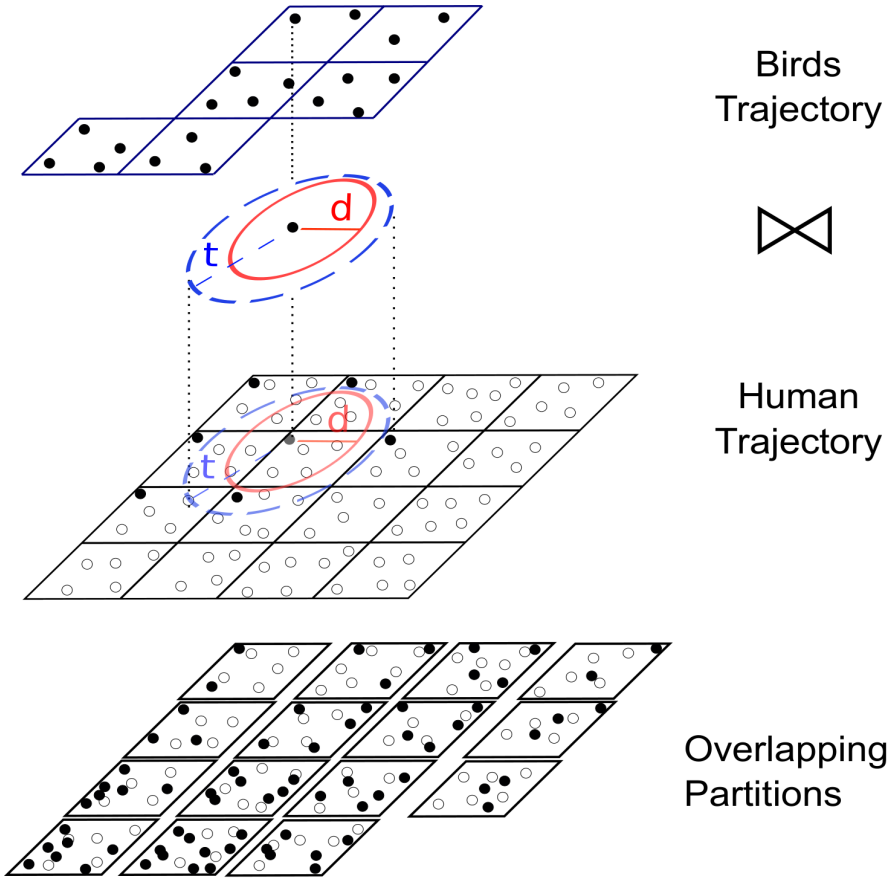


Figure 5.5: Spatio-temporal Join

cylinder intersects with the three partitions from the same temporal interval. In that case, the height of the cylinder is equal to zero. In Figure 5.3 we illustrate the test cylinder with a height equal to 3 temporal intervals. Similarly, the shaded partitions are the only one to be considered in the range query. For each time interval, a circle has a different radius, which collectively forms the test cylinder. Once we get the results, we apply the traditional kNN algorithm to find the final top-4 answers, i.e., $\{p5, p3, p6, p1\}$ in the refinement phase.

5.3 Spatio-temporal Join

Given two indexed dataset R and S of spatio-temporal records, and a spatio-temporal predicate θ. The join operation retrieves all pairs of records $\langle r, s \rangle$ that are similar to each other based on θ. For example, one might need to understand the relationship between the birds death and the existence of humans around them, which can be described as *"find every pairs from bird and human trajectories that are close to each other within a distance of 1 mile during the last week"*. The join algorithm runs in two steps as shown in Figure 5.5, *hash* and *join*.

In the **hashing** step, the map function scans the two input files and hashes each record to candidate buckets. The buckets are defined by partitioning the spatio-temporal space using the two-layer indexing of temporal and spatial, respectively. The granularity of the partitioning controls the tradeoff between partitioning overhead and load balance, where a more granular-partitioning increases the replication overhead, but improves the load balance due to the huge number of partitions, while a less granular-partitioning minimizes the replication overhead, but can result in a huge imbalance especially with highly skewed data. The hash function assigns each point in the left dataset, $r \in R$, to all buckets within an Euclidean distance d and temporal distance t, and assigns each point in the right dataset, $s \in S$, to the one bucket which encloses the point s. This ensures that a pair of matching records $\langle r, s \rangle$ are assigned to at least one common bucket. Replication of only one dataset (R) along with the use of single assignment, ensure that the answer contains no replicas.

In the **joining** step, each bucket is assigned to one reducer that performs a traditional in-memory spatio-temporal join of the two assigned sets of records from R and S. We use the plane-sweep algorithm which can be generalized to multidimensional space. The set S is not replicated, as each pair is generated by exactly one reducer, and thus no *duplicate avoidance* step is necessary.

5.4 Experiments

In our experiments, we compare the performance of a ST-Hadoop spatio-temporal range, kNN, and join query proposed in this chapter 5 to their spatial-temporal implementations on-top of SpatialHadoop and Hadoop. For range query, we use system throughput as the performance metric, which indicates the number of MapReduce jobs finished per minute. To calculate the throughput, a batch of 20 queries is submitted to the system, and the throughput is calculated by dividing 20 by the total time of all queries. The 20 queries are randomly selected with a spatial area ratio of 0.001% and a temporal window of 24 hours unless stated. This experimental design ensures that all machines get busy and the cluster stays fully utilized. For spatio-temporal join, we use the processing time of one query as the performance metric as one query is usually enough to keep all machines busy. The experimental results for range, kNN, and join queries are reported in Sections 5.4.1, 5.4.2 , and 5.4.3, respectively.

5.4.1 Spatiotemporal Range Query

In Figure 5.6, we increase the size of input from 1TB to 10TB, while measuring the job throughput. ST-Hadoop achieves more than two orders of magnitude higher throughput, due to the temporal load balancing of its spatio-temporal index. As for SpatialHadoop, it needs to scan more partitions, which explain why the throughput of SpatialHadoop decreases with the increase of data records in spatial space. Meanwhile, ST-Hadoop throughput remains stable as it processes only partition(s) that intersect with both space and time. Note that it is always the case that Hadoop needs to scan all HDFS blocks, which gives the worst throughput compared to SpatialHadoop and ST-Hadoop.

Figure 5.7 shows the effect of configuring the HDFS block size on the job throughput. ST-Hadoop manages to keep its performance within orders of magnitude higher throughput even with different block sizes. This is mainly because ST-Hadoop partitioning techniques utilize spatiotemporal data locality across HDFS blocks, in which this will result in much performance. However, increasing the HDFS block size will result in accommodates more data within the block; and thus, will incur overhead on the range query performance.

Extensive experiments are shown in Figure 5.8, analyzed how slicing ratio (α) can

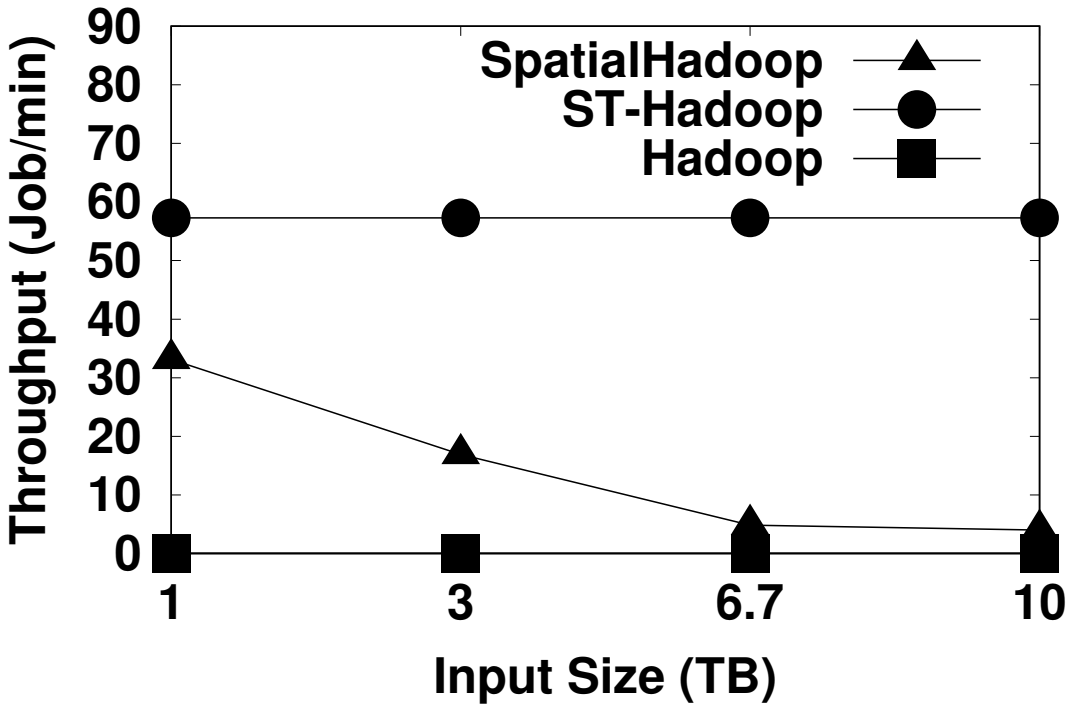


Figure 5.6: Range Query VS Input files (TB)

affect the performance of range queries. ST-Hadoop keeps its higher throughput around the default HDFS block size, as it maintains the load balance of data records in its two-layer indexing. As expected expanding the block size from its default value will reduce the performance on SpatialHadoop and ST-Hadoop, mainly because blocks will carry more data records.

5.4.2 K-Nearest-Neighbor Queries (kNN)

We extensively measure the performance of kNN query processing on Hadoop [78] and ST-Hadoop for 10 TB of twitter dataset. In experiments, 20 query locations are set at random points (i.e., random points in both date and time) sampled from the input file, α is set to 0.4, the number of k is set to 100. Unless otherwise mentioned.

Figure 5.9 measures system throughput when increasing the input size from 1 TB to 10 TB. ST-Hadoop has one to two orders of magnitude higher throughput. Hadoop and

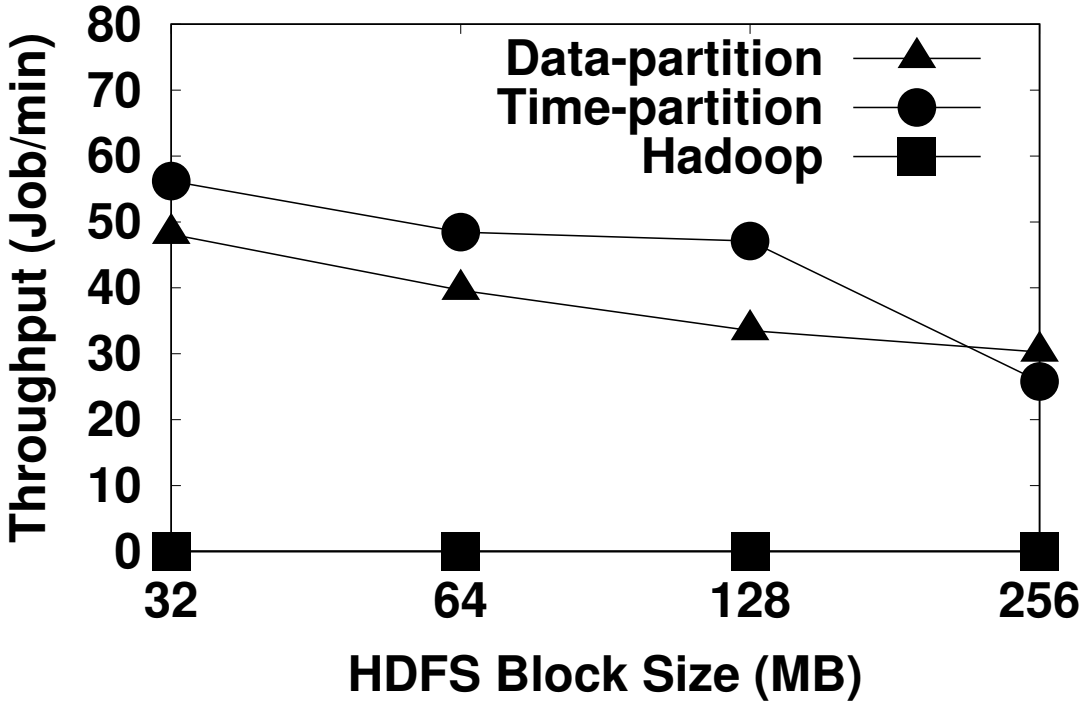


Figure 5.7: Range Query VS Block size (MB)

SpatialHadoop performances decrease dramatically as they need to process the whole file while ST-Hadoop maintains its performance as it processes one partition regardless of the file size. Since SpatialHadoop is not aware of the temporal locality of the data, it needs to process multiple partitions to finds the k nearest neighbor in a specific day, and in a worst case it might end up processing all partitions. Hence, ST-Hadoop keeps its speedup at two orders of magnitude.

Figure 5.10 gives the effect of increasing k from 1 to 40K on 10 TB dataset. ST-Hadoop gives an order of magnitude performance with both single level index and optimized query plan that uses ST-Hadoop hierarchy index. ST-Hadoop achieves two orders of magnitude performance compared to Hadoop kNN implementation. ST-Hadoop efficiently handles spatio-temporal kNN operation. However, we notice that the job throughput decreases when k is more than eight thousand, where more partitions are required to be processed. ST-Hadoop is consistently better than Hadoop. While the

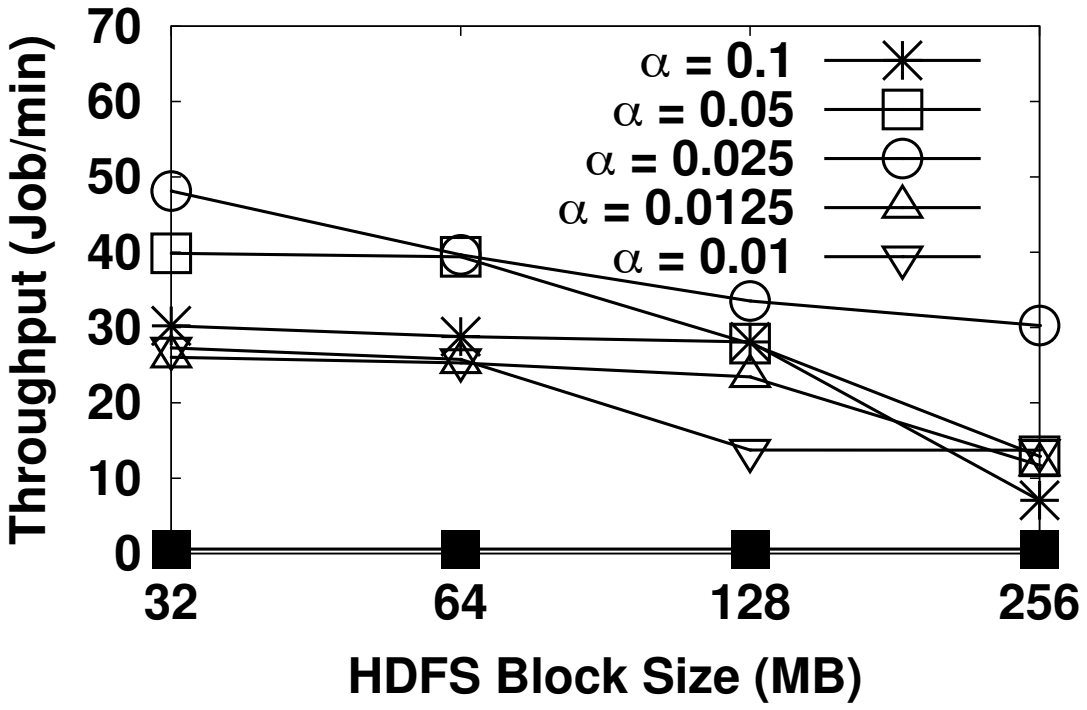


Figure 5.8: Range query with Block size VS Slicing ratio (α)

performance with single level index tends to decrease with the increased number of the nearest neighbor k. In the meantime, the optimized query that uses the hierarchy index remains stable for a higher number of neighbors. However, at some point, it will decrease. This is expected as the number of selected partitions increase with the increased of the k number.

In Figure 5.11, shows how the job throughput affected by the value of α in the ranking function. The query point Q is fixed at random location on the first day of a month. The increase of α means that ST-Hadoop might need to process several partitions from different days and also nearby partitions within the same days to find the nearest neighbor. As the α value increases, the performance of ST-Hadoop stays at two orders of magnitude higher than Hadoop. Without having ST-Hadoop hierarchy index, the performance slightly decrease. This is expected as query cares more about spatial proximity, and thus, 30 partitions will need to be processed. The reason for the

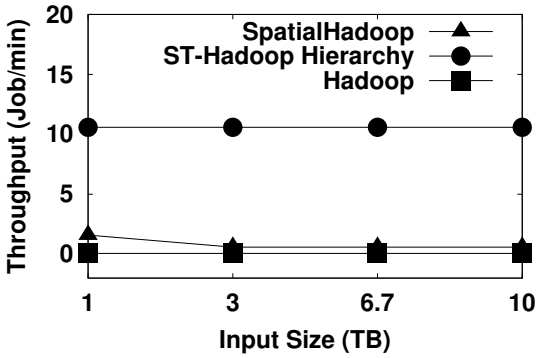


Figure 5.9: The execution of kNN query on different input files

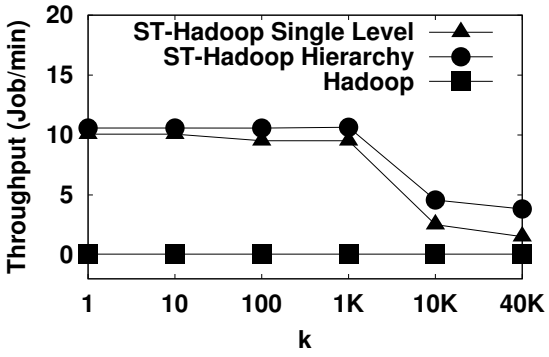


Figure 5.10: kNN query with various k

steady throughput by ST-Hadoop goes to the execution plan supplied by ST-Hadoop query optimizer. The query optimizer selects a single partition that overlap with the query point Q, which best fit to cover the whole temporal range. When $\alpha = 0$ the query optimizer overlaps Q with a single partition from the highest granularity (e.g, daily level). If $\alpha = 1$, then the query optimizer selects a single partition from a lower granularity level(e.g., month). Meanwhile, if α is in between that two extremes, then the query optimizer selects a single partition that either extends over the whole temporal range or partially cover it based on the ranking score of the furthest k.

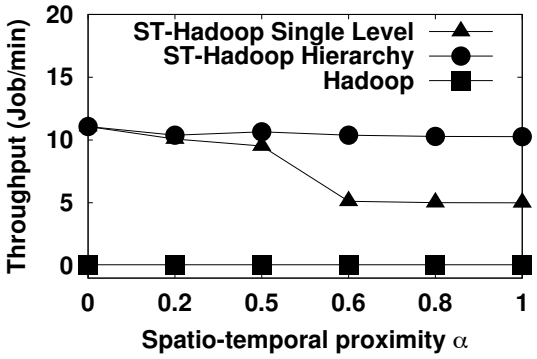


Figure 5.11: kNN throughput while varying (α) of Ranking Function

5.4.3 Spatiotemporal Join

Figure 5.12 gives the results of the spatio-temporal join experiments, where we compare our join algorithm for ST-Hadoop with MapReduce implementation of the spatial hash join algorithm [81]. Typically, in this join algorithm we perform the following query, "*find every pairs that are close within an Euclidean distance of* $1mile$ *and a temporal distance of* $2days$", this join query is executed on both ST-Hadoop and Hadoop and the response times are compared. The y-axis in the figure represents the total processing time, while the x-axis represents the join query on numbers of days×days in ascending order. With the increase of joining number of days, the performance of ST-Hadoops join increases, because it needs to join more indexes from the temporal hierarchy. In general, ST-Hadoop gives the best results as ST-Hadoop index replicates data in several layers, and thus ST-Hadoop significantly decreases the processing of non-overlapping partitions, as only partitions that overlap with both space and time are considered in the join algorithm. Meanwhile, the same joining algorithm without using ST-Hadoop index gives the worst performance for joining spatio-temporal data, mainly because the algorithm takes into its consideration all data records from one dataset. However, ST-Hadoop only joins the indexes that are within the temporal range, which significantly outperforms the join algorithm with double to triple performance.

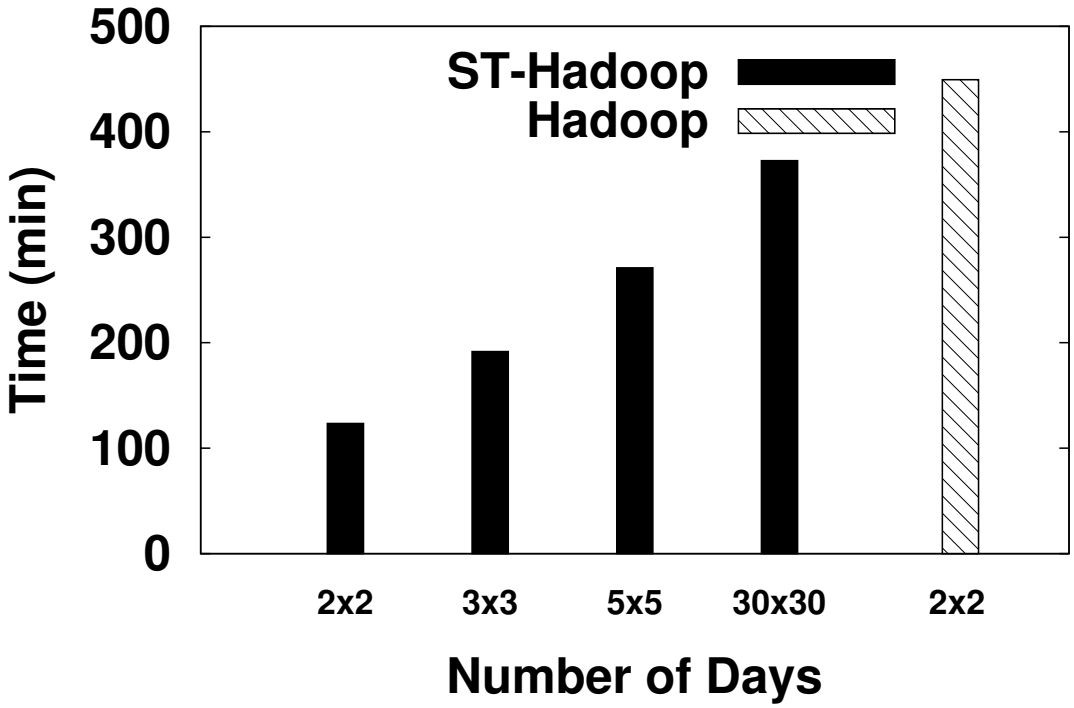


Figure 5.12: Spatio-temporal Join

Chapter 6

Spatio-temporal Query Optimizer in MapReduce

Figure 6.1 illustrates the conceptual visualization of ST-Hadoop index, where lines signify how the temporal index divided into a set of disjoint time intervals, e.g., months, weeks, or days. Triangles symbolize the spatial indexing, e.g., R-tree. ST-Hadoop stores its index information as a meta-data on the master node. The meta-data of the ST-Hadoop index provides a rich statistics about the spatial and temporal locality of partitions in the HDFS. Each record of the meta-data represents a partition, which contains information about the minimum boundary rectangle, temporal interval, temporal granularity (i.e., level), the number of data records within the HDFS block, and a unique identifier that acts as an entry pointer to access the partition block. Data records are replicated and spatiotemporally partitioned in each level. In ST-Hadoop we developed two optimization models of heuristic and cost-based to minimize query response time, respectively.

In a nutshell, the heuristic model deploys an algorithm to computes the *coverage ratio* r, that defined as the ratio of the time interval of a partition that overlaps with spatio-temporal query predicates. A partition was selected only if its *coverage ratio* is above a specific threshold $\mathcal{M}$. The algorithm runs in a top-down approach that starts with the top level and selects partitions that cover the temporal query interval T, If the query interval T is not covered at that granularity, then the algorithm continues to

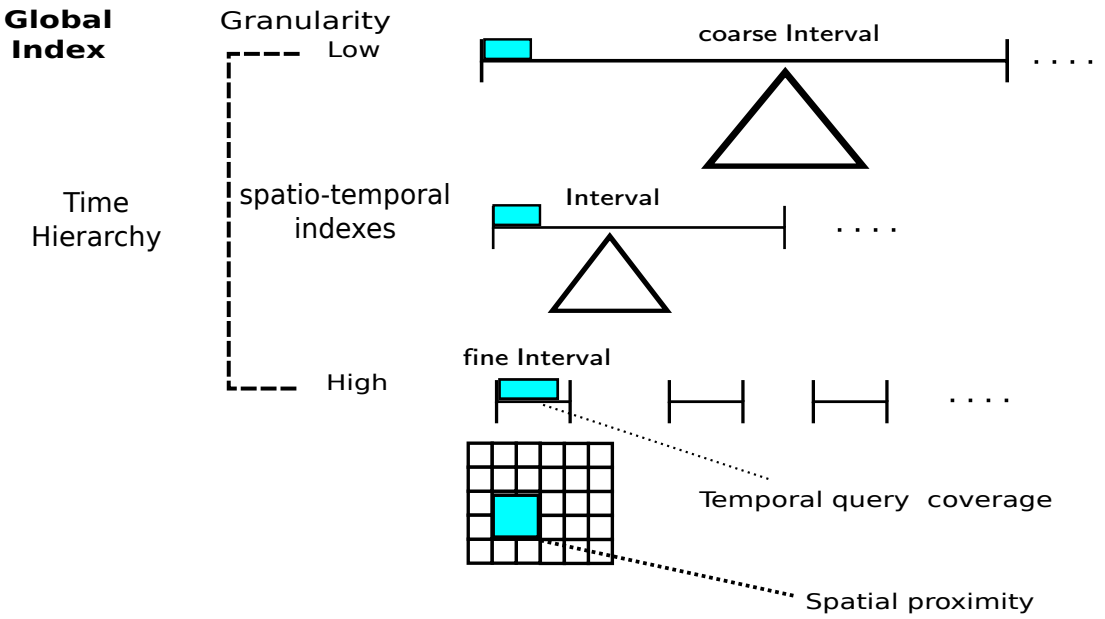


Figure 6.1: Conceptual representation of ST-Hadoop meta-data index

the next level. If the bottom level is reached, then all partitions overlap with T will be selected. Meanwhile, a cost-based model deploys a greedy algorithm that finds the minimum number of partitions need to be processed for any spatio-temporal operations. Similarly the algorithm starts from a lower granularity to the finest granularity on the bottom of ST-Hadoop meta-data index. We discuss both model in more details in the following sections 6.1 and 6.2. Followed by extensive experiments in section 6.3 comparing and verifying the two optimization models on queries response time.

6.1 Heuristic Query Optimization

The main goal of this optimization model is generate a constructive query plan based on a collection of heuristic statistics gather from ST-Hadoop meta-data. W examine the performance of the temporal hierarchy index in ST-Hadoop using both slicing techniques. We evaluate different granularities of time-partition slicing (e.g., daily, weekly, and monthly) with various data-partition slicing ratio. In the meantime, we fix the spatial range to a smallest area unit and increase the temporal range from 1 day to 31 days, while measuring the response time .

ST-Hadoop utilizes its temporal hierarchy index to achieve the best performance as it mixes and matches the partitions from different levels to minimize the running time, as described ST-Hadoop provides good performance for both small and large query intervals as it selects partitions from any level. When the query interval is very narrow, it uses only the lowest level (e.g., daily level), but as the query interval expand it starts to process the above level. The value of the parameter $\mathcal{M}$ controls when it starts to process the next level. At $\mathcal{M} = 0$, it always selects the up level, e.g., monthly. If $\mathcal{M}$ increases, it starts to match with lower levels in the hierarchy index to achieve better performance. At the extreme value of $\mathcal{M} = 1$, the algorithm only matches partitions that are completely contained in the query interval, e.g., at 18 days it matches two weeks and four days while at 30 days it matches the whole month. The optimal value in this experiment is $\mathcal{M} = 0.4$ which means it only selects partitions from a specific granularity (i.e., level) if $\mathcal{M}$ is at least 40% covered by the query temporal interval, as shown in the following equation.

$$\mathcal{M}(Q) = \quad \alpha \times \frac{Temporal\ Coverage(Q)}{Interval\ Coverage(level)} + (1-\alpha)\frac{Spatial\ Coverage(Q)}{Spaital\ Proximity(level)}$$

The α is a parameter that gears the execution of the query towards spatially, temporally, or spatio-temporally execution plan. This α parameter can be tuned from the system configuration files, or it can be inserted when executing the query as discussed in kNN operation 5.2. However, in ST-Hadoop the default value of the α is set to one, which means in this heuristic model favors more the temporal coverage over spatial approximation of any given queries.

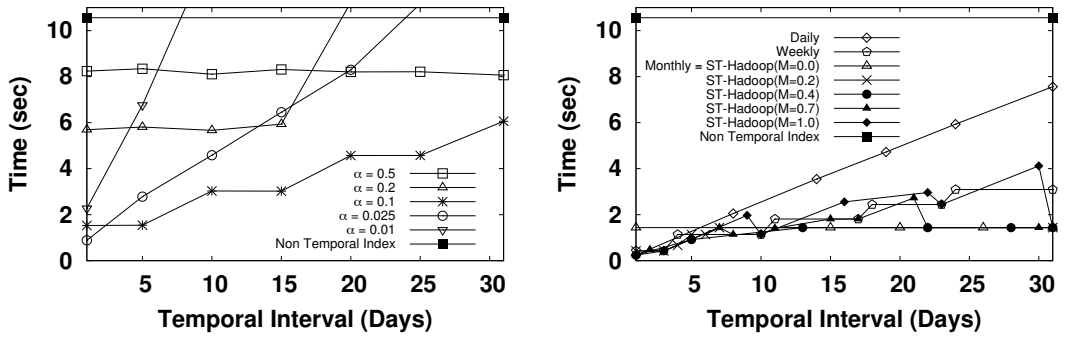
In this query optimization model, we study the effect of the spatio-temporal query range (σ) on the choice of $\mathcal{M}$. To measure the quality of $\mathcal{M}$, we define an optimal running time for a query Q as the minimum of all running times for all values of $\mathcal{M} \in [0, 1]$. Then, we determine the quality of a specific value of $\mathcal{M}$ on a query workload as the mean squared error (MSE) between the running time at this value of $\mathcal{M}$ and the optimal running time. This means, if a value of $\mathcal{M}$ always provides the optimal value, it will yield a quality measure of zero. As this value increases, it indicates a poor quality

as the running times deviates from the optimal.

6.2 Cost-based Optimization

Partitions in each level of ST-Hadoop index cover the whole time and space, which means a query can be answered from any level individually or we can mix and match partitions from different levels to cover the query predicates. To decide which partitions should be selected and from which levels, highly depends on the selectivity of the query predicates. Depending on which granularity is used to get the result, there is a trade-off between the number of partitions containing the query results and the amount of processing needed to process each partition. The HDFS block size is tuned in ST-Hadoop configuration files, which means partitions block size are the same across all computation nodes. Therefore, for any given query the bottleneck that hits the query performance is the number of partitions that contain the query answer.

The primary goal of this query optimizer model is to minimize the number of partitions that contain the final answer for each of its operation. We implemented in memory greedy algorithm that recursively iterates over ST-Hadoop meta-data to find the optimal execution plan. The algorithm runs in a top-down approach that starts from the lowermost granularity (e.g., monthly level) to the highest one (e.g., daily level). In each iteration, we examine the precise number of partitions N that contains the final answer for the given spatio-temporal query predicates. The algorithm reports the global optimal execution plan if the next granularity has a higher number of partitions, or it reaches the highest level. For example, consider a query that asks about 22 days of data records in a particular area. First, ST-Hadoop calculates the number overlap partitions from the monthly level, and then compares it with the next level from its temporal hierarchy index (e.g., week). ST-Hadoop query optimizer recursively computes and compares the number of partitions until the next explored level has more partitions from the current one. If the highest granularity is reached and has a fewer number of partitions, then all partitions overlap with query predicates will be selected.



(a) Data-partition Slicing

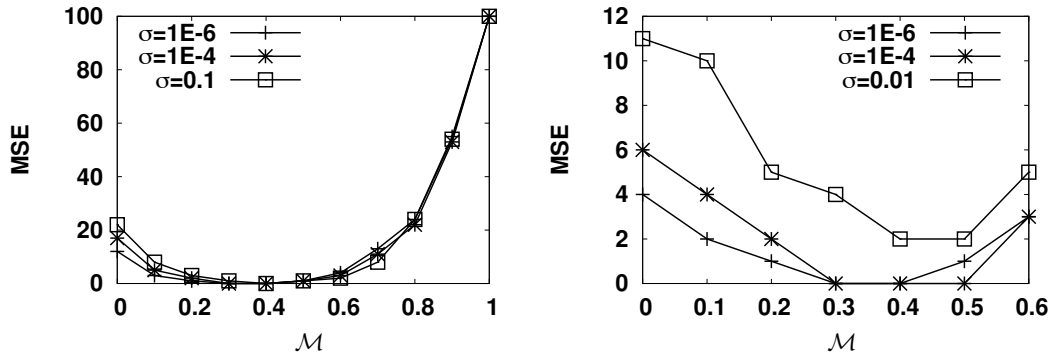
(b) Time-partition Slicing

Figure 6.2: Spatio-temporal Range Query Interval Window

6.3 Experiments

Experiments in Figure 6.2 examines the performance of the temporal hierarchy index in ST-Hadoop using both slicing techniques. We evaluate different granularities of time-partition slicing (e.g., daily, weekly, and monthly) with various data-partition slicing ratio. In these two figures, we fix the spatial query range and increase the temporal range from 1 day to 31 days, while measuring the total running time. As shown in the Figures 6.2(a) and 6.2(b), ST-Hadoop utilizes its temporal hierarchy index to achieve the best performance as it mixes and matches the partitions from different levels to minimize the running time, as described in Section 6. ST-Hadoop provides good performance for both small and large query intervals as it selects partitions from any level. When the query interval is very narrow, it uses only the lowest level (e.g., daily level), but as the query interval expand it starts to process the above level. The value of the parameter $\mathcal{M}$ controls when it starts to process the next level. At $\mathcal{M} = 0$, it always selects the up level, e.g., monthly. If $\mathcal{M}$ increases, it starts to match with lower levels in the hierarchy index to achieve better performance. At the extreme value of $\mathcal{M} = 1$, the algorithm only matches partitions that are completely contained in the query interval, e.g., at 18 days it matches two weeks and four days while at 30 days it matches the whole month. The best choice of $\mathcal{M}$ value in this experiment is $\mathcal{M} = 0.4$ which means it only selects partitions that are at least 40% covered by the temporal query interval.

In Figure 6.3 we study the effect of the spatio-temporal query range (σ) on the choice

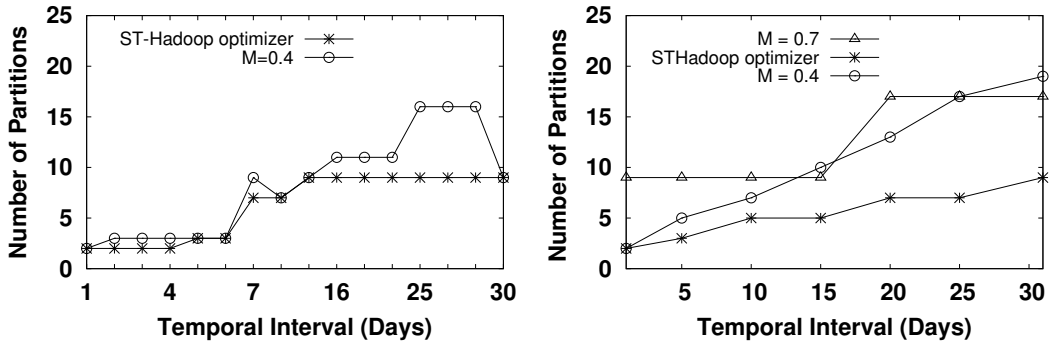


(a) Tuning of $\mathcal{M}$ for query intervals from 1 to 30 days (b) Tuning of $\mathcal{M}$ for query intervals from 1 to 400 days

Figure 6.3: The effect of the spatio-temporal query ranges on the best value of $\mathcal{M}$

of $\mathcal{M}$. To measure the quality of $\mathcal{M}$, we define the best running time for a query Q as the minimum of all running times for all values of $\mathcal{M} \in [0, 1]$. Then, we determine the quality of a specific value of $\mathcal{M}$ on a query workload as the mean squared error (MSE) between the running time at this value of $\mathcal{M}$ and the best running time. This means, if a value of $\mathcal{M}$ always provides the best value, it will yield a quality measure of zero. As this value increases, it indicates a poor quality as the running times deviates from the best running time. In Figure 6.3(a), We repeat the experiment with three values of spatial query ranges $\sigma \in \{1E-6, 1E-4, 0.1\}$. As shown in the figure, $\mathcal{M} = 0.4$ provides the best performance for all the experimented spatial ranges. This is expected as $\mathcal{M}$ is only used to select temporal partitions while the spatial range (σ) is used to perform the spatial query inside each of the selected partitions. Figure 6.3(b), shows the quality measures with a workload of 71 queries with time intervals that range from 1 day to 421 days. This experiment also provides a very similar result where the best choice value of $\mathcal{M}$ is around 0.4.

In Figure 6.4 we evaluate ST-Hadoop greedy algorithm implemented in the new query optimizer with the heuristic approach of $\mathcal{M}$, on both slicing techniques supported in ST-Hadoop. Certainly, the best choice of $\mathcal{M}$ is at least as far ahead as the optimal, but it is not optimal. In the heuristic approach, a partition is selected only if its *coverage ratio* is above a specific threshold $\mathcal{M}$, which is around 0.4. Meanwhile, in our new ST-Hadoop implementation we compute the exact number of partitions that need



(a) Selected HDFS blocks in Time-partition Slicing (b) Selected HDFS blocks in Data-partition Slicing

Figure 6.4: ST-Hadoop Greedy query optimizer VS heuristic $\mathcal{M}$

to be processed by employing a greedy algorithm that finds the minimum. ST-Hadoop employs a top-down approach that starts with the top level and selects partitions that cover query interval T, If the query interval T is not covered at that granularity, then the algorithm continues to the next level. ST-Hadoop finds the local optimal from each granularity until we reach the global optimal, i.e., the minimum number of partitions that covers query interval.

Figure 6.4(a), compares the number of selected partitions between $\mathcal{M}$ and the greedy algorithm. The input files indexed and sliced through Time-partition technique, i.e., daily, weekly, monthly levels. We fix the spatial range query and increase the temporal range from 1 day to 31 days. In these experiments, we eliminate other $\mathcal{M}$ value, as experimentally we found that the best choice of $\mathcal{M}$ value is equal to (0.4). As shown in the figure, the greedy algorithm always beats $\mathcal{M}$. As expected the algorithm selects the global optimal, which is the minimal number of partitions that contain the final answer. In Figure 6.4(b), we repeated the same experiment with various data-partition slicing ratio. ST-Hadoop greedy algorithm provides the best performance for both small and large query intervals as it selects the minimum number of partitions from any level. When the query interval is very narrow, it uses only the lowest granularity level. As as the query interval expand query optimizer starts to process the above level.

Chapter 7

Summit Trajectory library in ST-Hadoop

Driven by the ubiquity of location-based services, that produce a massive amount of trajectories. Querying and analyzing trajectory data become a must for a wide range of applications. This chapter presents a scalable data management system for large scale data. The proposed system is well-suited to efficiently support several basic queries, such as range, kNN, and similarity queries. These queries and the architectural design of the proposed library are extendable, in a way that it enables users to build various applications on trajectories.

7.1 introduction

Recent advances in mobile computing, sensor, GPS, and satellite technology have made it possible to produce a massive amount of trajectory data. This increasingly interests scientist and domain experts in performing analysis tasks over such huge data [59]. For example, NASA publicly archives over 4TB of stars and asteroids movement activity on a daily basis [4]. Sloan Digital Sky Survey project collects over 156TB of motion data from millions of outer-space objects [82]. MoveBank project gathers more than 20 years of animal movements [83]. New York Taxi & Limousine archives over a Billion of taxi trajectories [1]. National Hurricane Center stores comprehensive details of all storms' trajectories every year [84]. Besides the enormous amount of data, users should be able to explore and analyze trajectory data efficiently.

Domain experts who analyze trajectory data are either (a) use *Heterogeneous* multiple platforms [85, 86], in which trajectory operations built on-top of generic platforms, such as Hadoop, or Spark. Using these platforms as-is will result in sub-performance for trajectory applications that require indexing, e.g., Marmaray project from Uber uses Hadoop as a backbone platform for storing data as non-indexed heap files, or (b) use *Big Spatio-temporal Frameworks* [23, 42], in which they are efficient for processing spatio-temporal data on MapReduce platform. Yet, with a limited support for trajectory operations, as their index unable to accommodate storing the entire topology of trajectory objects, in which affects the performance of basic trajectory operations, such as finding similarity between trajectories. Thus, processing trajectory on MapReduce raised many challenges. Some of the most significant challenges are the inability of these systems to preserve the spatio-temporal topology of trajectories, load balancing efficiency, and the capability of supporting various trajectory operations.

This paper presents Summit; the first full open-source trajectory library on the MapReduce framework, shipped with the source code of ST-Hadoop [23]. Summit injects the trajectory data awareness inside each of ST-Hadoop layers, mainly, indexing, operation, and language layers. However, running a program that deals with trajectory data using Summit will have order(s) of magnitude better performance than ST-Hadoop. ST-Hadoop treats the spatio-temporal information of trajectory as a stationary data, as it loads only a basic geometrical feature (e.g., Point, Line, Rectangle) at a time.

```
Objects      =  LOAD 'point' AS (id:int, STPoint);
Intermediate =  FILTER Objects BY
                Overlaps ( Rectangle(x1, y1, x2, y2)
                ,Interval (t1, t2);
                GROUP Object BY (id)
                FOREACH $Object(id) Search Trajectory(id)
Result       =  SIMILAR Object threshold:T From Intermediate;
```

(a) Similarity query in ST-Hadoop

```
Objects  =  LOAD 'trajectory' AS (id:int, STTrajectory:
                                  <STPoint1,STPointm >);
Result   =  FILTER Objects BY
            Overlaps (Location,time,
             Rectangle(x1, y1, x2, y2),Interval (t1, t2))
            SIMILAR Object threshold:T;
```

(b) Similarity query in Summit

Figure 7.1: Similarity query in ST-Hadoop vs. Summit

In the meantime, trajectories are consist of a correlated sequence of features that are connected over time. This means that performing a basic trajectory operation such as similarity queries might end up scanning the whole dataset to check for trajectory connectivity before computing the similarity. Imagine a query that asks the similarity between trajectories in the last two years or so.

Figures 7.1(a) and 7.1(b) show code snippets that load and query similar trajectories from ST-Hadoop and Summit, respectively. The query finds similar trajectories within a specific rectangular area represented by two corner points and within a time interval. Running this query on ST-Hadoop will result in sub performance as opposed to Summit. As shown in the code 7.1(a), ST-Hadoop loads trajectories as spatio-temporal points. Next, it finds all overlap records from its index. The retrieved records need to be group by their trajectory id and then order by their time before start computing the similarity. This will incur significant I/O overhead, especially for a large spatio-temporal range that expands years. Meanwhile, Summit loads the entire sequence of trajectory and exploits its index to retrieve and compute the similarity between data records, and hence, achieves orders of magnitude better performance over ST-Hadoop.

The key idea behind the performance gain of Summit is that it tunes the MapReduce

paradigm to efficiently archiving, indexing, and querying the massive amount of trajectory data. In particular, Summit is powered by ST-Hadoop; an extension of MapReduce framework that deals efficiently with spatio-temporal data. Yet, ST-Hadoop can only manage stationary spatio-temporal geometrical shapes. In the meantime, analytical tasks on trajectories consider a trajectory as a sequence of basic shapes in motion. Thus, Summit design indexing techniques to support organizing trajectory data in the Hadoop Distributed File System (HDFS) in a way that preserves their geometrical shapes. Summit is open-source, and the code is available as a part of ST-Hadoop at `http://st-hadoop.cs.umn.edu`. We envision that the open source nature will act as a research vehicle for other researchers and application developers to build more complex operations to Summit.

7.2 background and related work

Distributed Generic Systems, Generic platforms have been used extensively in different analytic applications that include terabyte sorting [87], machine learning [88], The current effort for processing trajectory data are either: (A) Use *Heterogeneous* multiple platforms [52, 85, 86], in which trajectory operations built on-top of generic platforms, such as Hadoop [6], Spark [7], Cassandra [70], Kafka [89], or Storm [66], e.g., Marmaray project from Uber uses Hadoop as a backbone for storing data as non-indexed heap files, while carrying the execution of trajectory operations on another platform. (B) Implement specific operation on-top of a single *Generic* framework [15, 16, 46, 47, 55, 56, 57, 58, 59, 76, 90]. For example, a most recent research study investigated the kNN join query on Hadoop MapReduce employed five isolated map-reduce jobs to execute a single kNN join operation without indexing trajectory [47]. However, using generic distributed systems as-is will result in sub-performance for trajectory applications that require indexing, mainly because they store data as non-indexed heap files.

Distributed Spatial Systems, Extension of MapReduce platform has been developed and dedicated for spatial analytic operations in the last few years, this including SpatialHadoop [14], ScalaGiST [32], Hadoop-GIS [11], and ESRI-GIS tool on Hadoop [31].

There are also few systems extended Resilient Distributed Dataset(RDD) on Spark to support spatial operations, such as GeoSpark [13] and Simba [91]. Although, these big distributed spatial systems are efficient in spatial operations, yet they are not well designed to efficiently process spatio-temporal data such as trajectories, mainly because the infrastructure of their indexes and operations only support spatial queries.

Distributed Spatio-temporal Systems, There are some Big spatio-temporal systems like ST-Hadoop [23] GeoWave [42], and GeoMesa [41] that focus on supporting spatio-temporal applications. ST-Hadoop extends Hadoop and maintains a Hierarchical indexing structure that consists of two-layer indexing of a temporal and spatial. GeoMesa and GeoWave both are built upon Accumulo platform [67] and implemented a space-filling curve to combine the three dimensions of space-geometry and time. This class of systems did not attempt to enhance the contiguity and locality of trajectory data. Although, distributed spatio-temporal systems are efficient for processing basic spatio-temporal data (e.g., POINT, LINE, POLYGON), yet, they are limited in supporting trajectory (i.e., a connected sequence of basic geometry). This is mainly due to the inability of their index structure to accommodate storing the entire topology of trajectory objects. This, in turn, will affect the performance of basic trajectory operations, e.g., finding similarity between trajectories.

Distributed Time Series Systems Frameworks in this category are optimized for time series or time-stamped data. There are a large number of distributed time series systems, such as Informix [92], TSAR [93], OpenTSDB [94], and LittleTable [95]. Systems in this family of distributed frameworks significantly have different approaches for handling data, such that they mainly focus on building indexes and operations to efficiently performs analytical tasks on time series data. There is no sufficient support for spatial or spatio-temporal indexes or operations, and hence they do not support analysis tasks on trajectory data.

Distributed Graph Processing Systems, MapReduce has been extensively investigated for graph processing in both academia and industry [96, 97, 98, 99, 100]. The focus of these systems is to support basic graph models, where a graph consists

of a set of vertices and edges. Systems belong to this category are efficient for graph operations, such as complex traversal queries. However, they do not support spatio-temporal or trajectory operations mainly because their indexing structures are not well-suited for realizing both the spatial and the temporal property of trajectory.

Trajectory Operations, Existing research studies on trajectory implement operations on-top of distributed platforms, such as range [15, 16, 37, 40, 52, 54, 56, 57, 59], k-nearest neighbor [55, 59], Skyline [58], similarity search [76, 90], and join [46, 47]. Most recent efforts focus on supporting similarity search on-top of Spark [76, 90] and kNN join on Hadoop [47], yet they do not have any indexes in Hadoop Distributed File Systems (HDFS). Notably, all these research studies are limited to support specific operation. In the meantime, the Summit system is extendable in a way that it enables users to build various applications on trajectories and extends its operations library.

Similarity Measurements, Measuring the similarity between a pair of trajectories is essential for identifying portions that are common between two trajectories. The similarity measurement must satisfy three main criteria: (1) The flexibility to identify similar trajectories on various times, (2) Ignores outliers points in similarity computation, and (3) The ability to identify the similarity between portions of trajectories based on some distance measurement. Formally, we can say that a similarity function takes pairs of trajectories and it generates a score, that shows the closeness between their sequences. There are over a dozen similarity measurements in literature like Dynamic Time Warping (DTW) [90, 101, 102], Edit Distance on Real sequence (EDR) [103], Edit distance with Real Penalty (ERP) [104, 105], Longest Common Subsequence distance (LCSS) [106, 107], Fréchet similarity [108]. Interested readers can refer to previous study [109], which discussed trajectory similarity in great detail. In this paper, we are going to consider the Dynamic Time Warping (DTW) measurement. Originally DTW developed for matching speech signals in speech recognition [110], ever since it is considered as the one of the most rubst and wildly adopted similarity function for trajectories and time series data [51, 76, 107, 111, 112, 113, 114, 115, 116, 117, 118].

This chapter describes Summit; a full-fledged MapReduce framework with native

support for big trajectory data. Summit is a comprehensive extension library that injects trajectory data awareness inside ST-Hadoop layers.

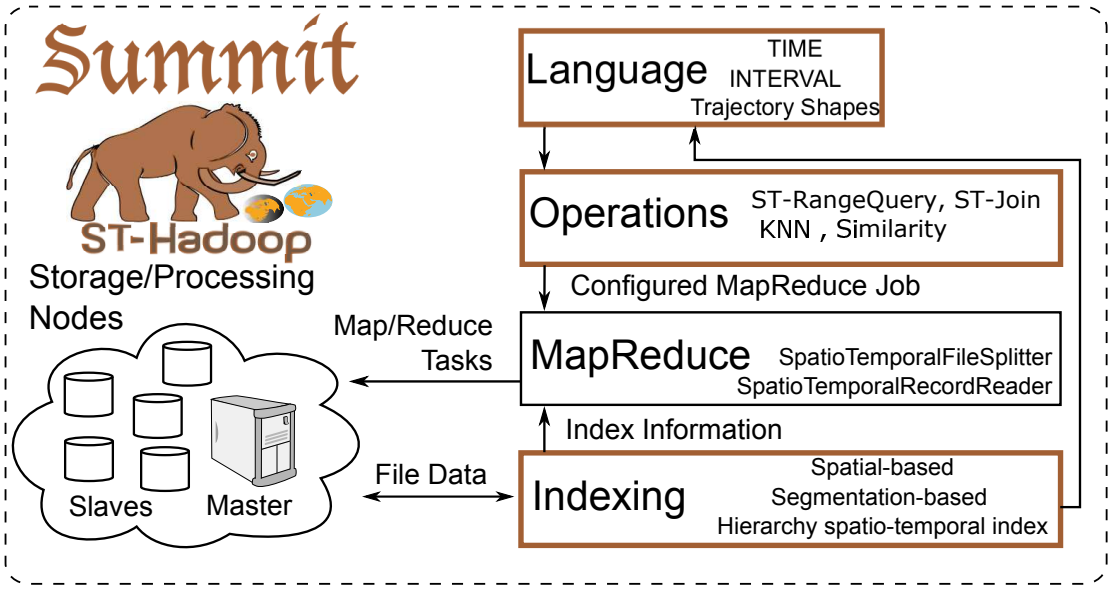


Figure 7.2: Summit Architecture

7.3 System Overview

Figure 7.2 gives an overview of Summit system architecture. Summit is a full-fledged open-source library on ST-Hadoop MapReduce framework [23] with *built-in* native support for trajectory data. Summit cluster contains one master node that breaks a map-reduce job into smaller tasks, carried out by slave nodes. Summit modifies three core layers of ST-Hadoop, namely, *Language*, *Indexing*, and *Operations*. The language layer adds new SQL-Like interface for trajectory operations and data types. The modifications and the implementation of the indexing and operation layers will be explained in the following sections.

7.4 Trajectory Indexing

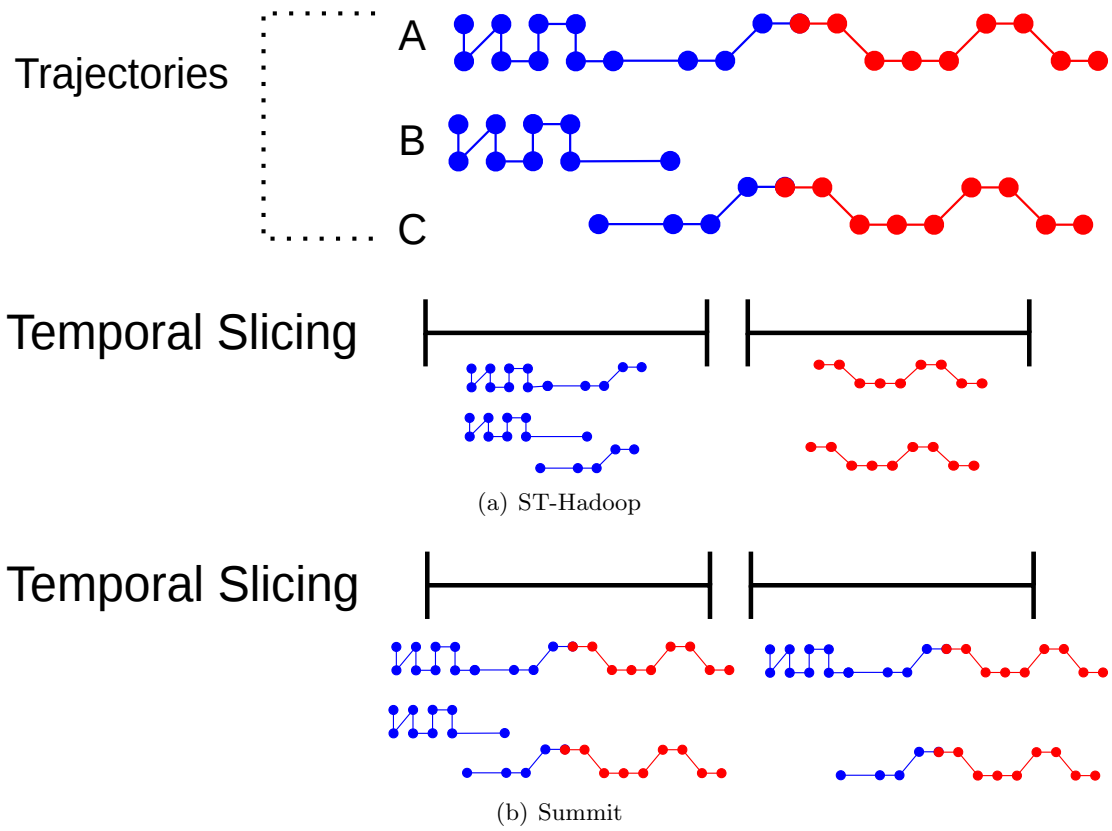
Input files in Hadoop Distributed File System (HDFS) are organized as heap files, where data is loaded into consecutive chunks, each of size 128MB. Though this was acceptable for analysis tasks that do not require indexing, it will result in sub-performance for applications, where indexing is essential. Recent efforts investigated in-memory indexing on Spark [90], yet it does not have any HDFS indices. In the meantime, spatiotemporally indexed HDFSs, as in ST-Hadoop [23, 42], are geared towards supporting queries with spatio-temporal predicates for basic geometrical shape, e.g., Point, Line, and Rectangle. On the other side, trajectories consist of a set of correlated sequence of spatio-temporal points, where ST-Hadoop is unable to realize the correlation between these sequences.

Summit organizes input files in HDFS in a way that preserves the geometrical topology of trajectories. In particular, data is spatiotemporally loaded and partitioned across computational nodes. Each partition holds the full sequence of trajectories that overlap with its spatio-temporal boundaries. Summit sacrifices storage to achieve higher performance by enforcing data replication across partitions. As a result, trajectory operations can have minimal data access to retrieve the query answer, reduce the computation complexity, and allow applications to run more sophisticated operations on the entire trajectories.

Summit employs a two-level indexing scheme of temporal indexing followed by a spatial one. The index is stored in the master node as auxiliary file, while actual partitions are divided across computation nodes. The process of index construction in Summit goes through the following three consecutive phases:

1. Sampling: The objective of sampling is to approximate the trajectory distribution and ensure the quality of partitioning. Due to the mass volume of data, Summit scans a representative sample that fits-in the main memory of the master node.

2. Bulkload Partitioning: Summit manipulates the sample to construct boundaries of the two-level indexing of temporal and spatial, respectively. System parameters in a configuration file guide the indexing of each level, such as the temporal granularity and the spatial partitioning mechanism. Expert users and data practitioners that have good



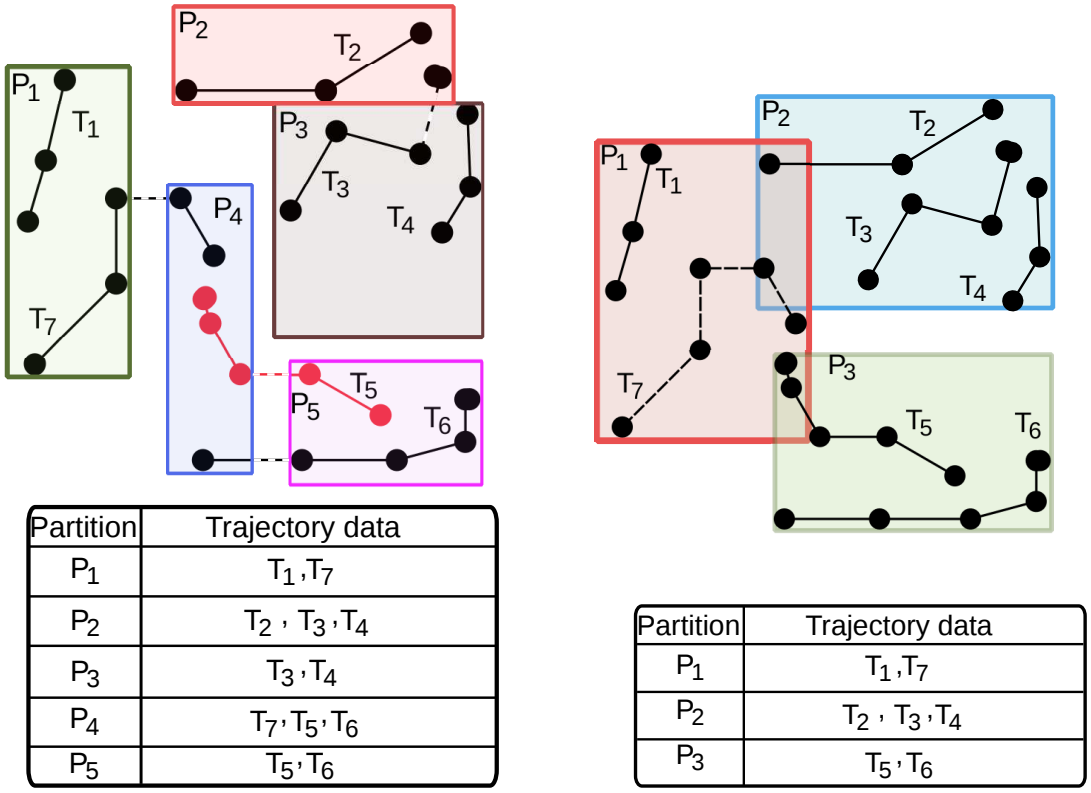
(a) ST-Hadoop

(b) Summit

Figure 7.3: Temporal Slicing

understanding of Summit and the nature of their datasets can tune these parameters. The construction of the two-level indexing scheme goes through two main steps:

1. *Temporal Slicing*: Figure 7.3 depicts the abstract idea of temporal slicing in Summit in comparison to ST-Hadoop. The temporal slicing mechanism in ST-Hadoop breaks trajectory into sub-sequences. Meanwhile, Summit slicing replicates trajectories if they overlap between temporal slices while maintaining non-overlapping disjoint. As shown in figure 7.3(b), the lifetime of trajectory `A` overlaps both the first and second slices, and thus, the entire trajectory will be replicated between those two temporal slices. As opposed to ST-Hadoop in figure 7.3(a) where `A` chopped into two sub-sequences, each stored separately in a different temporal partition.



Partition	Trajectory data
P_1	T_1, T_7
P_2	T_2 , T_3, T_4
P_3	T_3, T_4
P_4	T_7, T_5, T_6
P_5	T_5, T_6

(a) Spatial-based

Partition	Trajectory data
P_1	T_1, T_7
P_2	T_2 , T_3, T_4
P_3	T_5, T_6

(b) Segmentational-based

Figure 7.4: Trajectory Indexing

2. *Spatial Indexing*: Summit is equipped with the following two spatial indexing approaches for each temporal slice from the previous step, namely *Spatial-based* or *Segmentation-based*. Figure 7.4 illustrates the logical design of both methods, where rectangles represent the boundaries of the HDFS partitions while dots and lines depict the trajectory information. (a) *Spatial-based*: This approach preserves the spatio-temporal locality closeness between sub-trajectories. The boundaries of the HDFS partition split trajectories as shown in figure 7.4(a). (b) *Segmentation-based*: This is a data partitioning that guarantees that the entire trajectory is stored in a single HDFS block, as shown in figure 7.4(b). The minimum boundaries rectangles of this index might overlap. When a trajectory intersects with more than a single rectangle, its going to be replicated between partitions. This

partitioning is more in favor of operations that not only need to process the locality of trajectories but also their semantic or shapes over time, such as Similarity kNN and join queries.

3. Physical Assigning: The objective of this phase is to scan through the whole data and assign each record to the boundaries layout constructed from the previous phase. Summit initiates a map-reduce job that scans through the input file, physically partitions HDFS block, and assigns records to all overlapping partitions.

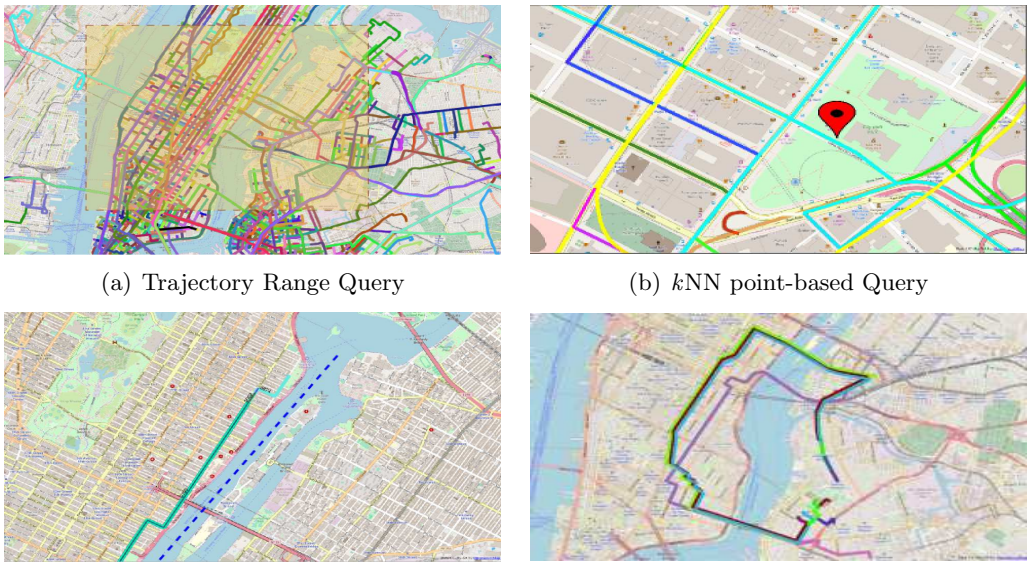
7.5 Trajectory Operations

In this paper, we discuss the internal execution of three basic operations in Summit, namely, range, nearest neighbor, and similarity queries. Other spatio-temporal operations on trajectories, e.g., reverse kNN, aggregation, and path queries, can be realized following similar approaches.

• **Trajectory Range Query (TRQ):** Given a three-dimensional query predicate, this query retrieves all trajectories that overlap with the query region in both space and time. Figure 7.5(a) shows an example of this type of query that " *Finds all taxis in downtown Manhattan between January and March 2019*". Regardless of the type of partitioning to answer the query, we employ an algorithm that runs in three steps namely, temporal filtering, spatial search, and spatio-temporal refinement. In the refinement phase, an extra processing is required to remove duplicates from the query answer, as trajectories might be replicated between partitions.

• **Trajectory *k* Nearest Neighbor Query (T*k*NN):** Summit supports the following two variants of the kNN operation:

- (1) ***k*NN point-based.** Given a query predicate that consists of query point $P_{(x,y)}$, and time interval $[t_1, t_2]$, find the k nearest trajectories to the query point during the given time interval. For example, "*Find the closest four animals to a Minnehaha waterfall between August and September*". Another example shown in



(a) Trajectory Range Query (b) kNN point-based Query

(c) kNN Trajectory-based Query (d) Similarity Query

Figure 7.5: Summit Trajectory Operations

Figure 7.5(b) of a kNN point-based query that "*finds 8-closest trajectories to New York city hall*".

- (2) ***k*NN trajectory-based.** Given a query trajectory Trj that consists of a sequence of spatio-temporal points, find the kNN trajectories to the whole trajectory points for every time instance according to some aggregate function, such as Min or Max. Figure 7.5(c) illustrates an example of a kNN trajectory-based query that "*finds 4-closest taxi trips aligned with East River*". This type of query is essential in many trajectory applications For example, environmental science a domain experts which to "*Find the closest two human traveled along a contaminated water stream in Jun 2018*" efficiently.

Answering both queries of kNN point-based or kNN trajectory-based, Summit employs an algorithm that consists of three phases, namely, partitioning, local computation, and global computation. In the partitioning phase, Summit decides which partition technique will be used. Once data is partitioned, Summit triggers a local computation algorithm to find a candidate set from the overlapping partitions. After performing a local computation, each computation node in Summit cluster will have

its own candidate set. The global computation phase is implemented in Summit as a reduce function, which runs on a single machine to compute the final result. Duplicate elimination is applied in this phase.

• **Trajectory Similarity Query (TSQ):** The objective of this query is to find similar trajectories to a given one based on some defined similarity function. This is a very useful query for many applications, such as transportation and advance pattern mining queries. A typical example of such queries shown in Figure 7.5(d) that "*Finds the k taxis that share similar routes with a given trajectory (e.g., another taxi) during some time interval (e.g., yesterday)*". Summit goes through two phases to find similar trajectories, namely, partitioning and computation phases. The partitioning phase indexes data by segmentation-based model. The computation phase runs in single map-reduce tasks for local and global computation. Duplicate removal takes place in the reduce phase. In Summit, we implemented the most robust and widely adopted similarity function, i.e., the Dynamic Time Warping [113], where we apply spatio-temporal thresholds. Other similarity measurements can be realized following the same approach.

7.6 Trajectory Range Query (TRQ)

Trajectory range query is specified by two predicates of a spatial area and a temporal interval, A and T, respectively. The query finds a set of trajectory records t that overlap with both a region A and a time interval T, such as "*Find all taxis in downtown Manhattan between January and March 2019*". Regardless of the type of trajectory partitioning to answer the query, Summit employs an algorithm that runs in three steps namely, *temporal filtering*, *spatial search*, and *refinement* with duplicate avoidance, described below.

In the **temporal filtering** step, the hierarchy index is examined to select a subset of partitions that cover the temporal interval T. The main challenge in this step is that the partitions in each granularity cover the whole time and space, which means the query can be answered from any level individually or we can mix and match partitions from different level to cover the query interval T. Depending on which granularities are used to cover T, there is a trade-off between the number of matched partitions and the amount of processing needed to process each partition. To decide whether a partition P is selected or not, ST-Hadoop computes the coverage ratio along with the number of partitions needed to be processed and then selects the granularity based on the minimum number of partitions.

In the **spatial search** step, Once the temporal partitions are selected, the *spatial search* step applies the spatial range query against each matched partition to select trajectories that spatially match the query range A. As partitions are indexed and distributed across nodes, computation carry out across Summit cluster for processing partitions, and thus maximizes the computing utilization of the machines.

Finally in the **refinement** step, compares individual records returned by the *spatial search* step against the query interval T, to select the exact matching records. This step is required as some of the selected temporal partitions might partially overlap the query interval T and they need to be removed. Similarly, Summit refines on the spatial query area A. In this refinement step duplicates avoidance take place, as trajectories are replicated between partitions.

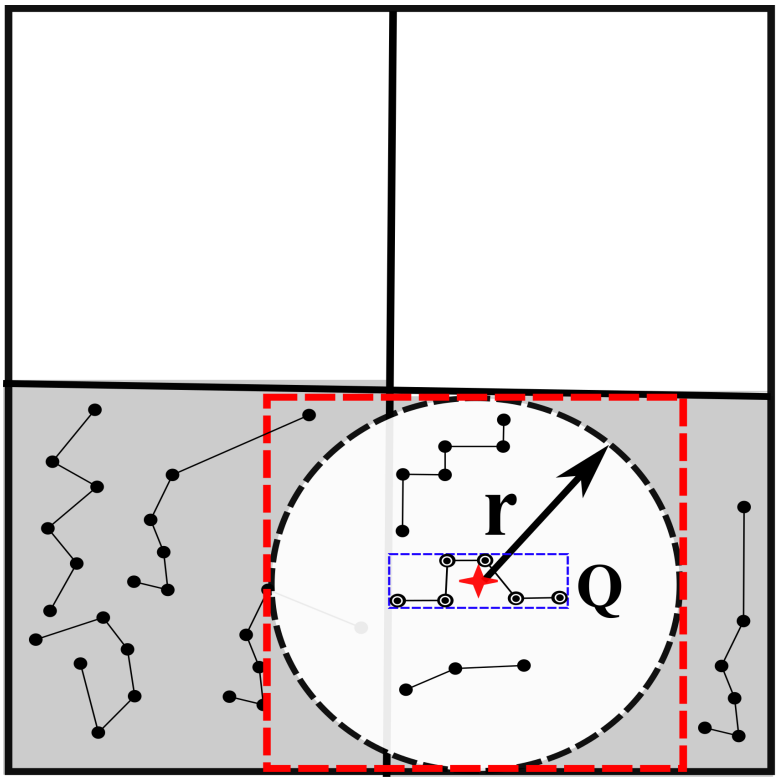


Figure 7.6: Local Computation: initial k answer

7.7 Trajectory nearest neighbor Query (TKNN)

7.7.1 (TKNN) Point-based

The trajectory point-based nearest neighbor query takes a trajectory Q, a spatio-temporal predicates θ, a spatio-temporal ranking function F_α, and an integer k as an input, and returns the k spatiotemporally closest points to Q such that: (1) The k points are within the temporal distance θ_{time}. (2) The k points are not far from any trajectory sequence with a spatial distance θ_{space}. (3) The top k points are ranked according to the spatio-temporal ranking function F_α that combines the spatial proximity and the temporal closeness of $p \in P$ to the a query point $q \in Q$. For example, "*Find the closest ten persons to a Jon commute to work in the last three months*". With the spatio-temporal information of the trajectory Q, Summit adds a spatio-temporal ranking function F_α to the kNN point-based query. The ranking function allows

Summit to compromise between spatial proximity and temporal closeness of its top-k points to the the trajectory query Q.

Definition Trajectory Spatio-temporal Ranking Function.
The ranking function F_α indicates whether a user query leans toward spatial proximity or temporal concurrency. If $\alpha = 1$, then the user cares about spatial closeness, i.e., the top-k results will be spatially closest to the trajectory query Q. If $\alpha = 0$, then the user cares about temporal concurrency, i.e., the top-k results will be temporally recent to trajectory query. Meanwhile, if α value is between zero and one, then the user cares about spatio-temporal proximity. The spatio-temporal proximity between any points in a trajectory to other points can be computed with the following mathematical equation.

$$\begin{aligned} F_\alpha(Q,p) = \quad & \alpha \times SpatialDist(Q_0, Q_n, p.location) \\ + \quad & (1-\alpha) \times TemporalDist(Q.interval, p.timestamp) \end{aligned}$$

The spatio-temporal ranking function F_α dependents on both *SpatialDist* and the *TemporalDist* functions, which they are normalized and monotonic. Each has a value range from zero to one. The *SpatialDist* is the Euclidean distance between trajectory Q and a location of p divided by the maximum spatial distance θ_{space}, where θ_{space} is the distance from a trajectory Q to the k^{th} furthest location. Meanwhile, the *TemporalDist* is that ratio of delta times of Q and p to the total temporal interval θ_{time}. The temporal interval θ_{time} is the time distance from the trajectory Q to the k^{th} furthest point in time.

Summit applies a simple and efficient technique that capable of pruning the search space to process only n number of partitions, which guarantee that those partitions will have the final k answers. Summit kNN point-based algorithm runs in three phases, partitioning, local computation, and global computation. Details of each phase discussed as following.

Phase 1: Partitioning

In this phase, Summit is spatiotemporally aware of trajectory locality in HDFS partitions. Summit applies spatial-based partitioning technique discussed in section 7.4. This partitioning technique is well suited with the point-based nearest neighbor query,

mainly because there is no need to preserve the trajectory sequences in kNN computation. Although other partitioning technique could be used in this step, they will result in sub-performance for any kNN point-based query.

Phase 2: Local computation

The objective of this phase is to find a set of candidate k result form overlapping partitions with trajectory query Q. Since a trajectory Q could intersect with multiple partitions, in the local computation Summit finds the initial answer for from each partition. Then it feeds each initial answer to the global computation, where it checks for the correctness of the final answer.

First, Summit search for overlapping partitions with the trajectory query Q. Then for each partition it finds the initial k answer. Summit locates the partition that intersects with trajectory Q, by feeding the *SpatioTmeporalFileSplitter* with a filter function that selects only the overlapping partition from the temporal interval. Summit exploits a *SpatioTmeporalRecordReader* to reads the selected partition, then executes a traditional kNN algorithm for every sequence in Q to produce the initial k answers. The function F_α computes the spatio-temporal distance between any trajectory points in the partition and trajectory Q.

Figure 7.6 shows an example of how Summit finds the initial k in the local computation. After executing a traditional kNN algorithm, Summit draw a local test circle to check for the correctness of the local kNN computations. The radius of the test circle is equal to the distance between the furthest k initial point in the answer with the furthest point in trajectory Q. The center of the circle is the midpoint of the diagonal trajectory query rectangle. If partitions overlap with the test circle, then Summit exploits its *SpatioTmeporalRecordReader* to reads the overlapped partitions, until the furthest k test circle does not overlap with any additional partitions.

Phase 3: Global Computation

The input of this phase is the initial k answers from all partitions overlapped with the trajectory query Q. Summit primary check for the correctness of the initial answer, and applies a kNN refinement to remove duplicates in the final answers.

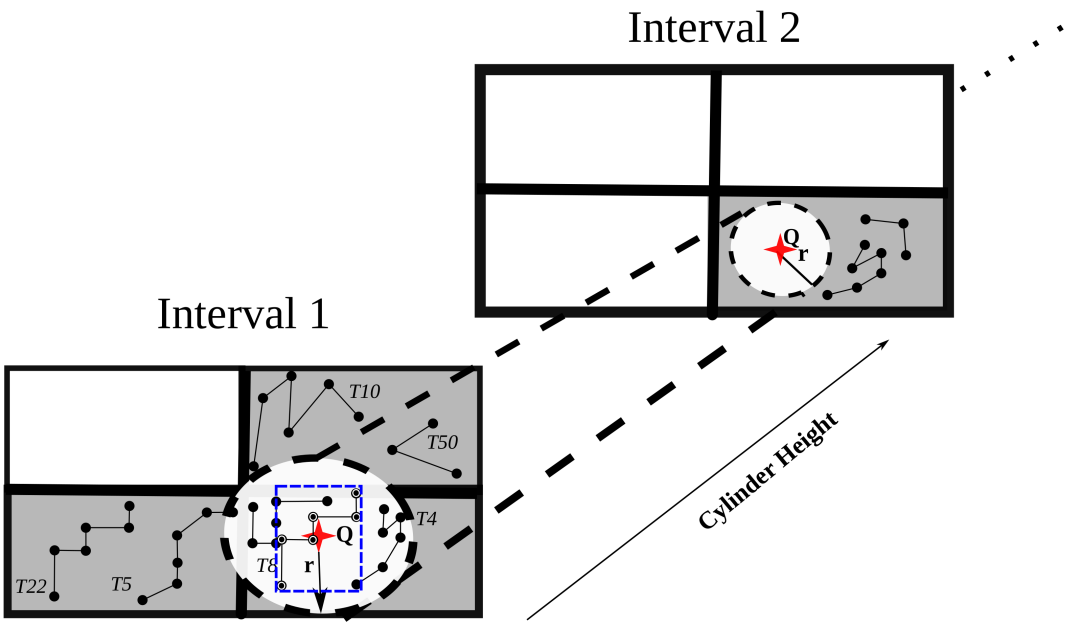


Figure 7.7: Correctness Check Cylinder

In the **correctness check step**, Summit checks if the initial k answer can be considered final. The main idea is similar to ST-Hadoop, where we draw a test Cylinder centered at Q with some radius r. The only different is in the radius computation, such that it consider trajectory query rather than query point. As shown in Figure 7.7, Summit radius is equal to the distance between the furthest k initial point in the answer with the furthest point in trajectory Q within a single temporal interval. The height l of the cylinder is equal to the temporal distance from Q to its k^{th} furthest neighbor in time. The radius and the height of the cylinder change only if there is potential point dominate the score of the ranking functions of the furthest k^{th} point in the initial results in any dimension, i.e., space or time. If the cylinder does not overlap with any partitions spatially or temporally, then we terminate the process, and the remove duplicates from the initial answer and it is considered as final answer.

Figure 7.7 illustrates the idea of test cylinder. The query point Q initially overlap with a single time interval. Summit check if some points either in the next or previous interval can dominate the score of the k^{th} furthest neighbor from the same initial interval, e.g., interval 1. If a dominance point exists, then Summit modifies the cylinder height

and radius accordingly in the next interval. Notice that the radius of the cylinder in next time interval is getting smaller, this is because we gradually draw the test cylinder. We continue this process until we reach a time interval that has no dominance point that can dominate the k^{th} furthest point.

In the ***k*NN refinement step**, The cylinder radius and height are obtained from the previous step. Summit locates partitions that overlap with test cylinder, by feeding the MBR of each circle in a temporal interval to the *SpatioTmeporalFileSplitter*. Finally, we scan over the trajectory records by reading partitions through *SpatioTemporalRecordReader* and process it with the traditional kNN algorithm to find the final answer. duplicate avoidance is applied in this step.

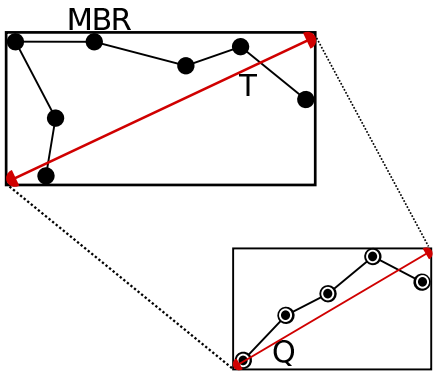


Figure 7.8: MBR Trajectory distance

7.7.2 (TKNN) Trajectory-based

The kNN trajectory-based query takes a trajectory Q, a spatio-temporal predicates θ, a spatio-temporal ranking function F_α, and an integer k as an input, and returns the k spatiotemporally closest trajectory to Q such that: (1) The k trajectories are within the temporal distance θ_{time}. (2) The k trajectories are not far from any Q with a spatial distance θ_{space}. (3) The top k trajectories are ranked according to the spatio-temporal ranking function F_α that combines the spatial proximity and the temporal closeness of $t \in T$. For example, "*Find the closest ten people commute along Hudson river in the last month*". With the trajectory query Q, Summit adds a spatio-temporal ranking function F_α to the kNN point-based query. The ranking function allows Summit to compromise between spatial proximity and temporal closeness of its top-k points to the the trajectory query Q.

The computation of the trajectory ranking function F_α discussed great details in section 7.7. The main modification to the ranking function is they way we compute the the spatial distance between two trajectories. In literature the distance between two trajectories are usually measured by some kind of aggregation function [119]. In Summit we consider the distance between the minimum bounding rectangles (MBR) of the two trajectories as a measurement [120], mainly because this will incur less computation than other aggregate distance measurements. Other aggregate measurements can be easily added to Summit library. Figure 7.8 illustrates the main idea of MBR distance

between two trajectories. Each MBR is identified by lower and upper bound points of $(x1, y1)$,and $(x2, y2)$, respectively. The equation of the distance between the two trajectories $Distance(Q, T)$ calculated as following:

$$\sqrt{\left(\Delta\left([Q_{x1}, Q_{x2}], [T_{x1}, T_{x2}]\right)\right)^2 + \left(\Delta\left([Q_{y1}, Q_{y2}], [T_{y1}, T_{y2}]\right)\right)^2}$$

Summit employs a simple and efficient technique that prune the search space to process fewest number of partitions, that guarantee having the final k nearest neighbor answers. Summit kNN trajectory-based algorithm runs in three phases, partitioning, local computation, and global computation. Details of each phase discussed as following.

Phase 1: Partitioning

In partitioning phase, Summit favor organizing the full sequence of a trajectory within the boundary of the HDFS partitions. Summit applies segmentational-based partitioning technique discussed in section 7.4, as this partitioning techniques preserves the shape and the full sequence of trajectories. Although spatial-based partitioning technique could be used in this step, it will result in sub-performance, especially that the full sequence of trajectory must be obtained and considered in the kNN computation to find the final answer.

Phase 2: Local computation

The objective of this phase is to find a set of candidate k result form overlapping partitions with trajectory query Q. The computation of finding the local candidates will be distributed, such that each map task will process a single partition and reports its initial k candidates set to the next phase. The local computation goes through two consecutive steps, of finding initial k and check the k set correctness. The two steps are described as following:

In the **local initial k step**, Summit assign a partitions to a map tasks, and each map task finds k candidate set. Figure 7.9 illustrates the algorithm for the local computation of kNN trajectory-based in Summit. As depicted the trajectory overlap with 3 partitions. Finding the initial answer of each overlapped partition will be carried by a separate map task. Summit locates partitions that intersect with trajectory Q,

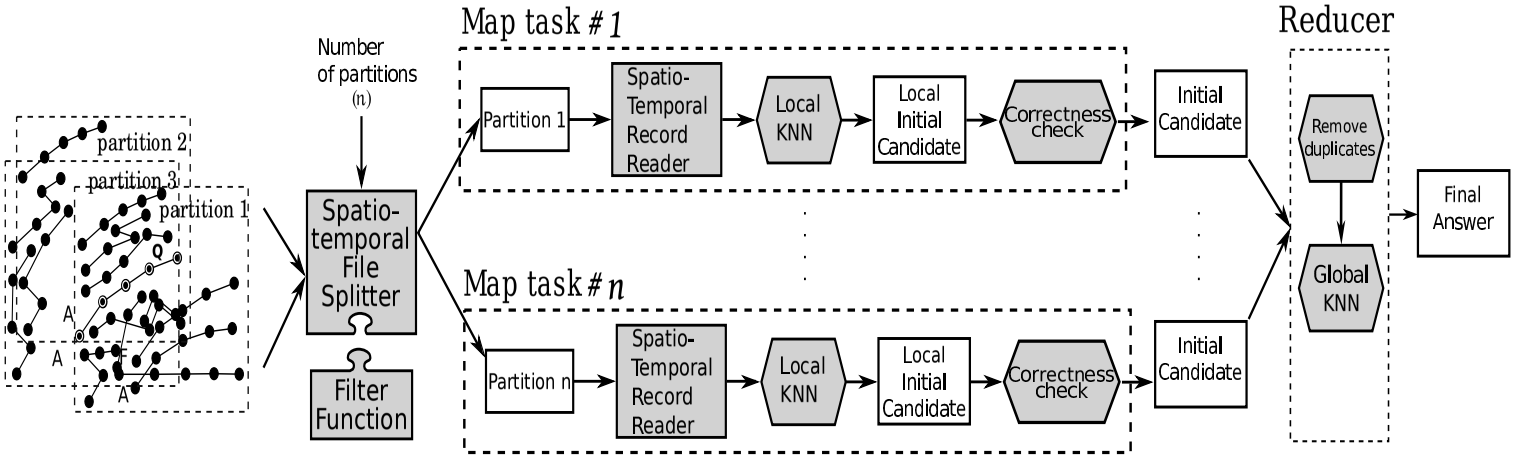


Figure 7.9: Nearest Neighbor Trajectory-based in Summit

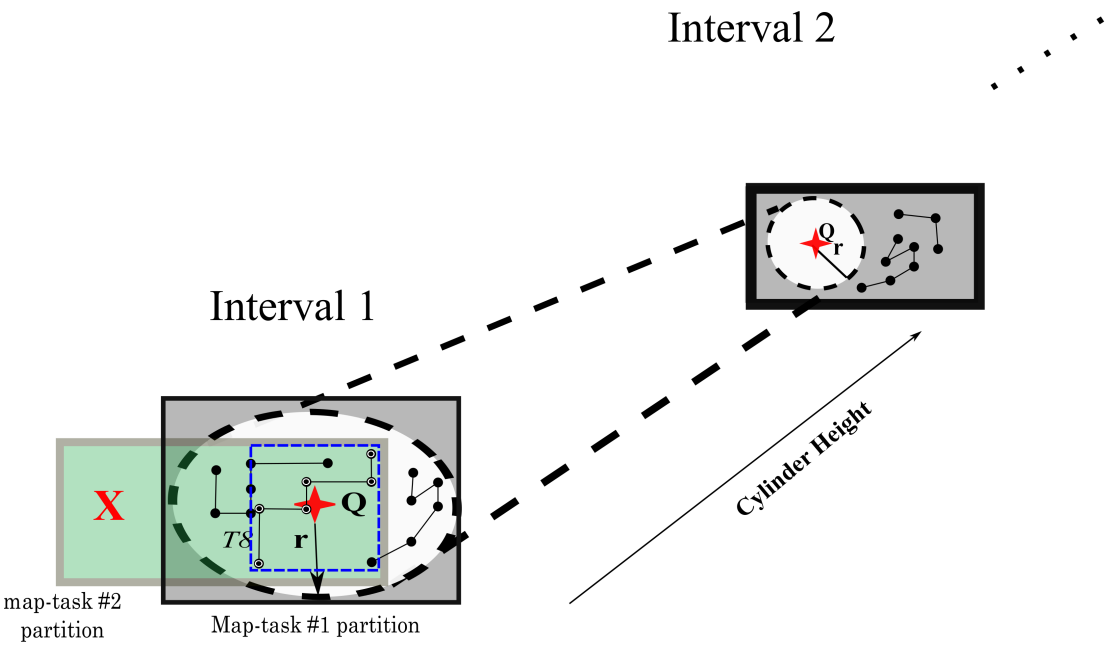


Figure 7.10: Nearest Neighbor Trajectory-based correctness check

by feeding the *SpatioTmeporalFileSplitter* with a filter function that selects only the overlapping partitions. Summit assign each partition to a map task, in which a single node in the cluster will carry the processing of that task. Each map task exploits a *SpatioTmeporalRecordReader* to reads the trajectories within its assigned partition, and performs a traditional kNN algorithm, to produce the initial k answer within the selected partition.

In the **correctness check step**, a map task check the correctness of its initial answer and feeds the k final candidate set to the next phase. Figure 7.10 shows an example of how Summit draw a local test circle to check for the correctness of the local kNN computations carried by the map task. The radius of the test circle is equal to the distance between the furthest k trajectory in the initial the answer and trajectory Q. The center of the circle is the midpoint of the diagonal trajectory query rectangle. If partitions overlap with the test circle, then only new partitions that have not been assigned to any other map task will be explored. Partitions that overlap with the test circle and already have been assigned to another map task by Summit will be ignored. For example as shown on the left most bottom of figure 7.10. A map-task 2 is already

processing a partition that overlap with the test circle of map-task 1. When map-task 1 check the correctness it will ignore map-task 2 partition. According to the value of function F_α, the drawing of the cylinder test follows the same technique discussed earlier in section 7.7. Summit exploits its record reader to reads trajectories from new partitions and executes a traditional kNN algorithm to recompute the k initial candidate sets. The final initial candidates will progress to the next phase.

Phase 3: Global Computation

In this phase Summit employs a reduce task to handle the processing of combining the output from each map task, remove duplicates, and generate the final answer. Each record from the previous phases represents a pair of the k trajectory and closeness score. There is no extra processing needed to check for the correctness or recompute the distance between trajectory Q and any of the initial candidates answer. The single reducer employs a priority heap to remove duplicates and generate the final results fast while scanning through the candidate sets generated from the previous phase.

7.8 Trajectory Similarity Query (TSQ)

The objective of this query is to find similar trajectories to a given one based on some defined similarity function. This is a very useful query for many applications, such as transportation and advance pattern mining [119, 120, 121, 122, 123, 124]. The similarity query takes a trajectory Q, a spatio-temporal predicates boundary, a similarity ranking function *Similarity*, and an integer k as an input, and returns the k most similar trajectories to a trajectory Q such that: (1) The k trajectories are within a temporal interval. (2) The k trajectories are inside a spatial area of the query predicate. For example, "*Find the three taxis that share similar routes with a given trajectory (e.g., another taxi) in downtown New York between January and March*". In Summit, we implemented the most robust and widely adopted similarity function, i.e., the Dynamic Time Warping [113]. Other similarity measurements can be realized following the same approach, such as Longest Common Sub-Sequence (LCSS), Edit Distance, Euclidean Distance, or Fréchet similarity functions. Interested readers can refer to a previous research study covers trajectory similarity functions in great details [107].

• Concept of Similarity Between Trajectories:

Measuring similarity between a pair of trajectories is essential for identifying portions that are common between the two. The similarity measurement must satisfy three main criteria: (1) The flexibility to identify similar trajectories on various times, (2) Ignores outliers points in similarity computation, and (3) The ability to identify the similarity between portions of trajectories. A couple of more complex metrics inspired from the sequence of similarity measures introduced in the literature [109]. Formally, we can say that a similarity function takes two trajectories and it generate a score indicating how the are similar based on some specification and criteria.

• Dynamic Time Warping (DTW):

The DTW was originally developed for matching speech signals in speech recognition [110], ever since it is consider as the one of the most robust and broadly adopted similarity function for trajectories and time series data [113, 117, 118] Formally, the

dynamic time warping con be defined as follow.

Given two trajectories $P(p_1, p_2, ..., p_n)$ and $Q(q_1, q_2, ..., q_m)$, every points in both $p_i \in P$ and $q_i \in Q$ is a spatio-temporal point. The distance between two pair points can be computed by any distance measurement between two points, such as Euclidean distance or Manhattan distance. In this paper we will consider using Euclidean distance. The following equation shows how the dynamic time warping computed. In Summit we use the Euclidean distance as `dist` function between two points.

$$DTW(P,Q) = \begin{cases} \sum_{i=1}^{n} dist(p_i, q_1) & \text{if } m = 1 \\ \sum_{i=1}^{m} dist(p_1, q_i) & \text{if } n = 1 \\ dist(p_n, q_m) + min\Big(DTW(P^{n-1}, Q^{m-1}), & \\ DTW(P^{n-1}, Q), DTW(P, Q^{m-1})\Big) & \text{otherwise} \end{cases} \tag{7.1}$$

Summit Similarity operation runs into two phases to find most similar trajectories, namely, partitioning and computation phases. The partitioning phase indexes data using segmentation-based technique. The computation phase runs as a single map-reduce tasks for local and global computation, respectively. Duplicate removal takes place in the reduce computation. Details of each phase discussed as follows.

Phase 1: Partitioning

In partitioning phase, Summit organizes the full sequence of a trajectory within a single HDFS partitions as shown in left most of Figure 7.11. In particular, Summit applies segmentational-based partitioning technique discussed in section 7.4, as this partitioning techniques preserves the shape and the full sequence of trajectories. Trajectory that overlaps with more than one partition will be replicated. The choice of this partitioning techniques will allows trajectory operations to have minimal data access when retrieve the query answer, reduce the computation complexity, and allow applications to run more sophisticated operations on the entire trajectories, such as finding similarity.

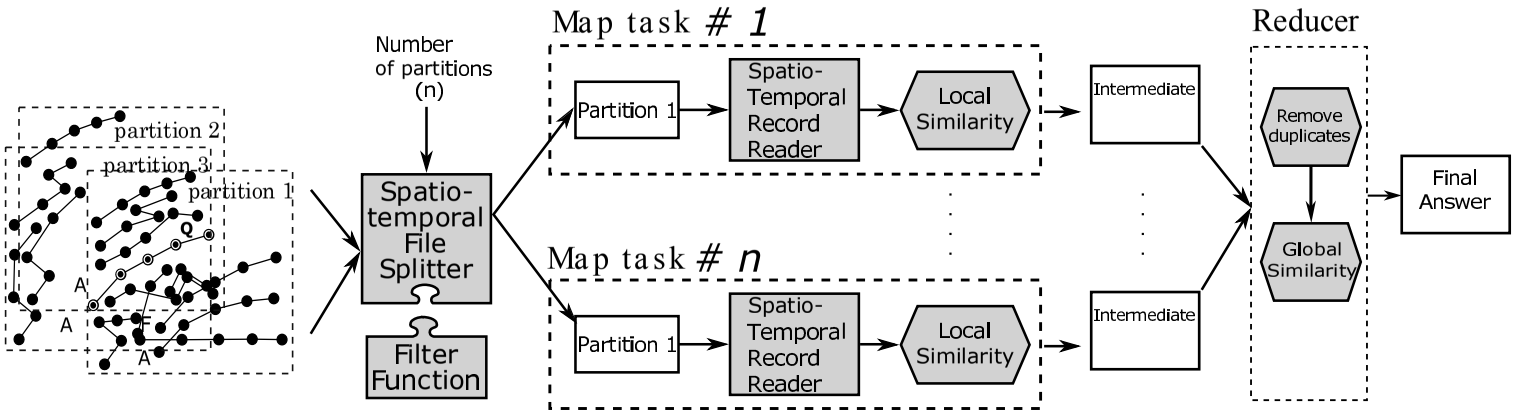


Figure 7.11: Abstract idea of similarity computation in Summit

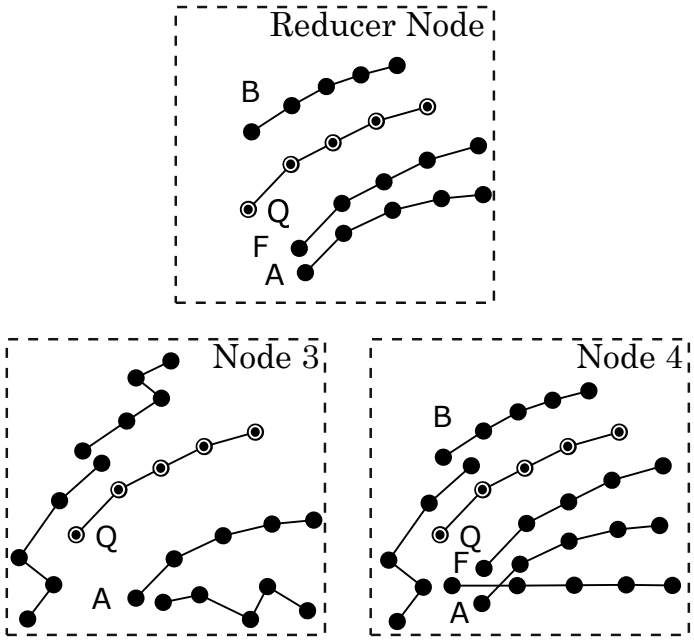


Figure 7.12: Summit global computation of similarity query

Phase 2: Local computation

The objective of this phase is to distribute the computation of the similarity measurements between nodes in cluster. The intermediate output of this phase is a pair of trajectory ID and similarity score, respectively. The score describes how similar a trajectory to the trajectory query Q. Summit exploits its *SpatioTemporalFileSplitter* to finds overlapping partitions with trajectory query Q. Next as illustrated in the Figure 7.11, Summit assign each overlapped partitions to a map tasks. The primary task of the map is to perform the similarity computation on all trajectories within its assigned partition.

Figure 7.11 illustrates the abstract idea of the algorithm for the local computation of similarity in Summit. First, Summit locates partitions that intersect with trajectory Q, by feeding the *SpatioTmeporalFileSplitter* with a filter function that selects only the overlapping partitions. As depicted the trajectory query Q overlaps with two partitions. Each map task exploits a *SpatioTmeporalRecordReader* to reads the trajectories within its assigned partition, and triggers a similarity computation measurement.

Phase 3: Global Computation

In this phase, as shown in Figure 7.12 Summit employs a reduce task to handle the processing of combining the intermediate output from each map task, remove duplicates, and generate the final answer. Each record from the previous phases represents a pair of a trajectory with its similarity score. There is no extra processing needed to check for the correctness or recompute the similarity between trajectories. The single reducer employs a priority heap of a length k, such that it scans through the intermediate output from the previous phase, removes duplicates on the fly, and generates the final result.

7.9 Experiments

This section provides an extensive experimental performance study of Summit compared to SpatialHadoop and Hadoop. We decided to compare with this two frameworks and not other spatio-temporal DBMSs for two reasons. First, as our contributions are all about supprting spatio-temporal trajectory data in Hadoop. Second, the different architectures of spatio-temporal DBMSs have great influence on their respective performance, which is out of the scope of this paper. Interested readers can refer to a previous study [77] which has been established to compare different large-scale data analysis architectures. In other words, Summit is targeted for Hadoop users who would like to process large-scale trajectory data but are not satisfied with its performance. The experiments are designed to show the efficient performance of Summit indexing and the overhead imposed by its new features compared to SpatialHadoop. However, Summit achieves two orders of magnitude improvement over SpatialHadoop and Hadoop.

In our experiments, we compare the performance of querying trajectories on Summit for spatio-temporal range, kNN-point, kNN-similarity, and join queries proposed in Section 5 to their spatio-temporal implementations on-top of SpatialHadoop and Hadoop, respectively. For range query, we use system throughput as the performance metric, which indicates the number of MapReduce jobs finished per minute. To calculate the throughput, a batch of 30 queries is submitted to the system, and the throughput is calculated by dividing 30 by the total time of all queries. The 30 queries are randomly selected with a spatial area ratio of 0.005% of New York City and a temporal window of 24 hours unless stated. This experimental design ensures that all machines get busy and the cluster stays fully utilized. For spatio-temporal join, we use the processing time of one query as the performance metric as one query is usually enough to keep all machines busy. The experimental results for range, nearest neighbor, and similarity queries are reported in Sections 7.9.2, 7.9.4, and 7.9.5, respectively. Meanwhile, Section 7.9.3 evaluates Summit the best execution plans for different workloads in Summit query optimizer.

Table 7.1: New York Taxi and Limousine Dataset

NYC	Size	#Records	#Points	Time segment
(Green+Yellow)	>2.6TB	1.3 Billion	> 77 Billion	2009-2016
Yellow	2.55TB	1.27 Billion	> 75 Billion	2009-2016
Green all	80GB	43 Million	2 Billion	2013-2016
Green-small	3GB	1.5 Million	180 Million	OCT-2016

Table 7.2: Summit Experiments Parameters Settings

Parameter	Values (default)
HDFS block capacity (B)	32, 64, (128), 256 MB
Cluster size (N)	5, 10, 15, 20, (24)
Selection ratio (ρ)	(0.01), 0.02, 0.05, 0.1, 0.2, 0.5, 1.0
Data-partition slicing ratio(α)	0.01, 0.02, 0.025, 0.05, (0.1), 1
Time-partition slicing granularity(σ)	(days), weeks, months, years
Similarity proximity (α)	0,0.2, (0.5), 0.6, 0.8, 1.0

7.9.1 Experiments Settings

Cluster Setup.

All experiments are conducted on a dedicated internal cluster of 24 nodes. Each has 64GB memory, 2TB storage, and Intel(R) Xeon(R) CPU 3GHz of 8 core processor. We use Hadoop 3.2.0 running on Java 10.0.2 and Ubuntu 18.04.1 LTS. Table 7.2 summarizes the configuration parameters used in our experiments. Default parameters (in parentheses) are used unless mentioned.

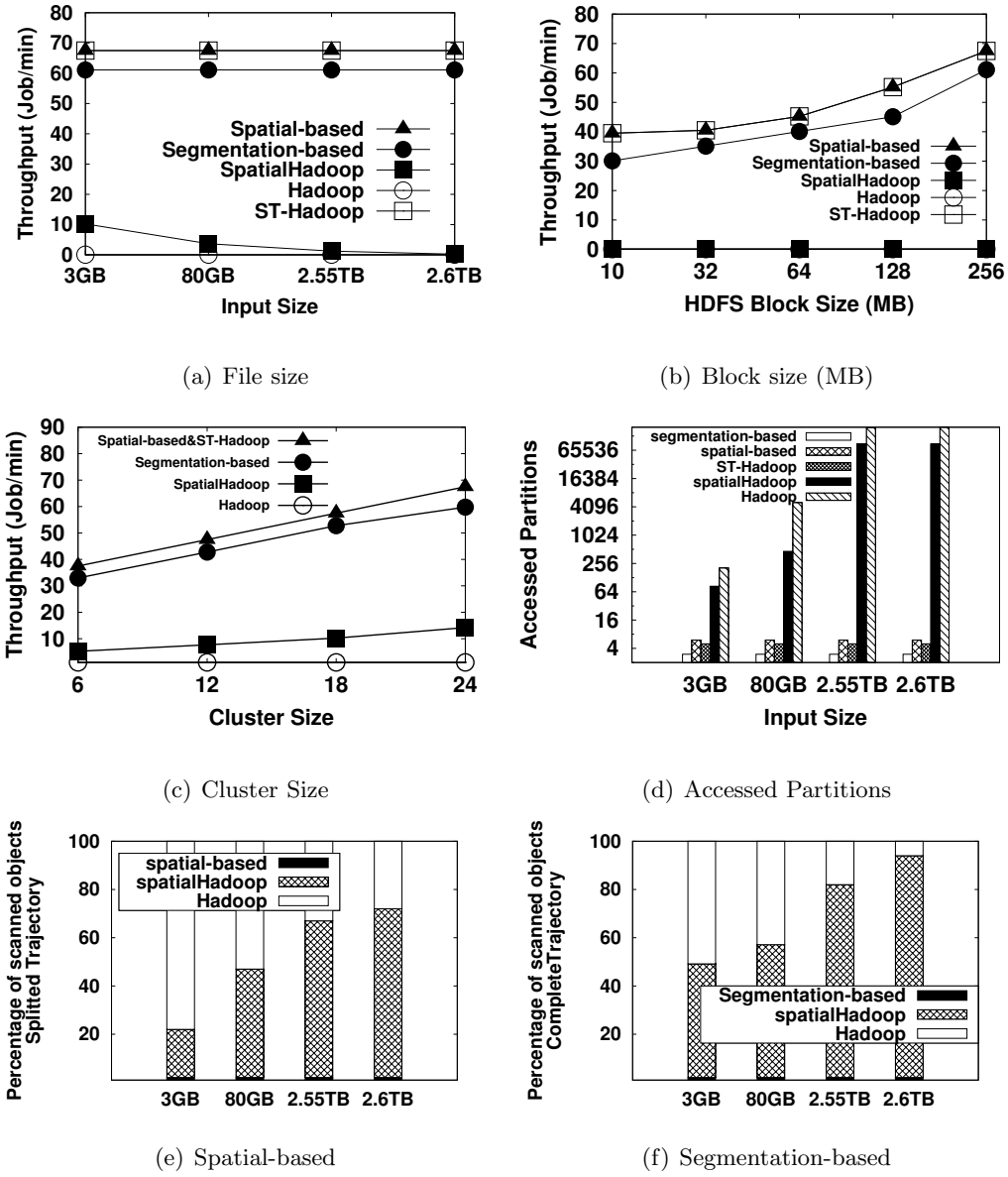
Datasets.

To test the performance of Summit we use the New York Taxi and Limousine commission (TLC) archived dataset [1]. The NYC (TLC) publicly released a dataset of taxi trips from January 2009 to June 2016 with GPS coordinates for both pick up and drop off locations. Later than Jun 2016 officials jeopardize GPS locations with area zone for commuters' privacy. However, the collected dataset contains over 1.3 Billion trips, were each trip spatiotemporally tagged with starting and ending for both location and time. The full trajectory of each trip is computed by dijkstra's algorithm on the New York road network obtained from [125]. The process of obtaining the full path of trajectories generated over 77 Billion spatio-temporal sequance for the 1.3 Billion trips with a

total size of 2.6 TB. To scale out time in our experiments we divided the dataset into different time segments and sizes, respectively as shown in Table 7.1. The default size used is grater than 2.6 TB which is big enough for our extensive experiments unless mentioned.

7.9.2 Range Query

In Figure 7.13(a), we increase the file size from 3GB to 2.6TB, while measuring the job throughput of Summit, SpatialHadoop, and Hadoop. Both partitioning techniques in Summit achieve more than two orders of magnitude higher throughput, due to its temporal load balancing of its spatio-temporal index. As the spatial-based technique retreives sub-sequance of trajectories, it achieves better performance than the segmentation-based in Summit. The main reason of this that in spatial-based partitioning technique all objects must be spatio-temporally `contained` inside the query predicate. In the meantime, Summit segmentation-based retreives the full trajectories that `overlap` with the spatio-temporal boundary of the query. As Hadoop needs to scan the whole file, its job throughput decreases with the increase of input file, which gives the worst throughput compared to SpatialHadoop and Summit. On the other hand, SpatialHadoop job thoughput decreases dramatically by adding the temporal predicate to the queries. As SpatialHadoop, needs to scan more partitions, which explain why the throughput of SpatialHadoop decreases with the increase of data records in the same spatial area over several years. Meanwhile, Summit throughput remains stable as it processes only partition(s) that intersect with both space and time.



(a) File size

(b) Block size (MB)

(c) Cluster Size

(d) Accessed Partitions

(e) Spatial-based

(f) Segmentation-based

Figure 7.13: Range Query

Figure 7.13(b) gives the impact of configuring the HDFS block size on the job throughput. Non-temporal index referred to both SpatialHadoop and Hadoop. Summit managed to keep its performance within orders of magnitude higher throughput even with various block sizes. The significance of this experiment confirms the outstanding

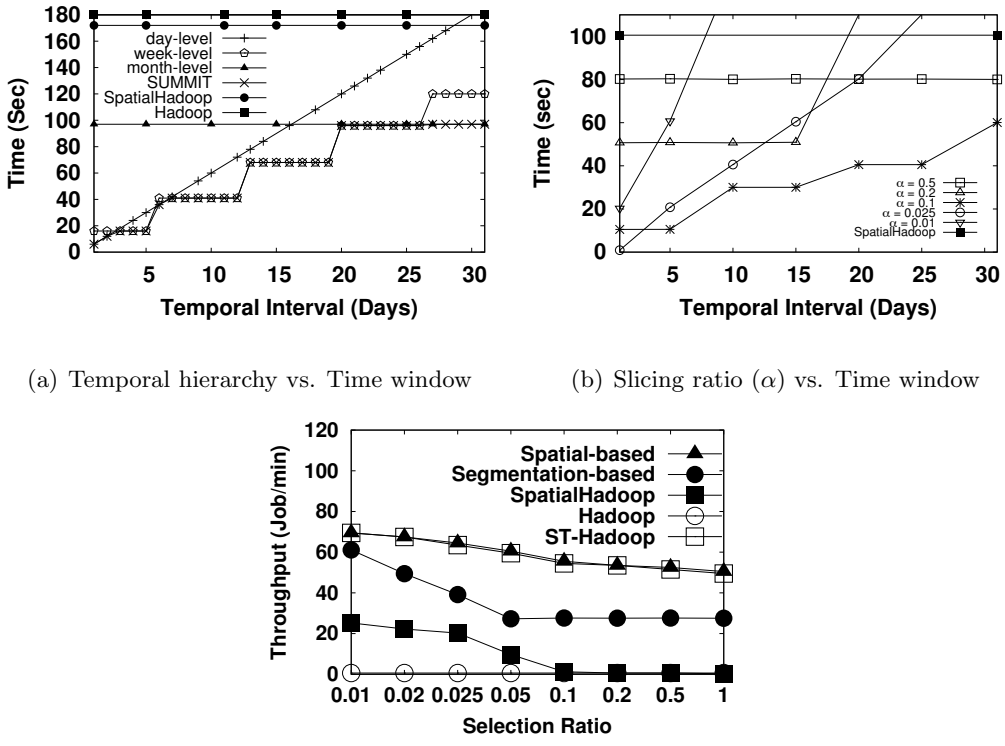
performance of Summit with increasing the HDFS block size. In the meantime, with decreasing the block size less than 32MB, the throughput slightly declines in Summit. As expected when minimizing HDFS block size, Summit performance slightly decreases for both segmentation-based and spatial-based, mainly because a query will require more blocks to process that leads to extra computation overhead in Summit cluster.

Figure 7.13(c), shows how Summit scales out with cluster size changing from 6 to 24 nodes when executing range queries with a spatio-temporal selection ratio of 0.01%. Summit, SpatialHadoop, and Hadoop smoothly scale with cluster size, while Summit is consistently more efficient than others. To scale-out the storage of 2.6TB on smaller clusters, we tuned the number of replica to zero in Summit cluster.

Extensive experiments are shown in Figure 7.13(d), analyzed the maximum number of accessed partitions of 30 queries submitted to each system. The queries are not overlapping and are randomly selected with a spatial area ratio of 0.005% of New York City and a temporal window of one month. Hadoop needs to access and scans all partitions for any input file. As for SpatialHadoop it slightly performs better by filtering partitions that do not overlap with the query. Yet, SpatialHadoop needs to access more than 40% of the total partitions. As the dataset is sparse and dense in a limited geographical space, SpatialHadoop fails to filter out partitions that overlap with the selected spatial boundary of a query. On the other hand, the number of accessed partitions in Summit remains stable as it only processes a fixed spatiotemporal area of the input file.

In Figures 7.13(e) and 7.13(f), we compute the average percent of the number of scanned objects to the total objects in each input file. Hadoop needs to scan all objects for any input files; thus, it scans 100% of records. As for SpatialHadoop, the percentage of scanned records increases with the increasing size of the input file. Expanding the size of the input file will lead to the fact that more data are being inserted to the same spatial region, in which SpatialHadoop adds more partitions within the same region. In the meantime, Summit recognize both space and time in its partitioning techniques, and thus, it only needs to scan less than 1% of data records.

7.9.3 Summit Stability



(a) Temporal hierarchy vs. Time window

(b) Slicing ratio (α) vs. Time window

(c) Selection ratio (ρ)

Figure 7.14: Summit stability

Extensive experiments in Figure 7.14 take Summit to an extreme edge by varying the temporal window and fluctuating the spatial minimum boundary rectangle in the query predicate. We analyzed the performance of trajectory indexing techniques on both space partitioning and data partitioning approach supported in Summit. When space partitioning is being adopted in Summit, a temporal hierarchy of trajectory index is created in the distributed file system as discussed earlier in Chapter 4 and Section 7.4. In contrary to the data partitioning technique, in which data are equally divided across computation nodes in each temporal level. Our experiments verify the efficiency and

the robustness of Summit compared to SpatialHadoop and Hadoop for every edge case.

Experiments in Figure 7.14(a) examines the performance of the temporal hierarchy index in Summit employing a temporal slicing with the segmentation-based for partitioning trajectory. We evaluate different granularities of time-partition slicing (i.e., daily, weekly, and monthly). In this figure, we fix the spatial query range and increase the temporal interval from 1 day to 30 days, while measuring the total running time. As shown in the Figures 7.14(a), Summit utilizes its temporal hierarchy index to achieve the best performance as it mixes and matches the partitions from different levels to minimize the running time.

Summit provides excellent performance for both small and large query intervals as it selects partitions from the level with the least number of partitions to process. When the query interval is very narrow, it uses only the lowest level (e.g., daily level), but as the query interval expand it starts to process the above level. In an edge case, when a query interval span two or more of the highest level in the temporal hierarchy (e.g., month), Summit computes the exact number of partitions that need to be processed by employing a greedy algorithm that finds the minimum number of partitions. Summit uses a top-down approach that starts with the top level and selects partitions that cover query temporal interval, If the query interval is not included at that granularity, then the algorithm continues to the next level. Summit finds the local optimal from each granularity until it reaches the global optimal, i.e., the minimum number of partitions that covers query predicate.

In Figure 7.14(b), we investigate the impact of various data-partition slicing ratio (α) on the query performance. Similarly to the previous experiments, we fix the spatial query range and increases the temporal interval up to a month. The trajectory data are sliced based on a different value of α. in which each ratio represents a level (i.e., a granularity) in Summit. The best query performance is shown around a slicing ratio of $\alpha = 0.1$. In case the query is less than a five days interval a higher slicing ratio might shine better than $\alpha = 0.1$, this is mainly a fewer HDFS block needed to be accessed for the query. However, this will dramatically change when the temporal query interval

expand.

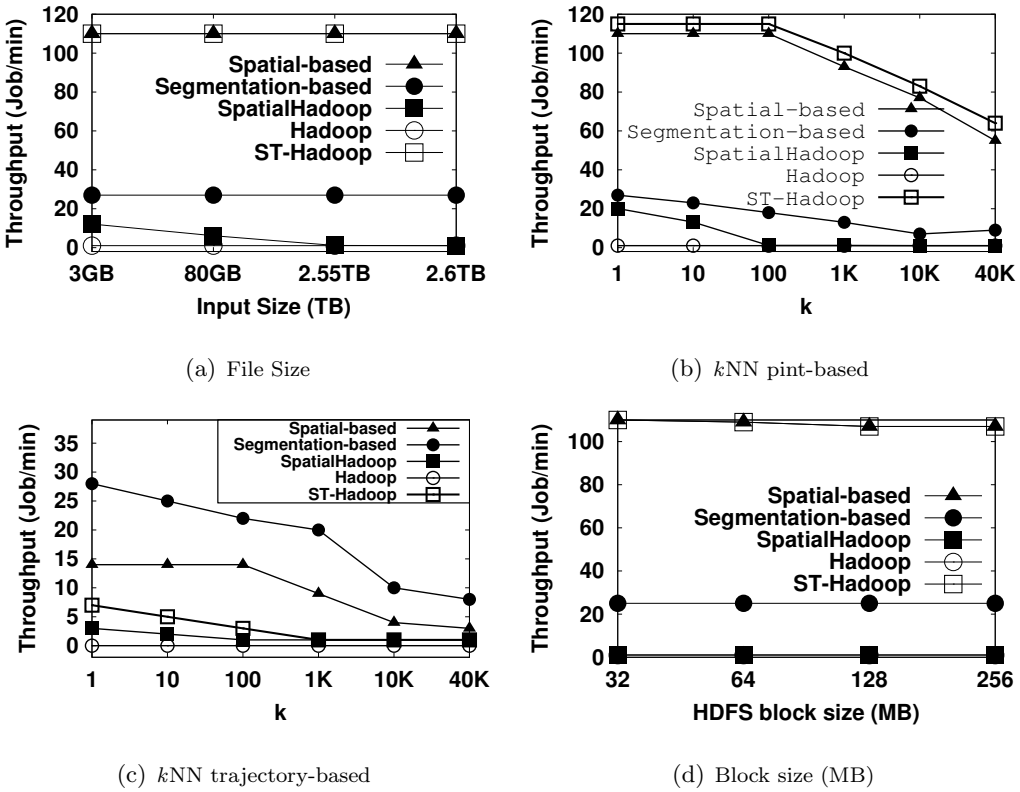
Experiment in figure 7.14(c) examines the performance of the range query with varying the spatial selection ratio from 0.01% to 1% of the entire area of New York City. We compute the average of time in seconds of 30 randomly selected queries, in which for each selection ratio we re-compute the spatial boundaries of the same selected queries. Regardless of the spatial selectivity Hadoop always scans the whole data file and filter in the reduce phase with the spatio-temporal predicate, in which it leads to the least performance. As for SpatialHadoop as expected the increase of the spatial ratio area demolish the performance dramatically.

In all cases, Summit empirically gives more than two orders of magnitude better throughput than Hadoop. The job throughput of all systems decreases with the increases of the query area, where more partitions needed to be accessed. Since Summit spatial-based retrieve sub-sequence of trajectories that entirely `contained` in the spatial predicate, it outperforms Summit segmentation-based. Thus, in segmentation-based (a) more partitions need to be accessed, and (b) the size of the result file is much larger.

7.9.4 Trajectory Nearest Neighbor query

In Figures 7.15 we extensively measure the performance of kNN-point and kNN-trajectory based queries implemented on Hadoop [78], SpatialHadoop, ST-Hadoop, and Summit for 2.6TB of NYC Taxi dataset. In these experiments, 30 query locations are set at random points (i.e., random points in both date and time) sampled from the whole input file, the number of k is set to 100. Unless otherwise mentioned.

Figure 7.15(a) measures system throughput with increasing the input file size. Summit has one to two orders of magnitude higher throughput. Hadoop and SpatialHadoop performances decrease dramatically as they need to process the whole file while Summit maintains its performance as it processes one partition regardless of the file size. As SpatialHadoop is not aware of the temporal locality of the data, it needs to process multiple partitions to finds the k nearest neighbor in a specific day, and in a



(a) File Size

(b) kNN pint-based

(c) kNN trajectory-based

(d) Block size (MB)

Figure 7.15: kNN Query

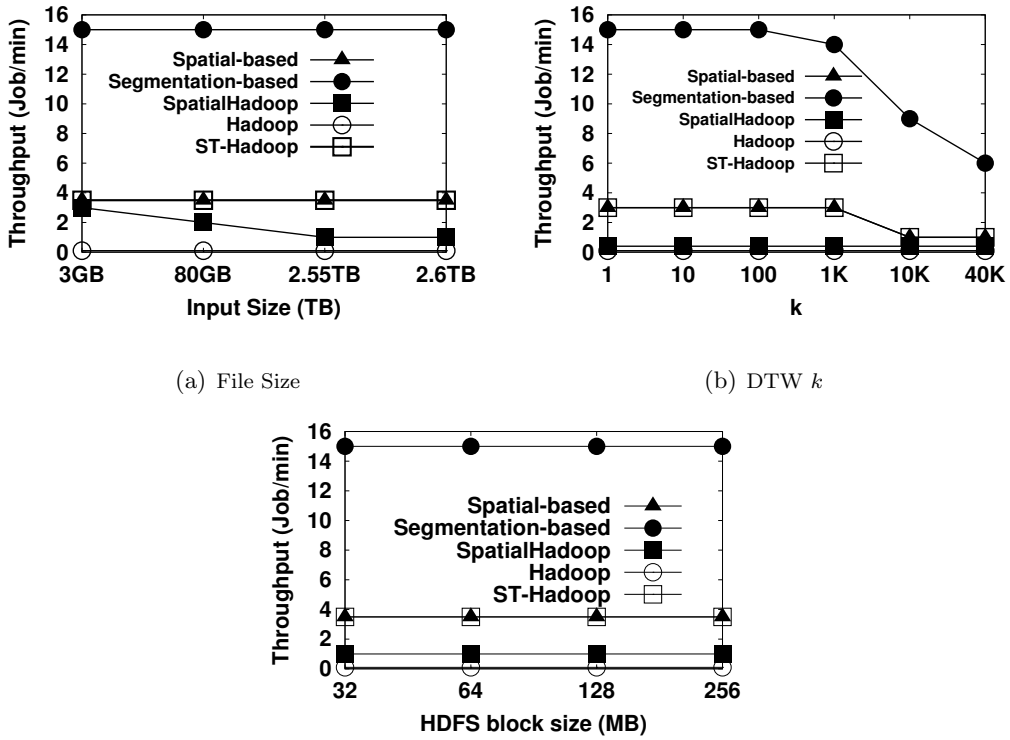
worst case it might end up processing all partitions. In the meantime, Summit keeps its speedup at two orders of magnitude, in which its index are spatio-temporally aware of trajectories locality.

Figures 7.15(b) and 7.15(c) give the effect of increasing k from 1 to 40K on the entire dataset. Summit gives an order of magnitude performance with both segmentation-based and spatial-based indexing. When varying the time window of the query Summit optimizes a query plan that uses the hierarchy index; thus, it achieves two orders of magnitude better performance compared to Hadoop kNN implementation. Summit efficiently handles spatio-temporal kNN operation. However, we notice that the job throughput decreases when k is roughly around and more than a thousand. This is expected as increasing the number of k will requires more partitions to be processed.

Summit is consistently better than the other systems. Since SpatialHadoop shortage in recognizing the temporal predicate of a query, the proposed algorithm needs to scan all neighbor partitions until the desire k is reached. Although, job throughput apparently decreases with the increased number of the nearest neighbor k, Summit optimize the execution of the query uses the hierarchy index to keep the performance stable. At some point of k, it will decrease, in which this is expected as the search area in both space and time will expand.

In Figure 7.15(d), shows the effects of different HDFS block size configuration on the job throughput. Thirty queries are fixed at a random location on the first day of a month. More trajectories can be stored when the size of the HDFS increases. Tuning the configuration of the HDFS block size does not influenced the performance of Summit kNN operation. The bottleneck of the performance is not the number of trajectories that fit a single block. Instead, it gets affected by how many blocks need to be accessed by a specific operation. In a case of kNN point-based, a query finds the nearest trajectory to a given trajectory in both space and time. Since Summit already consider the spatio-temporal locality of trajectories and the query time interval has been fixed in the queries, a kNN operation can efficiently locate the nearest trajectory from the same block that intersects with the trajectory query. As for comparing the two partitioning techniques in Summit, the spatial-based outperforms the segmentation-based, mainly because the spatial-based retrieve sub-sequence of a trajectory rather than the full trajectory as in the segmentation-based. Overall, Summit achieve orders of magnitude better performance than SpatialHadoop and Hadoop.

7.9.5 Trajectory Similarity query



(a) File Size

(b) DTW k

(c) Block size (MB)

Figure 7.16: DTW similarity Query

Experiments in figures 7.16 depict the processing performance of kNN-similarity query on Hadoop, SpatialHadoop, and Summit for 2.6TB of NYC Taxi dataset. In these experiments, we implemented the most robust and widely adopted non-metric similarity algorithm DTW on the top of Hadoop and SpatialHadoop. To compare the performance of other systems with Summit, thirty trajectories sampled from the whole input file. The location and the shape of each sampled trajectory are diverse, such that a trajectory Trj has a distinct neighborhood and journey from other ones in the sample. The number of k is set to 100. Unless otherwise mentioned.

Figure 7.16(a) measures the three system job throughput with increasing the input

file size. The query predicate consists of trajectory Trj and time window equivalent to the start and end time of Trj. Summit has a higher job throughput than Hadoop and SpatialHadoop. Evidently, the job throughput of Hadoop and SpatialHadoop decreases with increasing input files, as they need to process the whole file while Summit maintains its performance as it processes fewer partition(s) regardless of the input file size. Since SpatialHadoop is not aware of the temporal locality of the data, it needs to process multiple partitions to access trajectories that overlap with query trajectory Trj to finds the k similar ones from a specific day. In the meantime, Summit keeps its speedup at two orders of magnitude, in which its index are spatiotemporally aware of trajectories locality. Thus, a similarity query will access only partitions that overlap with the spatial and temporal query predicates, precisely, query trajectory Trj and time.

Figure 7.16(b) gives the effect of increasing k from 1 to 40K on the entire dataset. Summit gives an order of magnitude performance with both segmentation-based and spatial-based indexing. Summit gains its performance from its query optimizer, where a generated query plan selects few partitions from the hierarchy index; thus, it achieves two orders of magnitude better performance compared to Hadoop kNN DTW implementation. Summit efficiently handles spatio-temporal kNN-DTW similarity. However, we notice that the job throughput decreases when k is around and more than a thousand. This is expected as increasing the number of k will requires more partitions to be processed. Summit is consistently better than the other systems. Since SpatialHadoop shortage in recognizing the temporal predicate of a query, the proposed algorithm needs to scan all neighbor partitions until the desire k is reached. The length of trajectories in the similarity computation influences the job throughput, and that explains the small variance in job throughput between Summit spatial-based, and segmentation-based. In segmentation-based, the full trajectories are examined in the similarity computations. In contrary to the spatial-based, where a subset of trajectories gets to be analyzed by the similarity function.

In Figure 7.16(c), shows the effects of varying the HDFS block size on the job throughput. The time window in the query predicate of the thirty trajectory queries is fixed on the first day of a month. The kNN similarity operation, finds the k

nearest similar trajectory to a given trajectory Trj in both space and time. Since Summit already consider the spatio-temporal locality of trajectories and the query time interval has been fixed in the query predicate, a kNN similarity operation can efficiently locate the HDFS block that spatially intersects with Trj. As for comparing the two partitioning techniques in Summit, the spatial-based slightly outperforms the segmentation-based, mainly because the similarity function on spatial-based operates on sub-sequence of trajectories rather than the full sequence of trajectories as in the segmentation-based. Experimentally, Summit achieve orders of magnitude better performance than SpatialHadoop and Hadoop.

Chapter 8

Language Layer

ST-Hadoop and Summit do not provide a completely new language. Instead, they extend Pigeon language [20] by adding spatio-temporal data types, functions, and operations. Spatio-temporal data types are used to define the schema of input files upon their loading process. Meanwhile, spatio-temporal functions are used as interface to built-in operations that carry the processing on spatio-temporal data. Summit is a trajectory library added to ST-Hadoop; thus, both are going to be used interchangeably in this chapter, because they share the same language layer. In particular, we have added the following:

8.1 Basic Spatio-temporal Data types

ST-Hadoop extends basic geometrical shapes, such `STPoint`, `STLine`, and `STRectangle`. Also added `TIME`, and `INTERVAL`. The `TIME` instance is used to identify the temporal dimension of the data, while the time `INTERVAL` mainly provided to equip the query predicates. The following code snippet loads Twitter data from 'Twitter' file with a column of type `STPoint`.

```
tweets = LOAD 'Twitter' as
(id:int, STPoint(location:point, time:timestamp));
```

`Twitter` and `tweets` are the paths to the non-indexed heap file and the destination

indexed file, respectively. `location` and `time` are the columns that specify both spatial and temporal attributes.

Similarly if the shape of the spatio-temporal data is rectangle, then ST-Hadoop identifies the column of the rectangular shape as `STRectangle`. ST-Hadoop augment the basic spatial shape with time attributes. The following code snippet loads building shapes from OpenStreetMap (OSM) [125, 126] raw file, where `Rectangle` and `Time` are the geometrical shape of the building and the timestamp when building was added to OSM.

```
tweets = LOAD 'Building' as
(id:int, STRectangle(Shape:Rectangle, time:timestamp));
```

8.2 Basic Functions and Operations

Pigeon already equipped with several basic spatial predicates. ST-Hadoop changes the `overlap` function to support spatio-temporal operations. The other predicates and their possible variation for supporting spatio-temporal data are discussed in great details in [127]. ST-Hadoop encapsulates the implementation of three commonly used spatio-temporal operations, i.e., range, nearest neighbor, and Join queries, that take the advantages of the spatio-temporal index. The following example "*retrieves all tweets in Minneapolis city represented by its minimum boundary rectangle during the time interval of August 25th and September 6th*" from twitter indexed file.

```
tweets = FILTER twitter
 BY OVERLAP( STPoint,
    RECTANGLE(x1,y1,x2,y2),
    INTERVAL(08-25-2016, 09-06-2016));
```

ST-Hadoop extended the `JOIN` to take two spatio-temporal indexes as an input. The processing of the `join` invokes the corresponding spatio-temporal procedure. For example, one might need to understand the relationship between the birds death and the existence of humans around them, which can be described as "*find every pairs from*

birds and human trajectories that are close to each other within a distance of 1 mile during the last year".

```
human_bird_pairs = JOIN human_trajectory, bird_trajectory
 PREDICATE =  overlap( RECTANGLE(x1,y1,x2,y2),
               INTERVAL(01-01-2016, 12-31-2016),
               WITHIN_DISTANCE(1) );
```

ST-Hadoop extends `KNN` operation to finds top-k points to a given query point Q in space and time. ST-Hadoop computes the nearest neighbor proximity according to some α value that indicates whether the kNN operation leans toward spatial, temporal, or spaito-temporal closeness. The α can be any value between zero and one. A ranking function $F_\alpha(Q, p)$ computes the proximity between query point Q and any other points $p \in P$. The following code gives an example of kNN query, where a crime analyst is interested to find the relationship between crimes, which can be described as "*find the top 100 closest crimes to a given crime Q located in downtown that took place on the* 2^{nd} *during last year, with* $\alpha = 0.3$".

```
k_crimes = KNN crimes_data
 PREDICATE =  WITH_K=100
              WITH_alpha=0.3
              USING F(Q, crime);
```

8.3 Trajectory Spatio-temporal Data types

Additional to the basic spatio-temporal data types in ST-Hadoop, we have added A trajectory data type `STTrajectory`. The trajectory data type consists of a sequence of any of a basic spatio-temporal data type, such as `STPoint`, `STRectangle`, or `STLine`. Trajectory data types are fundamentally different than basic spatio-temporal shapes, where each sequence in trajectory have a different timestamp associated with it. Hence, a trajectory is defined by a set of basic geometrical shapes bounded by finite time interval derived for the trajectory sequence itself. The following code snippet loads

NYC taxi trajectories from 'NYC' file. Where `index`, `Level`, and `Granularity` are three parameters that indicate the trajectory partitioning techniques in the HDFS.

```
trajectory = LOAD 'NYC' as
(id:int, STTrajectory: <STPoint 1 - STPoint n >)
index:partition Level:1 Granularity:1-hour;
```

`NYC` and `trajectory` are the paths to the non-indexed heap file and the destination indexed file, respectively. The temporal interval are derived from the basic sequence shape, such that the `Interval` of a trajectory is equal to the period between the first entry and the last of a trajectory record.

8.4 Trajectory Functions and Operations

ST-Hadoop already added several basic spatio-temporal operation predicates. To exploit the Summit indexing and operations for processing trajectory, we added three function to support trajectory operations, namely, `KNN-point`, `KNN-trajectory`, and `Similarity`. Summit encapsulates the implementation of three commonly used spatio-temporal operations, i.e., nearest neighbor point-based, nearest neighbor trajectory-based, and similarity queries, that take the advantages of the trajectory index. In the meantime, Summit did not change the spatio-temporal `OVERLAP` predicate, as it is already recognize both basic and trajectory shapes.

The following example "*retrieves all cars in State Fair area represented by its minimum boundary rectangle during the time interval of August 25th and September 6th*" from trajectory indexed file.

```
cars = FILTER trajectory
 BY OVERLAP( STTrajectory,
    RECTANGLE(x1,y1,x2,y2),
    INTERVAL(08-25-2019, 09-06-2019));
```

Summit extends `KNN` operation to support two trajectory operations, of point-based and trajectory-based. In the `KNN-point` finds the top-k points to a given trajectory query

Q. Meanwhile, `KNN-trajectory` finds the top-k trajectory to the given trajectory Q. Summit for both version of the `KNN` operation computes the nearest neighbor proximity according to some α value that indicates whether the kNN operation leans toward spatial, temporal, or spaito-temporal closeness. The α can be any value between zero and one. A ranking function F_α computes the proximity between trajectory query Q and any other trajectory records.

The following code gives an example of kNN-point-based query, where a crime analyst is interested to find the relationship between crimes, with trajectory of specific person which can be described as "*find the top 3 closest crimes to a person trajectory Q on the 2^{nd} during last year, with $\alpha = 0.5$*".

```
k_crimes = KNN-point crimes_data
 PREDICATE =  WITH_K=100
              WITH_alpha=0.3
              INTERVAL(TIME)
              USING F(Q, crime);
```

The following code gives an example of Similarity query, where a zoologist is interested to find the migration of species, according to specified movement pattern which can be described as "*find the top 1000 animals traveled similarity to trajectory Q during last year*".

```
species = FILTER animals_movment
By OVERLAP( STTrajectory,
    RECTANGLE(x1,y1,x2,y2),
    INTERVAL(08-25-2019, 09-06-2019)
    SIMILAR(Q));
```

Chapter 9

Conclusion

In this thesis, we introduced ST-Hadoop [19] as a novel system that acknowledges the fact that space and time play a crucial role in query processing. ST-Hadoop is an extension of a Hadoop framework that injects spatio-temporal awareness inside MapReduce layers. The key idea behind the performance gain of ST-Hadoop is its ability to load the data in the Hadoop Distributed File System (HDFS) in a way that mimics spatio-temporal index structures. Hence, incoming spatio-temporal queries can have minimal data access to retrieve the query answer. ST-Hadoop is shipped with support for several fundamental spatio-temporal and trajectory operations, namely, spatio-temporal range, top-k nearest neighbor, similarity, and join queries. However, ST-Hadoop is extensible to support a myriad of other spatio-temporal operations. We envision that ST-Hadoop will act as a research vehicle where developers, practitioners, and researchers worldwide, can either use directly or enrich the system by contributing their operations and analysis techniques.

In Chapter 2 we provided a landscape and a comprehensive overview of existing research studies from both academia and industry in the area of supporting big spatio-temporal data. We have classified current works based on several criteria. In particular, the implementation approach, indexing techniques, operations, and language support.

In Chapter 3 we presented the architecture design of our proposed framework ST-Hadoop; as the first full-fledged open-source MapReduce framework with a built-in support for spatio-temporal data. The design distinguishes itself from existing work in the area of supporting spatio-temporal data.

In Chapter 4 we investigated two basic design of indexing in MapReduce to supports spatio-temporal data. The proposed indexing approaches incorporate the functionality of various big spatio-temporal batch workloads. In particular, we introduced data and space partitioning techniques for big spatio-temporal data. Also, we focus on supporting the incremental batch update nature of data in our design.

In Chapter 5 we detailed the implementation of three basic spatio-temporal operations, namely, spatio-temporal range, nearest neighbor, and join queries. We envision more operations can be added by professional developers, domain experts, and researchers following similar approaches discussed in this chapter.

In Chapter 6 we investigated the spatio-temporal query optimization. In particular, we developed two common query optimization models of heuristic and cost-based models.

In Chapter 7 we extend ST-Hadoop capability to support analytic operation on large scale trajectory data. We proposed a new extension Summit that is well-suited to efficiently support several basic trajectory queries, such as range, nearest neighbor, and similarity queries. These queries and the architectural design of the proposed library are extendable, in a way that it enables users to build various applications on trajectories and extends its functionality.

In Chapter 8 we described how casual users could interacts with ST-Hadoop through its language layer. We discussed basic spatio-temporal and trajectory data types, functions, and operations.

References

[1] Data from NYC Taxi and Limosuine Commission. `http://www.nyc.gov/html/tlc/`, 2019. accessed on April.

[2] `https://about.twitter.com/company`, 2019. accessed on April.

[3] Land Process Distributed Active Archive Center. `https://lpdaac.usgs.gov/about`, 2019. accessed on April.

[4] Data from NASA's Missions, Research, and Activities. `http://www.nasa.gov/open/data.html`, 2019. accessed on April.

[5] European XFEL: The Data Challenge. `http://www.xfel.eu/news/2012/the_data_challenge`, September 2012.

[6] `http://hadoop.apache.org`, 2019. accessed on April.

[7] `http://spark.apache.org`, 2019. accessed on April.

[8] Randall T. Whitman, Michael B. Park, Sarah M. Ambrose, and Erik G. Hoel. Spatial indexing and analytics on hadoop. In *SIGSPATIAL*, pages 73–82. ACM, 2014.

[9] Jiamin Lu and Ralf Hartmut Güting. Parallel secondo: Boosting database engines with hadoop. In *ICPADS*, pages 738–743. IEEE Computer Society, 2012.

[10] Shoji Nishimura, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. $\mathcal{MD}$-HBase: Design and Implementation of an Elastic Data Infrastructure for Cloud-scale Location Services. *DAPD*, 31(2):289–319, 2013.

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

[12] Ameet Kini and Rob Emanuele. Geotrellis: Adding Geospatial Capabilities to Spark. `http://spark-summit.org/2014/talk/geotrellis-adding-geospatial-capabilities-to-spa` 2014.

[13] Jia Yu, Jinxuan Wu, and Mohamed Sarwat. Geospark: a cluster computing framework for processing large-scale spatial data. In *SIGSPATIAL*, pages 70:1–70:4. ACM, 2015.

[14] Ahmed Eldawy and Mohamed F. Mokbel. Spatialhadoop: A mapreduce framework for spatial data. In *ICDE*, pages 1352–1363. IEEE Computer Society, 2015.

[15] Qiang Ma, Bin Yang, Weining Qian, and Aoying Zhou. Query processing of massive trajectory data based on mapreduce. In *CLOUDDB*, pages 9–16. ACM, 2009.

[16] Haoyu Tan, Wuman Luo, and Lionel M. Ni. Clost: a hadoop-based storage system for big spatio-temporal data analytics. In *CIKM*, pages 2139–2143. ACM, 2012.

[17] Zhenlong Li, Fei Hu, John L. Schnase, Daniel Q. Duffy, Tsengdar Lee, Michael K. Bowen, and Chaowei Yang. A spatiotemporal indexing approach for efficient processing of big array-based climate data with mapreduce. *International Journal of Geographical Information Science*, 31(1), 2017.

[18] Ahmed Eldawy, Mohamed F. Mokbel, Saif Al-Harthi, Abdulhadi Alzaidy, Kareem Tarek, and Sohaib Ghani. SHAHED: A mapreduce-based system for querying and visualizing spatio-temporal satellite data. In *ICDE*, pages 1585–1596. IEEE Computer Society, 2015.

[19] ST-Hadoop website. `http://st-hadoop.cs.umn.edu/`.

[20] Ahmed Eldawy and Mohamed F. Mokbel. Pigeon: A spatial mapreduce language. In *ICDE*, pages 1242–1245. IEEE Computer Society, 2014.

[21] Louai Alarabi. St-hadoop: A mapreduce framework for big spatio-temporal data. In *SIGMOD*, pages 40–42. ACM, 2017.

[22] Louai Alarabi, Mohamed F. Mokbel, and Mashaal Musleh. St-hadoop: A mapreduce framework for spatio-temporal data. In *SSTD*, pages 84–104. Springer, 2017.

[23] Louai Alarabi, Mohamed F. Mokbel, and Mashaal Musleh. ST-Hadoop: A MapReduce Framework for Spatio-temporal Data. *GeoInformatica*, 22(4):785–813, 2018.

[24] Louai Alarabi and Mohamed F. Mokbel. A demonstration of st-hadoop: A mapreduce framework for big spatio-temporal data. *PVLDB*, 10(12):1961–1964, 2017.

[25] Louai Alarabi. Summit: a scalable system for massive trajectory data management. In *SIGSPATIAL*, pages 612–613. ACM, 2018.

[26] Louai Alarabi. Summit: a scalable system for massive trajectory data management. *SIGSPATIAL Special*, 10(3):2–3, 2018.

[27] `https://flink.apache.org/`, 2019. accessed on April.

[28] Guozhang Wang, Marcos Antonio Vaz Salles, Benjamin Sowell, Xun Wang, Tuan Cao, Alan J. Demers, Johannes Gehrke, and Walker M. White. Behavioral simulations in mapreduce. *PVLDB*, 3(1):952–963, 2010.

[29] Gary Planthaber, Michael Stonebraker, and James Frew. Earthdb: scalable analysis of MODIS data using scidb. In Varun Chandola, Ranga Raju Vatsavai, and Chetan Gupta, editors, *Proceedings of the 1st ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data, BigSpatial@SIGSPATIAL 2012, Redondo Beach, CA, USA, November 6, 2012*, pages 11–19. ACM, 2012.

[30] Michael Stonebraker, Paul Brown, Donghui Zhang, and Jacek Becla. SciDB: A Database Management System for Applications with Complex Analytics. *Computing in Science and Engineering*, 15(3):54–62, 2013.

[31] GIS Tools for Hadoop. `http://esri.github.io/gis-tools-for-hadoop/`, February 2014.

[32] Peng Lu, Gang Chen, Beng Chin Ooi, Hoang Tam Vo, and Sai Wu. ScalaGiST: Scalable Generalized Search Trees for MapReduce Systems. *PVLDB*, 7(14):1797–1808, 2014.

[33] Ahmed Eldawy, Ibrahim Sabek, Mostafa Elganainy, Ammar Bakeer, Ahmed Abdelmotaleb, and Mohamed F. Mokbel. Sphinx: Empowering impala for efficient execution of SQL queries on big spatial data. In *SSTD*, volume 10411 of *Lecture Notes in Computer Science*, pages 65–83. Springer, 2017.

[34] Jia Yu, Raha Moraffah, and Mohamed Sarwat. Hippo in action: Scalable indexing of a billion new york city taxi trips and beyond. In *ICDE*, pages 1413–1414. IEEE Computer Society, 2017.

[35] Khaled Mohammed Al-Naami, Sadi Evren Seker, and Latifur Khan. GISQF: An Efficient Spatial Query Processing System. In *CLOUDCOM*, pages 681–688. IEEE Computer Society, 2014.

[36] Philippe Cudré-Mauroux, Eugene Wu, and Samuel Madden. Trajstore: An adaptive storage system for very large trajectory data sets. In *ICDE*, pages 109–120. IEEE Computer Society, 2010.

[37] Xike Xie, Benjin Mei, Jinchuan Chen, Xiaoyong Du, and Christian S. Jensen. Elite: an elastic infrastructure for big spatiotemporal trajectories. *VLDBJ*, 25(4):473–493, 2016.

[38] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak R. Borkar, Yingyi Bu, Michael J. Carey, Inci Cetindil, Madhusudan Cheelangi, Khurram Faraaz, Eugenia Gabrielova, Raman Grover, Zachary Heilbron, Young-Seok Kim, Chen Li, Guangqiang Li, Ji Mahn Ok, Nicola Onose, Pouria Pirzadeh, Vassilis J. Tsotras, Rares Vernica, Jian Wen, and Till Westmann. Asterixdb: A scalable, open source BDMS. *PVLDB*, 7(14):1905–1916, 2014.

[39] Sattam Alsubaiee, Alexander Behm, Vinayak R. Borkar, Zachary Heilbron, Young-Seok Kim, Michael J. Carey, Markus Dreseler, and Chen Li. Storage management in asterixdb. *PVLDB*, 7(10):841–852, 2014.

[40] Laipeng Han, Lan Huang, Xueyi Yang, Wei Pang, and Kangping Wang. A novel spatio-temporal data storage and index method for arm-based hadoop server. In Xingming Sun, Alex X. Liu, Han-Chieh Chao, and Elisa Bertino, editors, *Cloud Computing and Security - Second International Conference, ICCCS 2016, Nanjing, China, July 29-31, 2016, Revised Selected Papers, Part I*, volume 10039 of *Lecture Notes in Computer Science*, pages 206–216, 2016.

[41] Anthony D. Fox, Christopher N. Eichelberger, James N. Hughes, and Skylar Lyon. Spatio-temporal indexing in non-relational distributed databases. In *BIGDATA*, pages 291–299. IEEE Computer Society, 2013.

[42] Michael A. Whitby, Rich Fecher, and Chris Bennight. Geowave: Utilizing distributed key-value stores for multidimensional data. In *SSTD*, pages 105–122. Springer, 2017.

[43] Haozhou Wang, Kai Zheng, Jiajie Xu, Bolong Zheng, Xiaofang Zhou, and Shazia Wasim Sadiq. Sharkdb: An in-memory column-oriented trajectory storage. In *CIKM*, pages 1409–1418. ACM, 2014.

[44] Viorica Botea, Daniel Mallett, Mario A. Nascimento, and Jörg Sander. PIST: an efficient and practical indexing technique for historical spatio-temporal point data. *GeoInformatica*, 12(2):143–168, 2008.

[45] Zeyuan Shang, Guoliang Li, and Zhifeng Bao. DITA: A distributed in-memory trajectory analytics system. In *SIGMOD*, pages 1681–1684. ACM, 2018.

[46] Randall T. Whitman, Michael B. Park, Bryan G. Marsh, and Erik G. Hoel. Spatio-temporal join on apache spark. In *SIGSPATIAL*, pages 20:1–20:10. ACM, 2017.

[47] Yixiang Fang, Reynold Cheng, Wenbin Tang, Silviu Maniu, and Xuan S. Yang. Scalable algorithms for nearest-neighbor joins on big trajectory data. *TKDE*, 28(3):785–800, 2016.

[48] Wook-Shin Han, Jaehwa Kim, Byung Suk Lee, Yufei Tao, Ralf Rantzau, and Volker Markl. Cost-based predictive spatiotemporal join. *TKDE*, 21(2):220–233, 2009.

[49] Yifeng Geng, Xiaomeng Huang, Meiqi Zhu, Huabin Ruan, and Guangwen Yang. Scihive: Array-based query processing with hiveql. In *12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2013 / 11th IEEE International Symposium on Parallel and Distributed Processing with Applications, ISPA-13 / 12th IEEE International Conference on Ubiquitous Computing and Communications, IUCC-2013, Melbourne, Australia, July 16-18, 2013*, pages 887–894. IEEE Computer Society, 2013.

[50] Yifeng Geng, Xiaomeng Huang, and Guangwen Yang. Adaptive indexing for distributed array processing. In *IEEE International Congress on Big Data,*, pages 331–338. IEEE Computer Society, 2014.

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

[52] Jie Bao, Ruiyuan Li, Xiuwen Yi, and Yu Zheng. Managing massive trajectories on the cloud. In *SIGSPATIAL*, pages 41:1–41:10. ACM, 2016.

[53] Microsoft. Azure. `https://azure.microsoft.com/`, 2019. accessed on April.

[54] Zdravko Galic, Emir Meskovic, and Dario Osmanovic. Distributed processing of big mobility data as spatio-temporal data streams. *GeoInformatica*, 21(2):263–291, 2017.

[55] Ziqiang Yu, Yang Liu, Xiaohui Yu, and Ken Q. Pu. Scalable distributed processing of K nearest neighbor queries over moving objects. *TKDE*, 27(5):1383–1396, 2015.

[56] Ruichu Cai, Zijie Lu, Li Wang, Zhenjie Zhang, Tom Z. J. Fu, and Marianne Winslett. DITIR: distributed index for high throughput trajectory insertion and real-time temporal range query. *PVLDB*, 10(12):1865–1868, 2017.

[57] Hongzhi Wang and Amina Belhassena. Parallel trajectory search based on distributed index. *Inf. Sci.*, 388:62–83, 2017.

[58] Amina Belhassena and Hongzhi Wang. Distributed skyline trajectory query processing. In John C. S. Lui, Xinbing Wang, Alexander Wolf, Yunhao Liu, and

Chuanping Hu, editors, *Proceedings of the ACM Turing 50th Celebration Conference - China, TUR-C 2017, Shanghai, China, May 12-14, 2017*, pages 19:1–19:7. ACM, 2017.

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

[60] Amr Magdy, Louai Alarabi, Saif Al-Harthi, Mashaal Musleh, Thanaa M. Ghanem, Sohaib Ghani, and Mohamed F. Mokbel. Taghreed: a system for querying, analyzing, and visualizing geotagged microblogs. In *SIGSPATIAL*, pages 163–172. ACM, 2014.

[61] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[62] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In Steven D. Gribble and Dina Katabi, editors, *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, pages 15–28. USENIX Association, 2012.

[63] Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. Graphx: a resilient distributed graph system on spark. In Peter A. Boncz and Thomas Neumann, editors, *First International Workshop on Graph Data Management Experiences and Systems, GRADES 2013, co-loated with SIGMOD/PODS 2013, New York, NY, USA, June 24, 2013*, page 2. CWI/ACM, 2013.

[64] Matthias Boehm, Michael Dusenberry, Deron Eriksson, Alexandre V. Evfimievski, Faraz Makari Manshadi, Niketan Pansare, Berthold Reinwald, Frederick Reiss, Prithviraj Sen, Arvind Surve, and Shirish Tatikonda. Systemml: Declarative machine learning on spark. *PVLDB*, 9(13):1425–1436, 2016.

[65] Zhou Huang, Yiran Chen, Lin Wan, and Xia Peng. Geospark SQL: an effective framework enabling spatial queries on spark. *ISPRS*, 6(9):285, 2017.

[66] `http://storm.apache.org`, 2019. accessed on April.

[67] `https://accumulo.apache.org/`, 2019. accessed on April.

[68] `http://hbase.apache.org/`, 2019. accessed on April.

[69] `https://aws.amazon.com/dynamodb`, 2019. accessed on April.

[70] `http://cassandra.apache.org`, 2019. accessed on April.

[71] `https://asterixdb.apache.org/`, 2019. accessed on April.

[72] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, 1996.

[73] `https://ngageoint.github.io/geowave/`, 2019. accessed on April.

[74] Xiangyu Zhang, Jing Ai, Zhongyuan Wang, Jiaheng Lu, and Xiaofeng Meng. An efficient multi-dimensional index for cloud data management. In *CLOUDDB*, pages 17–24. ACM, 2009.

[75] Open Geospatial Consortium. `http://www.opengeospatial.org/`, 2019. accessed on April.

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

[77] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD*, pages 165–178. ACM, 2009.

[78] Shubin Zhang, Jizhong Han, Zhiyong Liu, Kai Wang, and Shengzhong Feng. Spatial queries evaluation with mapreduce. In *GCC*, pages 287–292. IEEE Computer Society, 2009.

[79] Takuya Yokoyama, Yoshiharu Ishikawa, and Yu Suzuki. Processing all k-nearest neighbor queries in hadoop. In *WAIM*, pages 346–351. Springer, 2012.

[80] Yunqin Zhong, Xiaomin Zhu, and Jinyun Fang. Elastic and effective spatio-temporal query processing scheme on hadoop. In *BIGSPATIAL*, pages 33–42. ACM, 2012.

[81] Ming-Ling Lo and Chinya V. Ravishankar. Spatial hash-joins. In *SIGMOD*, pages 247–258. ACM Press, 1996.

[82] Sloan Digital Sky Survey. `http://www.sdss.org/dr14/data_access/volume/`, 2019. accessed on April.

[83] Wikelski, M., and Kays, R. Movebank:s archive, analysis and sharing of animal movement data. Hosted by the Max Planck Institute for Ornithology. `www.movebank.org`, 2019. accessed on April.

[84] The National Hurricane Center. `www.nhc.noaa.gov`, 2019. accessed on April.

[85] Sijie Ruan, Ruiyuan Li, Jie Bao, Tianfu He, and Yu Zheng. Cloudtp: A cloud-based flexible trajectory preprocessing framework. In *ICDE*, pages 1601–1604. IEEE Computer Society, 2018.

[86] `https://github.com/uber/marmaray`, 2019. accessed on April.

[87] `http://sortbenchmark.org`, 2019. accessed on April.

[88] Amol Ghoting, Rajasekar Krishnamurthy, Edwin P. D. Pednault, Berthold Reinwald, Vikas Sindhwani, Shirish Tatikonda, Yuanyuan Tian, and Shivakumar Vaithyanathan. SystemML: Declarative Machine Learning on MapReduce. In *ICDE*, pages 231–242. IEEE Computer Society, 2011.

[89] `https://kafka.apache.org`, 2019. accessed on April.

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

[91] Dong Xie, Feifei Li, Bin Yao, Gefei Li, Liang Zhou, and Minyi Guo. Simba: Efficient in-memory spatial analytics. In *SIGMOD*, pages 1071–1085. ACM, 2016.

[92] Sheng Huang, Yaoliang Chen, Xiaoyan Chen, Kai Liu, Xiaomin Xu, Chen Wang, Kevin Brown, and Inge Halilovic. The next generation operational data historian for iot based on informix. In *SIGMOD*, pages 169–176. ACM, 2014.

[93] Peilin Yang, Srikanth Thiagarajan, and Jimmy Lin. Robust, scalable, real-time event time series aggregation at twitter. In *SIGMOD*, pages 595–599. ACM, 2018.

[94] OpenTSDB. `http://opentsdb.net`, 2019. accessed on April.

[95] Sean Rhea, Eric Wang, Edmund Wong, Ethan Atkins, and Nat Storer. Littletable: A time-series database and its uses. In *SIGMOD*, pages 125–138. ACM, 2017.

[96] Lu Qin, Jeffrey Xu Yu, Lijun Chang, Hong Cheng, Chengqi Zhang, and Xuemin Lin. Scalable big graph processing in mapreduce. In *SIGMOD*, pages 827–838. ACM, 2014.

[97] Wenqing Lin, Xiaokui Xiao, and Gabriel Ghinita. Large-scale frequent subgraph mining in mapreduce. In *ICDE*, pages 844–855. IEEE Computer Society, 2014.

[98] Jun Gao, Jiashuai Zhou, Chang Zhou, and Jeffrey Xu Yu. Glog: A high level graph analysis system using mapreduce. In *ICDE*, pages 544–555. IEEE Computer Society, 2014.

[99] `https://janusgraph.org`, 2019. accessed on April.

[100] `http://titan.thinkaurelius.com`, 2019. accessed on April.

[101] Byoung-Kee Yi, H. V. Jagadish, and Christos Faloutsos. Efficient retrieval of similar time sequences under time warping. In *ICDE*, pages 201–208. IEEE Computer Society, 1998.

[102] Christos Faloutsos, M. Ranganathan, and Yannis Manolopoulos. Fast subsequence matching in time-series databases. In *SIGMOD*, pages 419–429. ACM Press, 1994.

[103] Lei Chen, M. Tamer Özsu, and Vincent Oria. Robust and fast similarity search for moving object trajectories. In *SIGMOD*, pages 491–502. ACM, 2005.

[104] Lei Chen and Raymond T. Ng. On the marriage of lp-norms and edit distance. In *VLDB*, pages 792–803. Morgan Kaufmann, 2004.

[105] Elias Frentzos, Kostas Gratsias, and Yannis Theodoridis. Index-based most similar trajectory search. In *ICDE*, pages 816–825. IEEE Computer Society, 2007.

[106] Michail Vlachos, Dimitrios Gunopulos, and George Kollios. Discovering similar multidimensional trajectories. In *ICDE*, pages 673–684. IEEE Computer Society, 2002.

[107] Kevin Toohey and Matt Duckham. Trajectory similarity measures. *SIGSPATIAL Special*, 7(1):43–50, 2015.

[108] Helmut Alt and Michael Godau. Computing the fréchet distance between two polygonal curves. *IJCGA*, 5:75–91, 1995.

[109] Joachim Gudmundsson, Patrick Laube, and Thomas Wolle. Computational movement analysis. In Barbara Frank-Job, Alexander Mehler, and Tilmann Sutter, editors, *Die Dynamik sozialer und sprachlicher Netzwerke, Konzepte, Methoden und empirische Untersuchungen an Beispielen des WWW*, pages 423–438. Springer, 2012.

[110] Lawrence R. Rabiner and Biing-Hwang Juang. *Fundamentals of speech recognition*. Prentice Hall signal processing series. Prentice Hall, 1993.

[111] Yutaka Yanagisawa, Jun-ichi Akahani, and Tetsuji Satoh. Shape-based similarity query for trajectory of mobile objects. In *MDM*, volume 2574, pages 63–77. Springer, 2003.

[112] Jung-Rae Hwang, Hye-Young Kang, and Ki-Joune Li. Searching for similar trajectories on road networks using spatio-temporal similarity. In *ADBIS*, pages 282–295. Springer, 2006.

[113] Hui Ding, Goce Trajcevski, Peter Scheuermann, Xiaoyue Wang, and Eamonn J. Keogh. Querying and mining of time series data: experimental comparison of representations and distance measures. *PVLDB*, 1(2):1542–1552, 2008.

[114] Rares Vernica, Michael J. Carey, and Chen Li. Efficient parallel set-similarity joins using mapreduce. In *SIGMOD*, pages 495–506. ACM, 2010.

[115] Han Su, Kai Zheng, Haozhou Wang, Jiamin Huang, and Xiaofang Zhou. Calibrating trajectory data for similarity-based analysis. In *SIGMOD*, pages 833–844. ACM, 2013.

[116] Michael R. Evans, Dev Oliver, Shashi Shekhar, and Francis Harvey. Fast and exact network trajectory similarity computation: a case-study on bicycle corridor planning. In *SIGKDD,Computing, UrbComp@KDD*, pages 9:1–9:8. ACM, 2013.

[117] Haozhou Wang, Han Su, Kai Zheng, Shazia Wasim Sadiq, and Xiaofang Zhou. An effectiveness study on trajectory similarity measures. In *ADC*, pages 13–22. Australian Computer Society, 2013.

[118] Xiaoyue Wang, Abdullah Mueen, Hui Ding, Goce Trajcevski, Peter Scheuermann, and Eamonn J. Keogh. Experimental comparison of representation methods and distance measures for time series data. *Data Min. Knowl. Discov.*, 26(2):275–309, 2013.

[119] Yu Zheng. Trajectory data mining: An overview. *ACM TIST*, 6(3):29:1–29:41, 2015.

[120] Hoyoung Jeung, Man Lung Yiu, and Christian S. Jensen. Trajectory pattern mining. In *Computing with Spatial Trajectories*, pages 143–177. Springer, 2011.

[121] Fosca Giannotti, Mirco Nanni, Dino Pedreschi, Fabio Pinelli, Chiara Renso, Salvatore Rinzivillo, and Roberto Trasarti. Unveiling the complexity of human mobility by querying and mining massive trajectory data. *VLDBJ*, 20(5):695–719, 2011.

[122] Mark D. Rintoul and Andrew T. Wilson. Trajectory analysis via a geometric feature space approach. *Statistical Analysis and Data Mining*, 8(5-6):287–301, 2015.

[123] Chih-Chieh Hung, Wen-Chih Peng, and Wang-Chien Lee. Clustering and aggregating clues of trajectories for mining trajectory patterns and routes. *VLDBJ*, 24(2):169–192, 2015.

[124] Guojun Wu, Yichen Ding, Yanhua Li, Jie Bao, Yu Zheng, and Jun Luo. Mining

spatio-temporal reachable regions over massive trajectory data. In *ICDE*, pages 1283–1294. IEEE Computer Society, 2017.

[125] Louai Alarabi, Ahmed Eldawy, Rami Alghamdi, and Mohamed F. Mokbel. TAREEG: a mapreduce-based system for extracting spatial data from openstreetmap. In *SIGSPATIAL*, pages 83–92. ACM, 2014.

[126] Louai Alarabi, Ahmed Eldawy, Rami Alghamdi, and Mohamed F. Mokbel. TAREEG: a mapreduce-based web service for extracting spatial data from openstreetmap. In *SIGMOD*, pages 897–900. ACM, 2014.

[127] Martin Erwig and Markus Schneider. Spatio-temporal predicates. *TKDE*, 14(4):881–901, 2002.