Distributed Spatiotemporal Indexes for Querying and Mining in Key-Value Stores

Hong Van Le

Doctor of Philosophy

Department of Informatics
School of Multidisciplinary Sciences
The Graduate University for Advanced Studies,
SOKENDAI

Distributed Spatiotemporal Indexes for Querying and Mining in Key-Value Stores

Hong Van Le

Department of Informatics
School of Multidisciplinary Sciences
SOKENDAI (The Graduate University for Advanced Studies)

This dissertation is submitted
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

March 2019

Committee

Advisor	Dr. Atsuhiro TAKASU Professor of National Institute of Informatics/SOKENDAI
Subadvisor	Dr. Kenro AIHARA Associate Professor of National Institute of Informatics/SOKENDAI
Subadvisor	Dr. Keizo OYAMA Professor of National Institute of Informatics/SOKENDAI
Examiner	Dr. Norio KATAYAMA Associate Professor of National Institute of Informatics/SOKENDAI
Examiner	Takahiro HARA Professor of Osaka University

Acknowledgements

I would like to thank the people who helped and supported me during my Ph.D. studies.

Firstly, I would like to express my sincere gratitude to my advisor Professor Takasu Atsuhiro for the continuous support of my Ph.D study and related research, for his patience, motivation, and immense knowledge.

I want to thank all the members and internship students in Takasu Laboratory, especially Takenaka Akiko, for providing a motivating, fun, and positive working environment. Also, thanks to the people in NII for all the support and fruitful discussions about both research and life in Japan.

Last but not least, thanks to my understanding, supportive, and patient husband Lam Le.

Abstract

The recent ubiquity of sensors and GPS-enabled devices has resulted in an explosion of spatiotemporal data generated from probe cars, traffic sensors, and smartphones. To benefit from such data, applications need data storage that can handle the massive volume of data and support high-computational queries. Although key-value store databases (KVSs) efficiently handle large-scale data, they are not equipped with effective functions for supporting geographical data. To solve this problem, we present G$^+$-HBase, a high-performance spatiotemporal database based on HBase, a standard KVS.

First, we present G-HBase, a geographical database based on HBase. To index geographic data, we use Geohash as the rowkey in KVSs. Then, we propose a novel partitioning method, namely binary Geohash rectangle partitioning, to support spatial queries. Our extensive experiments on real datasets have demonstrated improved performance with k nearest neighbors and range query in G-HBase when compared with SpatialHadoop, a state-of-the-art framework with native support for spatial data. We also observed that performance of the spatial join query in G-HBase was on par with SpatialHadoop and outperforms SJMR algorithm in HBase.

Second, we extend G-HBase to G$^+$-HBase to support spatiotemporal data in an intelligent transportation system. In the spatiotemporal index, we adopted STCode, a longitude, latitude, and time-encoding algorithm, to build an index on top of HBase. Our proposed index structure allows continuous updates of objects and provides an efficient prefix filter for supporting spatiotemporal data retrieval. Experimental results demonstrate the high performance of spatiotemporal queries with response time meeting the requirements of real-time query-processing systems.

Finally, we further extend G$^+$-HBase to deal with user-generated geo-tagged social data. We study an efficient multidimensional index structure and parallel processing approaches for the top-k frequent spatiotemporal terms query, a basic analytic query on geo-tagged social data. Given a spatiotemporal range, the query aggregates frequencies of terms among the social posts in that range to find the most frequent terms. In order to reduce storage for indexing and to improve the query performance, we propose a distributed index structure

which transforms spatiotemporal coordinates into unique codes to generate rowkeys in KVSs and balances the data distribution across clusters. Then, we utilize data localization by calculating sorted term lists (STLs) inside storage servers in parallel. To reduce input/output between storage servers and the client, we theoretically estimate the necessary length of STLs to calculate top k frequent terms and send only a part of STLs to the client. From several experiments on real datasets, we observed lower space requirements but better query performance of our approach when compared with baseline approaches.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

1.1 Motivation

Recent and rapid improvement of positioning technologies such as satellites, the Global Positioning System (GPS), sensors, and wireless networks has resulted in an explosion of data generated by geographical objects. For instance, traffic management systems in busy cities such as Tokyo have been collecting a large number of location updates from probe cars such as taxis, buses, collaborating private cars, and GPS-enabled devices at a rate of multiple updates per minute from each vehicle. The collected data often include location data (longitude and latitude) and time data (a timestamp).

Many systems, in both scientific research and daily life, have taken advantage of the data collected from geographical objects. For example, location-based recommendation systems are exploiting the massive volume of geotagged social data from online social media to provide efficient advertisements and coupon distribution based on the current location of users. People can take advantage of these systems by continuously sending data about their current location and time, then receiving promotional discount information from a nearby restaurant or relevant service recommendation at the current area. Such systems need a database of spatiotemporal data, that can store a massive volume of data and support high-performance spatiotemporal queries.

There are three main challenges for spatiotemporal databases to be useful for these systems. The first challenge is the volume of data. Thanks to the ubiquity of location-aware devices, millions of location updates can be easily collected in a very short time, so the database needs to have high scalability, fault-tolerance, and availability while dealing with large volume of collected data. Second, high computational complexity is also a challenging problem. A candidate database would need to support many queries which involve geometric computations such as logical operations on spatial relationships. Those queries are highly

expensive and continuously changing. The last challenge is that high-performance analysis is required. The system has to guarantee satisfactory performance on queries, but when dataset become bigger, query time can dramatically increase.

Recently, MapReduce [26] based systems such as Hadoop have emerged as an effective solution for handling massive data. Although some systems based on Hadoop such as SpatialHadoop [32] and Hadoop-GIS [3] have been introduced as spatial data management systems that have scalability and efficient spatial query support, these systems lack support for temporal data and still have high latency compared with a low-latency system. Key–value stores (KVSs) such as HBase, with their scalability, fault tolerance, availability, and random real-time read/write capability, have shown promise. However, they do not have native support for spatial and spatiotemporal queries, which is required in a database of geographical data. In this thesis, we introduce a distributed framework, namely G$^+$-HBase, to index large-scale spatiotemporal data and provide various spatial and spatiotemporal queries on that data.

1.2 G$^+$-HBase System Overview

G$^+$-HBase is based on HBase. Figure 1.1 gives an overview of G$^+$-HBase architecture. G$^+$-HBase includes two components, namely, *Index Structure* and *Query Processing* components. The *Index Structure* component includes the key design and the data partition method. Input data including points and non-point objects are inserted using key design algorithms to transform multidimensional data into one-dimensional keys in KVSs. After data insertion, the table is partitioned based on the data density using the data partition methods.

G$^+$-HBase supports various spatial and spatiotemporal queries. Besides basic queries such as range query, k nearest neighbors query, G$^+$-HBase also supports spatial join query and analytic query on geo-tagged social data like top k frequent spatiotemporal terms query. Depending on the type of the query, the *Query Processing* component will design the query algorithms that utilize the *Index Structure* component to achieve the best performance. Because G$^+$-HBase is a distributed system, in many queries we design parallel processing algorithms based on the data partition method in the *Index Structure* component.

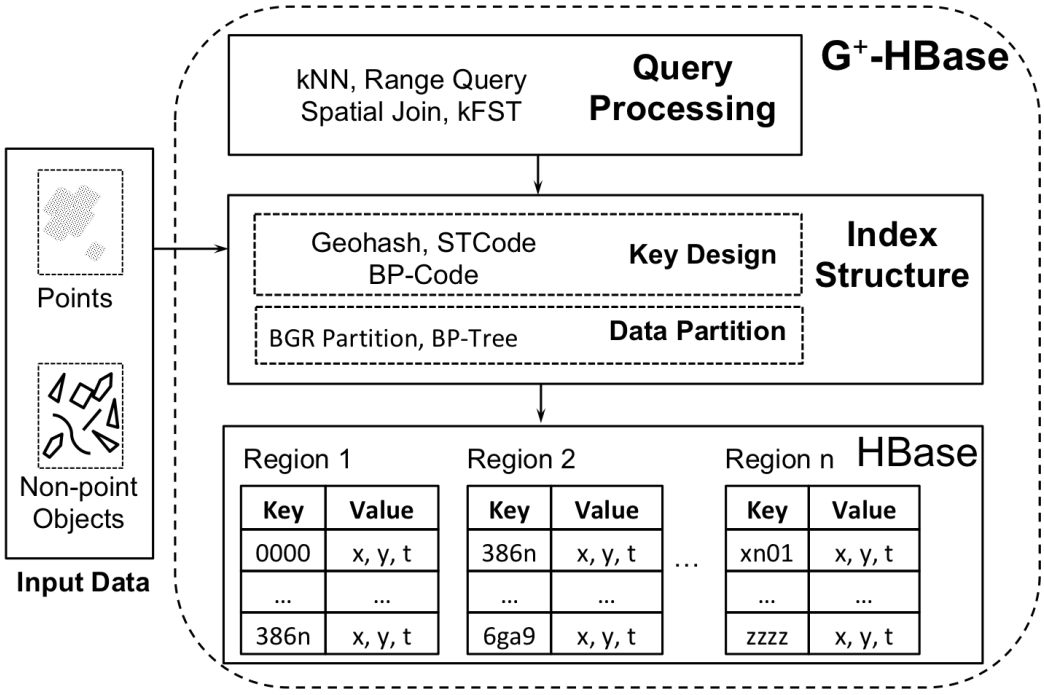


Figure 1.1: Architecture Overview of G+-HBase

1.3 Contribution

1.3.1 G-HBase: Geographical HBase

First, we propose a spatial index structure as described in Chapter 4. Some methods [63, 89] use Geohash [1], a linearization technique that transforms two-dimensional spatial data into a one-dimensional hashcode, to handle spatial data in HBase. However, they do not address edge cases and the Z-order limitations of Geohash, which can lead to incorrect results and can decrease the performance on spatial queries. In this thesis, we tackle the limitations of Geohash with spatial data by proposing an efficient distributed spatial index, which uses Geohash and the R-Tree [42] with HBase. We use Geohash as the key for key–value pairs for HBase, then further utilize the R-Tree, a multidimensional index structure for geographical data, to generate correct results and improve query performance. To bridge Geohash and the R-Tree, we propose a novel data structure, the binary Geohash rectangle-partition tree (BGRP Tree). By using the BGRP Tree, we partition a Geohash range into nonoverlapping subrectangles, then insert these subrectangles into the R-Tree.

Using Geohash as the key for HBase enables geographical databases to achieve a high insert throughput because Geohash encoding is not computation intensive. The R-Tree index

[1] http://geohash.org

helps improve the performance and the accuracy of spatial queries. Our experimental results indicate that querying that can take advantage of the Geohash and R-Tree index outperformed both querying that used only Geohash on HBase and querying in a MapReduce-based system. We also observed that our proposed index could process high-performance queries, with response times of less than one second.

1.3.2 G-HBase Extension: A Scalable Spatiotemporal Data Storage for Intelligent Transportation Systems

In the next step, we present a lightweight spatiotemporal index based on STCode [51], a hierarchical text-encoding algorithm aimed at overcoming the limitations of key-value stores in supporting spatiotemporal data in Chapter 5. There are several advantages for processing spatiotemporal queries using our index structure. First, our proposed index structure can leverage the lexicographical order of rowkey on key-value stores to store objects that are close to each other spatially and temporally at a nearby place in the database. Second, it utilizes a *column family* for efficient data distribution across a cluster without any additional internal modification requirements. Third, it provides a simple, but efficient, prefix filter for scanning relevant objects. Last but not least, because STCode encoding is not as computationally expensive as other tree-structured indexes, the proposed index structure could support continuous updates of spatio-temporal objects. We also demonstrate how spatio-temporal queries, such as k nearest neighbors or range queries, exploit the proposed index to enhance query performance.

1.3.3 G-HBase Extension: Efficient Distributed Spatiotemporal Index for Parallel Top-k Frequent Terms Query

Finally, in Chapter 6, we propose a distributed index structure for geotagged social data. Several online social media, such as Twitter, Instagram, Foursquare and Facebook, allow users to geotag their social posts. This creates novel data analytics problems, such as detecting popular topic trends or popular sites, most frequent trajectories, etc. The obtained data are not only spatial and temporal information but also text in social posts. To minimize storage requirement and to support spatiotemporal queries efficiently, we propose a spatiotemporal index which transforms spatial and temporal coordinates into unique codes, to generate rowkeys in KVSs. MD-HBase [85] couples Z-ordering with trie-based KD-Tree and Quadtree to partition and index multi-dimensional data in HBase. Even though the trie-based trees can capture data distribution statistics, they still cause unbalanced partitioning and even empty

partitions in some cases, leading to unbalanced parallel processing when querying. In our proposed spatiotemporal index, we first build a Balanced Partition Tree (BP-Tree) to balance the data partition across distributed clusters. Then we propose a Balanced Partition Code (BP-Code) based on BP-Tree to spatially and temporally generate rowkeys for spatiotemporal data on KVSs.

We also studied parallel algorithms to process top-k frequent spatiotemporal terms (kFST) query [2] based on the distributed spatiotemporal index. As an example, the user may want to know which terms have been popular around a specific location over the past week, and thus his query specifies a spatial circle with a radius one kilometer around the location and a temporal interval of a week. The kFST query problem is different from recent works on the intersection of keyword search and spatial querying. We apply data localization by bringing STL computation into storage servers to process in parallel. To further improve the query performance, when querying, we do on-the-fly estimation of the necessary STLs length and send only a part of STLs to the client, reducing I/O between storage servers and the client.

We experimentally evaluated the proposed parallel kFST queries by using large datasets of real Twitter data with various query selectivities and k values. Experimental evidence shows the improvement in performance of the parallel query with shortened length is compared with the parallel query with full length and the nonparallel query. We observed lower space requirements but better query performance of our approach when compared with the approach in [2]. Moreover, because we do on-the-fly estimation, our proposed query is more flexible with various query inputs in experimental results.

1.4 Thesis Organization

The remaining chapters are organized as follows. Chapter 2 introduces the background of KVSs and spatial access methods. Chapter 3 surveys the past studies of spatial and spatiotemporal extension in Big Data frameworks. Chapter 4 then presents G-HBase in detail. Chapter 5 describes the extension of G-HBase to support spatiotemporal data for intelligent transportation systems. Chapter 6 presents another extension of G-HBase to index geo-tagged social data and provide efficient parallel top-k frequent spatiotemporal terms computing. Finally, Chapter 7 summarizes the thesis.

Chapter 2

Background

2.1 Key-Value Stores

Recently, traditional Relational Database Management Systems had to face the problem of flexible scaling required for storing large-scale datasets [17] because the costs for scaling relational database are very high. As a result, a number of new systems have been proposed to scale to a large number of nodes while guaranteeing the quality of the services they deliver with a low budget. Many of the new systems are referred to as NoSQL data stores such as Google's BigTable [18], Amazon's Dynamo [27]. BigTable is a distributed storage system that is designed to scale to very large size: petabytes of data across thousands of commodity servers, which is useful for many systems with thousands or millions of users doing updates as well as reads. Therefore, many systems have been modeled after BigTable such as Apache HBase [41, 28], Cassandra [57], Accumulo [96], and Amazon Simple Storage [106]. In this work, we did experiments on HBase, the faithful, open source implementation of Google's Bigtable.

HBase is a distributed scalable database that takes advantage of the HDFS [99]. Tables in HBase include rows and columns like other databases, but they can scale to a large number of rows and columns. To store such large tables into a distributed cluster, tables are split into a number of smaller chunks, namely regions. Regions are stored in servers called RegionServers. Figure 2.1 describes in detail the architecture of HBase.

HBase uses Apache Zookeeper [48, 53] for distributed management. List of all regions in the system is kept in a table called hbase:meta. When a client wants to access a particular row, it will first ask Zookeeper the location of hbase:meta table and Zookeeper get that information from HMaster (see Figure 2.1). After accessing to hbase:meta table, the client can determine which RegionServer is hosting the region in the query.

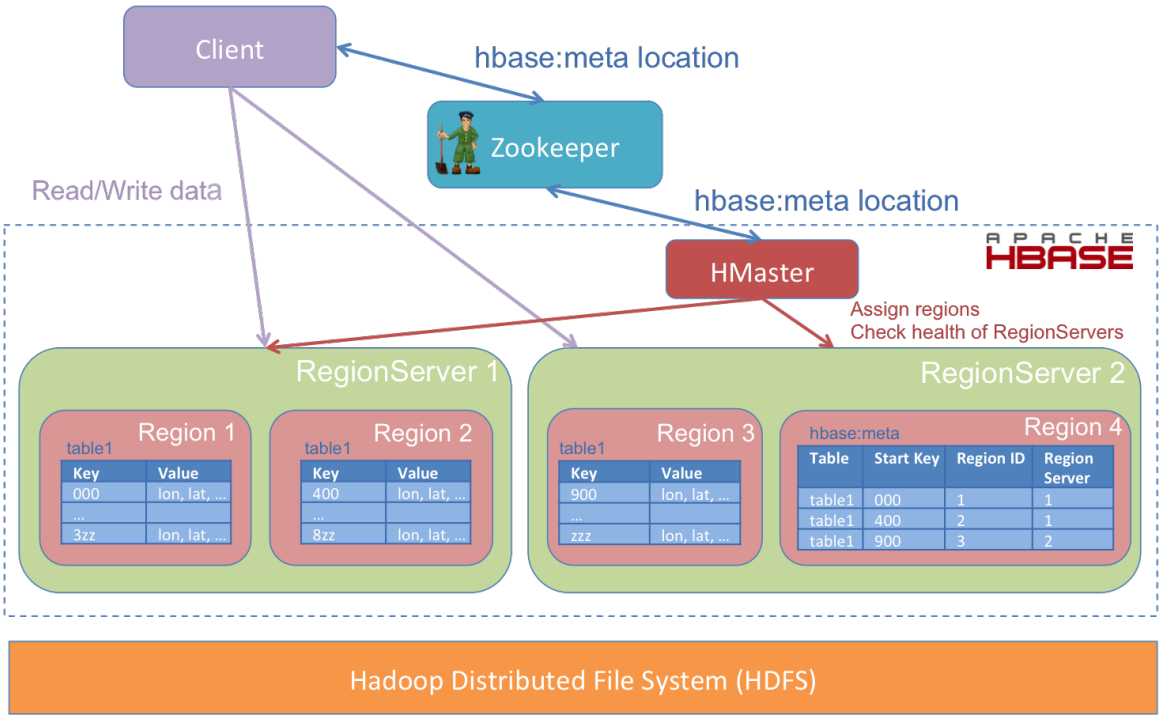


Figure 2.1: HBase architecture

HBase is also a KVS database since the data model in it is organized as a KVS. Within a table, data are stored as a sorted list of key-value pairs according to rows that are uniquely identified by their rowkeys, which therefore play an important role when searching or scanning. Rowkey design is one of the most important aspects of HBase schema.

The physical data model in HBase is column-oriented, therefore it is a column-oriented database. Rows in HBase are composed of columns and columns are grouped into column families. Columns in a family are stored together in a low-level storage file called HFile. The column family forms the basic unit of physical storage to which certain HBase features such as compression can be applied. Hence, proper design of column families is also essential when storing and processing data on HBase.

One of the advantages of HBase is its auto-sharding capability, which means it can dynamically split tables when the quantity of data becomes excessive. The basic unit of scalability and load balancing in HBase is called a region. Because HBase keeps rowkeys in lexicographical order, each region stores a range of rowkeys between a start rowkey and an end rowkey.

2.2 Multidimensional data indexes

The representation of multidimensional data has been an issue in database design for a long time. Many solutions have been proposed to solve this issue such as quadtrees, k-d trees, R-trees, etc. [95]. In this section, we briefly introduce them and their variants.

Quadtrees

There are two types of quadtrees. The first is a point quadtree [37] where the subdivision lines are based on the values of the data points. The second is trie-based and forms a decomposition of the embedding space from which the data points are drawn.

The main idea in a point quadtree is dividing the space in each node into four quadrants, namely NE, NW, SW and SE [37]. Each node stores one record and has up to four sons. The insertion is very simple and is analogous to the insertion to a binary search tree. If the tree is empty, just create a new node containing the insertion point and make it as the root. The next point is inserted into a quadrant of the root which contains the point.

An optimized point quadtree [37] was also proposed to guarantee the search performance with the assumption that all the data points are known in a prior. Although the insertion is simple and the search is fast, the deletion and the merging in point quadtrees are difficult operations.

The difference between a point quadtree and a trie-based quadtree is the point of decomposition. In a point quadtree, the data points themselves are chosen to be the points of decomposition, while in a trie-based quadtree, points of decomposition are the points which subdivide the regions into equal size quadrants. Variants of trie-based quadtree such as MX-quadtrees (MX denotes matrix) [94] and PR-quadtrees (P for point and R for region) [87] have been used in many applications.

K-d trees

In principle, a k-d tree is a binary tree. Instead of partitioning space based on the values of all k dimensions as in a quadtree, space is partitioned based on only one dimension in each level in a k-d tree. Therefore, k-d trees have less test in each level and simpler search algorithm because of fewer options in each recursive step, only two options instead of four ones as in quadtrees. However, decomposition in k-dimensional space become a sequential process whereas quadtrees allow partition simultaneously.

K-d trees also have two types, namely, point k-d trees and trie-based k-d trees. There are many variations of the point k-d tree such as the adaptive k-d tree [39], the fair-split tree [14], the VAMSplit k-d tree [109], etc. The structure of each variation depends on which dimension and value are chosen to partition. The trie-based k-d tree is a k-d tree which the positions of the decomposition do not depend on the data points. There are also many

variations of the trie-based k-d tree such as the PR k-d tree [87], the sliding-midpoint k-d tree [77], the bucket PR k-d tree [87].

R-trees

R-tree was originally proposed to be a dynamic index structure for multidimensional information, especially for non-point objects including rectangles or polygons. Since it was first introduced, it has been widely used in both theoretical and applied contexts. Common usages of R-tree are to index spatial objects such as streets, lakes, rivers, etc. and to query these objects as quick as possible. Moreover, many different variants of R-tree (e.g, R^+-tree [98], R*-tree [9], the Priority R-tree [8], the Hilbert R-tree [54], etc.) have been proposed to index different type of multidimensional data.

R-tree is a balanced search tree which can dynamically split and merge nodes. The key idea of the data structure is its *minimum bounding rectangle*. Nearby objects are grouped and represented with their minimum bounding rectangle as their parent node in the tree. Each leaf in the tree represents a single object; the non-leaf nodes represent the aggregation of their children. Since all children of a node lie within the bounding rectangle of the node, if a query does not intersect the node, it can not intersect any children and searching for all children can be eliminated. Therefore, R-tree is beneficial for searching data, especially for skewed data because it is a balanced tree.

Space-filling curves

Space-filling curves (SFCs) [93] transform multidimensional data into one dimensional data. One dimensional data often have natural ordering such as numbers, alphabetical order. The ordering helps to partition, search data easier. However, multidimensional data have no natural ordering. Queries on multidimensional data, e.g., spatial queries are mostly related to proximity. In general, SFCs are used to reduce proximity problems of higher dimensional data. There are many types of SFCs such as Z-order curve [81] and Hilbert curve [46].

Chapter 3

Related Work

Traditional relational database management systems (DBMSs) have managed and processed multidimensional spatial data efficiently by exploiting tree-structured spatial indices. For example, PostGIS[1] adds R-Tree [42] support to PostgreSQL, and Oracle Spatial supports spatial data by using the Quadtree [37]-based index. For spatiotemporal data, TerraLib [15] is an open source software library extending object-relational DBMSs that allows spatial, temporal, and attribute queries on the database. However, relational DBMSs can no longer keep up with the increasing volume of data because of insufficient scalability.

When considering scalable data processing systems for large datasets, systems based on MapReduce [26] have dominated. Many researchers have studied on processing spatial queries using MapReduce such as [16, 70, 4, 117, 31, 69, 111, 113]. Also, there are some proposed systems including SpatialHadoop [32, 33, 10], an spatial extension of Hadoop [110], Hadoop-GIS [3], a Hive [104] integrated spatial data processing system, SECONDO [68], SATO [107] and ESRI Hadoop [111] that are based on MapReduce for processing large volumes of spatial data. Some of them utilize R-Tree [42] family such as R+-Tree [98], R*-Tree [9], STR [64] to efficiently handle spatial data. Beside spatial queries, many researchers also study spatiotemporal queries on MapReduce such as [65, 70, 112, 40, 102, 5, 6, 34]. Though, such systems are suitable in the context of batch processing; they still have a high latency compared with the requirements of a low-latency system.

3.1 Spatial Databases

Spatial support has also been extended to NoSQL-based solutions such as [103, 119, 101]. MD-HBase [85] layers a multidimensional index over the key-value store by using Z-ordering

[1] http://postgis.net

[81] as linearization technique and utilizes multidimensional index structures, such as the K-d tree [11] and the Quadtree [37] to solve the Z-Order limitation on top of HBase. However, MD-HBase physically splits regions in HBase based on k-d tree or quadtree, so it requires effort to modify HBase split policy to distribute data into regions. The modification may lead to an inconsistent state when an analytical query concurrent to a split. Our index requires neither additional modification in HBase base code nor concern about consistency while splitting regions, therefore it can be utilized on top of not only HBase but also other key-value stores such as Accumulo, Cassandra. Furthermore, because our BGR partitioning does not require physical partition, it can be applied to a spatial join query with multiple tables and only costs minimal, whereas MD-HBase needs a further study about how to apply its index to process such query.

[63, 89] adopt Geohash to handle spatial data in HBase. HGrid [44] builds a hybrid index structure, combining a quad-tree and a grid as primary and secondary indices in HBase. GeoWave [108] uses a tiered and binned Space Filling Curve [43] to index multidimensional data into single dimensional KVSs in order to preserve locality in all dimension. Hsu et al. [47] proposed a novel key formulation scheme based on R^+-tree, called KR+-tree, on HBase and Cassandra, and designed spatial query algorithm of kNN query and range query based on the proposed key design. KR+-tree is able to balance the number of false-positive and the number of sub-queries so that it improves the efficiency of range query and kNN query in compared with MD-HBase. Nonetheless, these systems did not concern with spatial join query, which is also an important and expensive spatial query.

M-Grid [56] was introduced in 2017 as a multidimensional data distributing and indexing framework for location-aware services on the cloud platform. The authors use a binary-trie structure to partition the data and then exploited the Hilbert Space-Filling Curve based linearization technique to convert multidimensional data into one-dimensional binary keys. The framework is scalable, platform independent and efficiently process point, range, and kNN queries. However, it partitions the data in a binary-trie structure which still cause unbalanced partitioning when working with skewed multidimensional data. In G^+-HBase, we proposed BP-Code to better balance the partitions of the data. Moreover, the framework only provides basic multidimensional queries, thus it needs further study to supports other queries such as spatial join, kFST.

There are numerous research efforts to develop both in-memory [78, 92, 86] and disk-based [88, 66, 67, 90, 55] spatial join algorithms. In the past decade, with the explosion of collected data, many studies have been proposed to process spatial join in parallel. [91] presents two parallel partition-based spatial-merge join algorithms (PPBSM) based on the partition-based spatial-merge join (PBSM) [90], a classic spatial join algorithm which

partitions input records according to a uniform grid. PPBSM reduces the effects of skewed data distribution by declustering space into a number of grid tiles, then evenly assigning tiles into nodes in the cluster. Recently, with the emergence of MapReduce, Spatial Join with MapReduce (SJMR) [118] has been proposed to adopt PPBSM algorithm into MapReduce. Though SJMR helps to distribute tiles in a dense area into different partitions, it does not map the tiles based on data density, so still has imbalance problem with skewed data. In G-HBase, instead of using grid partition, we utilized the proposed BGR Partitioning to split the space based on data density of the two input tables to get the better load balance in parallel processing.

3.2 Spatiotemporal Databases

With the popularity of spatiotemporal applications, e.g., traffic analysis and navigation applications, several spatiotemporal indexes have been proposed [79, 84, 76]. Recently, there were many studies that extend the spatiotemporal index in HBase such as [116, 30]. Chen et al. [20, 21] proposed a two-level lookup mechanism, STEHIX (Spatio-TEmporal Hbase IndeX), which is based on the retrieval mechanism of HBase. First, they used the Hilbert curve to linearize geo-locations and store the converted one-dimensional data in the meta table. Then, inside each region, they built a region index using the StoreFiles in HBase regions. They focused on range queries and kNN queries for the designed index and showed improvement of query performance over MD-HBase. However, the index was built based on the specific inner structure of HBase, which leads to the index inflexible to be applied to other KVSs.

GeoMesa [38] introduces a spatiotemporal index structure based on Geohash on top of Apache Accumulo[2]. It interleaves Geohash and timestamps into the design index and achieves acceptable results when storing data at 35-bit resolution. Nonetheless, the performance of its proposed spatiotemporal index significantly depends on the number of Geohash characters in rowkey and on the resolution level, hence the study of their impact on each query and each dataset is required.

3.3 Top-k Frequent Spatiotemporal Terms Computation

GeoMesa [38] presents a Geohash-based spatiotemporal index structure on top of Apache Accumulo. However, it needs further study to efficiently support an aggregation query

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

like kFST since they only support basic spatiotemporal queries. MD-HBase [85] layers a multidimensional index over HBase by using multidimensional index structures, such as the K-d tree [11] and the Quadtree [37], and Z-ordering [81] as linearization technique. Though, the trie-based trees in MD-HBase still cause unbalanced data distribution in clusters. Our index structure provides more balanced data distribution, thus support better for parallel processing. We applied the index structure in MD-HBase to support parallel kFST queries and observed better query performance of our proposed index.

There are some studies of indexing spatio-textual data into KVSs. Spatial Textual HBase (ST-HBase) [71] proposed Spatial and Textual Based Hybrid Index (STbHI) and Term Cluster Based Inverted Spatial Index (TCbISI) using KD-tree based on HBase. Even though ST-HBase can provide high insert throughput and both STbHI and TCbISI support better query performance when compared with other existing spatio-textual index, the study did not consider time dimension, which is also an important dimension in geo-tagged social data. Chen et al. Chen et al. [22] supports spatiotemporal keywords queries in HBase with an access model using Hilbert curve [80] and Bloom filter [12]. However, this work is different from our work since we focus on top-k frequent spatiotemporal term computation.

Top-k spatiotemporal query has been supported in some recent studies. Mercury [73] uses a multi-level partitioning in-memory pyramid [7] to support top-k spatiotemporal-textual queries over geotagged social data. GARNET [52] addresses various queries on microblogs data by initially storing data in main memory then periodically flush to disk. AFIA [100] builds a multi-layer grid index where each cell in the grid stores a list of k+1 most frequent terms in the cell. However, both systems only return approximate results, do not guarantee the exact top-k terms can be computed. GeoTrend [72] and GeoScope [13] compute top-k frequent and trending keywords over a spatiotemporal range, Tornado [75, 74] introduces a spatio-textual indexing layer to the architecture of Storm [105], but the problem is different since these systems work on stream data stored in main memory, where our work targets large historical data which need a scalable distributed system to store and query.

The study of STL-Li in [2] is closet to our work since it solves the same problem in the same data. However, our approach is different with theirs in three main aspects: (1) We directly build a spatiotemporal index on KVS where STL-Li needs a huge extra index in main memory. (2) We estimate STL length on-the-fly which makes the queries more flexible with various inputs. (3) We calculate STLs in parallel to improve query performance, so no need to store STLs in advance, saving storage space. Our experiments evidence also show that our approach outperforms STL-Li approach in both space requirements and query performance.

3.4 Discussions

Table 3.1: KVS-based Multidimensional Systems

Systems	KVSs	Data Types	Partition Techniques	SFCs	Queries
GeoMesa [38]	Accumulo	Points	Grid	Z-Order	Range
MD-HBase [85]	HBase	Points	KD-Tree, Quadtress	Z-Order	Range, kNN
HGrid [44]	HBase	Points	Grids, Quadtress	Z-Order	Range, kNN
GeoWave [108]	HBase, Cassandra, Accumulo, DynamoDB	Points	Quadtrees	Hilbert	Range
M-Grid [56]	HBase	Points	P-Grid	Hilbert	Range, kNN
G^+-HBase	HBase	Points, Non-point Objects	BGRP-Tree, BP-Tree	Z-Order	Range, kNN, Spatial Join, kFST

Table 3.1 summarizes the full-fledged systems based on KVS to support big multidimensional data. Each row in the table designates a system, while each column represents one aspect of the system as described below.

KVSs. Because in a KVS, designing the key is very important to improve query performance, all systems focus on building index structures on top of KVSs. Except for MD-HBase which requires additional modification in HBase storage layer to distribute data, most of the indexing frameworks are platform independent. Therefore, most of the systems can adapt to various KVS systems. For example, GeoMesa originally conducted their experiments on Accumulo, but they have already extended the system on HBase, Cassandra [1]. G^+-HBase also designs index structures which are independent of KVS platforms, so it can adapt to other KVSs besides HBase.

Data Types. To index multidimensional data into KVSs, it is essential to transform multidimensional data into one-dimensional keys. Therefore, all systems focus on supporting points. G^+-HBase presents index structures for not only points but also non-point objects.

Partition Techniques. The partition techniques implemented in systems vary and include both uniform partition (grids) and basic multidimensional partitions (Quadtrees, KD-trees). Instead of using existing partition techniques, G^+-HBase proposed BGRP-Tree and BP-Tree which dynamically partition data based on data density.

SFCs. Z-Order and Hilbert Curve are the two SFCs used in the systems. Most systems utilized Z-Order because of its simplicity to compute. Hilbert Curve offers better locality preservation but is more compute-intensive. In G^+-HBase, same as most systems, we used Z-Order to avoid expensive computing for SFCs.

Queries. The queries supported by the systems are often basic operations such as range and kNN queries. Besides basic operations, join operation is also an important query in a multidimensional system. Spatial join is an expensive query which requires a full table scan, therefore it is suitable for a parallel framework. To the best of our knowledge, G^+-HBase is the first KVS-based framework which supports spatial join query by using MapReduce parallel processing. Among these systems, G^+-HBase is also the only system that supports kFST, a basic analytical query on geo-tagged social data. kFST finds the k most frequent terms among the posts in a specified spatiotemporal region.

Chapter 4

G-HBase: A High Performance Geographical Database based on HBase

4.1 Introduction

Recent and rapid improvement of positioning technologies such as the Global Positioning System (GPS), sensors, and wireless networks has resulted in an explosion of geographic data generated by users. McKinsey Global Institute says that the pool of personal location data was at the level of 1 PB in 2009 and is growing at a rate of 20% per year [62]. Many systems, in both scientific research and daily life, have taken advantage of the collected location data. For example, intelligent transportation systems are exploiting the massive volume of sensor data from probe cars and GPS-enabled devices to provide efficient route planning and traffic balancing based on the current traffic situation. Such systems need a database of geographic data, that can store a massive volume of data and provide high-performance spatial queries.

To meet these requirements, systems need a database management system (DBMS) that has good scalability while guaranteeing satisfactory performance with low-latency spatial queries. Although some frameworks based on MapReduce [26] such as SpatialHadoop [32], Hadoop-GIS [3] are reasonable choices for handling large volumes of spatial data, MapReduce is built for batch processing and lack of the ability to access data randomly to provide real-time query processing [41] [28]. Recently, key–value stores such as HBase [41] [28], with their scalability, fault tolerance, and random read/write capability, have shown promise. However, they do not have native support for geographic data and spatial queries, which is required in a database of geographic data.

Some methods [63, 89] use Geohash[1], a linearization technique that transforms two-dimensional spatial data into a one-dimensional hashcode, to handle spatial data in HBase. However, there are two major limitations of Geohash which they do not address. The first limitation is *data skew problem*, which lead to load imbalance of parallel tasks in a distributed system and increase response time. Secondly, although Geohash uses Z-order to maintain proximity of geographic data, there are still some points that are close geographically, for example, *7* and *e* in Fig. 4.2, but have widely separated Geohash codes. This *Z-order problem* would cause false-positives in range query, and far apart objects are included in the same partition when partitioning data using Geohash. Besides, it is necessary to design spatial queries to adapt to the spatial indexes and the HBase structure.

We have developed G-HBase - a high-performance geographical database based on HBase. The goal of the system is to provide a distributed and efficient system that can handle a large volume of data, and support computationally expensive spatial queries. First, we used Geohash as rowkeys in HBase to distribute large datasets across a multi-server cluster. Then we proposed a novel partitioning method, the binary Geohash rectangle partitioning (BGR Partitioning), to partition tables in HBase based on the data density. We also designed algorithms to process various intensive spatial queries based on the designed index structure and proposed partitioning method. Main contributions are:

- We provide a distributed database for geographic data based on HBase which can store a large volume of data and support high-performance spatial queries.
- We propose a novel partitioning method named BGR Partitioning to handle limitations of Geohash and improve the performance of spatial queries.
- We introduce an algorithm to process spatial join using MapReduce on HBase. To the best of our knowledge, G-HBase is the first geographical database that supports spatial join query on HBase (Sect. 4.4.3, Sect. 4.5.5).
- We also present an algorithm using range bounding geohashes to process range query that can avoid Z-order problem (Sect. 4.4.2, Sect. 4.5.4).

4.2 Overview of G-HBase

Figure 4.1 gives the overview of G-HBase architecture. G-HBase includes two components, namely, *Index Structure* component and *Query Processing* component.

[1] http://geohash.org

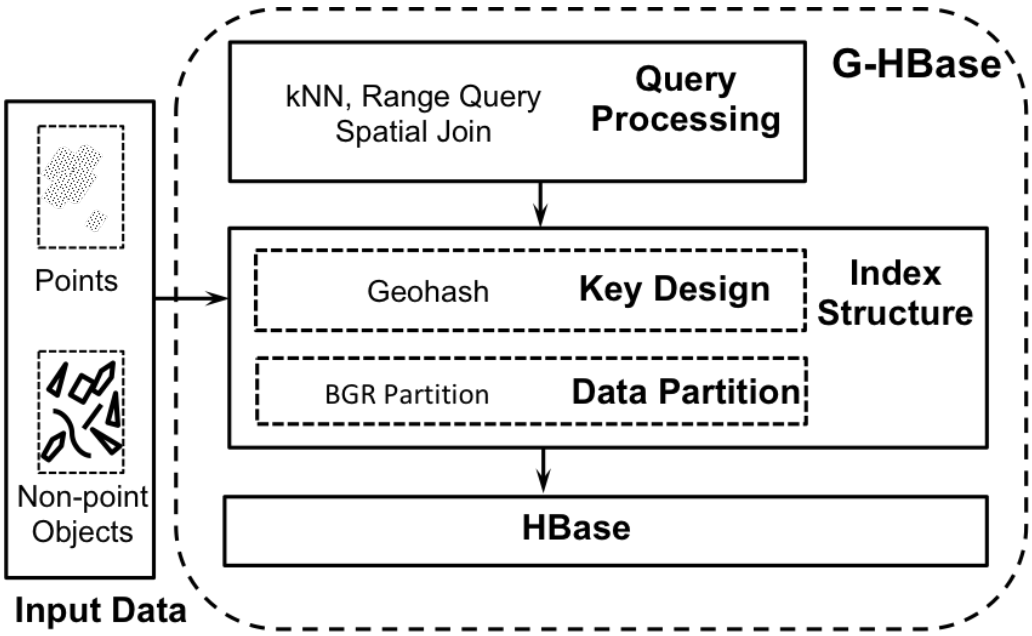


Figure 4.1: Architecture Overview of G-HBase

The *Index Structure* component includes the key design (Geohash) and the data partitioning method (BGR Partition). Input data including points and non-point objects data are inserted using Geohash to encode. After data insertion, the table is partitioned based on the data density using BGR Partition method. Depending on the type of query, the *Query Processing* component will decide to use the BGR Partition method or not and how to utilize the method to achieve the best performance.

4.3 Indexes Component

4.3.1 Geohash as rowkeys

The first part of the *indexes* component is Geohash. HBase is a key-value store (KVS) in which data are stored as a list of key–value pairs sorted by uniquely identified rowkeys. Therefore designing the unique rowkeys is one of the most important aspects of HBase schema. However, geographic data are represented by two coordinates (longitude and latitude), which are equally important in defining a location. Geohash provides a solution to transform longitude/latitude coordinates into unique codes.

Geohash is a hierarchical spatial data structure that subdivides space into uniform rectangles, then uses Z-order (Fig. 4.2) to encode rectangles. To best take advantage of Geohash, we store the spatial data in HBase where the rowkeys are geohashes. HBase stores data in

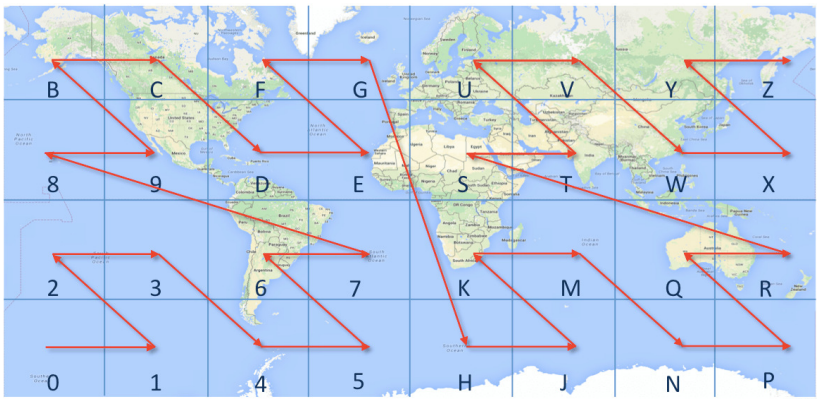


Figure 4.2: Z-order traversal with 1-character geohashes

lexicographical order of rowkeys, thus objects that are nearby geographically will also be close to each other in the database. Geohashes having a common prefix are always close to each other geographically, enabling us to do proximity search using a prefix filter.

Storing Points

Figure 4.3 shows three steps to generate a geohash for a point. In the first step, longitude and latitude dimensions are represented by longitude and latitude trees, respectively. The root node has the bound S, the maximum range of the corresponding dimension (for example, S_x is 360, S_y is 180). Geohash recursively divides nodes in the trees into two equal-size children, and then assigns bit 0 (resp. 1) if the location is in the left (resp. right) child. Then, Geohash interleaves bits from the two dimensions to generate a bit sequence. Finally, each five bits in the bit sequence is encoded into a character in Base32.

Storing Non-Point Objects

There are two approaches to index a non-point object using Geohash. First, single-assignment approach uses only one geohash which covers the whole object. Second, multiple-assignment approach decomposes a non-point object into a list of disjoint geohashes that cover the non-point object. The drawback of the multiple-assignment is that objects are duplicated, which leads to increase of data redundancy and space requirement. Whereas, single-assignment approach can minimize the data redundancy. However, it leads to wasted overlapped area and more false positives during query processing. For example, if the object in Fig. 4.4 is indexed using single-assignment approach, its rowkey is *w*. The object only overlaps four geohashes *ww, wx, wy, wz*, therefore 28 geohashes from *w0* to *wv* are wasted overlapped area.

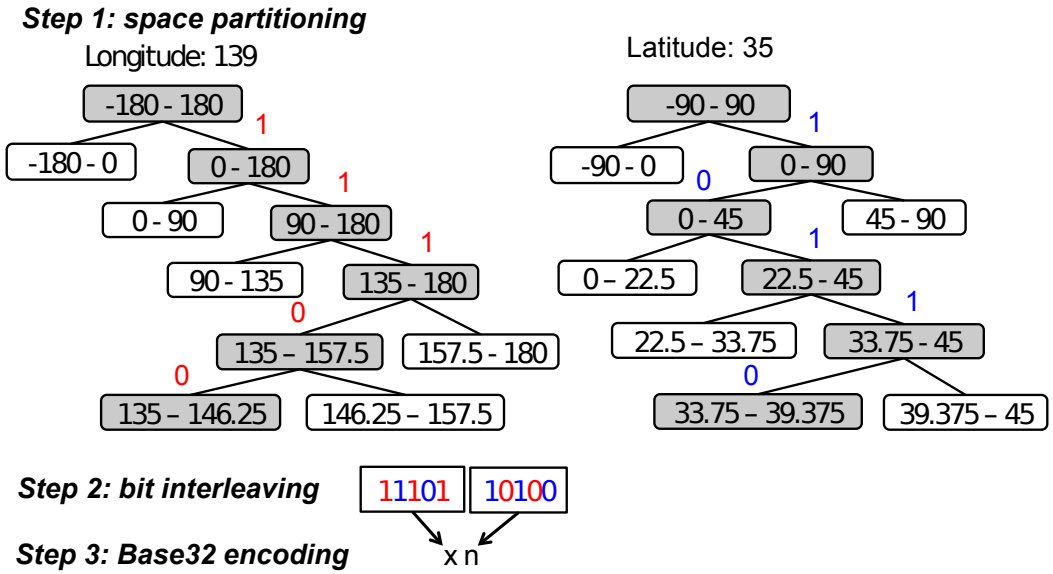


Figure 4.3: Generation of a two-character geohash for a point

If a query scans the wasted overlapped area, e.g. *w0*, the object is obtained even though it does not overlap the area *w0*.

There is trade-off between data redundancy and the wasted overlapped area to determine which approach to store non-point objects. In the study [88] about redundancy in spatial databases using indexes based on z-order, it is said that small amounts of redundancy (between 30% and 70% depending on the dataset) provide the best results. Therefore, in this work, we use multiple-assignment approach, but minimize the data redundancy.

To minimize data redundancy, we first use single-assignment approach by calculating a *candidate* – the smallest geohash which contains the centroid of the minimum bounding rectangle (MBR) of the object, and both edges are longer than the edges of the MBR. If the *candidate* covers the whole object, it is indexed using one geohash. Otherwise, we can decrement the length of the *candidate* to achieve a geohash which covers the whole object. However, this will cause a large wasted overlapped area. Therefore, instead, all eight adjacent geohashes of the *candidate* are checked if they intersect the object. An extended study from [88] about redundancy in spatial databases based on key-value stores is needed. We leave such extension as future work.

A geohash primarily denotes a rectangle. To simplify the explanation, here we consider the longitude dimension of the rectangle. The latitude dimension is analogous. The horizontal edge of the rectangle corresponds to a node in the longitude tree. For example, the 2-character geohash *xn* in Fig. 4.3 represents a rectangle with the horizontal edge corresponding to the

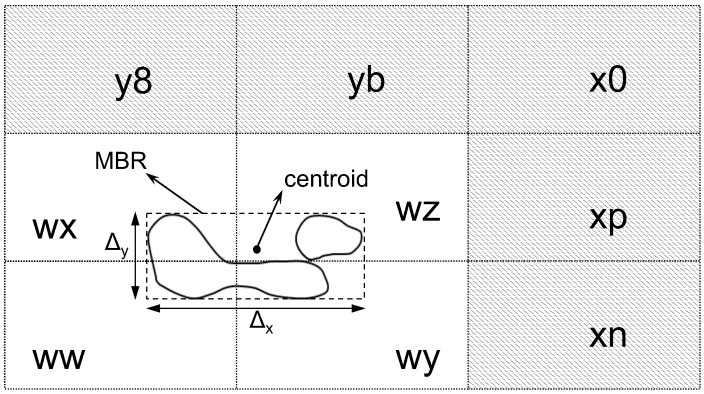


Figure 4.4: Bounding geohashes for a non-point object

node (135-146.25) at the fifth level of the longitude tree. The edge length of the node in level n is given by:

$$\Delta = S/2^n \tag{4.3.1}$$

where S is the length of the root node.

Definition 1. Horizontal MinRangeHash

For a rectangle r, the *MinRangeHash* corresponding to the horizontal range of r is defined as the geohash g_x that satisfies:

1. g_x contains the centroid of r,

2. the horizontal edge of g_x is larger than the horizontal edge of r, and

3. g_x is the longest geohash that satisfies the conditions 1 and 2.

The vertical *MinRangeHash* g_y corresponding to the vertical range of r is analogous.

To compute the horizonal *MinRangeHash* g_x, we approximate the horizontal range to the smallest node in the longitude tree of which the node size is larger than the range. Based on [51], the level of the node can be understood as the precision of such approximation and is calculated as:

$$n = \left\lfloor \frac{ln(S/\Delta)}{ln(2)} \right\rfloor \tag{4.3.2}$$

The length l_x of the *MinRangeHash* is calculated based on the precision as in Eq.(4.3.3). Similarly, we can compute the length of its vertical *MinRangeHash* as in Eq.(4.3.4).

$$l_x = \begin{cases} 2\lfloor n_x/b \rfloor + 1, & \text{if } (n_x \mod b) = 0 \\ 2\lfloor n_x/b \rfloor + 1, & \text{if } (n_x \mod b) \geq \lfloor b/2 \rfloor \\ 2\lfloor n_x/b \rfloor, & \text{otherwise} \end{cases} \tag{4.3.3}$$

$$l_y = \begin{cases} 2\lfloor n_y/b \rfloor + 1, & \text{if } (n_y \mod b) = 0 \\ 2\lfloor n_y/b \rfloor + 1, & \text{if } (n_y \mod b) \geq \lceil b/2 \rceil \\ 2\lfloor n_y/b \rfloor, & \text{otherwise} \end{cases} \tag{4.3.4}$$

In Eq. (4.3.3) and Eq. (4.3.4), b is the number of bits to encode a character. In the case of Geohash, b is five because Geohash uses Base32 to encode.

Algorithm 1 describes the algorithm to calculate the list. To reduce the data size inflation when assigning an object to multiple geohashes, we try to make the list with the minimum number of geohashes. Therefore, we first compute a *candidate* – the smallest geohash which contains the centroid of the minimum bounding rectangle (MBR) of the object, and both edges are longer than the edges of the MBR. To determine the *candidate*, we get the *MinRangeHash* g_x and g_y of the MBR, and calculate the maximum geohash of the two *MinRangeHashes*.

Algorithm 1: Get a list of geohashes that cover a non-point object

input : a non-point object o
output : list of geohashes that cover the object

```
1: r ⟵ MBR of o
2: g_x ⟵ horizontal MinRangeHash of o
3: g_y ⟵ vertical MinRangeHash of o
4: candidate ⟵ max(g_x, g_y)
5: results ⟵ candidate
6: for ad ∈ candidate.getAdjacent() do
7:     if ad intersects with the object then
8:         results ∪ ad
9: return results
```

After getting the *candidate*, it is added to the resultant list. Because both edges are longer than the edges of the MBR, if the centroid is close to the centroid of the *candidate*, the *candidate* may cover the object. In the case the *candidate* does not cover the object, all its

eight adjacent geohashes are checked if they intersect the object then added to the result list. Finally, the object is inserted into HBase with keys corresponding to the geohashes in the list.

4.3.2 BGR Partitioning

The second part of the *indexes* component is BGR Partitioning. To store large tables into a distributed cluster, HBase provides an auto-sharding capability, which dynamically splits tables into a number of smaller chunks, namely regions, when the quantity of data becomes excessive. However, region split is based on one dimensional geohash code, and the Z-order problem of geohash could lead to a region may contains objects which are non-consecutive in the space. We propose BGR Partitioning method that partitions a table into disjoint rectangles in two dimensional space based on the density of the table.

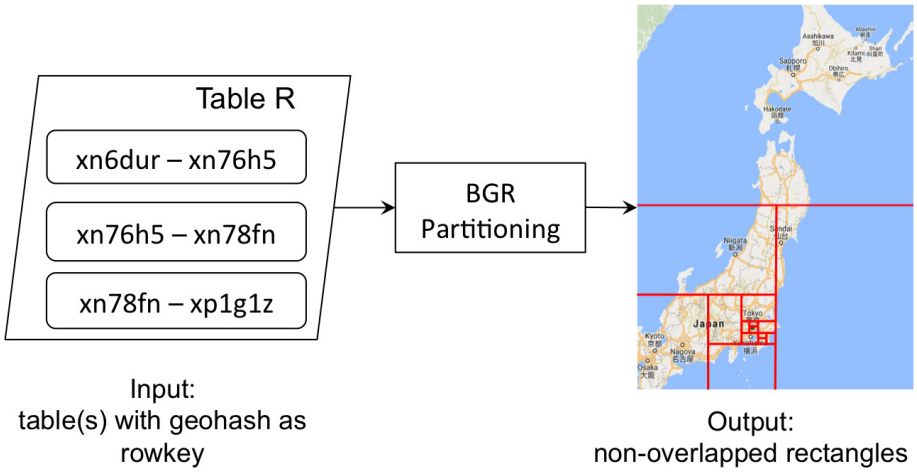


Figure 4.5: Input and output of the BGR partitioning

In G-HBase, after data is ingested using geohash as rowkey and the table is already physically splitted into regions, a BGR partitioning logically partitions data into non-overlapped rectangles. Figure 4.5 shows input and output of the BGR partitioning method. The input of the method could be a single table or multiple tables. The method includes two steps as in Algorithm 2. The first step is dividing regions of the table(s) into segments based on Longest Common Prefix (LCP) of geohashes. Secondly, all segments are inserted into a BGRP Tree, a temporal data structure to subdivide the segments into disjoint rectangles. The output of the method is all leaves of the BGRP Tree. Each leaf in the tree is a rectangle which can be represented by a geohash range. The output of the BGR Partitioning could be used as input of an R-Tree as in kNN query or partitions to process spatial join in parallel using MapReduce.

The range of regions in a table is updated only when region is splitted. Because the BGR partition is based on the range of regions, we only apply it when an insertion or a update

causes a region split. When deleting data, the data is marked as deleted instead of being physically deleted. Therefore, the deletion in HBase does not cause any change in the range of regions and we do not need to apply BGR partitioning after each deletion.

Algorithm 2: BGR Partitioning for a single table

input : a table T
output : a list of disjoint partitions
1: *tree*.initialize()
2: **foreach** *region* $\in T$ **do**
3: $S \longleftarrow$LCPSegment(*region*)
4: **for** $s \in S$ **do**
5: insert(*tree*, s)
6: **return** *tree*.allLeaves()

The first step is LCP-based segmentation. For each region, we find the LCP of the start and end rowkeys, then obtain the character next to the LCP from the start rowkey (c_1) and the end rowkey (c_2). Geohash uses Base32 to encode spatial points, so we concatenate the LCP with each character from c_1 to c_2 in Base32 to create a new geohash. By using this algorithm, we can ensure that the list of segments will cover all points in the region.

Table 4.1: An example of the LCP-based segmentation

Region	Start Rowkey	End Rowkey	LCP	Rectangles
R1	**xn**6dur	**xn**76h5	xn	xn6, xn7
R2	**xn7**6h5	**xn7**8fn	xn7	xn76, xn77, xn78
R3	**x**n78fn	**x**p1g1z	x	xn, xp

Table 4.1 shows an example of the LCP-based segmentation. In this example, we have three regions in the geohash range from *xn6dur* to *xp1g1z*. The result of the segmentation is a list of rectangles that cover all points in the three regions. In this example, we can assume that the areas involving rectangles with longer geohashes, i.e., *xn76*, *xn77* and *xn78*, have the higher density than other areas.

In the example in Table 4.1, some rectangles contain others, such as *xn*, *xn7* and *xn77*. To partition the table into non-overlapped partitions, we do further partition using a BGRP Tree. The BGRP Tree is the core of BGR Partitioning. We build the tree satisfies the following properties.

- Each node (rectangle) is within a Geohash range
- There is no overlap between nodes

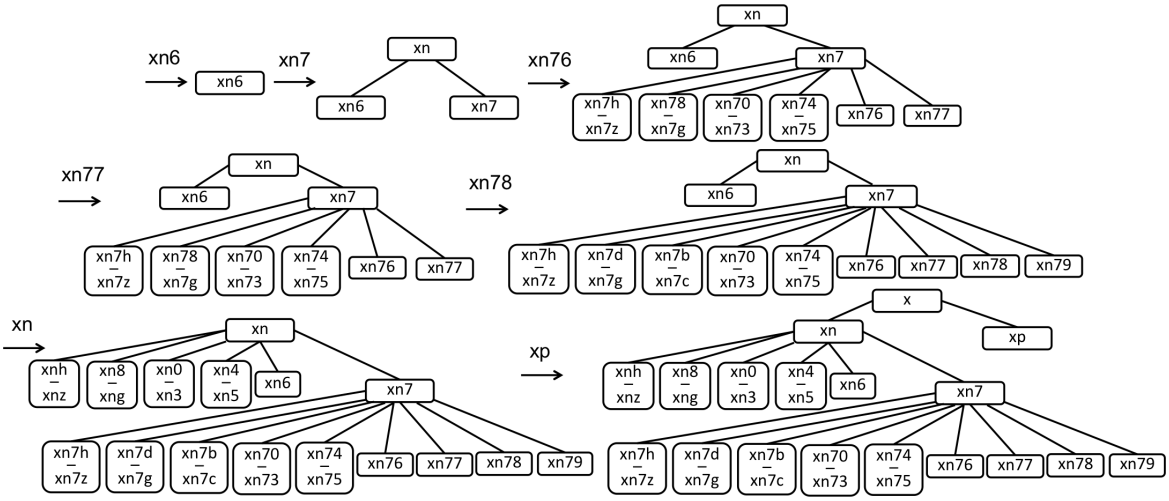


Figure 4.6: Insertion in BGRP Tree

- The leaves cover the whole surface of the original rectangles

- Tree construction does not depend on the order of rectangle insertion, so we can use the BGRP Tree with multiple tables.

Figure 4.6 shows the insertion of all segments in Table 4.1 into the BGRP Tree. Algorithm 3 describes the algorithm to insert a segment into the BGRP Tree. We first do the search which descends from the root, recursively searching subtrees for the highest-level node that contains the input segment. If a node does not contain the segment, we do not search inside that node.

After finding the containing node, the segment is inserted into the node. A BGRP task is a binary partition task which is necessary for some cases in the insertion. The input for this task is two rectangles, with the bigger rectangle containing the smaller one. Both could be presented by a geohash or a range of geohashes. The output is a set of subrectangles that includes the smaller rectangle and covers all the surface of the bigger one.

Each subrectangle in the output need to correspond to a geohash range. As explained in Sec. 4.3.1, geohash is generated by binary partitioning, so if we divide a geohash into two equal rectangles, each rectangle will correspond to a geohash range. Therefore, the BGRP task recursively subdivides a rectangle into two subrectangles until one of the two subrectangles matches the smaller input rectangle. This method is similar to the binary space partitioning method [97]. Figure 4.7 describes the BGRP task for the inputs *xn* and *xn6*. For a rectangle whose geohash range length is r, the task will require $log_2 r$ binary partition steps and will result in a set of $log_2 r + 1$ subrectangles.

Algorithm 3: Insert a segment to a BGRP Tree

```
   input  : a tree t, a segment s
1: if t is empty then
2:     t.setRoot(s)
3:     return
4: n ⟵ search(t, s)
5: if n is null then
6:     newRoot ⟵ getLCP(root, s)
7:     newRoot.addChild(root)
8:     newRoot.addChild(s)
9:     t.setRoot(newRoot)
10:    return
11: if n is leaf then
12:    BGRPTask(n, s)
13: else
14:    if n = s then
15:        for c ∈ n.getChildren() do
16:            BGRPTask(n, c)
17:    else
18:        n.addChild(s)
```

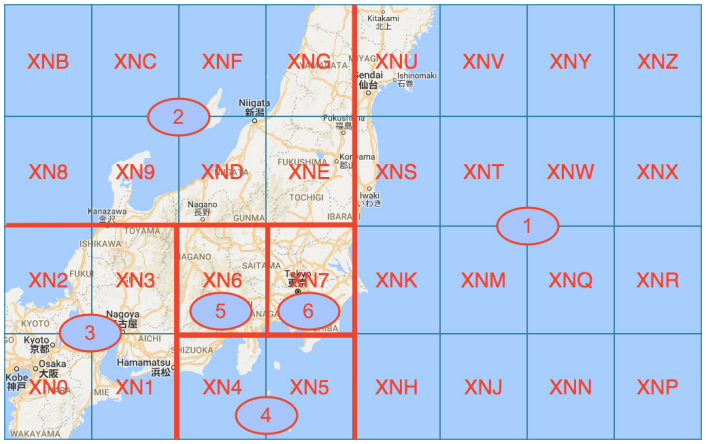


Figure 4.7: The BGRP task

We execute a BGRP task when inserting into a leaf, as shown for the insertion of *xn76* and *xn78* in Fig. 4.6. The BGRP task is also applied when the insert segment exists in the tree as a nonleaf node. Insertion of *xn* in Fig. 4.6 is an example of this case. Before being inserted, *xn* was a nonleaf node with only two children, *xn6* and *xn7*, which do not cover the entire surface of *xn*. Therefore, we need to partition *xn* to ensure that the surface of *xn* is

completely covered in leaves. After inserting all the segments into the tree, all leaves in the tree are partitions of the table.

4.4 Query Processing Component

Spatial query processing strategy in the *query processing* component includes two steps. *Filter step:* Using the spatial index to quickly eliminate objects that do not satisfy the query. The output of this step is a set of candidates which possibly are included in the query result. *Refinement step:* Examining all the candidates to eliminate false objects.

4.4.1 k Nearest Neighbors

Filter step: With Geohash as the rowkey for storing data in HBase, when processing a kNN query, the client first sends a scan request with a prefix filter to the regions. Because points that share the same prefix are near each other in the space, the scan returns all points that are near the query point.

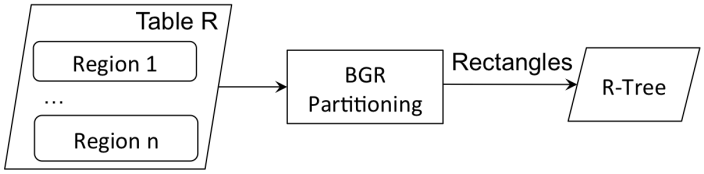


Figure 4.8: The index to process kNN query

Inspired by Google's BigTable [18] coprocessors, HBase also supports efficient computational parallelism by using "coprocessors". By using this parallelism, in G-HBase, we execute the refinement step inside each of the regions. Processing kNN in parallel helps reducing I/O in case scan results is larger than k. However, if all the regions return results to the client, it is still heavy I/O load when there are a large number of regions. To prune unnecessary scanning, we use an R-Tree as an additional index tier as in Fig. 4.8.

First, data is inserted into a table using geohash as rowkey. The table is splitted into regions. Then, we apply the BGR partitioning into the table. Finally, the output of the partitioning, all leaves of the BGRP Tree, are inserted into the R-Tree. When processing kNN, we find the rectangles in the R-Tree that may contain query results, then scan only the found rectangles, thereby pruning the scanning on unrelated regions. All the steps for R-Tree creation are processed in the background, removing the need for R-Tree creation overhead when processing queries.

Refinement step: After obtaining the scan results, the client then refines the results to get the k nearest neighbors of the query point. If not enough nearest neighbors are found, the filter step is conducted again with shorter prefix until enough k points are found. Otherwise, we check the accuracy of the result. Same as in [32], if the distance from the point to its k^{th} furthest neighbor is shorter than the distance from the point to four edges of the prefix rectangle, the query is terminated. Otherwise, we draw a circle centered at the input point with the distance from the point to its k^{th} furthest neighbor is the radius. Then, we scan the neighbors of the prefix which overlap the circle and then get the final k nearest neighbors.

4.4.2 Range Query

Filter step: Query range can be simply processed by scanning between start geohash and end geohash of the range, making many false positives due to Z-order traversal. For example, in Fig. 4.9, the query range starts with geohash $1x$ and ends with geohash 62, causing a lot of redundant scan in rectangles 2, 3, 4, 5. To reduce the false positives, we scan using range bounding geohashes (RBG) of the query range.

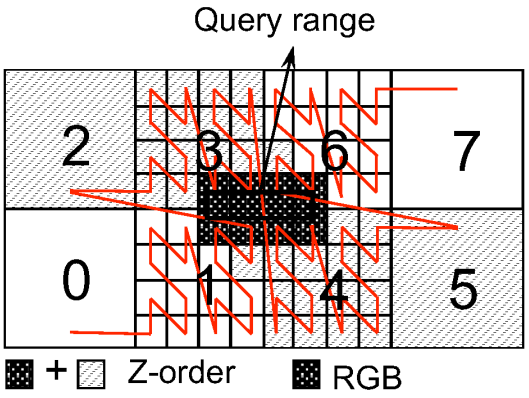


Figure 4.9: Scanning area

Algorithm 4 describes how to determine RBG of a query range. Same as Sect. 4.3.1, we calculate the *MinRangeHash* g_x and g_y corresponding to the horizontal and vertical edge of the query range, respectively. As experienced in the work [60], the imbalance between the horizontal and vertical edge in the range query would lead to higher query response time. The query range could have a wide horizontal edge much longer its vertical edge or vice versa. In such case, if we choose the maximum of g_x and g_y as in Sect. 4.3.1 as the hash length of the RBG, it would contain many points outside the query range. To reduce the redundant scan area, the algorithm starts with computing the minimum geohash g of the two *MinRangeHash*. Then g is expanded by adding all its neighbors into a list until the query

range is covered by the list. With each geohash in the list, G-HBase generates a scan request with the geohash as a prefix filter. We do not further reduce the length because it leads to longer list of geohashes, causing more scan requests between the client and the cluster.

Algorithm 4: Get RBG for a query range

input : a query range q
output : list of geohashes that cover the query range

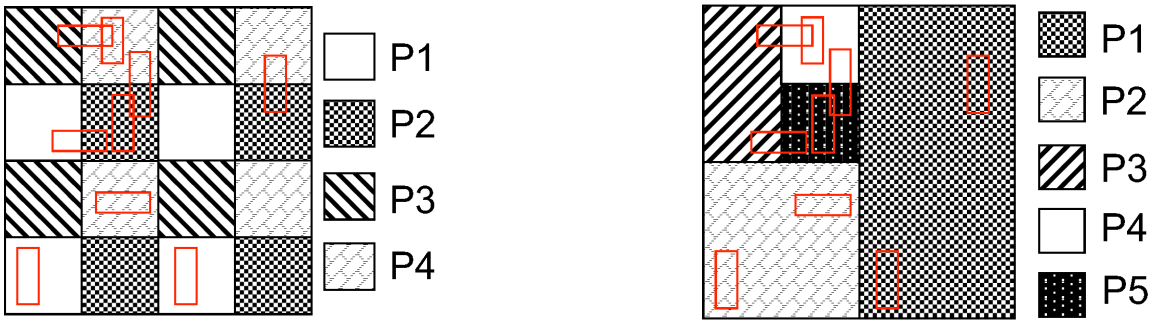
```
1:  r ⟵ MBR of q
2:  g_x ⟵ horizontal MinRangeHash of q
3:  g_y ⟵ vertical MinRangeHash of q
4:  g ⟵ min(g_x, g_y)
5:  results.add(g)
6:  while results do not cover r do
7:      for r ∈ results do
8:          candidates.add(r)
9:          for ad ∈ r.getAdjacent() do
10:             if ad intersects with q then
11:                 candidates.add(ad)
12:     results ⟵ candidates
13: return results
```

Refinement step: After getting scan results, G-HBase checks whether the query range contains points in the scanned results and eliminate all points that are not the query range. In G-HBase, refinement step in range query is executed inside each region of HBase, so that we can prune unrelated points sent from regions to the client. We named the refinement step in each region "range filter".

4.4.3 Spatial Join

Filter step: The spatial join query is a fundamental operation in a geographical database. The inputs of the query are two spatial datasets R and S and a spatial relation such as intersection, enclosure, nearness. The output of the query is a set of all pairs (r,s) where $r \in R$, $s \in S$ and r has the spatial relation with s. The input datasets can be complex objects such as lines, polygons, multi-polygons and calculating whether two such complex objects satisfy a join condition is very computationally expensive. Furthermore, because spatial join query often involves processing the whole datasets, it is a costly operation.

SJMR [118] was proposed to process spatial join in a distributed system using the MapReduce paradigm. In map stage, it partitions data into tiles, numbers the tiles according to Z-order and maps the tiles to partitions with a round robin scheme. Though it helps to



(a) SJMR partitioning

(b) BGR partitioning

Figure 4.10: Partition data to parallel spatial join processing

distribute tiles in a dense area into different partitions, it makes redundant tiles in a sparse area and does not map the tiles based on data density. For example, in Fig. 4.10b, SJMR partitions the space into 16 tiles and assigns them to four partitions. While the fourth partition contains five objects, the third partition has only one objects. In G-HBase, we also use the MapReduce paradigm to process spatial join query. However, to achieve better balance in parallel processing, we present BGRP join algorithm which utilizes the BGR Partitioning to partition space based on the density of the input data.

The BGRP join algorithm includes three phases as in Algorithm 5. In *phase A*, BGR Partitioning is applied with both input tables to create a BGRP tree. We assumed that the two input tables were ingested using geohash as rowkey. As explained in Sec. 4.3.2, we divided each region in table R and S into segments based on LCP of start rowkey and end rowkey of the region, then inserted the segments into the BGRP tree. The BGR partitioning algorithm does not physically partitions the records in the two table. It only defines the geohash ranges to be processed in map functions. Therefore, the overhead for this step is very small as in the experiments in Sec. 4.5.5.

Then, *phase B* generates combined splits corresponding to leaves in the tree. For each leaf, the data in the geohash range in the two tables may be located in different machines. We scan in table R and S to get the lists R_i and S_i of objects which are in the geohash range of the leaf, respectively. A combined split is a pair of the two lists R_i and S_i. Each combined split contains the server names hosting its geohash range. To make use of data locality, the MapReduce framework checks the server names, and if MapReduce worker nodes process is running on the same machines, it will preferably run it on one of those servers. Because HBase is colocated with MapReduce on a same cluster, the scan in the server will be able to retrieve all data from the local disk. Finally, in *phase C*, inside each map function, we use the plane sweep [92], a widely used in-memory spatial join approach, to join the two lists.

Algorithm 5: BGRP Join

input : table R and table S
output : table J which contains all overlapped pair (r, s) from R and S
/* Phase A: applying BGR Partitioning for both tables */
1: $tree$.initialize()
2: **foreach** $region_r \in R$ **do**
3: **for** $s_r \in LCPSegment(region_r)$ **do**
4: insert$(tree, s_r)$
5: **foreach** $region_s \in S$ **do**
6: **for** $s_s \in LCPSegment(region_s)$ **do**
7: insert$(tree, s_s)$
/* Phase B: generating combined splits */
8: **foreach** $p_i \in tree.allLeaves()$ **do**
9: $R_i \longleftarrow R$.scan(p_i)
10: $S_i \longleftarrow S$.scan(p_i)
/* Phase C: processing map function with each split */
11: **function** $map(p_i, pair(R_i, S_i))$**:**
12: $results \longleftarrow$ planeSweep(R_i, S_i)
13: **for** $j \in results$ **do**
14: **if** $j.referencePoint \in p_i$ **then**
15: J.ingest(j)

Refinement step: Because a non-point object may belong to multiple partitions, there are duplicated records after the filter step. We use the reference-point technique [29] to avoid duplication in the results. For each detected overlapping pair, we first compute the intersection of the pair. Then, if the reference-point (e.g., top-left corner) of the intersection is contained in the partition, the pair is reported as a final answer.

4.5 Experimental Results

4.5.1 Cluster Configuration

We built a cluster with 64 nodes. Each node had two virtual cores, 16 GB memory and a 288 GB hard drive. The operating system for the nodes was CentOS 7.0 (64-bit). We set up one HMaster, 60 Region Servers and three Zookeeper Quorums using Apache HBase 0.98.7 with Apache Hadoop 2.4.1 and Zookeeper 3.4.6. Replication was set to two on each datanode. Policy for region splitting was set to *ConstantSizeRegionSplitPolicy* and maximum file size of each HRegion was set to 512 MB. To conduct queries using MapReduce, we installed

SpatialHadoop v2.4 and configured one master and 60 slaves. The system was configured to run one map or reduce instance on each node.

4.5.2 Dataset Description

We used four real datasets with points (T-Drive [115], OpenStreetMap (OSM[2] Nodes) and non-point objects (OSM Lakes, OSM Parks) in our experiments.

T-Drive. T-Drive was generated by 30,000 taxis in Beijing during Feb. 2 to Feb. 8, 2008 within Beijing. The total number of records in this dataset is 17,762,390. It required more than 700 MB and was split into eight regions in the HBase cluster.

OSM Nodes. OSM includes datasets of spatial objects presented in many forms such as points, lines, and polygons. We used the dataset of nodes for the whole world. It is a 64 GB dataset that includes 1,722,186,877 records. We inserted the OSM Nodes dataset into 860 regions in the HBase cluster.

To evaluate kNN query and range query, we chose two groups of points for the experiments, namely, a group of high-density points and a group of low-density points. We summarized the number of points in the area of each 25-bit geohash and sorted in descending order. High-density points are the points in crowded areas, which are the first ten geohashes in the list. Conversely, low-density points are the points in uncongested areas, which are the last ten geohashes in the list.

OSM Lakes and OSM Parks. We used two datasets, OSM Parks and OSM Lakes,

Table 4.2: OSM Lakes datasets

	0.01%	0.1%	1%	10%	100%
# of objects	841	8,419	84,193	841,931	8,419,319
File size	6 MB	56 MB	490 MB	2.5 GB	9.7 GB
# of rows	1,461	15,162	152,809	1,509,175	14,977,600
# of regions	1	1	4	18	86

Table 4.3: OSM Parks datasets

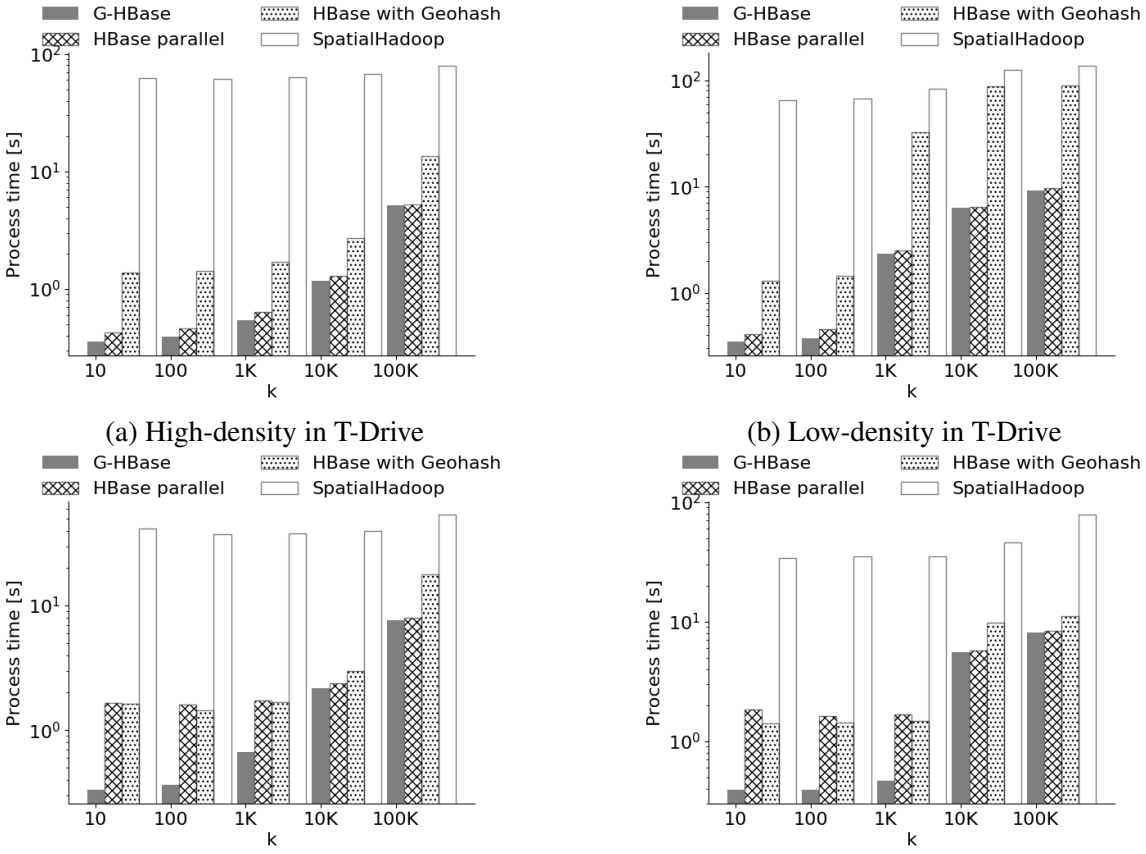
	0.01%	0.1%	1%	10%	100%
# of objects	996	9,961	99,618	996,188	9,961,883
File size	8 MB	58 MB	563 MB	3.1 GB	10 GB
# of rows	1,817	18,049	180,599	1,792,808	17,816,197
# of regions	1	1	4	23	86

to conduct spatial join query experiments. To evaluate the query with various data sizes,

[2] http://www.openstreetmap.org

we extracted 0.01%, 0.1%, 1%, 10%, 100% of each dataset. Table 4.2 and Table 4.3 describe detail information of all extracted datasets from OSM Lakes and OSM Parks dataset, respectively.

4.5.3 k Nearest Neighbors



(a) High-density in T-Drive

(b) Low-density in T-Drive

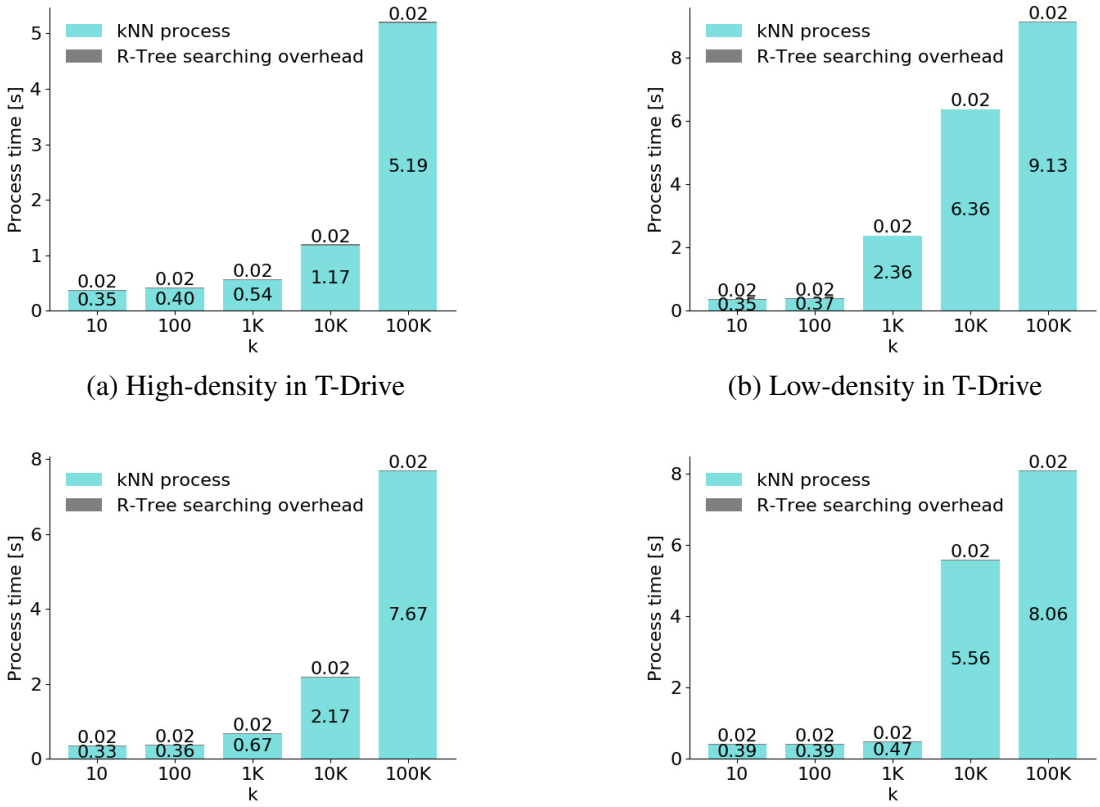
(c) High-density in OSM Nodes

(d) Low-density in OSM Nodes

Figure 4.11: Performance of kNN queries

We evaluated the performance of kNN queries using the two datasets, T-Drive and OSM Nodes. We conducted kNN query processing method on G-HBase and compared with the parallel query in HBase, the query using only Geohash and the query in SpatialHadoop as a baseline method. Figure 4.11 plots the process time in seconds for the performance of kNN queries with the T-Drive and OSM Nodes dataset. We vary k as 10, 100, 1,000, 10,000 and 100,000. The process time is measured by calculating the elapsed time between when the system starts scanning data and when all results are returned.

With both datasets, we observed that the kNN query in G-HBase outperformed all other kNN query methods. Note that response times of kNN query in G-HBase are proportional to



(a) High-density in T-Drive

(b) Low-density in T-Drive

(c) High-density in OSM Nodes

(d) Low-density in OSM Nodes

Figure 4.12: R-Tree searching overhead

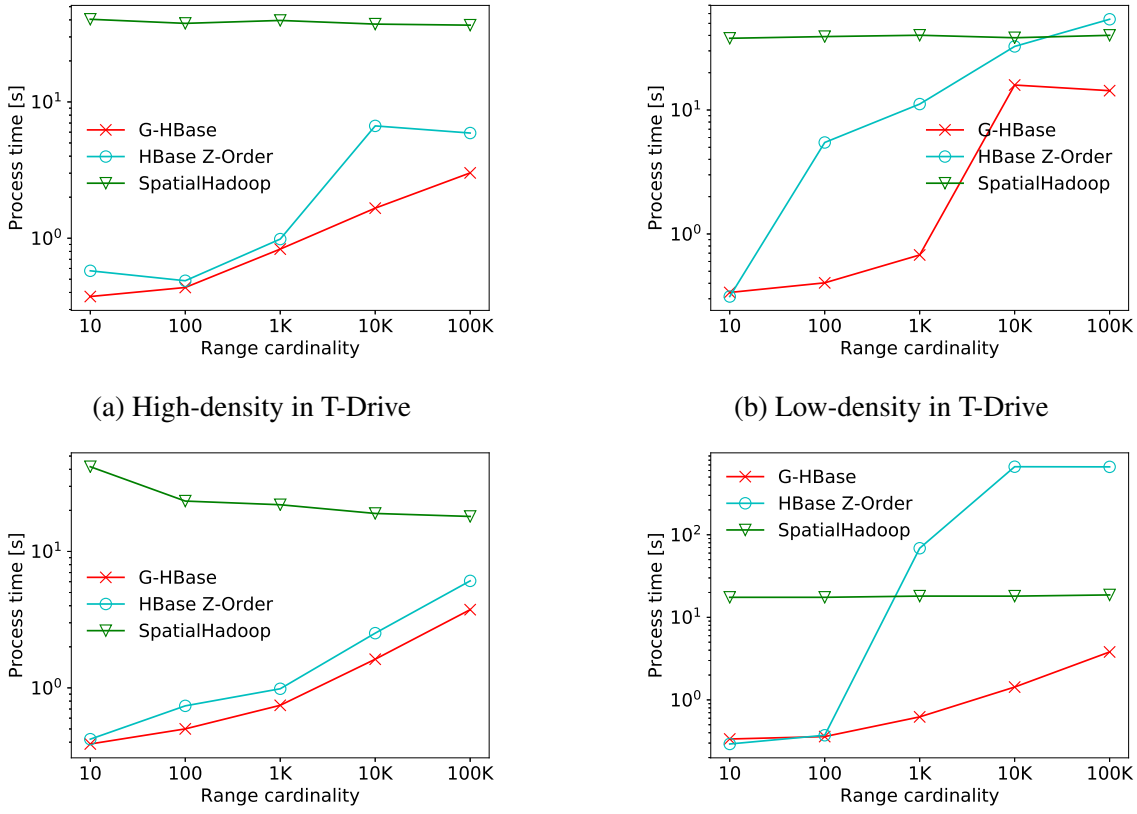
k and about tenfold to hundredfold lower than queries using MapReduce in SpatialHadoop. Queries using MapReduce presented the poor performance regardless of the k because of the brute force parallel scan of the entire table. In G-HBase, by using R-Tree, we can define the regions that may store the results before processing kNN, then only process kNN in those regions, so we do not need to scan the whole regions. The overhead for searching on R-Tree in G-HBase is only about 20 milliseconds as in Fig. 4.12, whereas MapReduce processing has overhead for the startup, cleanup, shuffling and sorting tasks, which are always cost at least one or two seconds for each task.

Experimental results differed in performance, depending on whether high-density or low-density points were being processed in either T-Drive or OSM Nodes dataset. With the T-Drive dataset, for low-density points and larger values of k, the queries could not find sufficient neighbors during the initial search. Therefore, queries had to search again over a larger area, which led to higher latency. By using the R-Tree to search rectangles near the

query point, the queries could reduce the scanning of unrelated areas, thereby achieving an improved performance in G-HBase.

R-Tree searching overhead. We also measured the overhead involved in the R-Tree searching step. Figure 4.12 shows the performance of kNN query in G-HBase with an R-Tree searching overhead. As shown in the figure, the cost for the R-Tree searching is around 20 milliseconds, which account for 0.21% to 5.71% of the total process time of kNN query in G-HBase. The overhead does not change much when we increase k. We only insert big rectangles representing dataset partitions instead of whole points in the dataset into the R-Tree, implying only a small number of nodes and a correspondingly small cost for the R-Tree searching. Furthermore, because the R-Tree is small, we can store it in memory and reduce the reading time latency.

4.5.4 Range Query



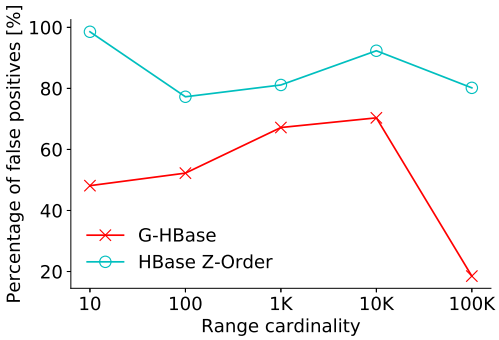
(a) High-density in T-Drive

(b) Low-density in T-Drive

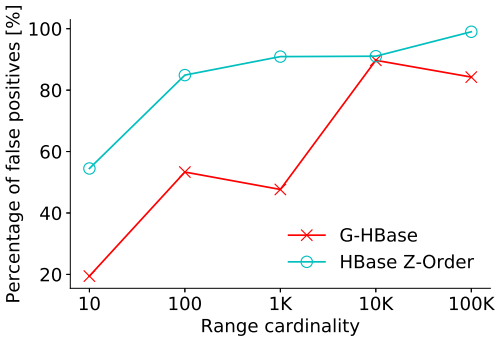
(c) High-density in OSM Nodes

(d) Low-density in OSM Nodes

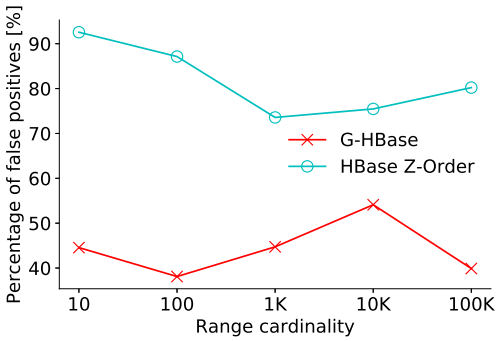
Figure 4.13: Performance of range queries



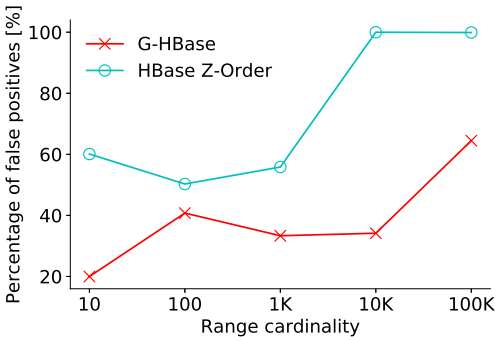
(a) High-density in T-Drive



(b) Low-density in T-Drive



(c) High-density in OSM Nodes



(d) Low-density in OSM Nodes

Figure 4.14: False positives in range queries

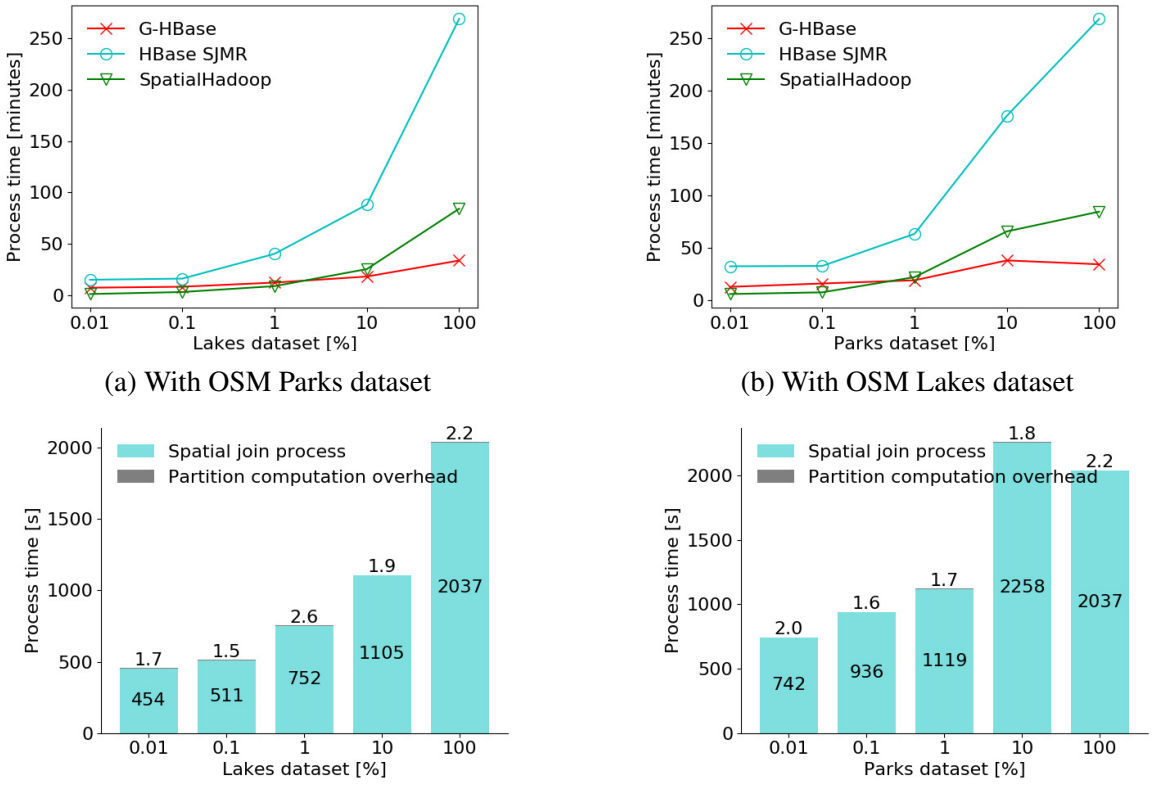
We evaluated the performance of the range query in G-HBase in comparison with SpatialHadoop and the range query using Z-order which scans between start geohash and end geohash of the range. For evaluation, we randomly sampled query ranges with the range cardinality varying from 10 to 100K. Same as the kNN query experiment, we conducted range query experiment in high-density and low-density areas.

The process times of three range query types with T-Drive and OSM Nodes datasets were plotted in Fig. 4.13. Query performance of G-HBase differs according to datasets and the range cardinality, but in all cases, the query in G-HBase outperformed the query in SpatialHadoop. Even though in most cases, the query using Z-order showed better performance than SpatialHadoop, it still had higher process time than in G-HBase, especially for queries in low-density areas. The main reason for the worse performance of Z-order is the false positive scans in the filter step.

Figure 4.14 plots the percentage of false positives of the query with Z-order and the query in G-HBase. We observed from Fig. 4.13 and Fig. 4.14 that the process times are

proportional to the number of scanned points in the filter step. As illustrated in the figure, using Z-order caused more false positives in the filter step, whereas G-HBase computes the RBG to scan, therefore can improve the performance. For example, in low-density areas in OSM Nodes dataset, response times of queries using Z-order were exceptionally high. The number of scanned points in these areas is also very high with a large number of false positives, causing the decrease of query performance.

4.5.5 Spatial Join



(a) With OSM Parks dataset

(b) With OSM Lakes dataset

(c) Overhead with OSM Parks dataset

(d) Overhead with OSM Lakes dataset

Figure 4.15: Performance of spatial join queries

We used OSM Parks and OSM Lakes datasets with various sizes as described in Section 4.5.2 to conduct spatial join experiments. Each extracted dataset from OSM Lakes dataset is joined with OSM Parks dataset and vice versa. Figure 4.15 exhibits performance of the BGRP join algorithm in G-HBase in comparison with SJMR in HBase and SpatialHadoop. In experiments in Fig. 4.15a, we joined the OSM Parks dataset with the extracted datasets from OSM Lakes dataset (as described in Table 4.2). Figure 4.15b shows results of the

experiments in which we joined the OSM Lakes dataset with all the extracted datasets from OSM Parks dataset (as described in Table 4.3).

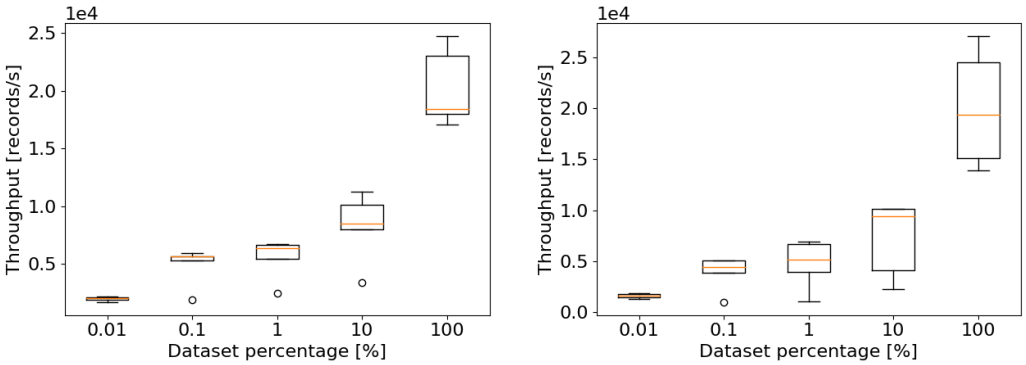
In both cases, we observed that using the BGRP join algorithm in G-HBase showed 2-fold to 7-fold speedup over the SJMR algorithm in HBase. SJMR gave a poor performance as it uses a uniform grid to partition, causing higher response times in some partitions which have a large number of objects to join. Meanwhile, G-HBase partitions data based on data density by using BGR Partitioning, therefore can reduce the influence of skew data. Moreover, SJMR algorithm uses both map and reduce tasks, thus costs more for shuffling and sorting between map and reduce.

SpatialHadoop also showed good performance as it uses R-Tree to deal with skew data. When joining a small dataset with a big dataset, SpatialHadoop are more efficient because it has a preprocessing step to reduce the number of partitions, then reduce the number of map tasks. In G-HBase, region partitions from the larger table created more leaves in the BGRP Tree than the smaller one, thus generating more map tasks to be processing. However, when the two input tables are both big datasets, G-HBase provided better performance. G-HBase pre-splits result table based on the leaves in the BGRP Tree, so it can distribute the insertion into multiple regions, reducing the cost of inserting a huge number of resulted objects into HBase.

BGR Partitioning overhead. We also measured the overhead of the BGR Partitioning in spatial join query (Fig. 4.15c, 4.15d). Same as in kNN query, the cost for the BGR Partitioning is very small (around one second) in comparison with the cost for spatial join process.

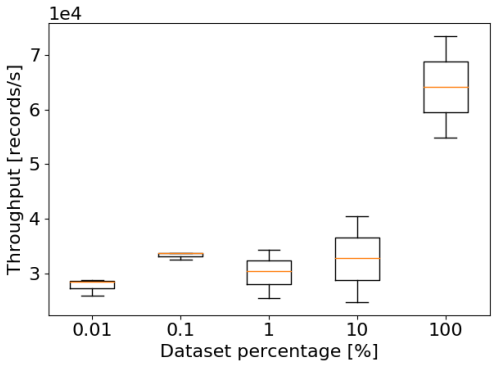
4.5.6 Insertion Throughput

We evaluated the insert performance with both points and non-point objects. With non-point objects, we used OSM Lakes and OSM Parks datasets. With points, we used OSM Nodes dataset. Figure 4.16a, 4.16b, 4.16c show the insertion throughput when inserting OSM Parks, OSM Lakes, OSM Nodes datasets with the number of records varied from 0.01% to 100% of the original datasets. We observed that in all cases, the more records inserted, the higher the insertion throughput. HBase dynamically split regions when the quantity of data in regions excesses a pre-defined threshold, so number of regions increases when more data are inserted. With more regions, data are inserted to multiple regions concurrently, leading to higher insertion throughput. Because generating Geohash is not computationally expensive, G-HBase can sustain a peak insertion throughput of 70,000 records per second.



(a) OSM Parks dataset

(b) OSM Lakes dataset



(c) OSM Nodes dataset

Figure 4.16: Insertion throughput

4.6 Conclusions

We have described G-HBase, a distributed database with native support for geographic data. G-HBase is equipped various fundamental spatial queries such as k nearest neighbors, range query, spatial join in the *Query Processing* component by utilizing our proposed BGR Partitioning in the *Index Structure* component. Experiments with four real-world datasets including both points and non-point data demonstrated the high performance of the compute-intensive spatial queries in comparison with SpatialHadoop and other query algorithms in HBase. The results also showed that the overhead when using the proposed BGR Partitioning is a very small fraction of the process time of the queries.

In the future, we plan to extend this work to support spatiotemporal data in HBase. Beside spatial information, we now can also collect data with temporal information. Spatiotemporal data play an important role in many systems such as intelligent transportation systems, location-based recommendation systems. Those systems pose many new challenges, such as

querying data when its spatial relationships are continuously changing and maintaining high performance while guaranteeing high insertion throughput.

Chapter 5

A Scalable Spatio-temporal Data Storage for Intelligent Transportation Systems Based on HBase

5.1 Introduction

The last few years have witnessed an explosion of spatio-temporal data because of the rapid improvement in location-aware devices such as GPS-enabled devices and sensor-based vehicles. Intelligent transportation systems (ITS) are exploiting a massive volume of real-time sensor data from probe cars and GPS-enabled devices to provide efficient traffic balancing and route recommendation services. This information includes spatial (longitude and latitude) and temporal (time) data. Many other applications such as route planners based on current traffic situations have also taken advantage of these data. Therefore, creating support for high-performance spatio-temporal queries on large-scale data becomes essential in both scientific research and daily life.

To handle the massive amounts of spatio-temporal data, applications need a scalable data storage facility capable of supporting various complicated spatio-temporal queries in real-time. There are two challenges to address these requirements. *The Volume Challenge*: the system needs to have high scalability, resilience, and availability while dealing with large volume of collected data. *The Complexity Challenge* Most moving objects involve geometric computations such as logical operations for spatial relationships that are often highly expensive. Moreover, geometric computations are associated with an exact temporal constraint (time-instant or time-interval) that traditional spatial indexes are inefficient in supporting.

Although MapReduce[26]-based systems such as Hadoop have emerged as an effective solution for handling massive data, the batch-processing technique on the MapReduce framework results in high latency compared with the requirements of a real-time system. Key-value store databases such as HBase [41] have shown promise with the scalability and random real-time read/write capability. However, they are incompetent at storing and processing spatio-temporal data.

In this paper, we present a lightweight spatio-temporal index based on STCode [51], a hierarchical text-encoding algorithm aimed at overcoming the limitations of key-value store databases in supporting spatio-temporal data. There are several advantages for processing spatio-temporal queries using our index structure. First, our proposed index structure can leverage the lexicographical order of rowkey on key-value store databases to store objects that are close to each other spatially and temporally at a nearby place in the database. Second, it utilizes a *column family* for efficient data distribution across a cluster without any additional internal modification requirements. Third, it provides a simple, but efficient, prefix filter for scanning relevant objects. Last but not least, because STCode encoding is not as computationally expensive as other tree-structured indices, the proposed index structure could support continuous updates of spatio-temporal objects.

In this paper, we extend G-HBase to support spatio-temporal data on top of HBase as in Figure 5.1. First, we present the key design using STCode. We also demonstrate how spatio-temporal queries, such as K-nearest neighbors or range queries, exploit the proposed index to enhance query performance. Our experimental results showed the efficiency of our indexing technique as well as real-time query processing capability with response times of less than 1 s.

5.2 Spatiotemporal Index Structure

5.2.1 STCode

The rowkey is designed uniquely to sort and identify rows in HBase. However, spatio-temporal data are represented by three coordinates (longitude, latitude and time), which are all equally important in defining an object. To store spatio-temporal data in HBase, we adopted STCode [51], which provides an algorithm for encoding longitude, latitude, and time into unique codes.

STCode defines latitude $\lambda \in (0, 180)$ and longitude $\varphi \in (0, 360)$. Time is represented in minutes within 1 year $m \in (0, 527040)$. The minute values are chosen to cover the whole year including a leap year $527040 = 366 \times 24 \times 60$. For each dimension, STCode recursively

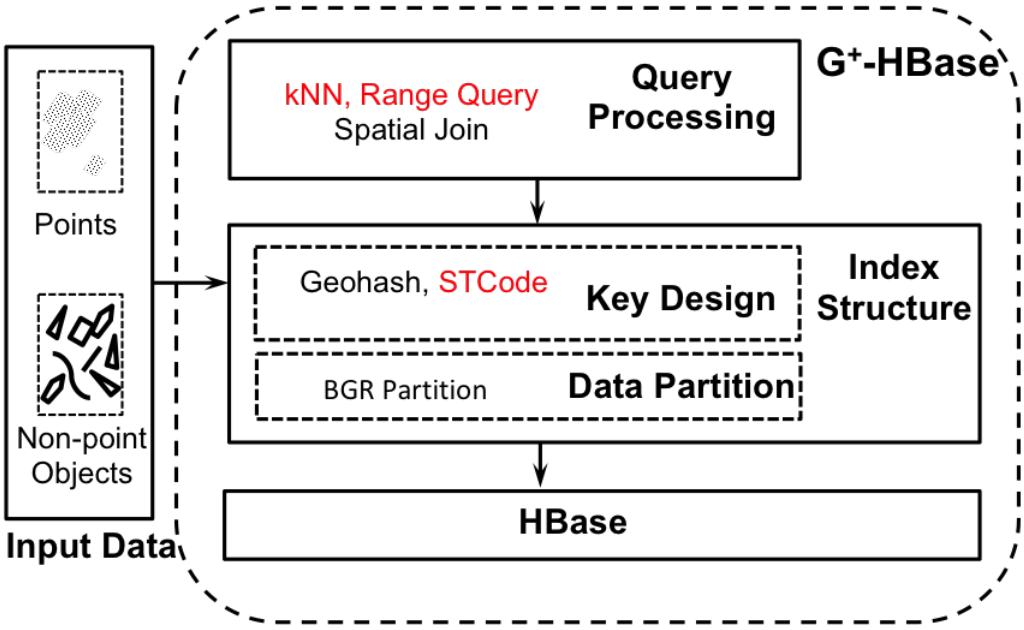


Figure 5.1: G-HBase Extension to Support Spatiotemporal Data

performs binary partition to divide the range into two equal parts, and then assign bit 0 for the left and bit 1 for the right part, respectively.

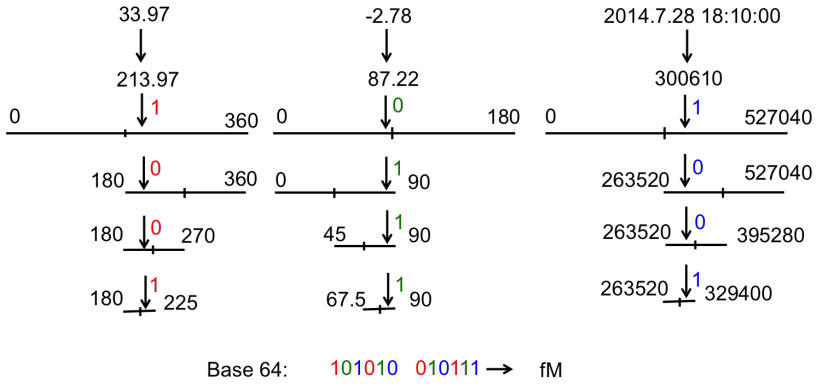


Figure 5.2: STCode generation

Figure 5.2 shows an example of generating an STCode of two characters for a point with longitude, latitude, time of 33.97, -2.78, 2014.7.28 18:10:00, respectively. In the first step, it customizes the values according to its definition, then it has $\lambda = 213.97$, $\varphi = 87.22$, $m = 300610$. The process of binary partitioning the range and selecting a bit is repeated until the desired precision is achieved. STCode uses base64[1] for encoding and each character in base64 is represented by a 6-bit sequence. A bit sequence is calculated by interweaving

[1] http://en.wikipedia.org/wiki/Base64

the longitude, latitude, and time bits with the order of longitude first, then latitude and time. Once the bit sequence is calculated, it is encoded to produce the final code value. In this way, all points with longitude $\lambda \in (202.5, 225)$, latitude $\varphi \in (78.75, 90)$, and time $m \in (296460, 329400)$ will share a common prefix *fM*. Therefore, points sharing the same prefix are close to each other geographically and temporally and all STCodes with the same prefix are included in the cube presented by the prefix.

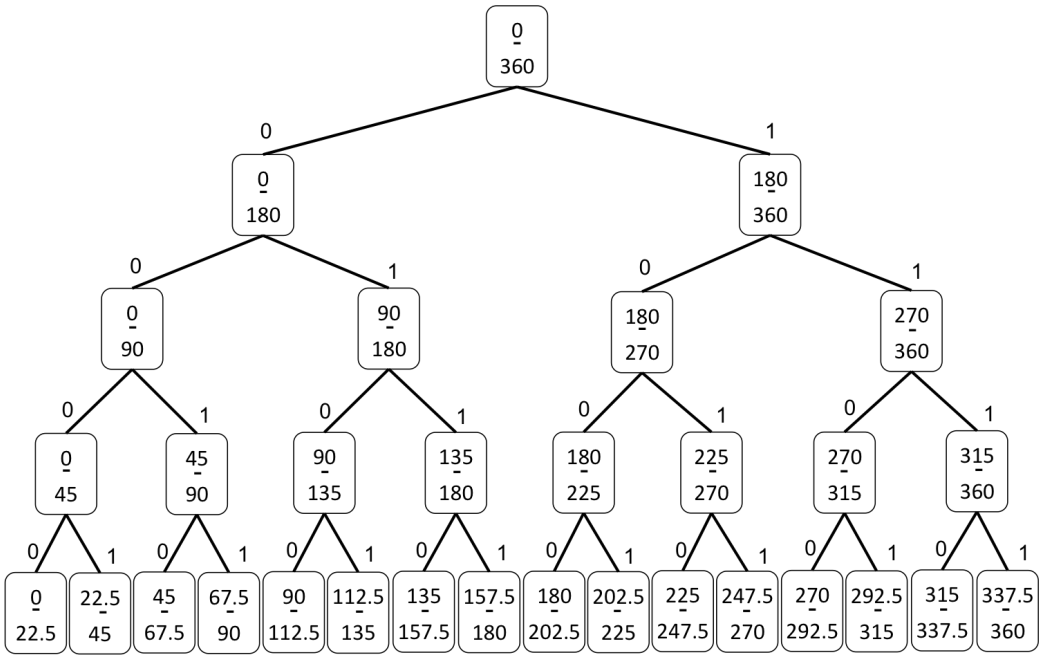


Figure 5.3: STCode tree

The range binary partition technique in STCode can be presented as a tree for each dimension. Figure 5.3 shows the 4-level tree representing the longitude encoding of the example in Figure 5.2. The number of bits for each dimension in sequence corresponds to the number of levels in the tree. For instance, in the above example, longitude provides four bits in sequence; therefore, the tree for longitude has four levels. Each character in base64 encodes a sequence of six bits, and so each dimension provides two bits. Then, we have $length_{code} = l/2$ where l is tree level.

Each node on the tree has size c and the worst-case error $\sigma = c/2$. The tree level l can be calculated from the following equation.

$$l = \left\lfloor \frac{ln(N_{rs}/\sigma)}{ln(2)} \right\rfloor, \tag{5.2.1}$$

where N_{rs} is the size of the root node. In the case of longitude, N_{rs} = 360.

5.2.2 Index Structure

Because HBase keeps rowkeys in lexicographical order, we used STCode as a rowkey on HBase. As mentioned in Section 5.2.1, points that are spatially and temporally close tend to share the same prefix; therefore, we can store nearby points at a near place in HBase. The STCode is suitable for the rowkey because it is not computationally intensive and the same prefix sharing between nearby points makes it easier to find nearest neighbors.

Designing rowkey as STCode provides a simple, but powerful, prefix filter for scanning data in HBase. When scanning points in a specific area, instead of scanning and comparing all points in a table, we just need to scan points that have the same prefix with the area. Therefore, the prefix filter helps reduce the latency for searching and calculating, and also reduce the input/output (I/O) load between client and server. Sometimes, it is difficult to determine the length of prefix for the prefix filter because it depends on the density of the area and on the characteristic of each query types. In Section 5.3, we will explain how to use the prefix filter to improve the spatio-temporal query performance in more detail.

A limitation of STCode is that it cannot encode the year information. This causes the same STCode to be the same time for different years. To handle this limitation, we stored the year information as a column family in HBase. The power of the column family is that it is additional information to identify a row beside the rowkey, and it is the basic unit of physical storage in HBase. Our proposed index structure exploits these advantages to distinguish between data for every year and to store or to compress all data of a year in the same physical storage. Because HBase is a distributed database, the year data can then be distributed to the same machine or to nearby machines, thus decreasing the search latency for close points in the database.

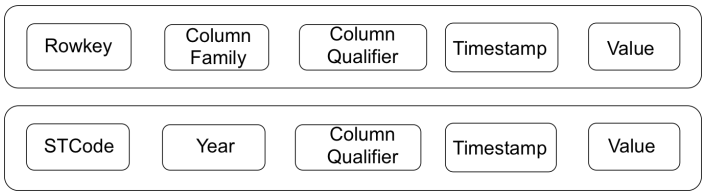


Figure 5.4: The spatio-temporal index structure over HBase

Figure 5.4 describes our proposed index structure. A row in HBase was constructed from rowkey, column family, column qualifier, timestamp, and value. We adopted STCode as rowkey and used the year information as column family. This index structure could be applied easily to existing storage systems without any requirements for modifying their internal implementation.

5.3 Spatio-temporal Queries

5.3.1 Range Query

A spatio-temporal range query is one of the most basic queries in ITS. Algorithm 6 describes the pseudocode for range-query processing. The input of this query is in the three-dimensional (3D) range, where (*minLon, maxLon*), (*minLat, maxLat*), (*minT, maxT*) are the ranges of longitude, latitude, and time, respectively. The output consists of all points included in the query range.

Algorithm 6: Range Query

```
input  : minLon, minLat, minT, maxLon, maxLat, maxT
output : all points inside query range
1: coveringCubes ⟵ getScanningArea(minLon, minLat,
   minT, maxLon, maxLat, maxT)
2: for cube ∈ coveringCubes do
3:     scanResults ⟵ scanWithPrefixFilter(cube)
4:     for point ∈ scanResults do
5:         if query range contains point then
6:             results.add(point)
7: return results
```

Finding the scanning area Before scanning the data in the database, we need to determine the prefixes for the prefix filters by finding the list of cubes that cover the query range. The scanning area is the group of all covering cubes found. The algorithm for this process is shown in Algorithm 7. We first calculated the centroid of the range and the tree levels corresponding to each dimensions based on Equation 5.2.1, then get the maximum of these levels and the corresponding length of code. In STCode, the longer the code, the smaller the cube presented by that code, so in this way, we get the smallest bounding cube that may cover the query range.

We expand the bounding cube by adding all its 26 adjacent cubes. An example of a bounding cube including adjacent cubes of oT is shown in Figure 5.5. If the bounding cube covers the query range, it returns the list of subcubes that intersect with the query range. In Figure 5.5, there are six subcubes that intersect the query range: $qC, qB, q5, oU, oT, oM$. If the bounding cube does not cover the query range, we continue reducing the length of code until the query range is covered by the bounding cube.

Adding adjacent cubes helps us limit the area to scan in the case the query range intersects with the boundary of STCode's cube. For instance, in Figure 5.5, the query range intersects with the boundary of cubes o and q. When oT does not cover the query range, if we reduce

Algorithm 7: Get scanning area

input : minLon, minLat, minT, maxLon, maxLat, maxT
output : list of cubes in STCode that cover query range

1: $lonLevel \longleftarrow$ lonLevel$(maxLon - minLon)$
2: $latLevel \longleftarrow$ latLevel$(maxLat - minLat)$
3: $timeLevel \longleftarrow$ timeLevel$(maxT - minT)$
4: $centroid \longleftarrow$ getCentroid$(minLon, minLat, minT,$ $maxLon, maxLat, maxT)$
5: $level \longleftarrow \max(lonLevel, latLevel, timeLevel)$
6: **for** $length_{code} \longleftarrow level/2$ **to** *0* **do**
7: $candidate \longleftarrow$ STCode$(centroid, length_{code})$
8: $boundingCubes$.add$(scanCube)$
9: initialize $results$
10: $results$.add$(scanCube)$
11: **for** $ad \in candidate.getAdjacent()$ **do**
12: $boundingCubes$.add(ad)
13: **if** *ad intersects with query range* **then**
14: $results$.add(ad)
15: $boundingCube \longleftarrow$ convexHull$(boundingCubes)$
16: **if** *boundingCube covers query range* **then**
17: return results

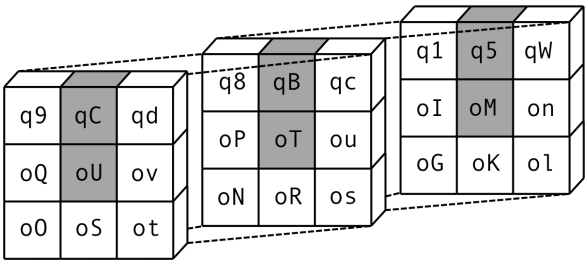


Figure 5.5: Bounding cube with centroid oT and its 26 adjacent cubes

the code length, we have to scan all objects inside the two big cubes o and q. As we check the adjacent cubes before reducing the code length, the scanning area is only six small cubes as depicted in Figure 5.5.

Range filter: Lines 4 to 6 in Algorithm 6 are the process of checking whether the query range contains points in the scanned results. We execute this process, called "range filter", inside each region of HBase, so that we can prune unrelated points sent from regions to client. By using the range filter, instead of returning all points in the scanning area, regions only return points inside the query range to the client; therefore, the I/O load is reduced.

5.3.2 K-Nearest Neighbors Query

The spatio-temporal K-nearest neighbor search (kNN) is more challenging than the spatial kNN because it needs to consider temporal constraints. The input of this query includes a spatial point (longitude, latitude), time interval, and k. The output is k nearest points of the query point in the time interval.

The algorithm for the kNN query is shown in Algorithm 8. To determine the prefix code for the prefix filter, we calculated the tree level corresponding to the query time interval. The scanned points were stored in a fixed-size queue in ascending order of distance to the query point. The size of the queue is fixed as k, so every time the size exceeds k, the queue automatically removes its furthest point from the query point. If not enough nearest neighbors are found, we cut down the length of the prefix and loop until enough k points are found.

Algorithm 8: kNN Query

input : lon, lat, minT, maxT, k
output : k nearest points in time interval from minT to maxT

1: $level \longleftarrow$ minimumTimeLevel$(maxT - minT)$
2: $results \longleftarrow$ MinMaxPriorityQueue(k)
3: **while** *size of results* $< k$ **do**
4: $prefix \longleftarrow$ STCode$(lon, lat, (minT + maxT)/2, level/2)$
5: $scanResults \longleftarrow$ scanWithPrefixFilter$(prefix)$
6: $results$.addAll$(scanResults)$
7: **if** *size of scanResults* $< k$ **then**
8: $level \longleftarrow level - 2$
9: return *results*

Parallel kNN: A kNN query does not use adjacent cubes as in the range query because scanning all 26 adjacent cubes will cause overhead of connecting to region servers. We reduce the I/O load by applying a "divide and conquer" algorithm for the kNN query. When processing the kNN query, the client first sends a scan request with the prefix filter to region servers, gets scan results, then calculates kNN on the received data. In parallel kNN, the kNN is processed inside each region. In this way, instead of returning the unprocessed scan results, each region only returns the kNN inside it to the client. This algorithm is supported by HBase coprocessors, an efficient computational parallelism inspired by Google's BigTable [18] coprocessors.

5.3.3 Average Speed Query

In ITS, we often need aggregation of stored data such as the average speed of cars in a specific area. For example, consider an analysis that uses the average speed of a road link (e.g., from Awajicho station to Tokyo station on Sotobori street between 7:30 and 8:30 am last month) to manage traffic at rush hours. The input of an average speed query consists of a road link and a time interval. The output is the average speed of all cars passing by the query road link in the query time interval.

We use a range query to retrieve all points in the input segment within the input time interval. Points are sorted based on car id and points with the same car id are sorted based on timestamp. The speed of each car is calculated as in Equation 5.3.1.

$$s = \frac{\sum_{i=1}^{n} \Delta d_i}{\sum_{i=1}^{n} \Delta t_i}, \tag{5.3.1}$$

where Δd_i and Δt_i are distance and time intervals between two consecutive points of the car, respectively, and n is the number of distance intervals of the car in the query segment within the query time interval. To eliminate parking time, we only calculate when the distance interval is greater than 0.

5.4 Experimental Results

We built a cluster comprising 1 HMaster, 60 region servers and 3 Zookeeper quorums to conduct experiments with HBase. Each node had 1 virtual core, 8 GB memory and a 64 GB hard drive. The operating system on the nodes was CentOS 7.0 (64 bit). We used Apache HBase 0.98.7 with Apache Hadoop 2.4.1 and Zookeeper 3.4.6. The replication factor on HDFS was set to 2 on each datanode.

We used the real-world dataset, T-Drive [115, 114], in the experiments. T-Drive was generated by 30,000 taxis in Beijing over a period of 3 months. The total number of records in this dataset was 17,762,390. It required about 700 MB and was split into 16 regions for the HBase cluster. The response time of queries was measured by calculating the elapsed time between when the system started scanning data and when all results were returned.

5.4.1 kNN Query

To evaluate the performance of kNN queries, we executed nonparallel and parallel kNN queries (as explained in Section 5.3.2) on T-Drive data. We chose two groups of points to run experiments, one group of high- and one of low-density points. High-density points were

the points in crowded areas that contained many points. Conversely, low-density points were the points in uncongested areas that contained a few points. Figure 5.6 and Figure 5.7 plot the kNN query response times for a kNN query with groups of high- and low-density points, respectively.

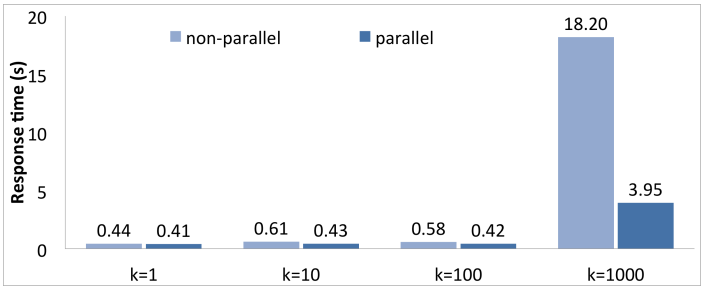


Figure 5.6: Performance of kNN queries with low-density points

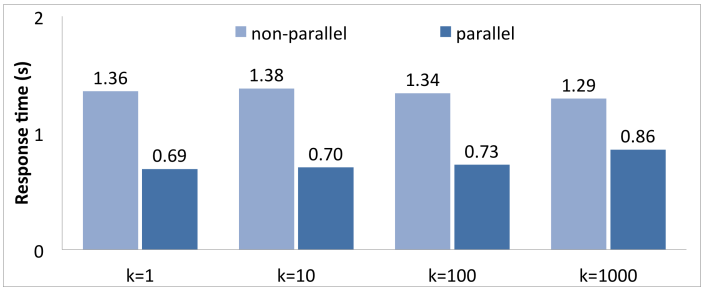


Figure 5.7: Performance of kNN queries with high-density points

As shown in Figure 5.6 and Figure 5.7, for both groups of high- and low-density points, the parallel kNN query outperformed the nonparallel kNN query. In particular, with the group of high-density points, the performance of parallel kNN query was faster by a factor of nearly 2 over the nonparallel query when k was less than 100. Because of the high number of points in a high-density scanning area, the I/O bottleneck between client and region servers could become serious. However, in a parallel kNN query, the region servers only return k nearest points to client and the I/O load is reduced.

For larger k values such as 1000, the response time became much longer. With low-density points and a large k, the query could not find enough neighbors in its early stage. Therefore, it needs to scan larger areas continuously, which might lead to performance deterioration.

5.4.2 Range Query

This experiment was designed to capture the relationship between the query's response time and the number of points in query range and to evaluate the performance of the range query. We made implementations of the range query with and without the range filter (as described in Section 5.3.1). For evaluation, we randomly sampled query ranges with the number of resultant points ranging from 1 to more than 3000. The response times of two range query types were plotted in Figure 5.8. Each circle stands for each examined range query. The red and blue circles correspond to range queries with and without range filter, respectively. Each type of query had a linear regression line and a shaded 95% confidence region for the regression fit. The size of the circle represents the number of points in the scanning area.

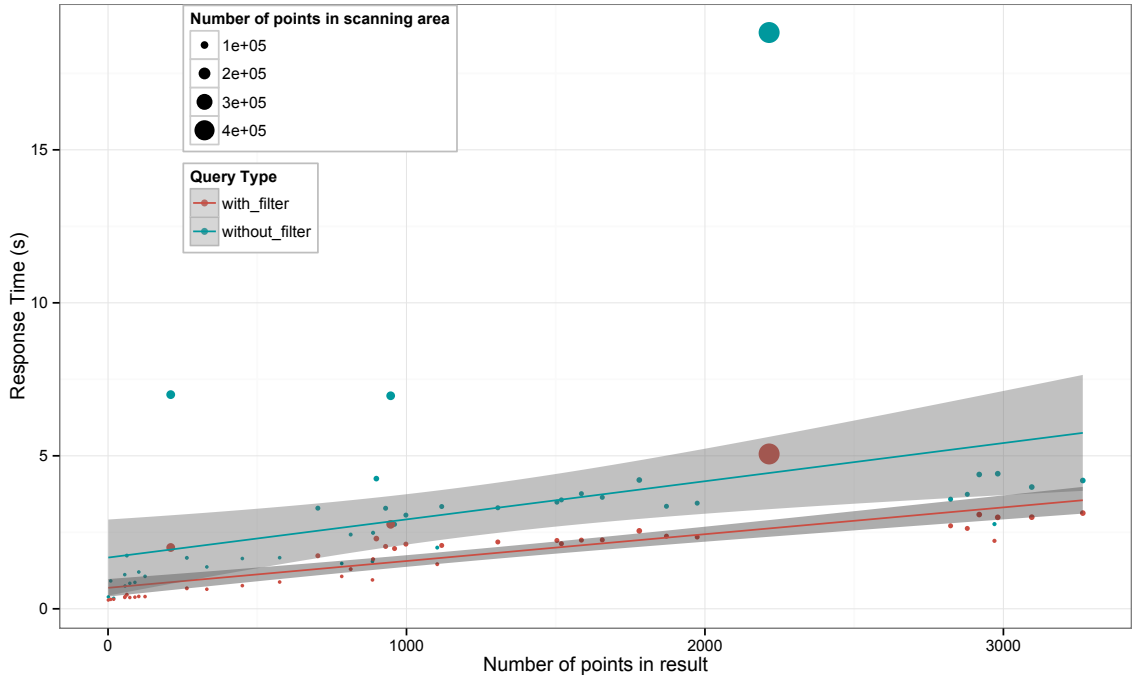


Figure 5.8: The performance of range query

As is evident from the two linear regression lines in the figure, the range query with range filter has better performance than without range filter. Without range filter, the performance of the range query depends significantly on the number of points in the scanning area as the bigger circles have longer response times. On the other hand, as the query with range filter has a narrower confidence region, it remains more stable even with high-density scanning area. The main reason for this stability is because the range filter pushes the process of verifying points in query ranges into region servers; therefore, network overheads are drastically reduced.

Another remarkable result of this experiment was that our indexing structure made responses to queries in just a few seconds. With the range filter, when the number of resultant points was less than 500, most of the queries could be processed in less than 1 s.

5.4.3 Average Speed Query

We executed this query with T-Drive data to capture the average speed of taxis in Beijing. The experiments were performed on three main ring roads of Beijing, i.e., the 2nd, 3rd, and 4th Ring Road. For each ring road, four segments in the North, South, East, and West were chosen, with lengths from 10 to 12 km. The time interval was every two hours from 7:00 AM to 11:00 PM.

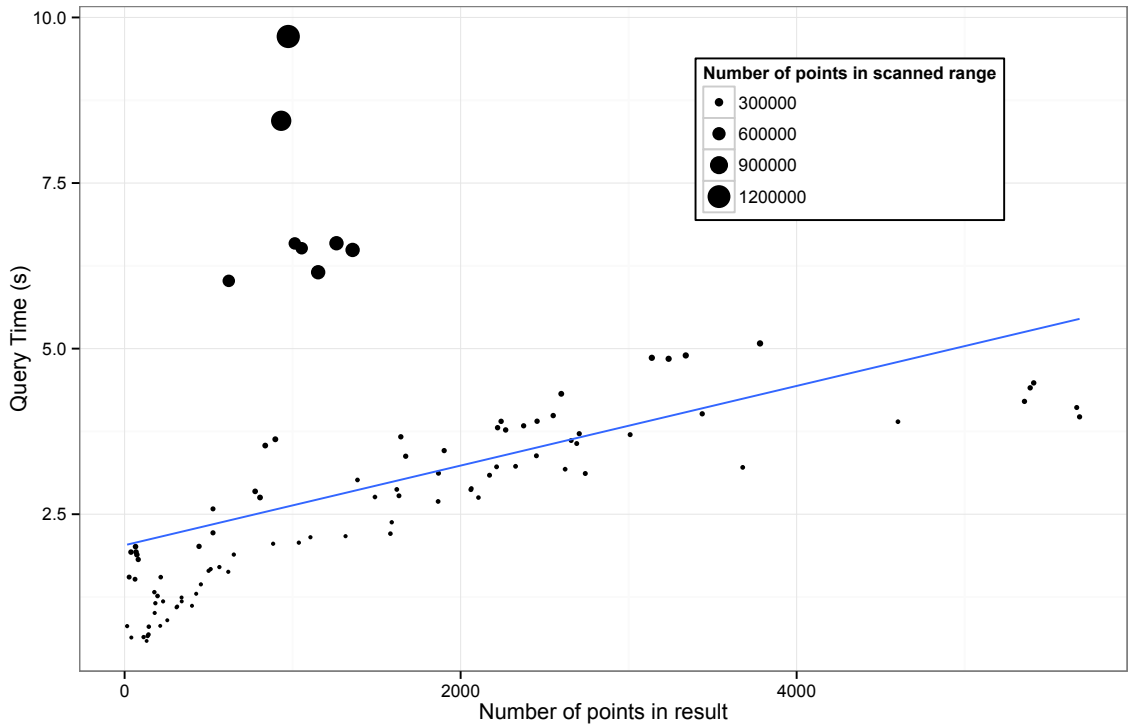


Figure 5.9: Performance of average speed query

As shown in Figure 5.9, the response time of an average speed query was proportional to the number of points in the query range. Most response times were below 5 s. On some roads that were not so crowed and not during rush hours, the response time was less than 1 s.

There were some outliers for which the query's response time (from 6 to 9 s) was higher than for others. As the size of dots in the figure shows the number of points in the scanning area, all outliers had a significantly larger number of points in the scanning area, from 0.6 to

1.2 million points. The main reason for these outliers was the imbalance between the latitude and longitude ranges. Some long segments had a wide latitude range and a narrow longitude range or vice versa, while each STCode had fixed longitude and latitude range depending on the length of code (as explained in Section 5.2.1). The scanning area had to cover all query ranges; therefore, a large STCode was required for a wide query longitude (or latitude) range coverage. If the query latitude (or longitude) range is too narrow, the large area of STCode will contain many points outside the query range, therefore causing time to be consumed on scanning.

5.4.4 Insertion Throughput

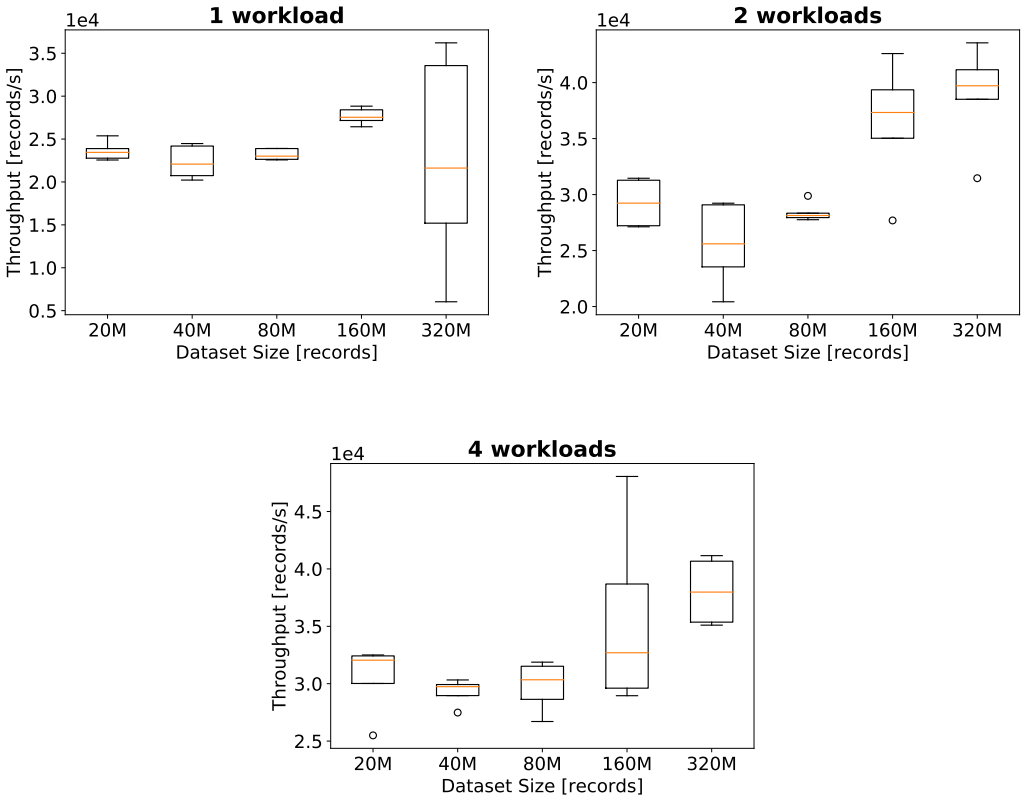


Figure 5.10: Insertion throughput

Supporting high insert throughput for location updates is critical to sustaining a system such as ITS. We evaluated the insert throughput with one, two, and four load generators. Each generator created 10,000 inserts per second. We ran the load generators simultaneously on different machines. The size of insertion data was varied from 20 to 320 million inserts.

We used a Zipfian distribution to generate spatially skewed data. We evaluated the insertion using YCSB benchmarking tool [23].

The results were summarized in Figure 5.10. We observed that the system can sustain a peak throughput of nearly 50K location updates per second. When increasing the size of insertion datasets, in most cases, the throughput also increased. However, there is performance deterioration when the dataset size becomes larger (e.g. inserting 320M dataset with one workload). The main reason for this case is the cost associated with the splitting a region which blocks other operations until its completion. When a dataset is large, the number of regions is also large. Thus, in some cases when splitting multiple regions, the system has to wait until all the region splits finished to continue inserting. HBase provides pre-splitting techniques to avoid this problem. However, in a system where location devices continuously register their location updates, we assume that data are not known beforehand, which makes pre-splitting data very difficult.

5.5 Conclusion

In this paper, we proposed an efficient spatio-temporal index structure on top of HBase, a structure we consider to be essential in ITS. The efficiency of our index structure and the query algorithm was demonstrated through several experiments. Our evaluation showed real-time performance of common spatio-temporal queries such as kNN, range query with response times of less than 1 s. Advanced queries for ITS such as average speed query could also be processed in just a few seconds. In future work, we plan to consider more spatio-temporal queries, e.g. a convex hull, spatial join, as long as other high-performance queries and analytics of ITSs such as online traffic monitoring queries or path planner, and accident detector. Another ongoing work is further query design to handle the imbalance among ranges of 3D.

Chapter 6

Efficient Distributed Spatiotemporal Index for Parallel Top-k Frequent Spatiotemporal Terms Query

6.1 Introduction

Several online social network media, such as Twitter, Instagram, and Facebook have recently become very popular among users. Such microblogs have provided a large volume of user-generated geo-tagged data, which can be exploited in several data analytic systems, e.g., events detection system, location-based recommendation, etc. In this study, we focus on processing top-k frequent spatiotemporal terms (kFST) query, a basic analytic query on geo-tagged social data. The purpose of the query is to find the k most frequent terms among the posts in a specified spatiotemporal region. For example, a recommendation system may want to know which terms have been popular in an area every Friday night to enhance the recommendation at that area on Friday night.

There are some studies addressing the kFST query on stream data in the main memory such as [100, 52]. However, this work targets large historical geo-tagged social data which do not fit in memory. The goal of the study is to provide a high-performance kFST query on huge geo-tagged social datasets with minimum storage requirement but still guarantee the query accuracy. To handle a large volume of data, key–value stores (KVSs) such as HBase [41, 28], with their scalability, fault tolerance, and availability, have shown promise. However, they do not have native support for spatiotemporal data and queries, therefore it is necessary to design a spatiotemporal index and query processing algorithms to adapt the KVS structure.

Ahmed et al [2] introduced an index using R-Tree and sorted terms list (STL) to support the kFST query on historical data. The authors first use an R-Tree to index data geographically, then in each node inside the tree, they build an STL of all terms the node. The STLs are stored in a KVS with an STL id number as rowkey. To reduce space requirement and improve query performance, they only store a part of STLs by estimating STL length needed for top-k calculation before storing. Even though they achieved performance improvement, they have large space requirement for both R-Tree in memory and STLs in KVS. Moreover, the STL length is estimated when indexing with fixed query selectivity and k, thus the index is not flexible with different query inputs.

In this work, we further extend G^+-HBase to support kFST on geotagged socical data as in Figure 6.1. To reduce extra in-memory spatiotemporal index and to support spatiotemporal queries efficiently, we propose a spatiotemporal index which transforms spatial and temporal coordinates into unique codes, to generate rowkeys in KVSs. Some works such as [63, 89, 59] use Z-ordering to generate spatiotemporal rowkeys in KVSs, but they uniformly partition the data space whereas spatiotemporal data are often skewed. MD-HBase [85] couples Z-ordering with trie-based KD-Tree and Quadtree to partition and index multi-dimensional data in HBase. Even though the trie-based trees can capture data distribution statistics, they still cause unbalanced partitioning and even empty partitions in some cases, leading to unbalanced parallel processing when querying. In our proposed spatiotemporal index, we first build a Balanced Partition Tree (BP-Tree) to balance the data partition across distributed clusters. Then we propose a Balanced Partition Code (BP-Code) based on BP-Tree to spatially and temporally generate rowkeys for spatiotemporal data on KVSs.

We also present parallel algorithms to process the aggregating computation based on the distributed spatiotemporal index. We apply data localization by bringing STL computation into storage servers to process in parallel. To further improve the query performance, when querying, we do on-the-fly estimation of the necessary STLs length and send only a part of STLs to the client, reducing I/O between storage servers and the client.

We experimentally evaluated the proposed parallel kFST queries by using large datasets of real Twitter data with various query selectivities and k values. Experimental evidence shows the improvement in performance of the parallel query with shortened length is compared with the parallel query with full length and the nonparallel query. We observed lower space requirements but better query performance of our approach when compared with the approach in [2]. Moreover, because we do on-the-fly estimation, our proposed query is more flexible with various query inputs in experimental results.

This paper is organized as follows. Section 6.2 provides the problem definition. The design of our distributed index is presented in Section 6.3, followed by details of kFST

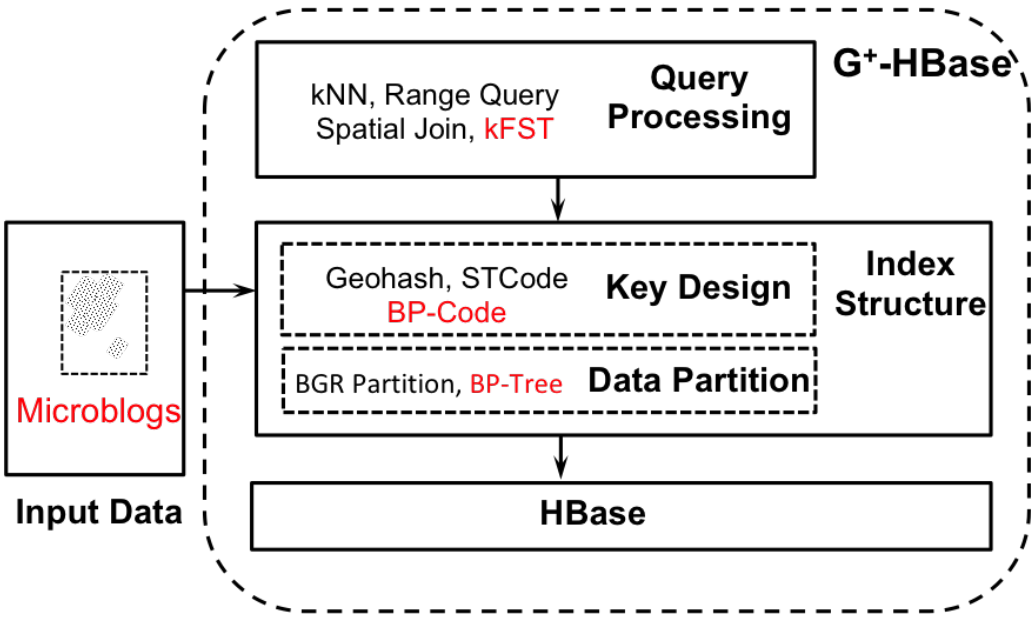


Figure 6.1: G-HBase Extension to Support Spatiotemporal Textual Data

query workflow and proposed parallel queries in Section 6.4, 6.5, respectively. Experimental evaluation is discussed in Section 6.6. Finally, Section 6.7 concludes the paper.

6.2 Problem Definition

Let $D = \{o_1, o_2, ..., o_N\}$ be a dataset with N objects. Each object $o_i \in D$ corresponds to a post in microblogs. Each post includes a tuple of 3 attributes $< id, ST, Terms >$. $o.id$ is a unique id number of the post. $o.ST$ is a 3-dimentional point that provides the spatiotemporal information of the post in space and time. $o.ST$ is often represented by a triplet (x, y, z) where x, y, z are longitude, latitude and time, respectively. $o.Terms$ denotes a list $\{t_1, t_2, ...\}$ of terms in the post. The list can contains duplicated terms.

Let T be list of distinct terms in D. The frequency of a term $t \in T$ is denoted as $f(t) = \{\sum_{i=1}^{N} f_{o_i}(t)\}$, where $f_{o_i}(t)$ is the number of time t appears in o_i. The frequency of a term t in a given spatiotemporal region R is denoted as $f_R(t) = \{\sum f_{o_i}(t) | o_i.ST \in R\}$

Definition of kFST Query: Input of a kFST Query is a tuple $< R_Q, k >$, where R_Q is a spatiotemporal range and k is number of output terms. Let $D_{R_Q} = \{o_i | o_i.ST \in R_Q\}$ be list of all objects in R_Q and T_{R_Q} be list of distinct terms in D_{R_Q}. Output of the query is a list of k terms $\{t_1, t_2, ..., t_k\}$ whose frequencies $f_{R_Q}(t_1), f_{R_Q}(t_2), ..., f_{R_Q}(t_k)$ are the largest among all terms in T_{R_Q}.

6.3 Distributed Spatiotemporal Index Structure

Key-value store (KVS) stores data as a sorted list of key-value pairs. Rows are identified and sorted according to their unique rowkeys, so the rowkey plays an important role for inserting and scanning data. Designing rowkeys is one of the most important steps while designing data schema using KVS.

Apache HBase [41] is a popular distributed KVS inspired by Google BigTable [18] implementation. The rowkey is designed uniquely to sort and identify rows in HBase. However, spatiotemporal data are represented by three coordinates (longitude, latitude and time), which are all equally important in defining an object. To store spatiotemporal data in HBase, linearization technique such as space-filling curves [95] could be applied. Many techniques such as Geohash, STCode [51] use Z-ordering [81], an example of a space-filling curve, to transform multi-dimensional data points into unique codes to store data into KVSs [60], [63], [38]. However, linearization alone is not enough for efficient spatiotemporal query processing; a linearization is based on uniform space partition whereas spatiotemporal data are often skewed.

MD-HBase [85] uses KD-Tree [11] and Quadtree [37] to partition data and Z-ordering to transform multi-dimensional data into one-dimensional data. The trees split the multi-dimensional space recursively into subspaces and organize these subspaces as a search tree. The split happens when the number of data points in a subspace exceeds a threshold. A trie-based approach and a point-based approach [95] are two common approaches to split a subspace. The trie-based approach splits the space at the mid-point of a dimension, resulting in equal size splits while the point-based technique splits the space by the median of data points, resulting subspaces with an equal number of data points. To couple with Z-ordering, MD-HBase employs a trie-based approach because it allows the trees to be coupled with Z-ordering. For example, in Figure 6.3a, space is split using a trie-based KD-Tree and all the smaller subspaces inside the space are connected in a Z-ordering traversal.

However, the tree-based approach is not robust to skew data, leading to unbalanced partition. In this work, we build a tree, namely Balanced Partition Tree (BP-Tree), which partition the space based on the data density. Then we propose a Balanced Partition Code (BP-Code) to encode multi-dimensional data into one-dimensional data based on the partition and distribute data into KVS.

6.3.1 Data Partitioning Approach

In a distributed system such as HBase, data partitioning is an essential initial step to effectively store and process the data. Especially, when querying data in parallel, well-balanced data

distribution plays an important role to improve query performance. There are two major considerations for spatiotemporal data partitioning in KVS. The first consideration is to preserve locality property when distributing data in a KVS to process spatiotemporal queries efficiently. The second consideration is to avoid high density partitioned regions. This is mainly due to potential high data skew in the spatiotemporal dataset, which could cause load unbalances among servers in a cluster environment.

For spatial and spatiotemporal data, there is a long history of data partitioning study with a lot of efficient data structures such as R-Tree family [42, 98, 9]. However, they do not cover the whole data space; making it hard to couple with linearization techniques to index data into one-dimensional keys in KVSs.

Trie-based KD-Tree is efficient to implement with linearization techniques as it covers the whole data space and results in regularly shaped subspaces. Compared to a naive linearization based index structure trie-based KD-Tree can capture data distribution statistics. However, because the trie-based KD-Tree splits the data space at the mid-point of a dimension, it stills cause unbalanced partitioning and even empty partitions in some cases. For instance, in Figure 6.2, the KD-Tree splits the space into six subspaces with the maximum number of data points inside each subspace is 50. However, there are some subspaces have a large difference in the number of data points. For example, the subspaces *1110* has 49 data points whereas its neighbor, *1111*, includes only two data points. The worst case is the empty subspaces *10*, leading to a redundant partition.

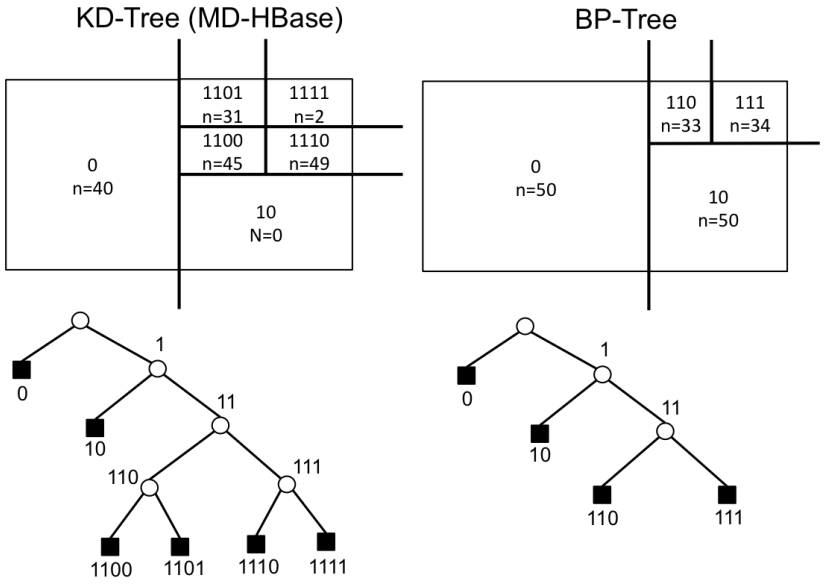


Figure 6.2: BP-Tree and KD-Tree with threshold T= 50: the resulting partition of space and the tree representation.

To support parallel query processing, in this work, we aim to balance the data distribution across the cluster. In HBase, the region forms the basic unit of parallel processing, therefore, we build a BP-Tree to split the data space based on data distribution, then assign each leaf of the tree to a region. The BP-Tree is a binary tree with the following properties.

1. (*Global cover*) Each non-leaf node has two children.
2. (*Split*) Each leaf node stores at most T objects and at least $T/2$ objects.
3. (*Key prefix*) An object with Z-ordering r is stored at a leaf node whose label is a prefix of r.

The first property ensures that the tree covers the whole data space. The second one manages how the prefix tree partition high-density regions. The last property ensures all object in the same region are close together in multi-dimensional space, therefore offer better support for multi-dimensional queries.

When querying kFST, if the input spatiotemporal range R_Q overlaps m leaves in the BP-Tree, the query will be processed in parallel in m regions. However, in multi-dimensional space, the more number of dimensions is, the more number of leaves that R_Q overlaps. If m is larger, the parallel processing becomes inefficient in a small cluster. Moreover, a large number of parallel processing causes more input/output communication between the cluster and the client. Therefore, when inserting data into BP-Tree, we utilize the *local redistribution* algorithm in [45] to improve the storage utilization of the tree, thus reducing the number of parallel units when querying.

Algorithm 9: Insertion in BP-Tree

```
   input : a BP-Tree tree, a data point p
 1: if tree is empty then
 2:     tree.root.add(p)
 3:     return
    /* search the leaf that overlaps the point                    */
 4: leaf ⟵ search(tree, p)
 5: if leaf is full then
 6:     sibling ⟵ getSibling(tree, leaf)
 7:     if sibling is not full then
 8:         redistribute(leaf, sibling, p)
 9:     else
10:         leaf.split(p)
11: else
12:     leaf.add(p)
```

Algorithm 9 describes the insertion of a point into the BP-Tree. The algorithm first searches the tree to find the leaf that overlaps the point. If the leaf is full, instead of splitting the leaf, we check its sibling. If the sibling has not been full yet, we do *local redistribute* between the two nodes to balance the data distribution. Otherwise, the split is conducted on the leaf.

The *local redistribute* approach has a number of advantages. First, it helps reduce the number of splits, thus reduce the height of the tree and time for searching. Second, as mentioned above, it balances the data partition, increases the fanout of the tree. For example, in Figure 6.2, with the same number of data points and threshold T, the fanout of the KD-Tree is 28 whereas the BP-Tree has the fanout of 42.

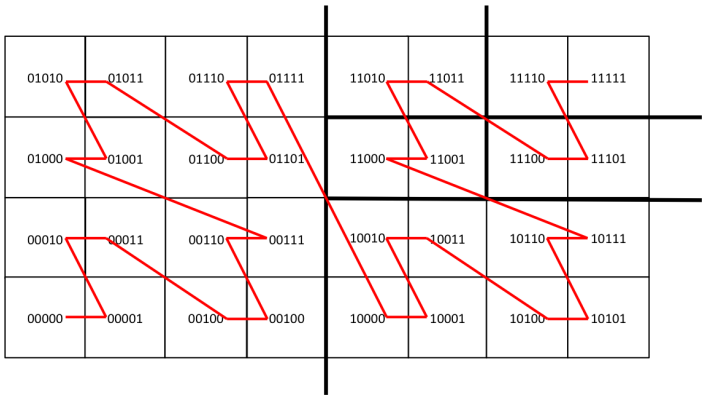
6.3.2 Balanced Partition Code

After partitioning the data using BP-Tree, we need to transform multi-dimensional data into one-dimensional unique codes, then use these codes as rowkeys in KVSs. However, unlike trie-based KD-Trees, BP-Tree partition the space into subspaces based on data density, so linearization techniques cannot be applied directly. To solve this problem, we propose **Local Linearization Approach** to couple linearization techniques with point-based KD-Trees such as BP-Tree.

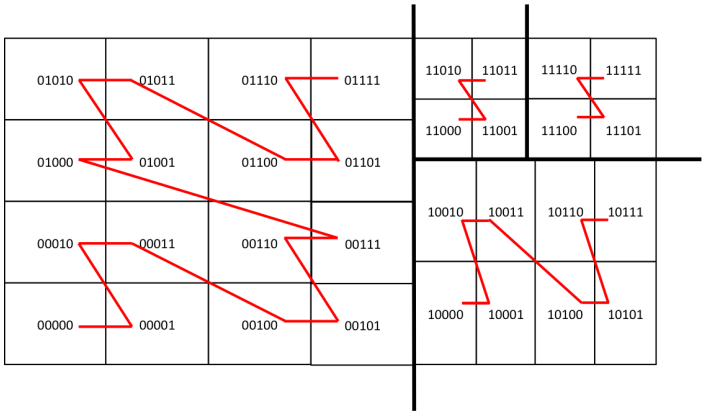
The main idea of the **Local Linearization Approach** is that instead of applying linearization on the whole data space, we only apply it locally inside each subspace in the tree. Figure 6.3b shows an example of the **Local Linearization Approach** when applying Z-ordering to BP-Tree. As shown in the figure, even though the Z-ordering values have different sizes of rectangles in different subspace, they are continuous inside subspace. The advantage of this approach is that we can preserve the locality property of the data inside each data partition, therefore can support efficient multi-dimensional queries.

Using the **Local Linearization Approach**, we propose a Balanced Partition Code (BP-Code) to transform multi-dimensional data into one-dimensional data. The BP-Code derives a code for a data point from a BP-Tree, so it can capture the data distribution statistics. Moreover, using BP-Code helps ensure the *key prefix* property of BP-Tree, which means all data in the same data partition shares the same prefix of their bits string.

There are three steps to generate a BP-Code for a point. Algorithm 10 explains the generation of a BP-Code. With an input of a data point, a BP-Tree and a desired length of the BP-Code, the algorithm will return a BP-Code corresponding to the input point with the desired length. In the first step, we search on the tree to find a leaf that overlaps the point. Thanks to the *lobal cover* property of the BP-Tree, we always can find the leaf. In the next step, the algorithm derives a bit string corresponding to the point from the leaf. Start from



(a) Z-ordering in MD-HBase



(b) Z-ordering in BP-Code

Figure 6.3: Z-ordering in MD-HBase and BP-Code

the bits string of the leaf, the algorithm recursively does binary partition on the bounding rectangle of the leaf. If the point is in the left partition, a bit 0 is concatenated to the bits string. Otherwise, a bit 1 is added. This step stops when we get the desired length of the bits string. Because the key in KVSs should not be too long, we use some binary-to-text encoding schemes such as Base32[1] and Base64[2] to encode the bits string to a shorter code in the third step. This step is optional.

[1]https://en.wikipedia.org/wiki/Base32
[2]https://en.wikipedia.org/wiki/Base64

Algorithm 10: BP-Code Generation

input : a BP-Tree *tree*, a data point p, length of the code l
output : a BP-Code corresponding to the point p with length l

```
/* Step 1: search the leaf that overlaps the point p */
```
1: $leaf \longleftarrow$search$(tree, p)$
```
/* Step 2: derive a bits string corresponding to the point p from
   the leaf */
```
2: $bitString \longleftarrow leaf$.getBitString()
3: $br \longleftarrow leaf$.getBoundingRectangle()
4: $bitLength \longleftarrow$calculateBitLength(l)
5: **for** $i = bitString.length$; $i \leq bitLength$; $i = i+1$ **do**
6: $left, right$ = br.binaryPartition()
7: **if** *left overlaps p* **then**
8: $bitString$.concatBit(0)
9: $br \longleftarrow left$
10: **else**
11: $bitString$.concatBit(1)
12: $br \longleftarrow right$
```
/* Step 3: encode the bits string into a short code (optional)
   */
```
13: $bpcode \longleftarrow$ encode$(bitString)$

6.4 Top k Frequent Spatiotemporal Terms Computation Workflow

The main goal of this work is to provide a high-performance kFST query that can take advantage of parallel processing on HBase clusters. To achieve that goal, it is essential to identify time-consuming phases, break them down into small tasks, and process these tasks in parallel. An intuitive approach is to index and partition the data into regions spatiotemporally and process the computation in each region in parallel. The query processing problem then becomes the problem in designing querying methods that can run on these regions independently, while preserving the accuracy of the computation. We propose the following steps on processing a kFST query on HBase using BP-Code as rowkey, as shown in Algorithm 11.

There are two main phases to process a kFST query. In the first phase, a spatiotemporal range query is processed to get all objects in the query range. Then, top k frequent terms are calculated based on all the objects achieved in the first phase.

Algorithm 11: Workflow of kFST query using BP-Code as rowkey in HBase

input : a spatiotemporal range R_Q, a number k
output : top k frequent terms in the range R_Q
A. Estimate λ (optional);
B. Calculate RBC;
C. Scan data;
D. Apply range filter;
E. Calculate STL;
F. Apply top-k algorithms (optional);

6.4.1 Spatiotemporal Range Query Phase

Spatiotemporal range query processing strategy includes three steps B, C, D in Algorithm 11. Query range can be simply processed by scanning between the minimum and maximum Z-ordering values of the range if Z-ordering is used as the index. However, BP-Code does not preserve Z-ordering globally; some Z-ordering values in different regions do not continue. Therefore, instead of scanning globally, we scan data locally inside each region.

Thanks to the *Local Linearization Approach* in BP-Code, we can scan between the minimum and maximum Z-ordering values of the range inside each region. However, Z-ordering may cause many false positives in this scan. For example, in Fig. 6.4, the query range starts with code 1x and ends with code 62, causing a lot of redundant scan in rectangles 2, 3, 4, 5. Therefore, we calculate range bounding codes (RBC) of the query range inside each region to reduce the false positives when scanning data in step C.

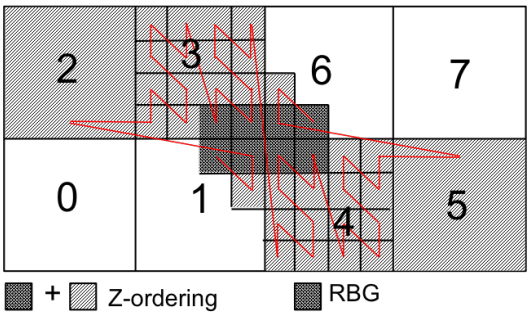


Figure 6.4: Range Bounding Geohashes

The RBC is based on the range bounding geohashes (RBG) [61]. RBG is used for spatial data. Fig. 6.4 shows an example of RBG for range query in two-dimensional space. In this work we have three-dimensional data, so we just simply modified the RBG to calculate RBC. Given a query range R_Q and a region *region*, we first calculate the overlap between R_Q and *region*. Then we calculate MinRangeHash g_x, g_y, g_z corresponding to the longitude, latitude, time edge of the overlap (for details see [61]). The calculation starts with computing the minimum code c of the three MinRangeHash. Then c is expanded by adding all its neighbors inside the region into a list until the overlap is covered by the list (Algorithm 12)

Algorithm 13 describes the details of step B, calculate RBC. It starts with searching all the leaves in the BP-Tree that overlap the input query range R_Q. As each leaf corresponding

Algorithm 12: calculateRBC($R_Q, region$)

input : a range R_Q, the region *region*
output : list of BP-Codes inside the region *region* that cover the overlap between the region *region* and the range R_Q

```
1:  q ⟵ overlap(R_Q, region)
2:  c ⟵ getCentroid(q)
3:  g_x ⟵ MinRangeHash(c, q.maxX − q.minX)
4:  g_y ⟵ MinRangeHash(c, q.maxY − q.minY)
5:  g_z ⟵ MinRangeHash(c, q.maxZ − q.minZ)
6:  g ⟵ min(g_x, g_y, g_z)
7:  results.add(g)
8:  while results do not cover q do
9:      for r ∈ results do
10:         candidates.add(r)
11:         for ad ∈ r.getAdjacentsInRegion(region) do
12:             if ad intersects with q then
13:                 candidates.add(ad)
14:     results ⟵ candidates
15: return results
```

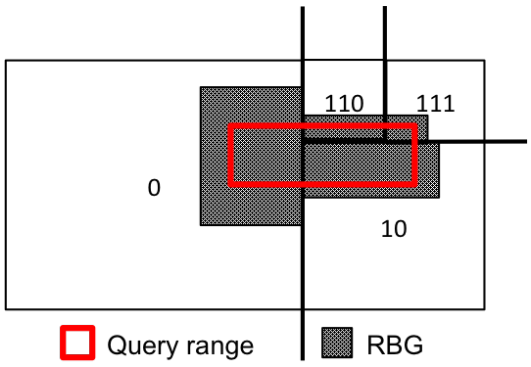


Figure 6.5: RBG of a query range using BP-Code

to a region in HBase, RBC of R_Q in each region is calculated. The scanning area is now all codes in RBC from all the overlap regions. Figure 6.5 shows an example of scanning area with BP-Code.

The output of step C is the results from the scan in step B. It is a set of candidates which possibly are included in the query results. Step D applies a range filter to examine all the candidates and eliminate objects that are not the query range.

Algorithm 13: Get list of RBC for a query range

input : a query range R_Q, a BP-Tree *tree*
output : list of BP-Codes that cover the query range R_Q

```
1: leaves ⟵ tree.search(q)
2: for region ∈ leaves do
3:     results.addAll(calculateRBC(R_Q, region))
4: return results
```

6.4.2 Top k Calculation Phase

After achieving all objects in the query range, in step E, we calculate frequencies of all terms in those objects. Then, the terms are sorted based on their frequency to build an STL. In a nonparallel query, step E is done in the client, after all the scan results were returned from HBase. Results of the query are the first k terms in the STL. In parallel queries, step E is executed in parallel inside each region in HBase. When regions return STLs after step E, the client needs to apply top-k aggregation algorithms on achieved STLs in step F to get final results.

Table 6.1: List of kFST Query Algorithms

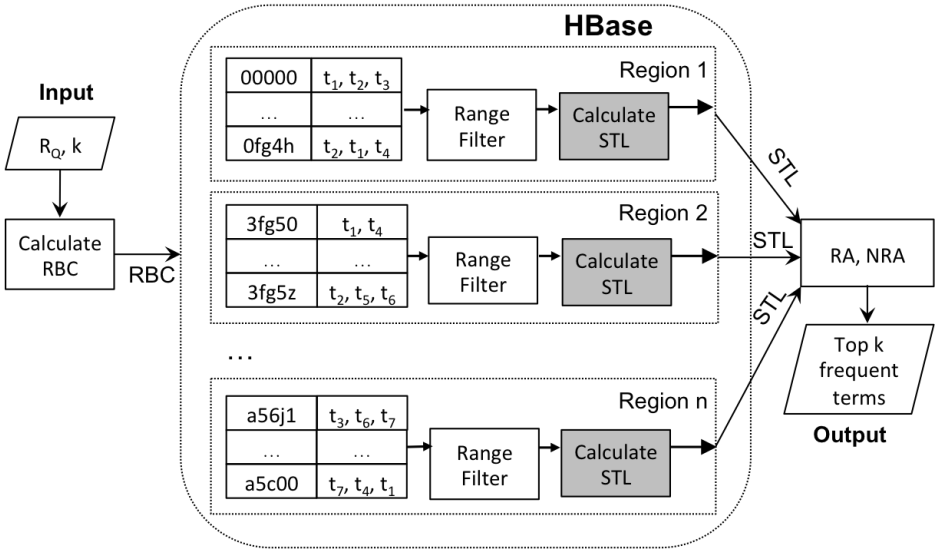
Algorithms	Workflow	Description
BPCodeQuery	B, C, D, E	Nonparallel query using BP-Code
ST-P	B, C, D, E, F	Parallel query with full STL
ST-λP	A, B, C, D, E, F	Parallel query with shortened STL

Table 6.1 lists all algorithms to process a kFST query using BP-Code index in this study. BPCodeQuery is a nonparallel query. We propose two parallel queries $ST-P$, $ST-\lambda P$ which process step E inside storage servers in parallel. Section 6.5 presents the details of proposed parallel queries.

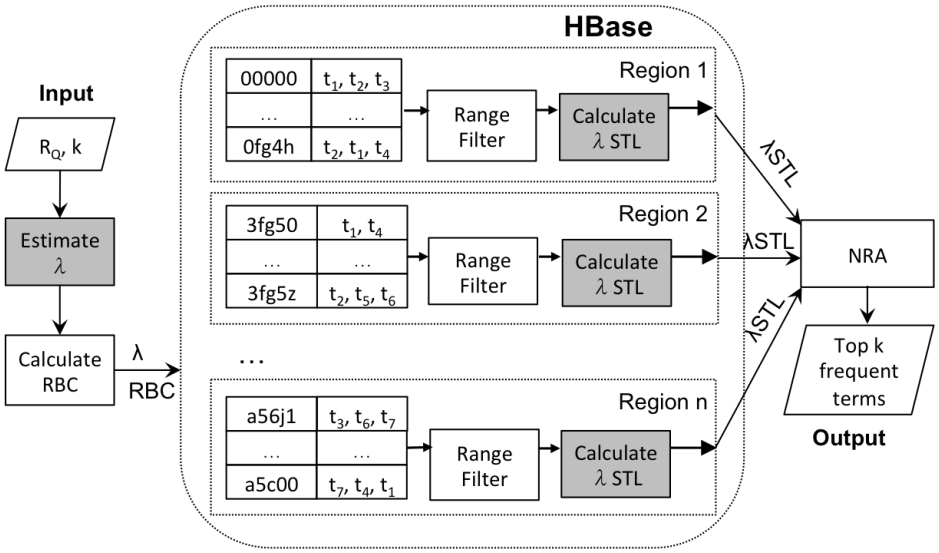
6.5 Proposed Parallel kFST Queries

6.5.1 Parallel query with full STL (ST-P)

As in table 6.1, ST-P includes spatiotemporal range query phase with steps B, C, D and top k calculation phase with steps E, F. As shown in figure 6.6a, step E, STL calculation, is



(a) ST-P Query



(b) ST-λP Query

Figure 6.6: Proposed Parallel kFST Queries

performed in parallel inside data regions. In BPCodeQuery, after spatiotemporal range query phase, all scan results are sent to the client. However, high frequent terms which appear in many objects will be sent multiple times to the client, causing more I/O communication. Moreover, the client has to aggregate and sort terms from all scan results, which is time-consuming. To reduce I/O communication between the client and HBase cluster, and also to avoid aggregating and sorting large volume of data in the client, we utilize data localization by processing STL calculation into storage servers in parallel.

Because the output of step E is STLs from data regions, we need to apply a top-k algorithm to get top-k frequent terms from achieved STLs. There are several top-k algorithms were proposed to calculate top-k items from multiple resources. According to the survey in [49], there are three categories of top-k processing techniques involving accessing multiple data sources. Top k processing techniques in the first category, *Both Sorted and Random Access*, assume the availability of both sorted and random access in all the data sources. The second category, *No Random Access*, assumes random access is not supported and finds the top-k answers by exploiting only sorted accesses. Finally, the *Sorted Access with Controlled Random Probes* category assumes the availability of at least one sorted access source and uses random accesses in a controlled manner. In this work, we use two most popular algorithms, *No Random Access (NRA)* [36] in the second category and *Random Access (RA)* [35, 82] in the first category, to calculate top-k frequent terms from achieved STLs.

RA algorithm

Algorithm 14 describes the details of the RA algorithm. Figure 6.8 shows an example of how to process the NRA algorithm. Consider two input STLs L_1, L_2 sorted by the frequency f_1, f_2 respectively. k is set to two. In the first iteration, with a new term a, the algorithm do random access to L_2 and calculate its frequency value as 170. Similarly, term e is added to the *SeenTerms* list with a frequency value of 98. The threshold θ at this iteration is 178 which is larger than 98, so the algorithm continues. The second step scans a new term b and updates the *SeenTerms* list. In the third step, the frequency value of the k_{th} term in the *SeenTerms* list, e, is larger than the threshold θ. This means no unseen terms has a chance to have frequency value larger than e. Therefore the algorithm can stop. Top k terms in the *SeenTerms* list is returned as the result of the algorithm.

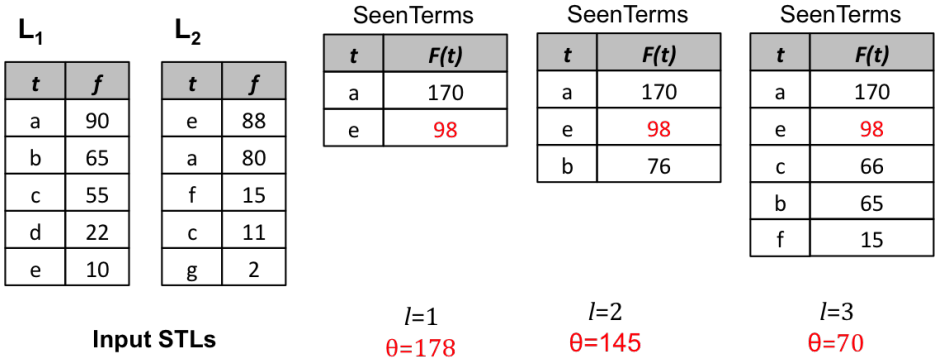


Figure 6.7: RA Algorithm with k=2

Algorithm 14: Random Access Algorithm [35, 82]

input : a list of m STLs $L = \{L_1, L_2, ..., L_m\}$, k
output : top k frequent terms in L
1. Do sorted access in parallel to list L. At each iteration l do:
   - For every new term t, do random access to other lists to find $p_i(o)$ in every other list L_i. Compute the frequency value $F(t) = F(p_1, ..., p_m)$ of term t.
   - Maintain a SeenTerms list which includes all seen terms with their frequency values.
   - Maintain the last seen values $\bar{x}_1^{(l)}$, $\bar{x}_2^{(l)}$, ..., $\bar{x}_m^{(l)}$ in the m lists. Define the threshold value θ to be $F(\bar{x}_1^{(l)}, \bar{x}_2^{(l)}, ..., \bar{x}_m^{(l)})$.
2. Stop when at least k terms have been seen with frequency values at least equal to θ.
3. Return k terms in the SeenTerms list with most frequency values.

NRA algorithm

The details of the NRA algorithm are described in algorithm 15. Figure 6.8 shows an example of how to process the NRA algorithm. Consider two input STLs L_1, L_2 sorted by the frequency f_1, f_2 respectively. k is set to two. In the first iteration, term a has the frequency range of [90, 178] since the value of its known frequency f_1 is 90, while the value of its unknown frequency f_2 cannot exceed 88. The second step scans two terms b, a and updates the score bounds of other seen objects. Since the lower bound of term e does not exceed the upper bound of term b in *Candidates*, the algorithm has to continue. In the third step, the lower bound of term e in $Topk$ is larger than the maximum upper bound in *Candidates*. This means no seen terms in *Candidates* has a chance to have frequency larger than e. Moreover, an upper bound for the frequency of an unseen term at this step is computed as 55+15=70, which is the result of applying F to the last seen frequencies in both sorted lists. Unseen terms also can not have a larger frequency than e, thus the algorithm can stop. $Topk$ is returned as the result of the algorithm.

6.5.2 Parallel query with shortened STL (ST-λP)

Even though ST-P approach reduces I/O load between client and HBase cluster by return STLs instead of objects list, we realized that NRA algorithm only uses a small part of each STL, causing redundant parts in returned STLs. For example, in figure 6.8, even though the input STLs have five terms each, NRA algorithm only needs to access three first terms. We thus propose ST-λP query that exploits this to further improve query performance. Instead of full STLs, ST-λP returns shortened STLs, namely λSTLs, to the client. With input as

Algorithm 15: No Random Access Algorithm [36]

input : a list of m STLs $L = \{L_1, L_2, ..., L_m\}$, k
output : top k frequent terms in L

1. Do sorted access in parallel to list L. At each iteration l do:
   - Maintain the last seen values $\bar{x}_1^{(l)}$, $\bar{x}_2^{(l)}$, ..., $\bar{x}_m^{(l)}$ in the m lists
   - For every term t, compute a lower bound $W^{(l)}(t)$ and a upper bound $B^{(l)}(t)$ by substituting each unknown value $x_i^{(l)}$ with 0, $\bar{x}_i^{(l)}$, respectively. If term t has not been seen at all, $W^{(l)}(t) = F(0,0,...,0)$ and $B^{(l)}(t) = F(\bar{x}_1^{(l)}, \bar{x}_2^{(l)}, ..., \bar{x}_m^{(l)})$.
   - Let $Topk$ be the set of the k terms with the largest lower bound value W seen so far.
   - Let M_k be the k^{th} largest W in $Topk$.
   - Let $Candidates$ be the set of seen terms which are not in $Topk$.
2. Call a term t *viable* if $B(t) > M_k$. Stop at iteration s when
   - At least k distinct terms have been seen.
   - No terms in $Candidates$ is *viable*. This means $\forall t \in Candidates, B^{(s)}(t) \leq M_k$
3. Return $Topk$

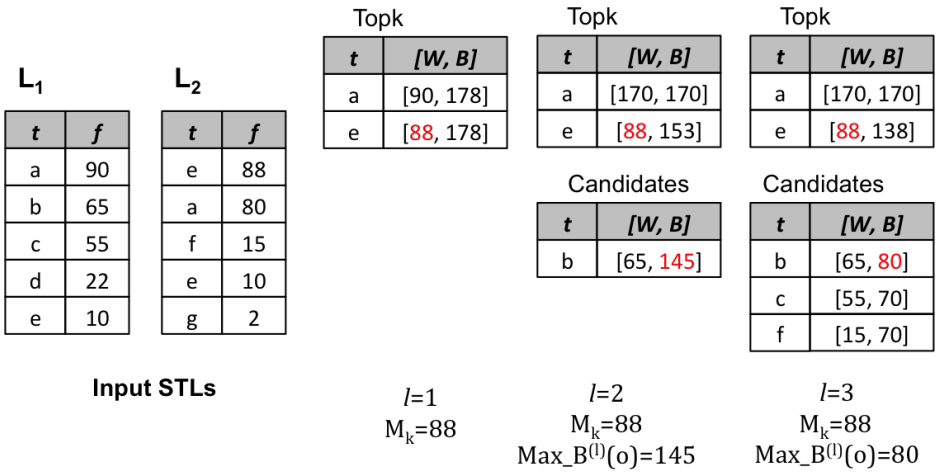


Figure 6.8: NRA Algorithm with k=2

shortened STLs, random access is not supported because it may cause incorrect results. For example, the algorithm may need to do random access to the part which was cut in the shortened STLs. Therefore, in this query, RA algorithm cannot be applied.

The problem now becomes how to estimate the length of shortened STLs (λ). With v is the actual number of terms NRA algorithm accesses, to ensure query accuracy, λ has to larger than v. As shown in table 6.1 and figure 6.6b, ST-λP has additional step A, λ estimation, before scanning in HBase. In this step, λ is carefully estimated based on the input query range R_Q and k.

Estimate STL length

Table 6.2: Parameters

Symbol	Description
n	number of dimensions
μ	average number of terms in an object
c	Zipf parameter
M	total number of regions
N	total number of objects
m	number of regions involved in the query
$I = \{I_1, I_2, ..., I_m\}$	expected number of objects inside the query area from involved regions

λ estimation step takes advantage of the observation that keywords popularity in microblogs follows the Zipf distribution [50, 24, 83]. Let μ is the average number of terms in an object and N is the total number of objects in the whole dataset. Thus, the total number of terms including duplicates in the whole dataset is Nμ. According to the Zipf law, the frequency of a term is inversely proportional to its rank in the frequency list. Therefore, with a collection specific Zipf parameter c, frequency $f(r, N\mu)$ of the r^{th} term in an STL containing Nμ terms can be computed by:

$$f(r, N\mu) = \frac{cN\mu}{r} \tag{6.5.1}$$

Let N_q is the expected number of objects in the input query region R_Q. Similarly, frequency of the k^{th} term in the STL of the query area is:

$$f(k, N_q\mu) = \frac{c_q N_q \mu}{k} = \frac{c_q \sum_{i=1}^{m} I_i \mu}{k} \tag{6.5.2}$$

Let m is number of regions in HBase table that are involved in the query area. In each involved region, let I_i is number of objects in region i that are inside query area. As mentioned in algorithm 15, the upper bound θ for the frequency of a unseen term at iteration r^{th} is sum

of all r^{th} frequency values in STLs from m involved regions. Given m, $I = \{I_1, I_2, ..., I_m\}$, μ, θ at iteration r^{th} is computed as:

$$\theta(r, m, I, \mu) = \sum_{i=1}^{m} \frac{c_i I_i \mu}{r} \tag{6.5.3}$$

The main goal of λ estimation is to calculate the minimum length of STLs which the NRA algorithm iterates. Note that the NRA algorithm stops at the iteration λ^{th} when the lower bound $M_k^{(\lambda)}$ of the k^{th} best term so far is higher than the maximum upper bound $Max\_B^{(\lambda)}(t)$ of terms in *Candidates*. However, computing $M_k^{(\lambda)}$ and $Max\_B^{(\lambda)}(t)$ requires knowledge of the correlation of the term frequencies across lists, which is very difficult to obtain before scanning the data inside HBase.

According to algorithm 15, we have the lower bound $M_k^{(\lambda)}$ is less than the actual frequency of the k^{th} best term, that is, $M_k^{(\lambda)} \leq f(k, \sigma N \mu)$. Also, the maximum upper bound $Max\_B^{(\lambda)}(t)$ of terms in *Candidates* is higher than the upper bound for the frequency of a unseen term, which means $\theta(\lambda, m, I, \mu) \leq Max\_B^{(\lambda)}(t)$. Combining the inequalities we have shown, we have

$$\theta(r, m, I, \mu) \leq Max\_B^{(r)}(t) \leq M_k^{(r)} \leq f(k, N_q \mu) \tag{6.5.4}$$

We understand that this approach may cause underestimation of λ. To reduce the probability of λ underestimation, we assume that λ is proportional to k. Therefore, in this work, we estimate λ that satisfies

$$\theta(\lambda, m, I, \mu) \leq k f(k, N_q \mu) \tag{6.5.5}$$

Estimate I

Estimating λ requires data of number of objects that inside the query area in each involved region $I = \{I_1, I_2, ..., I_m\}$. Let M is total number of regions in a HBase table. In section 6.3, data are uniformly distributed across the cluster, so we assume that all regions have the same number of objects. If the whole dataset has N objects, each region stores N/M objects. Given an input query area R_Q, I_i in region i depends on the proportion between area of region i and the overlap of query area and region area. Let $S_i = \{s_{i,1}, s_{i,2}, ..., s_{i,n}\}$ is size of region R_i and $q_i = \{q_{i,1}, q_{i,2}, ..., q_{i,n}\}$ is size of the overlap (hyper-)rectangle between region R_i and the query.

$$I_i = \frac{N \prod_{j=1}^{n} q_{i,j}}{M \prod_{j=1}^{n} s_{i,j}} \tag{6.5.6}$$

6.6 Experimental Results

6.6.1 Experiment Setup

Computing server: All experiments are performed on a server with 32 Cores, 288 GB memory and 12 TB storage. The operating system for the server is Debian Jessie.

HBase clusters: We evaluate the system on three clusters as in Table 6.3. Each server has two virtual cores, 16 GB memory and a 288 GB hard drive using CentOS 7.0 (64-bit). The virtual servers are created on 4 physical machines. We set up one HMaster and three Zookeeper Quorums using Apache HBase 0.98.7 with Apache Hadoop 2.4.1 and Zookeeper 3.4.6. Replication was set to two on each datanode.

Table 6.3: HBase clusters

Cluster name	Size
Small	10 region servers
Medium	35 region servers
Large	60 region servers

Datasets: We crawled 54,941,496 geotagged, English-based tweets from 2017.07.29 to 2018.01.27 using the Twitter streaming API. The spatial domain covered the whole world. After removing all the stop keywords and swear keywords, each tweet has 8 terms on average. For our experiments, we extracted two datasets as in table 6.4.

Table 6.4: Datasets

Name	Number of tweets	Start	End
15M	15,230,974	2017.07.29	2017.9.11
50M	50,088,665	2017.07.29	2018.01.04

To evaluate the performance of our proposed kFST query processing, we executed the queries using the two datasets, 15M and 50M. We evaluated parallel kFST queries and compared with NRA-STL-Li and BPCodeQuery as baseline methods. The selectivity was varied from 0.001 to 0.016 (based on the number of tweets).

Query Distribution: We evaluated the proposed index structure and queries with various query selectivities. Selectivity σ of a query is n_q/N where n_q is the number of points inside the query and N is the total number of points in the dataset. We varied the selectivities from 0.001 to 0.016. As each region in HBase is a parallel processing unit, we also evaluated the query performance with different numbers of regions that the query area overlaps. For

example, if a query overlaps four regions, the query will be processed in parallel in four servers. We chose queries with the number of overlapped regions from one to 32.

NRA-STL-Li: We conducted NRA-STL-Li query in [2] as a baseline. The query uses R-Tree as an index for tweets. The R-Tree is stored in memory. Then, in each leaf of R-Tree, the corresponding STL is built and then stored in HBase. Similarly, STL is built in each non-leaf node. The model stores only a part of STLs corresponding to non-leaf nodes in HBase. We used the same length of STLs in non-leaf nodes as in the paper, with 220 and 430 for 15M and 50M dataset, respectively. The authors of [2] have proposed various queries, but through experiments, they found out that NRA-STL-Li has the best performance. Therefore, we chose NRA-STL-Li query as a baseline for our work.

Table 6.5: kFST queries

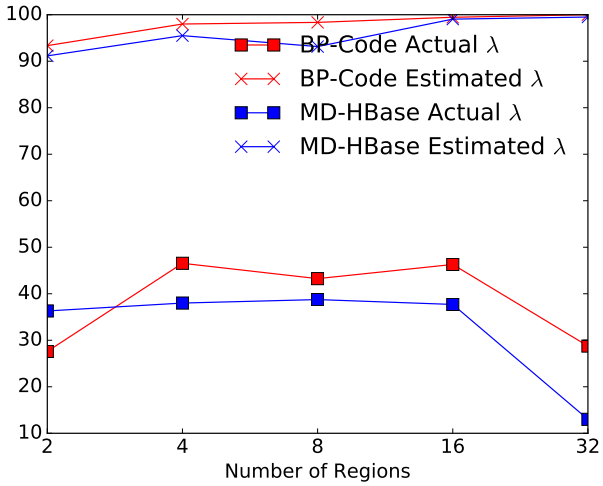
Algorithm	Description
NRA-STL-Li	Baseline
BPCodeQuery	Baseline
MD-HBase ST-P-NRA	Parallel kFST query with full STLs using NRA algorithm based on the index structure in MD-HBase
MD-HBase ST-P-RA	Parallel kFST query with full STLs using RA algorithm based on the index structure in MD-HBase
MD-HBase ST-λP	Parallel kFST query with shortened STLs based on the index structure in MD-HBase
BP-Code ST-P-NRA	Parallel kFST query with full STLs using NRA algorithm based on the index structure of BP-Code
BP-Code ST-P-RA	Parallel kFST query with full STLs using RA algorithm based on the index structure of BP-Code
BP-Code ST-λP	Parallel kFST query with shortened STLs based on the index structure of BP-Code

BPCodeQuery: As explained in section 6.4, BPCodeQuery is a nonparallel kFST query using BP-Code as index in HBase.

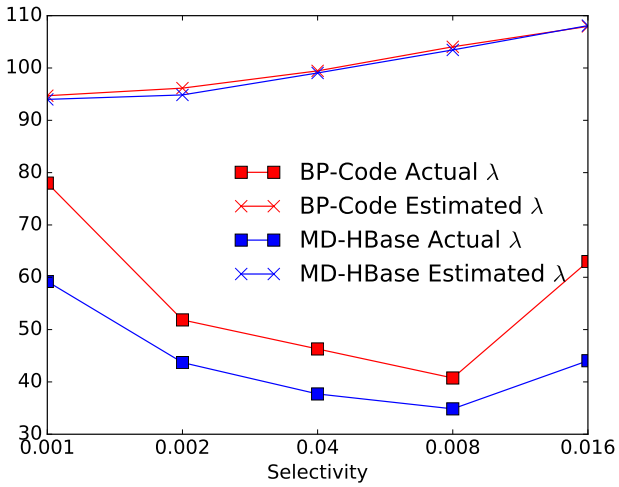
6.6.2 Parallel kFST Comparison

We evaluated the proposed parallel queries with our proposed index using BP-Code and the index structure in MD-HBase. Please note that we run these experiments using 50M dataset. Most of the experiments are conducted on the medium cluster, except the experiment with various clusters.

λ Estimation: We first compared the actual average STL list length accessed by the NRA algorithm and the theoretically estimated λ using the KD-Tree Partition Approach



(a) selectivity=0.004



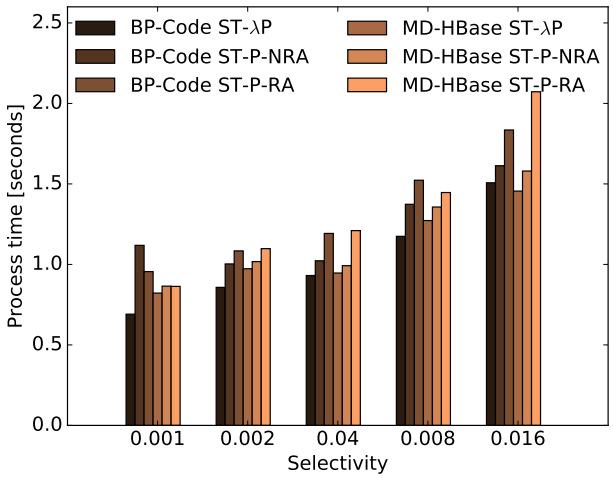
(b) 16 regions

Figure 6.9: λ Estimation

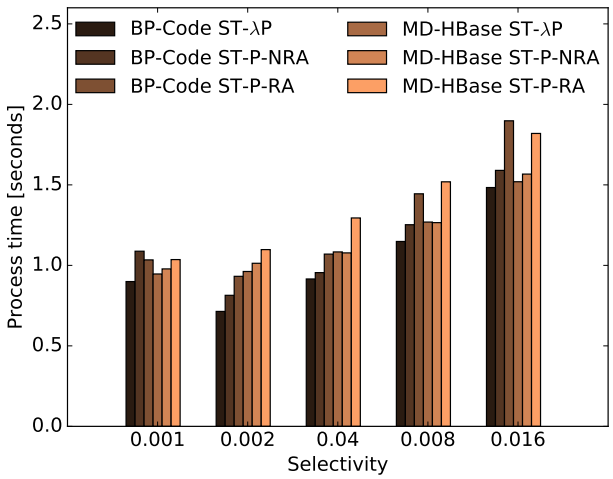
and the BP-Tree Partition Approach. Figures 6.9a shows the results in the 50M dataset with selectivity is 0.004 and number of regions varies from 2 to 16. Figure 6.9b describes the results in the 50M dataset with number of regions is 16 and selectivity varies from 0.001 to 0.016. As expected, the estimated λ overestimated the actual SLT length accessed in both approaches. Thanks to that, the ST-λP query can guarantee the query accuracy.

Because we added λ estimation step before processing kFST, the query had overhead for this step. However, in all cases, the overhead is very small in compared with the time for processing kFST computation. Therefore, the estimation step does not affect the query performance much.

Query Performance: We conducted experiments with all parallel kFST queries proposed in section 6.5. Figures 6.10a and 6.10b present the experimental results of all six queries with queries overlaps 8 and 16 regions, respectively. In each experiment, selectivity was varied from 0.001 to 0.016. Experiments that varies number of regions that the query overlaps was reported in Figures 6.11a and 6.11b. In all cases, with both index structure, the ST-λP queries outperformed the ST-P-NRA queries with all query selectivities. Returning shortened STLs instead of full STLs reduces I/O load between the client and HBase cluster. To ensure the query accuracy, if λ is underestimated, the query had to scan the database again to get full STLs. However, because we achieved overestimated λ, the query times were not affected.



(a) 8 regions



(b) 16 regions

Figure 6.10: Performance of Parallel kFST Queries with Various Selectivities

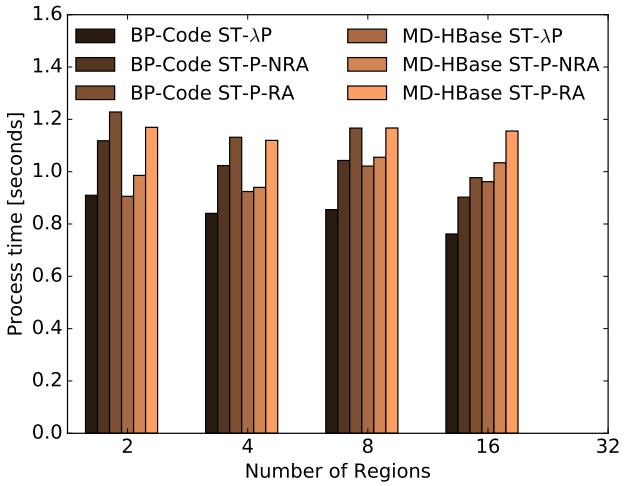
The ST-P-RA queries with both index structures often have longer query time than the ST-P-NRA query because it has to perform random access into STLs, causing more processing time. However, the RA algorithm often spends fewer iterations than the NRA algorithm, so in some cases, the ST-P-RA query offers better performance than the ST-P-NRA query. Nevertheless, we observed that in all cases, the ST-*lambda*P query outperformed the ST-P-RA query.

From the experimental results, we observed that the query time of all queries increased when the selectivity increased and decreased when the number of overlapped regions increased. When querying a larger selectivity, the number of data transferred between the client and the HBase cluster increased, causing to longer query time. On the other hand, the more number of overlapped regions is, the more number of parallel processing units is, leading to performance improvement. However, we did not achieve a linear increment of query performance because we only parallelized a part of the query. Other steps are still single processed in the client. Moreover, the more parallel processing units may cause more I/O bottleneck between the client and the distributed cluster. We further analyze the query performance in this case in Appendix A.

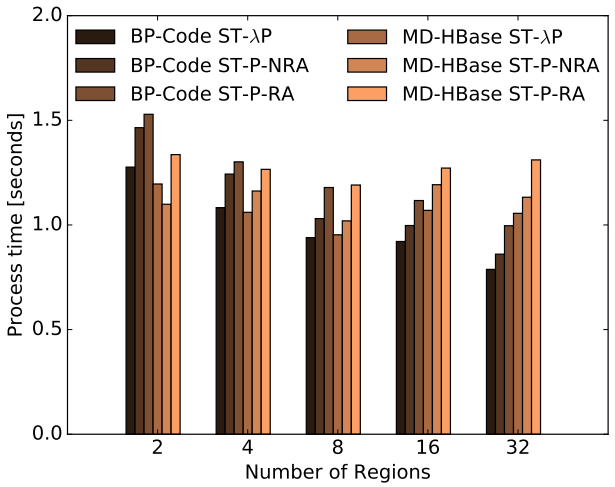
When looking at the contribution of the two index structure to the query performance, it is interesting that the queries using the MD-HBase index structure outperformed our proposed index when the number of overlapped regions are very small. However, with larger the number of regions, our proposed index offers better performance. Note that the index structure in MD-HBase splits the data space into more subspaces and each subspace stores a smaller number of points. Thus a query tends to overlap more region in MD-HBase than in our index structure. When the number of regions is still small, parallel processing in small regions provides a quicker response from the cluster, leading to lower processing time. However, as mentioned above, the more parallel processing units causes more I/O bottleneck, thus the MD-HBase index decreased the query performance when the number of regions increased.

To further compare the two index structure, we conducted the queries on all small, medium and large clusters (Figure 6.12a and 6.12b). As expected, queries using MD-HBase index structure increase the processing time drastically in the small clusters. Because the MD-HBase index often provides more number of overlapped regions, it requires more computing resources than our proposed index. BP-Code slightly increased query time in the small cluster, but it is more stable with different sizes of the cluster.

Estimation Overhead: Because we added λ estimation step before processing kFST, the query had overhead for this step. The overheads in ST-λP using BP-Code and MD-HBase index are shown along with query times in figures 6.13 and 6.14. In both datasets with all



(a) selectivity=0.002



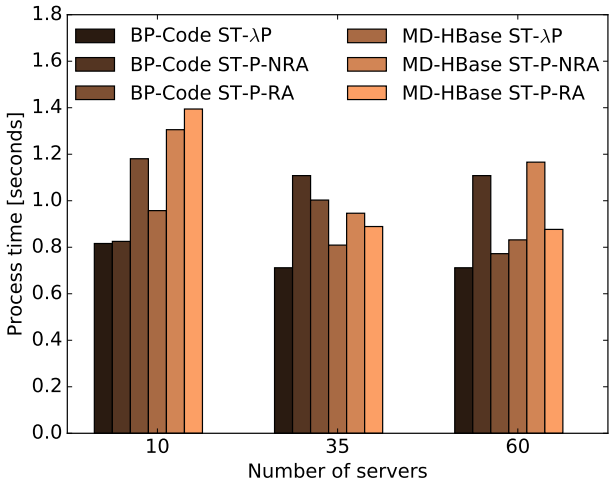
(b) selectivity=0.004

Figure 6.11: Performance of Parallel kFST Queries with Various Number of Regions

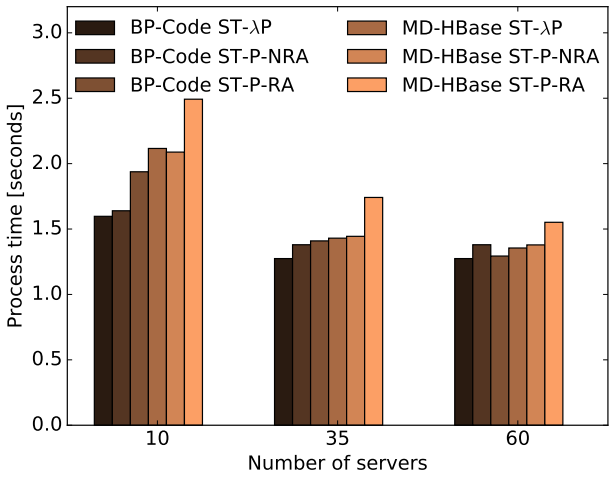
query selectivities, the overhead is very small in compared with the time for processing kFST computation. Therefore, the estimation step does not affect the query performance much.

6.6.3 Comparison with Baseline Approaches

As discussed in section 6.6.2, the BP-Code ST-λP has shown the best performance among proposed parallel kFST queries. We further proceeded a comparison between BP-Code ST-λP and the two baseline methods, NRA-STL-Li and BPCodeQuery in both query performance and space requirement aspects.



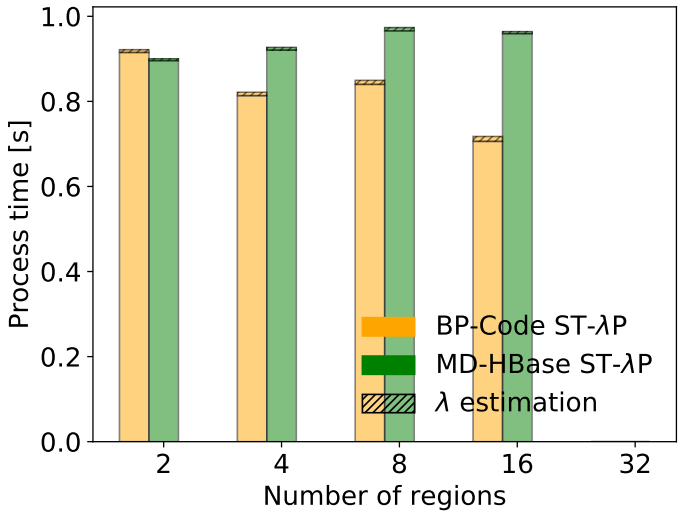
(a) selectivity=0.001, 8 regions



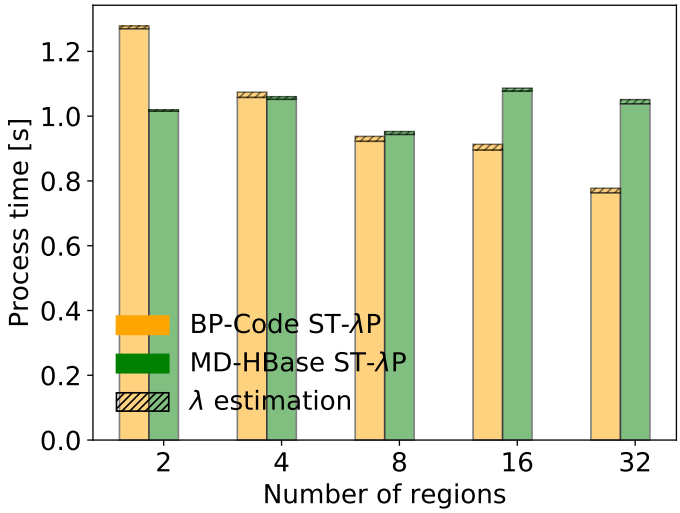
(b) selectivity=0.008, 32 regions

Figure 6.12: Performance of Parallel kFST Queries with Various Clusters

Space Requirements: Figure 6.15 shows the space requirements of our approach in compared with NRA-STL-Li for 15M and 50M datasets. NRA-STL-Li stores an R-Tree in memory and STLs in HBase. Our approach stores BP-Tree in the memory and stores data with BP-Code as the index in HBase. The in-memory BP-Tree is used almost for searching overlapped regions, so we do not need to keep all objects in the leaves of BP-Tree in memory. Therefore, the in-memory BP-Tree is only tens of kilobytes. Meanwhile, R-Tree stores all objects in the dataset, it requires almost the same space as the index the dataset by BP-Code in HBase. Moreover, NRA-STL-Li approach requires a large space to store STLs in HBase, causing the total space requirement is almost three times more than our approach.



(a) selectivity=0.002

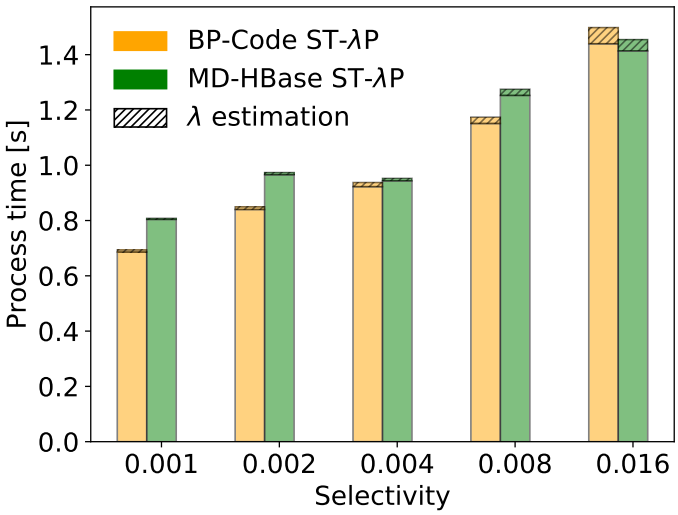


(b) selectivity=0.004

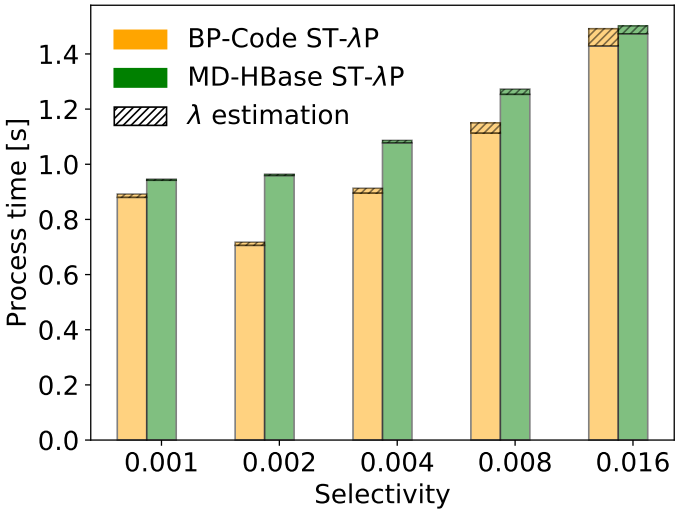
Figure 6.13: λ Estimation Overhead with Various Selectivities

Query Performance: We conducted experiments to compare our proposed parallel query ST-λP with the two baselines. We varied both query selectivity and number of overlapped regions to evaluate all three queries. The experiment with selectivity is fixed to 0.004 and the number of regions is varied from two to 32 is reported in Figure 6.16a. Figure 6.16b shows the experimental results of queries overlapping 16 regions with various selectivities.

We observed that the BPCodeQuery performed the worst because it has to scan and send all objects in query range to the client, causing a huge I/O overhead. The client also has



(a) 8 regions



(b) 16 regions

Figure 6.14: λ Estimation Overhead with Various Regions

to aggregate and sort the frequencies. NRA-STL-Li approach calculates and stores STLs beforehand, so it does not have to build STLs from scratch as BPCodeQuery.

However, NRA-STL-Li query times are still much larger than ST-λP. In Figure 6.16a, ST-λP was about three times faster than NRA-STL-Li. With various selectivities, ST-λP outperformed NRA-STL-Li from 2 to 11 times. The main reason is NRA-STL-Li involves more STLs in the computation than our proposed method. In NRA-STL-Li approach, each STL corresponds to a node in an R-Tree with maximum 100 entries. Meanwhile, in our approach, each STL is calculated from a region with hundreds of thousands to millions

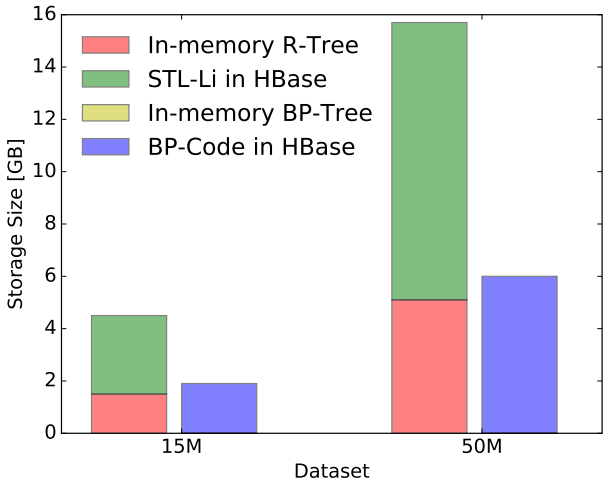


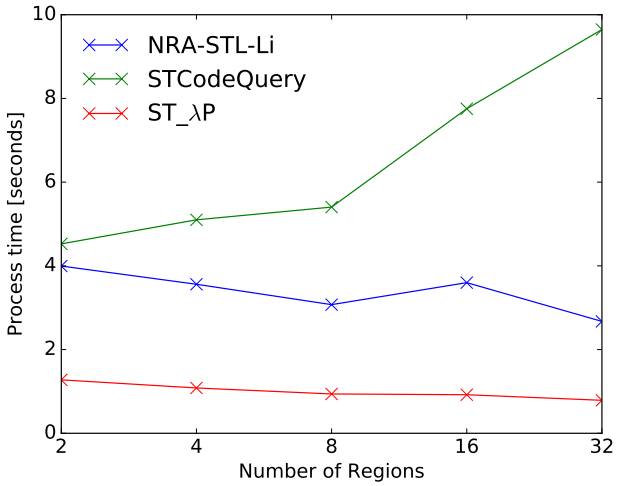
Figure 6.15: Space requirements

of records, therefore the number of involved STLs is much smaller than in NRA-STL-Li. Because top-k algorithms are known to degrade in performance when the number of lists is very large [2], the top-k algorithms in NRA-STL-Li take longer time. Moreover, even though ST-λP has to calculate the STLs from scratch, it brings this time-consuming step inside storage servers, executes the step in parallel, thus reduces processing time.

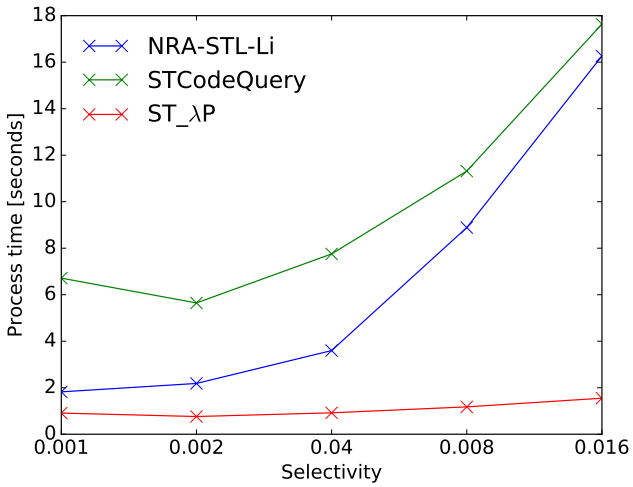
k Sensitive: We conducted an experiment with a various number of k to examine the behavior of NRA-STL-Li and ST-λP with different k. Query selectivity was varied from 0.01 to 0.05. In NRA-STL-Li approach, the authors estimate λ when indexing data into the database by assuming that k is fixed and known in advance. Therefore, as shown in figure 6.17b, the performance of NRA-STL-Li with some query selectivities was degraded greatly when k was increased. Note that, λ of NRA-STL-Li in this experiment was computed using $k = 10$. Conversely, our approach is more stable (as shown in figure 6.17a) when varying because we estimate λ on the fly based on the input k of the query.

6.7 Conclusions and Discussions

We presented a distributed spatiotemporal index on HBase that minimizes space requirements but better supports spatiotemporal query. We also proposed two parallel queries, ST-P, and ST-λP, to process kFST computation on top of the distributed index. Our proposed queries use an on-the-fly theoretical estimation to reduce the size of the STLs, thus improve the query performance.



(a) selectivity=0.004

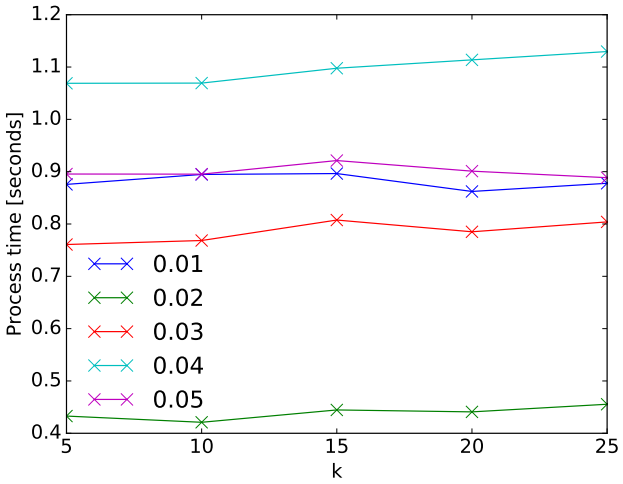


(b) 16 regions

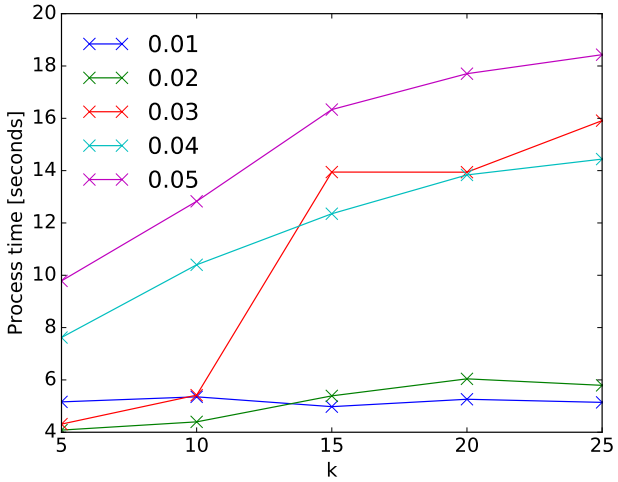
Figure 6.16: Baselines

In this paper, we evaluated BP-Code in experiments with kFST computation. However, because BP-Code generally indexes spatiotemporal data in KVS, it can support not only kFST but also basic spatiotemporal queries such as range query, kNN. Note that the spatiotemporal range query phase is included in the workflow to process kFST (as described in Section 6.4, therefore the index can directly be applied on range query. The parallel algorithm in ST-P query also can be utilized to process kNN in parallel using BP-Code.

Another problem that we would like to tackle is the top-k trending spatiotemporal terms (kTST) query because, in many applications, frequency measure does not capture trending terms effectively. To solve such problems, we can use BP-Code to index data, then apply



(a) ST-λP_OS Query



(b) NRA-STL-Li

Figure 6.17: Varies k in 15M dataset

ST-P to process trending computation in parallel. Instead of using top-k algorithms as in kFST, kTST requires trending term measurement methods to define trending terms. Thus, we need to propose effective trending term measurement methods, then based on the methods to design better parallel algorithms for kTST computation.

BP-Code is not limited to spatiotemporal data. It also can support spatial data with two or more dimensions. Hence, it is possible to use the index to store and query spatial and spatio-textual data in KVS. Besides basic spatial queries and join queries, there are some other queries that G-$^+$HBase is potential to provide. For example, spatial aggregation query that computes the aggregate function (e.g. count or sum) of a value (e.g., purchases,

incomes) in an application [58]. Our current setting would solve the problem of computing the frequency of a specific keyword given a spatial area.

Another noted query when working with spatio-textual data is top-k spatial keyword query which retrieves the k objects that are closest to the query location and contain the query keywords [25, 19]. This query is the combination of nearest neighbor search problem on spatial data and keyword search problem on text data. Even though the indexes in G-$^+$HBase efficiently support kNN query in spatial data, this query requires an index of both spatial and text information. Therefore, we need a further study to extend G-$^+$HBase in order to support such queries.

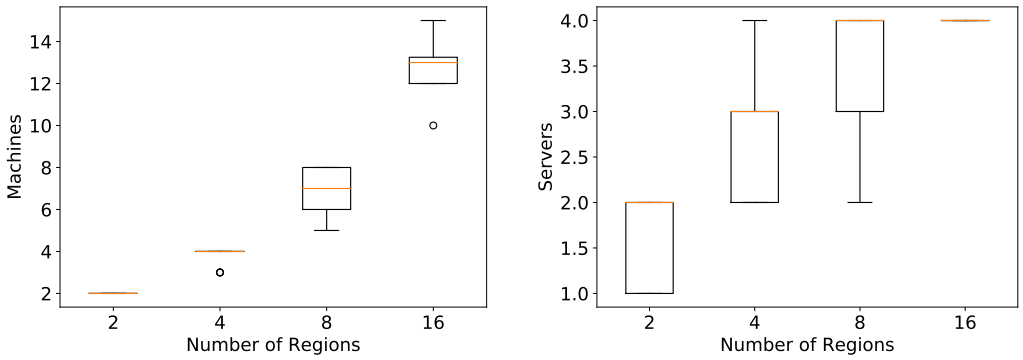
Appendix

A. Analysis of Query Performance with Various Number of Regions

We conducted the experiments of parallel processing in an HBase cluster including 36 virtual machines, one HMaster, and 35 RegionServers. The virtual machines were installed on four physical servers where each physical server hosts nine virtual machines. Each virtual machine has two cores.

In HBase, a RegionServer can host multiple regions. Therefore, the number of virtual machines to process a query is not equal to the number of regions involved in the query. For instance, as shown in Figure 6.18a, queries with 16 regions were processed in 10 to 15 virtual machines. Furthermore, each physical server hosts multiple virtual machines, thus the number of physical servers involved to a query is much smaller than the number of regions in the query (as shown in Figure 6.18b).

In order to balance the parallel computation, the number of regions of a query to be processed in a physical server should be even. If a physical server holds more regions than other servers, the process will take a longer time to finish. Figure 6.19 plots the maximum number of regions per physical server in queries with a selectivity of 0.002. The blue line plots the average number of regions in a physical server to balance the parallel computation. We observed that there are some queries that process more regions in a physical server than the average number of regions, leading to imbalance computation between physical servers and performance deterioration. Moreover, since all virtual machines on a host system share the same hardware resources, running too many virtual machines at the same time in a physical server can lead to bottlenecks.



(a) Virtual Machines (b) Physical Servers

Figure 6.18: Number of servers involved in queries (selectivity=0.002)

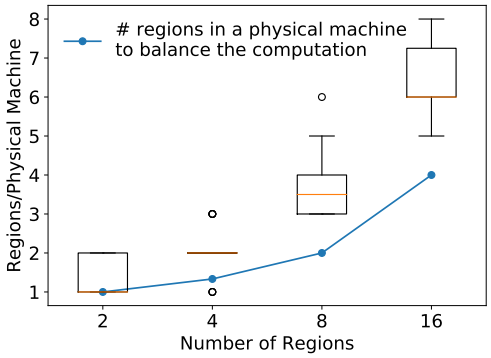


Figure 6.19: Maximum number of regions per physical server (selectivity=0.002)

We build G$^+$-HBase as a platform independent system, therefore we do not modify the storage layer in HBase. We try to balance the data partition on regions, then let HBase balance the number of regions in RegionServers. Therefore, there are queries that put the computation in a small number of RegionServers, leading to imbalance computation and unexpected query performance results. Additional tuning of cluster designation or a deeper modification on the storage layer in HBase would improve this situation.

Chapter 7

Summary

This thesis presents a novel distributed framework, G^+-HBase, for processing a large volume of spatial and spatiotemporal data.

We first have described G-HBase, a distributed database with native support for geographic data. G-HBase is equipped with various fundamental spatial queries in the *query processing* component by utilizing our proposed BGR Partitioning in the *indexes* component. Experiments with four real-world datasets including both points and non-point data demonstrated the high performance of the compute-intensive spatial queries in comparison with SpatialHadoop and other query algorithms in HBase. The results also showed that the overhead when using the proposed BGR Partitioning is a very small fraction of the process time of the queries.

Then, we proposed an efficient spatiotemporal index structure on top of HBase, a structure we consider to be essential in ITS. The efficiency of our index structure and the query algorithm was demonstrated through several experiments. Our evaluation showed the real-time performance of common spatiotemporal queries such as kNN, range query with response times of less than one second. Advanced queries for ITS such as average speed query could also be processed in just a few seconds. In future work, we plan to consider more spatiotemporal queries, e.g. a convex hull, spatial join, as long as other high-performance queries and analytics of ITSs such as online traffic monitoring queries or path planner, and accident detector. Another ongoing work is further query design to handle the imbalance among ranges of 3D.

To handle the imbalance among ranges of spatiotemporal data in the previous work, we proposed BP-Code, a balanced-partition code that transforms spatiotemporal data into one-dimensional code based on the density of the data. Then, we presented a distributed spatiotemporal index using BP-Code on HBase that minimizes space requirements but better supports spatiotemporal query. We also proposed two parallel queries, ST-P, and ST-λP,

to process kFST computation on top of the distributed index. Our proposed queries use an on-the-fly theoretical estimation to reduce the size of the STLs, thus improve the query performance.

Bibliography

[1] (2019). Geomesa. https://www.geomesa.org/. Last accessed 7 January 2019.

[2] Ahmed, P., Hasan, M., Kashyap, A., Hristidis, V., and Tsotras, V. J. (2017). Efficient computation of top-k frequent terms over spatio-temporal ranges. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD)*, pages 1227–1241. ACM.

[3] Aji, A., Wang, F., Vo, H., Lee, R., Liu, Q., Zhang, X., and Saltz, J. (2013). Hadoop-GIS: a high performance spatial data warehousing system over MapReduce. *Proceedings of the VLDB Endowment*, 6(11):1009–1020.

[4] Akdogan, A., Demiryurek, U., Banaei-Kashani, F., and Shahabi, C. (2010). Voronoi-based geospatial query processing with MapReduce. In *Proceedings of IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 9–16. IEEE.

[5] Alarabi, L. (2017). ST-Hadoop: a MapReduce framework for big spatio-temporal data. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD)*, pages 40–42. ACM.

[6] Alarabi, L., Mokbel, M. F., and Musleh, M. (2018). ST-Hadoop: a MapReduce framework for spatio-temporal data. *GeoInformatica*, 22(4):785–813.

[7] Aref, W. G. and Samet, H. (1990). Efficient processing of window queries in the pyramid data structure. In *Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 265–272. ACM.

[8] Arge, L., De Berg, M., Haverkort, H. J., and Yi, K. (2004). The priority R-tree: A practically efficient and worst-case optimal R-tree. In *Proceedings of the 2004 ACM International Conference on Management of Data (SIGMOD)*, pages 347–358. ACM.

[9] Beckmann, N., Kriegel, H.-P., Schneider, R., and Seeger, B. (1990). The r*-tree: an efficient and robust access method for points and rectangles. In *ACM SIGMOD Record*, volume 19, pages 322–331. ACM.

[10] Belussi, A., Migliorini, S., and Eldawy, A. (2018). Detecting skewness of big spatial data in SpatialHadoop. In *Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 432–435. ACM.

[11] Bentley, J. L. (1975). Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517.

[12] Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426.

[13] Budak, C., Georgiou, T., Agrawal, D., and El Abbadi, A. (2013). Geoscope: Online detection of geo-correlated information trends in social networks. *Proceedings of the VLDB Endowment*, 7(4):229–240.

[14] Callahan, P. B. and Kosaraju, S. R. (1995). A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *Journal of the ACM (JACM)*, 42(1):67–90.

[15] Câmara, G., Vinhas, L., Ferreira, K. R., De Queiroz, G. R., De Souza, R. C. M., Monteiro, A. M. V., De Carvalho, M. T., Casanova, M. A., and De Freitas, U. M. (2008). TerraLib: An open source gis library for large-scale environmental and socio-economic applications. In *Open source approaches in spatial data handling*, pages 247–270. Springer.

[16] Cary, A., Sun, Z., Hristidis, V., and Rishe, N. (2009). Experiences on processing spatial data with MapReduce. In *Proceedings of International Conference on Scientific and Statistical Database Management*, pages 302–319. Springer.

[17] Cattell, R. (2011). Scalable sql and nosql data stores. *ACM SIGMOD Record*, 39(4):12–27.

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

[19] Chen, L., Cong, G., Jensen, C. S., and Wu, D. (2013). Spatial keyword query processing: an experimental evaluation. *Proceedings of the VLDB Endowment*, 6(3):217–228.

[20] Chen, X., Zhang, C., Ge, B., and Xiao, W. (2015a). Spatio-temporal queries in HBase. In *Proceedings of IEEE International Conference on Big Data (IEEE Big Data)*, pages 1929–1937. IEEE.

[21] Chen, X., Zhang, C., Ge, B., and Xiao, W. (2016). Efficient historical query in HBase for spatio-temporal decision support. *International Journal of Computers Communications & Control*, 11(5):613–630.

[22] Chen, X., Zhang, C., Shi, Z., and Xiao, W. (2015b). Spatio-temporal keywords queries in HBase. *Big Data and Information Analytics*, 1(1):81–91.

[23] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. (2010). Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud Computing*, pages 143–154. ACM.

[24] Cunha, E., Magno, G., Comarela, G., Almeida, V., Gonçalves, M. A., and Benevenuto, F. (2011). Analyzing the dynamic evolution of hashtags on twitter: a language-based approach. In *Proceedings of the Workshop on Languages in Social Media*, pages 58–65. Association for Computational Linguistics.

[25] De Felipe, I., Hristidis, V., and Rishe, N. (2008). Keyword search on spatial databases. In *Proceedings of IEEE 24th International Conference on Data Engineering (ICDE)*, pages 656–665. IEEE.

[26] Dean, J. and Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113.

[27] Decandia, G. (2007). Dynamo: Amazon's highly available key-value store. In *Proceedings of 21st ACM Symposium on Operating Systems Principles (SOSP)*, pages 205–220. ACM.

[28] Dimiduk, N., Khurana, A., Ryan, M. H., and Stack, M. (2013). *HBase in action*. Manning Shelter Island.

[29] Dittrich, J.-P. and Seeger, B. (2000). Data redundancy and duplicate detection in spatial join processing. In *Proceedings of 16th International Conference on Data Engineering (ICDE)*, pages 535–546. IEEE.

[30] Du, N., Zhan, J., Zhao, M., Xiao, D., and Xie, Y. (2015). Spatio-temporal data index model of moving objects on fixed networks using HBase. In *Proceedings of IEEE International Conference on Computational Intelligence & Communication Technology (CICT)*, pages 247–251. IEEE.

[31] Eldawy, A., Li, Y., Mokbel, M. F., and Janardan, R. (2013). Cg_Hadoop: computational geometry in MapReduce. In *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 294–303. ACM.

[32] Eldawy, A. and Mokbel, M. F. (2013). A demonstration of SpatialHadoop: An efficient MapReduce framework for spatial data. *Proceedings of the VLDB Endowment*, 6(12):1230–1233.

[33] Eldawy, A. and Mokbel, M. F. (2015). SpatialHadoop: A MapReduce framework for spatial data. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 1352–1363. IEEE.

[34] Eldawy, A., Mokbel, M. F., Alharthi, S., Alzaidy, A., Tarek, K., and Ghani, S. (2015). Shahed: A MapReduce-based system for querying and visualizing spatio-temporal satellite data. In *Proceedings of IEEE 31st International Conference on Data Engineering (ICDE)*, pages 1585–1596. IEEE.

[35] Fagin, R. (1999). Combining fuzzy information from multiple systems. *Journal of Computer and System Sciences*, 58(1):83–99.

[36] Fagin, R., Lotem, A., and Naor, M. (2003). Optimal aggregation algorithms for middleware. *Journal of computer and system sciences*, 66(4):614–656.

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

[38] Fox, A., Eichelberger, C., Hughes, J., and Lyon, S. (2013). Spatio-temporal indexing in non-relational distributed databases. In *Proceedings of 2013 IEEE International Conference on Big Data (IEEE Big Data)*, pages 291–299. IEEE.

[39] Friedman, J., Bentley, J., and Finkel, A. R. (1977). An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209–226.

[40] Fries, S., Boden, B., Stepien, G., and Seidl, T. (2014). PHiDJ: Parallel similarity self-join for high-dimensional vector data with MapReduce. In *Proceedings of IEEE 30th International Conference on Data Engineering (ICDE)*, pages 796–807. IEEE.

[41] George, L. (2011). *HBase: the definitive guide*. O'Reilly Media, Inc.

[42] Guttman, A. (1984). R-trees: A dynamic index structure for spatial searching. In *Proceedings of ACM SIGMOD*, pages 47–57. ACM.

[43] Hamilton, C. H. and Rau-Chaplin, A. (2008). Compact hilbert indices: Space-filling curves for domains with unequal side lengths. *Information Processing Letters*, 105(5):155–163.

[44] Han, D. and Stroulia, E. (2013). Hgrid: A data model for large geospatial data sets in HBase. In *Proceedings of IEEE Sixth International Conference on Cloud Computing (CLOUD)*, pages 910–917. IEEE.

[45] Henrich, A. (1996). Improving the performance of multi-dimensional access structures based on kd-trees. In *Proceedings of the Twelfth International Conference on Data Engineering (ICDE)*, pages 68–75. IEEE.

[46] Hilbert, D. (1935). Über die stetige abbildung einer linie auf ein flächenstück. In *Dritter Band: Analysis· Grundlagen der Mathematik· Physik Verschiedenes*, pages 1–2. Springer.

[47] Hsu, Y.-T., Pan, Y.-C., Wei, L.-Y., Peng, W.-C., and Lee, W.-C. (2012). Key formulation schemes for spatial index in cloud data managements. In *Proceedings of IEEE 13th International Conference on Mobile Data Management (MDM)*, pages 21–26. IEEE.

[48] Hunt, P. (2010). Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of USENIX Annual Technical Conference*, pages 145–158.

[49] Ilyas, I. F., Beskales, G., and Soliman, M. A. (2008). A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys (CSUR)*, 40(4):11.

[50] Inches, G., Carman, M. J., and Crestani, F. (2010). Statistics of online user-generated short documents. In *Proceedings of European Conference on Information Retrieval*, pages 649–652. Springer.

[51] Ježek, J. and Kolingerová, I. (2014). STCode: The text encoding algorithm for latitude/longitude/time. In *Connecting a Digital Europe Through Location and Place*, pages 163–177. Springer.

[52] Jonathan, C., Magdy, A., Mokbel, M. F., and Jonathan, A. (2016). Garnet: A holistic system approach for trending queries in microblogs. In *Proceedings of IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 1251–1262. IEEE.

[53] Junqueira, F. and Reed, B. (2013). *ZooKeeper: distributed process coordination*. O'Reilly Media, Inc.

[54] Kamel, I. and Faloutsos, C. (1994). Hilbert R-tree: An improved R-tree using fractals. *Proceedings of the VLDB Endowment*, pages 500–509.

[55] Koudas, N. and Sevcik, K. C. (1997). Size separation spatial join. *ACM SIGMOD Record*, 26(2):324–335.

[56] Kumar, S., Madria, S., and Linderman, M. (2017). M-Grid: a distributed framework for multidimensional indexing and querying of location based data. *Distributed and Parallel Databases*, 35(1):55–81.

[57] Lakshman, A. and Malik, P. (2010). Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40.

[58] Lazaridis, I. and Mehrotra, S. (2001). Progressive approximate aggregate queries with a multi-resolution tree structure. *ACM SIGMOD Record*, 30(2):401–412.

[59] Le, H. V. and Takasu, A. (2015a). An efficient distributed index for geospatial databases. In *Proceedings of 26th International Conference on Database and Expert Systems Applications (DEXA)*, pages 28–42. Springer.

[60] Le, H. V. and Takasu, A. (2015b). A scalable spatio-temporal data storage for intelligent transportation systems based on HBase. In *Proceedings of IEEE 18th International Conference on Intelligent Transportation Systems (ITSC)*, pages 2733–2738. IEEE.

[61] Le, H. V. and Takasu, A. (2018). G-HBase: A high performance geographical database based on HBase. *IEICE Transactions on Information and Systems*, 101(4):1053–1065.

[62] Lee, J.-G. and Kang, M. (2015). Geospatial big data: challenges and opportunities. *Big Data Research*, 2(2):74–81.

[63] Lee, K., Ganti, R. K., Srivatsa, M., and Liu, L. (2014). Efficient spatial query processing for big data. In *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 469–472. ACM.

[64] Leutenegger, S. T., Lopez, M. A., and Edgington, J. (1997). STR: A simple and efficient algorithm for R-tree packing. In *Proceedings of 13th international conference on Data Engineering (ICDE)*, pages 497–506. IEEE.

[65] Li, Z., Hu, F., Schnase, J. L., Duffy, D. Q., Lee, T., Bowen, M. K., and Yang, C. (2017). A spatiotemporal indexing approach for efficient processing of big array-based climate data with MapReduce. *International Journal of Geographical Information Science*, 31(1):17–35.

[66] Lo, M.-L. and Ravishankar, C. V. (1994). Spatial joins using seeded trees. *ACM SIGMOD Record*, 23(2):209–220.

[67] Lo, M.-L. and Ravishankar, C. V. (1996). Spatial hash-joins. *ACM SIGMOD Record*, 25(2):247–258.

[68] Lu, J. and Guting, R. H. (2012). Parallel Secondo: boosting database engines with Hadoop. In *Proceedings of IEEE 18th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 738–743. IEEE.

[69] Lu, W., Shen, Y., Chen, S., and Ooi, B. C. (2012). Efficient processing of k nearest neighbor joins using MapReduce. *Proceedings of the VLDB Endowment*, 5(10):1016–1027.

[70] Ma, Q., Yang, B., Qian, W., and Zhou, A. (2009). Query processing of massive trajectory data based on MapReduce. In *Proceedings of the first International Workshop on Cloud Data Management*, pages 9–16. ACM.

[71] Ma, Y., Zhang, Y., and Meng, X. (2013). ST-HBase: a scalable data management system for massive geo-tagged objects. In *Proceedings of International Conference on Web-Age Information Management (WAIM)*, pages 155–166. Springer.

[72] Magdy, A., Aly, A. M., Mokbel, M. F., Elnikety, S., He, Y., Nath, S., and Aref, W. G. (2016). GeoTrend: spatial trending queries on real-time microblogs. In *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, page 7. ACM.

[73] Magdy, A., Mokbel, M. F., Elnikety, S., Nath, S., and He, Y. (2014). Mercury: A memory-constrained spatio-temporal real-time search on microblogs. In *Proceedings of IEEE 30th International Conference on Data Engineering (ICDE)*, pages 172–183. IEEE.

[74] Mahmood, A. and Aref, W. G. (2017). Query processing techniques for big spatial-keyword data. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD)*, pages 1777–1782. ACM.

[75] Mahmood, A. R., Aly, A. M., Qadah, T., Rezig, E. K., Daghistani, A., Madkour, A., Abdelhamid, A. S., Hassan, M. S., Aref, W. G., and Basalamah, S. (2015). Tornado: A distributed spatio-textual stream processing system. *Proceedings of the VLDB Endowment*, 8(12):2020–2023.

[76] Mahmood, A. R., Punni, S., and Aref, W. G. (2018). Spatio-temporal access methods: a survey (2010-2017). *GeoInformatica*, pages 1–36.

[77] Maneewongvatana, S. and Mount, D. M. (1999). It's okay to be skinny, if your friends are fat. In *Proceedings of Center for Geometric Computing 4th Annual Workshop on Computational Geometry*, volume 2, pages 1–8.

[78] Mishra, P. and Eich, M. H. (1992). Join processing in relational databases. *ACM Computing Surveys (CSUR)*, 24(1):63–113.

[79] Mokbel, M. F., Ghanem, T. M., and Aref, W. G. (2003). Spatio-temporal access methods. *IEEE Data Engineering Bulletin*, 26(2):40–49.

[80] Moon, B., Jagadish, H. V., Faloutsos, C., and Saltz, J. H. (2001). Analysis of the clustering properties of the hilbert space-filling curve. *IEEE Transactions on Knowledge and Data Engineering*, 13(1):124–141.

[81] Morton, G. M. (1966). *A computer oriented geodetic data base and a new technique in file sequencing*. International Business Machines Company.

[82] Nepal, S. and Ramakrishna, M. (1999). Query processing issues in image (multimedia) databases. In *Proceedings of 15th International Conference on Data Engineering (ICDE)*, pages 22–29. IEEE.

[83] Nguyen, K. and Tran, D. A. (2011). An analysis of activities in facebook. In *Proceedings of IEEE Consumer Communications and Networking Conference (CCNC)*, pages 388–392. IEEE.

[84] Nguyen-Dinh, L.-V., Aref, W. G., and Mokbel, M. F. (2010). Spatio-temporal access methods: Part 2 (2003-2010). *Data Engineering*, page 46.

[85] Nishimura, S., Das, S., Agrawal, D., and Abbadi, A. E. (2011). MD-HBase: a scalable multi-dimensional data infrastructure for location aware services. In *Proceedings of 12th IEEE International Conference on Mobile Data Management (MDM)*, volume 1, pages 7–16. IEEE.

[86] Nobari, S., Tauheed, F., Heinis, T., Karras, P., Bressan, S., and Ailamaki, A. (2013). TOUCH: in-memory spatial join by hierarchical data-oriented partitioning. In *Proceedings of the 2013 ACM International Conference on Management of Data (SIGMOD)*, pages 701–712. ACM.

[87] Orenstein, J. A. (1982). Multidimensional tries used for associative searching. *Information Processing Letters*, 14(4):150–157.

[88] Orenstein, J. A. (1989). Redundancy in spatial databases. *ACM SIGMOD Record*, 18(2):295–305.

[89] Pal, S., Das, I., Majumder, S., Gupta, A. K., and Bhattacharya, I. (2015). Embedding an extra layer of data compression scheme for efficient management of big-data. In *Information Systems Design and Intelligent Applications*, pages 699–708. Springer.

[90] Patel, J. M. and DeWitt, D. J. (1996). Partition based spatial-merge join. *ACM SIGMOD Record*, 25(2):259–270.

[91] Patel, J. M. and DeWitt, D. J. (2000). Clone join and shadow join: two parallel spatial join algorithms. In *Proceedings of 8th ACM International Symposium on Advances in Geographic Information Systems (ACM GIS)*, pages 54–61. ACM.

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

[93] Sagan, H. (2012). *Space-filling curves*. Springer Science & Business Media.

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

[95] Samet, H. (2006). *Foundations of multidimensional and metric data structures*. Morgan Kaufmann.

[96] Sawyer, S. M., O'gwynn, B. D., Tran, A., and Yu, T. (2013). Understanding query performance in accumulo. In *Proceedings of IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE.

[97] Schumacker, R. A., Brand, B., Gilliland, M. G., and Sharp, W. H. (1969). Study for applying computer-generated images to visual simulation. Technical report, DTIC Document.

[98] Sellis, T. K., Roussopoulos, N., and Faloutsos, C. (1987). The R+-Tree: A dynamic index for multi-dimensional objects. *Proceedings of the VLDB Endowment*, pages 507–518.

[99] Shvachko, K., Kuang, H., Radia, S., and Chansler, R. (2010). The Hadoop distributed file system. In *Proceedings of IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10. IEEE.

[100] Skovsgaard, A., Sidlauskas, D., and Jensen, C. S. (2014). Scalable top-k spatio-temporal term querying. In *Proceedings of IEEE 30th International Conference on Data Engineering (ICDE)*, pages 148–159. IEEE.

[101] Sun, H., Tang, Y., Wang, Q., and Liu, X. (2017). Handling multi-dimensional complex queries in key-value data stores. *Information Systems*, 66:82–96.

[102] Tan, H., Luo, W., and Ni, L. M. (2012). Clost: a Hadoop-based storage system for big spatio-temporal data analytics. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management (CIKM)*, pages 2139–2143. ACM.

[103] Tang, X., Han, B., and Chen, H. (2016). A hybrid index for multi-dimensional query in HBase. In *Proceedings of 4th International Conference on Cloud Computing and Intelligence Systems (CCIS)*, pages 332–336. IEEE.

[104] Thusoo, A., Sarma, J. S., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., and Murthy, R. (2009). Hive: a warehousing solution over a Map-Reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629.

[105] Toshniwal, A., Taneja, S., Shukla, A., Ramasamy, K., Patel, J. M., Kulkarni, S., Jackson, J., Gade, K., Fu, M., Donham, J., et al. (2014). Storm@ twitter. In *Proceedings of the 2014 ACM International Conference on Management of Data (SIGMOD)*, pages 147–156. ACM.

[106] Varia, J. (2008). Cloud architectures. *White Paper of Amazon*, 16.

[107] Vo, H., Aji, A., and Wang, F. (2014). Sato: a spatial data partitioning framework for scalable query processing. In *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 545–548. ACM.

[108] Whitby, M. A., Fecher, R., and Bennight, C. (2017). Geowave: utilizing distributed key-value stores for multidimensional data. In *International Symposium on Spatial and Temporal Databases (SSTD)*, pages 105–122. Springer.

[109] White, D. A. and Jain, R. (1996). Algorithms and strategies for similarity retrieval. In *Proceedings of the SPIE Conference*.

[110] White, T. (2012). *Hadoop: The definitive guide*. O'Reilly Media, Inc.

[111] Whitman, R. T., Park, M. B., Ambrose, S. M., and Hoel, E. G. (2014). Spatial indexing and analytics on Hadoop. In *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 73–82. ACM.

[112] Yokoyama, T., Ishikawa, Y., and Suzuki, Y. (2012). Processing all k-nearest neighbor queries in Hadoop. In *Proceedings of International Conference on Web-Age Information Management (WAIM)*, pages 346–351. Springer.

[113] You, S., Zhang, J., and Gruenwald, L. (2015). Large-scale spatial join query processing in cloud. In *Proceedings of 31st IEEE International Conference on Data Engineering Workshops (ICDEW)*, pages 34–41. IEEE.

[114] Yuan, J., Zheng, Y., Xie, X., and Sun, G. (2011). Driving with knowledge from the physical world. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 316–324. ACM.

[115] Yuan, J., Zheng, Y., Zhang, C., Xie, W., Xie, X., Sun, G., and Huang, Y. (2010). T-drive: driving directions based on taxi trajectories. In *Proceedings of the 18th SIGSPATIAL International conference on advances in geographic information systems*, pages 99–108. ACM.

[116] Zhang, C., Chen, X., Feng, X., and Ge, B. (2016). Storing and querying semi-structured spatio-temporal data in HBase. In *Proceedings of International Conference on Web-Age Information Management (WAIM)*, pages 303–314. Springer.

[117] Zhang, S., Han, J., Liu, Z., Wang, K., and Feng, S. (2009a). Spatial queries evaluation with MapReduce. In *Proceedings of Eighth International Conference on Grid and Cooperative Computing (GCC)*, pages 287–292. IEEE.

[118] Zhang, S., Han, J., Liu, Z., Wang, K., and Xu, Z. (2009b). Sjmr: Parallelizing spatial join with MapReduce on clusters. In *Proceedings of IEEE international conference on Cluster Computing and Workshops (CLUSTER)*, pages 1–8. IEEE.

[119] Zhou, X., Zhang, X., Wang, Y., Li, R., and Wang, S. (2013). Efficient distributed multi-dimensional index for big data management. In *Proceedings of International Conference on Web-Age Information Management (WAIM)*, pages 130–141. Springer.